

Freeman • Hupfer • Arnold

JavaSpaces™ Principles, Patterns, and Practice

Foreword by David Gelernter, Yale University

The Jini™ Technology Series

Includes
Official Specs
from Sun
Microsystems

... from the Source™

Sun
microsystems

JINI™

JavaSpacesTM

Principles, Patterns, and Practice

The Jini™ Technology Series

Lisa Friendly, Series Editor

Ken Arnold, Technical Editor

For more information see: <http://java.sun.com/docs/books/jini/>

This series, written by those who design, implement, and document the Jini™ technology, shows how to use, deploy, and create Jini applications. Jini technology aims to erase the hardware/software distinction, to foster spontaneous networking among devices, and to make pervasive a service-based architecture. In doing so, the Jini architecture is radically changing the way we think about computing. Books in **The Jini Technology Series** are aimed at serious developers looking for accurate, insightful, thorough, and practical material on Jini technology.

The Jini Technology Series web site contains detailed information on the Series, including existing and upcoming titles, updates, errata, sources, sample code, and other Series-related resources.

Ken Arnold, Bryan O'Sullivan, Robert W. Scheifler, Jim Waldo, Ann Wollrath, *The Jini™ Specification*

ISBN 0-201-61634-3

Eric Freeman, Susanne Hupfer, and Ken Arnold, *JavaSpaces™ Principles, Patterns, and Practice*

ISBN 0-201-30955-6

JavaSpacesTM

Principles, Patterns, and Practice

**Eric Freeman
Susanne Hupfer
Ken Arnold**



ADDISON-WESLEY

An imprint of Addison Wesley Longman, Inc.
Reading, Massachusetts • Harlow, England • Menlo Park, California
Berkeley, California • Don Mills, Ontario • Sydney
Bonn • Amsterdam • Tokyo • Mexico City

Copyright © 1999 by Sun Microsystems, Inc.
901 San Antonio Road, Palo Alto, California 94303 U.S.A.
All rights reserved.

Duke™ designed by Joe Palrang.

Sun Microsystems, Inc. has intellectual property rights ("Sun IPR") relating to implementations of the technology described in this publication ("the Technology"). In particular, and without limitation, Sun IPR may include one or more U.S. patents, foreign patents, or pending applications. Your limited right to use this publication does not grant you any right or license to Sun IPR nor any right or license to implement the Technology. Sun may, in its sole discretion, make available a limited license to Sun IPR and/or to the Technology under a separate license agreement. Please visit <http://www.sun.com/software/communitysource/>.

Sun, Sun Microsystems, the Sun Logo, and all Sun, Java, Jini, and Solaris based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd.

THIS PUBLICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION. SUN MICROSYSTEMS, INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN ANY TECHNOLOGY, PRODUCT, OR PROGRAM DESCRIBED IN THIS PUBLICATION AT ANY TIME

Library of Congress Cataloging-in-Publication Data

Freeman, Eric, 1965-

JavaSpaces principles, patterns, and practice / Eric Freeman,
Susanne Hupfer, Ken Arnold.

p. cm. – (Java series)

Includes bibliographical references and index.

ISBN 0-201-30955-6 (alk. paper)

1. Java (Computer program language) 2. Javaspaces technology.
3. Electronic data processing—Distributed processing. I. Hupfer, Susanne, 1965-. II. Arnold, Ken. III. Title. IV. Series.

QA76.73.J38F74 1999

005.2'76-DC21

99-27366

CIP

The publisher offers discounts on this book when ordered in quantity for special sales. For more information, please contact: Corporate, Government and Special Sales; Addison Wesley Longman, Inc.; One Jacob Way; Reading, Massachusetts 01867.

Text printed on recycled and acid-free paper.

ISBN 0201309556

2 3 4 5 6 7 CRS 02 01 00 99

2nd Printing August 1999

With love, to Elisabeth – EF

*To my loving parents, Bernard and Monika, and
in memory of my grandparents – SH*

With love, to Jareth and Cory – KA

Contents

Preface	xv
Foreword	xix
1 Introduction.....	1
1.1 Benefits of Distributed Computing	2
1.2 Challenges of Distributed Computing.....	3
1.3 What Is JavaSpaces Technology?.....	4
1.3.1 Key Features	6
1.3.2 JavaSpaces Technology in Context.....	8
1.4 JavaSpaces Technology Overview	9
1.4.1 Entries and Operations.....	9
1.4.2 Going Further	11
1.5 Putting It All Together	14
1.6 Advantages of JavaSpaces Technologies	16
1.7 Chapter Preview	17
1.8 Exercises	19
2 JavaSpaces Application Basics	21
2.1 Entries	22
2.1.1 The <code>Entry</code> Interface	22
2.1.2 Instantiating an Entry.....	23
2.1.3 Adding Fields and Methods.....	23
2.2 Building an Application	25
2.2.1 The <code>SpaceAccessor</code>	26
2.3 Writing Entries into a Space	28
2.3.1 A Closer Look at <code>write</code>	30
2.4 Reading and Taking Entries	31
2.4.1 Associative Lookup.....	31
2.4.2 The Basics of <code>read</code> and <code>take</code>	32
2.4.3 The Rules of Matching	33
2.4.4 Dealing with <code>null</code> -valued Fields in Entries.....	35
2.4.5 Primitive Fields	36

2.4.6 A Closer Look at <code>read</code> and <code>take</code>	36
2.5 Going Further with the Example	38
2.5.1 Subclassing an Entry	39
2.5.2 Adding a Few More Methods	40
2.5.3 Trying the Game	42
2.5.4 Post-Game Analysis	44
2.6 Serialization and Its Effects	44
2.6.1 Entry Serialization	45
2.6.2 Matching and Equality	47
2.6.3 The No-arg Constructor	47
2.6.4 The <code>Entry</code> Interface Revisited	48
2.6.5 Improving Entry Serialization Using <code>snapshot</code>	48
2.7 Summary	49
3 Building Blocks	51
3.1 Introduction to Distributed Data Structures	52
3.1.1 Building Distributed Data Structures with Entries	53
3.2 Shared Variables	56
3.2.1 Atomic Modification	57
3.2.2 Additional Operations	58
3.2.3 Creating a Web Counter	59
3.2.4 Stepping Back	62
3.3 Unordered Structures	63
3.3.1 Bags	64
3.3.2 Task Bags and Result Bags	65
3.4 Ordered Structures	66
3.4.1 Distributed Arrays Revisited	67
3.5 Summary	73
3.6 Exercises	74
4 Synchronization	75
4.1 Semaphores	76
4.1.1 Implementing a Semaphore	77
4.1.2 Implementing a License Manager	79
4.1.3 The License Manager Installer	79
4.1.4 The License Manager Client Library	80
4.2 Using Multiple Semaphores	81
4.2.1 The Dining Philosophers	82
4.3 Fairly Sharing a Resource	87
4.3.1 Using a Queue to Take Turns	88
4.3.2 Round-Robin Synchronization	92

4.4	Barrier Synchronization	95
4.5	Advanced Synchronization: The Readers/Writers Problem	97
4.5.1	Implementing a Readers/Writers Solution	98
4.5.2	Implementing a Counter	99
4.5.3	Implementing a Space-based Readers/Writers Application	101
4.6	Summary	105
4.7	Exercises	105
5	Communication.....	107
5.1	Basic Message Passing	108
5.1.1	Playing Ping-Pong.....	109
5.2	Characteristics of Space-based Communication.....	113
5.2.1	Tightly Coupled Communication	113
5.2.2	Loosely Coupled Communication.....	114
5.2.3	Benefits of Loose Coupling.....	115
5.3	Beyond Message Passing	116
5.4	A Basic Channel	116
5.4.1	The Channel Message	117
5.4.2	The Channel Tail	118
5.4.3	Creating a Channel	118
5.4.4	Appending a Message to a Channel.....	119
5.4.5	Implementing a Channel Writer	121
5.4.6	Implementing a Channel Reader.....	123
5.4.7	Demonstrating the Channel Writer and Reader.....	126
5.5	Building a Chat Application with Channels	127
5.5.1	The Graphical User Interface.....	127
5.5.2	Combining Channel Writing and Reading	128
5.6	A Consumer Channel	130
5.6.1	Implementing a Pager Service.....	131
5.6.2	The Pager Message Entry	133
5.6.3	Tracking the Start and End of a Channel.....	133
5.6.4	The Index Entry.....	134
5.6.5	Creating a Consumer Channel.....	135
5.6.6	Sending Messages to the Channel	136
5.6.7	Reading and Removing Messages from the Channel	138
5.6.8	Demonstrating the Pager	141
5.7	Bounded Channels.....	141
5.7.1	The Status Entry	142
5.7.2	Channel Creation Revisited.....	143
5.7.3	Writing to a Bounded Channel	144
5.7.4	Taking from a Bounded Channel.....	146
5.7.5	Demonstrating the Bounded Channel	148

5.8 Summary.....	149
5.9 Exercises.....	149
6 Application Patterns	153
6.1 The Replicated-Worker Pattern	153
6.1.1 Computing the Mandelbrot Set.....	155
6.1.2 Task and Result Entries	157
6.1.3 The Master	159
6.1.4 The Worker.....	164
6.2 The Command Pattern	166
6.2.1 Implementing a Compute Server	167
6.2.2 The Generic Worker	167
6.2.3 The Generic Master	169
6.2.4 Creating Specialized Tasks and Results	170
6.2.5 Creating a Specialized Master	173
6.2.6 Running the Compute Server	175
6.3 The Marketplace Pattern.....	176
6.3.1 An Automobile Marketplace	176
6.3.2 Interaction in the Marketplace	177
6.3.3 The Bid Entry	180
6.3.4 The Application Framework.....	181
6.3.5 The Buyer	182
6.3.6 The Seller	186
6.3.7 Running the Marketplace	191
6.4 Other Patterns.....	192
6.4.1 Specialist Patterns.....	192
6.4.2 Collaborative Patterns.....	193
6.5 Summary.....	194
6.6 Exercises.....	194
7 Leases	197
7.1 Leases on Entries	197
7.2 The Lease Object	199
7.2.1 Lease Expiration	200
7.2.2 Renewing a Lease	200
7.2.3 Cancelling a Lease	201
7.3 Lease Maps.....	202
7.3.1 Creating a Lease Map	202
7.3.2 Adding and Removing Leases	203
7.3.3 Renewing Leases	204
7.3.4 Cancelling Leases	205

7.4	Automated Lease Renewal	206
7.4.1	The LeaseManager Interface.....	206
7.4.2	Implementing the Constructors	207
7.4.3	Adding Leases	208
7.4.4	Cancelling a Lease.....	209
7.4.5	Renewing Leases	210
7.4.6	Checking Leases	211
7.4.7	Processing Renewal Failures.....	213
7.4.8	Putting It All Together.....	214
7.5	Summary	215
7.6	Exercises	215
8	Distributed Events	217
8.1	Events in the Distributed Environment.....	217
8.2	Hello World Using <code>notify</code>	220
8.3	The Notification API.....	222
8.3.1	The JavaSpace <code>notify</code> Method.....	223
8.3.2	The EventRegistration Class	224
8.3.3	The RemoteEventListener Interface	225
8.3.4	The RemoteEvent Object	225
8.4	Putting the Pieces Together.....	226
8.4.1	The Channel Relay Application	227
8.4.2	The Channel Reader Thread	229
8.4.3	The Relayer Listener	229
8.4.4	The Notify Handler Thread	231
8.5	Summary	234
8.6	Exercises	234
9	Transactions	237
9.1	The Distributed Transaction Model	238
9.2	Creating a Transaction	239
9.3	Web Counter Revisited	240
9.4	The Space's Transactional Properties	243
9.5	Operational Semantics Under Transactions	244
9.5.1	Writing Entries Under a Transaction.....	244
9.5.2	Reading Entries Under a Transaction	245
9.5.3	<code>read</code> versus <code>readIfExists</code>	245
9.5.4	Taking Entries Under a Transaction	245
9.5.5	Notifications Under Transactions	246
9.6	Summary	246
9.7	Exercises	246

10 A Collaborative Application	247
10.1 The Messenger	247
10.2 Implementing a Messenger User	249
10.3 The Account	249
10.3.1 The Account Object	250
10.4 User Sessions	254
10.4.1 The Session Object	255
10.5 Friends List	257
10.5.1 The FriendsList Object	258
10.5.2 The FriendsListMonitor Object	261
10.6 Communication Channel	263
10.6.1 Message Entries and Indices	263
10.6.2 The Channel Object	265
10.6.3 Retrieving Messages from the Channel	267
10.7 The Messenger Applet	270
10.7.1 Logging in via the Login Object	271
10.7.2 The loginCallback Method	273
10.7.3 Adding to the Friends List	274
10.7.4 Chatting with Friends	275
10.7.5 Sending Messages	277
10.7.6 Listening for Messages	278
10.8 Summary	279
10.9 Exercises	279
11 A Parallel Application	281
11.1 The Compute Server	281
11.1.1 The Command Interface Revisited	281
11.1.2 The Task Entry	282
11.1.3 The Generic Worker	283
11.1.4 A Generic Master	285
11.2 The Crypt Application	287
11.2.1 Background	287
11.2.2 The Generic Worker and Crypt Task	288
11.2.3 The Crypt Master	291
11.2.4 Generating Tasks	294
11.2.5 Improving Space Usage with Watermarking	296
11.2.6 Collecting Results	297
11.2.7 A Little Poison	299
11.2.8 Exploring Crypt	301
11.3 Summary	301
11.4 Exercises	302

12 Further Exploration 303

12.1 Online Resources	303
12.2 Related Java and Jini Technologies	304
12.3 Background Reading	304
12.3.1 Distributed Systems	304
12.3.2 Transactions	305
12.4 Historical Resources	305

Appendix A:**The Jini™ Entry Specification 307**

A.1 Entries and Templates	307
A.1.1 Operations	307
A.1.2 <i>Entry</i>	308
A.1.3 Serializing <i>Entry</i> Objects	308
A.1.4 <i>UnusableEntryException</i>	309
A.1.5 Templates and Matching	311
A.1.6 Serialized Form	311
A.1.7 Comments	311

Appendix B:**The Jini™ Entry Utilities Specification 313**

B.1 Entry Utilities	313
B.1.1 <i>AbstractEntry</i>	313
B.1.2 Serialized Form	314
B.1.3 Comments	314

Appendix C:**The JavaSpaces™ Specification 315**

C.1 Introduction	315
C.1.1 The JavaSpaces Application Model and Terms	315
C.1.2 Benefits	318
C.1.3 JavaSpaces Technology and Databases	319
C.1.4 JavaSpaces System Design and Linda Systems	320
C.1.5 Goals and Requirements	322
C.1.6 Dependencies	322
C.1.7 Comments	323
C.2 Operations	323
C.2.1 Entries	323
C.2.2 <i>net.jini.space.JavaSpace</i>	323
C.2.3 <i>write</i>	326

C.2.4 <code>readIfExists</code> and <code>read</code>	326
C.2.5 <code>takeIfExists</code> and <code>take</code>	327
C.2.6 <code>snapshot</code>	327
C.2.7 <code>notify</code>	328
C.2.8 Operation Ordering.....	329
C.2.9 Serialized Form.....	330
C.3 Transactions	330
C.3.1 Operations Under Transactions	330
C.3.2 Transactions and ACID Properties.....	331
C.4 Further Reading	332
C.4.1 Linda Systems.....	332
C.4.2 The Java Platform.....	333
C.4.3 Distributed Computing	333
Index	334

Preface

Over the next decade the computing landscape will change dramatically as devices become ubiquitous, network-connected, and ready to communicate. As the landscape changes, the way in which we design and build software will change as well: The distributed application (one that involves multiple processes and devices) will become the natural way we build systems, while the standalone desktop application will become nearly extinct.

Designing distributed software is remarkably hard, however. The fundamental characteristics of a networked environment (such as heterogeneity, partial failure, and latency) and the difficulty of “gluing together” multiple, independent processes into a robust, scalable application present the programmer with many challenges that don’t arise when designing and building desktop applications.

JavaSpaces™ technology is a simple, expressive, and powerful tool that eases the burden of creating distributed applications. Processes are loosely coupled—communicating and synchronizing their activities using a persistent object store called a *space*, rather than through direct communication. This method of coordinating distributed processes leads to systems that are flexible, scalable, and reliable. While simple, the space-based model is powerful enough to implement advanced distributed applications—from e-commerce systems to groupware to heavy-duty parallel computations. Space-based programming also leverages the Jini™ technology’s leasing, distributed event, and transaction features, making it suitable for building robust, commercial-quality distributed systems.

This book teaches you how to use JavaSpaces technology to design and build distributed applications. It is intended for computer professionals, students, and Java enthusiasts—anyone who wants experience building networked applications. Through experimentation with the code examples, you’ll develop a repertoire of useful techniques and patterns for creating space-based systems. We assume that you already have some programming experience and basic working knowledge of Java programming language fundamentals, but this book doesn’t require any specific knowledge of network programming.

JavaSpaces technology is new, and writing a book before the technology is in widespread use presents a unique challenge. We've approached the project from complementary perspectives. Two of the authors, Eric Freeman and Susanne Hupfer, spent much of the past decade designing and building space-based systems as part of the Linda research group at Yale University and used the JavaSpaces technology during the two years of its development. The third, Ken Arnold, was in charge of the JavaSpaces project at Sun Microsystems, working with a team of engineers to design and build the technology this book is all about.

In this book, we present the foundations of programming with JavaSpaces technology, and a set of common patterns and frameworks for approaching space-based programs. As the technology becomes more widely used, the JavaSpace programming community will discover new ways of using it. We would like future editions of this book to incorporate these new patterns, and we invite you to send comments, suggestions, and ideas to javaspaces@aw1.com and to make use of the book's web site at <http://java.sun.com/docs/books/jini/javaspaces>.

How to Access And Run the Example Code

The code examples in this book have been compiled and run against the following packages from Sun Microsystems:

- ◆ The Java™ 2 SDK, Standard Edition, v1.2

which is available for download at <http://www.java.sun.com/products/>, and

- ◆ The Jini Technology Starter Kit (JSK) release 1.0
- ◆ The JavaSpaces Technology Kit (JSTK) release 1.0

both of which are available for download at <http://developer.java.sun.com/developer/products/jini/product.offerings.html>.

You can obtain the complete source code of the examples in the book at <http://java.sun.com/docs/books/jini/javaspaces>. This site is the official web site for the book and contains links to resources and information relating to the JavaSpaces technology, errata, and supplementary material generated after this book went to press.

Acknowledgments

We are indebted to many people for their contributions to the development of the JavaSpaces technology. Bill Joy pushed the Linda-style distributed computing idea hard enough to make Sun take it seriously. Jim Waldo, Ann Wollrath, Roger Riggs, and Bob Scheiffler worked on the design, with input from Peter Jones. Gary Holness and John McClain joined the JavaSpaces implementation team and made a difference both in the detailed semantics and the implementation design. Bob Resendes joined near the end but was important in getting the product finished, and Jimmy Torres and Frank Barnaby built and ran the release processes that got the it out the door. And, of course, nothing could have been done without Helen Leary, but then, nothing ever is.

We are equally grateful to the many people who helped us turn the idea for this book into a reality. We especially want to thank our editor Mike Hendrickson at Addison-Wesley and Series editor Lisa Friendly at Sun Microsystems, who both recognized the value of the project early on. Mike provided enthusiastic support, ideas, push and encouragement along the way. Lisa held the book to the highest standards.

The team at Addison-Wesley was an enormous help at every stage of the project. In particular, Marina Lang and Julie DeBaggis were supportive, made the publishing process run smoothly, and steered us clear of many potholes. Without the dedicated efforts of Sarah Weaver, Marty Rabinowitz, Diane Freed, Bob Russell, Tracy Russ, Katherine Kwack, Sara Connell, Simone Payment, and others working behind the production and marketing scenes, this book would not have been possible.

A number of reviewers greatly improved the book's content through their careful reading and helpful suggestions. Jen McGinn supplied a thorough technical and editorial review, and was instrumental in refining our usage of Sun Microsystems trademarks. Bob Resendes, Gary Holness, and John McClain provided in-depth technical review of the manuscript based on their intimate knowledge of the technology. Andreas Doerr, Laird Dornin, Howard Lee Harkness, Bruce Hopkins, Pascal Ledru, and Nigel Warren contributed useful technical and editorial commentary. Our appreciation also goes to Elisabeth Freeman for her careful reading of many drafts, and to Peter Sparago for his comments.

The wonderful illustrations throughout the book are due to the creative energy and talent of Jim Dustin. Tony Welch of Sun Microsystems gave helpful advice on the use of trademarks in the illustrations.

We also wish to thank the Jini team at Sun Microsystems, in particular Mark Hodapp and Theresa Lanowitz, who were generous with their time and assistance.

Finally, we'd especially like to thank David Gelernter, who planted the seed for the JavaSpaces technology nearly two decades ago, and who was supportive of this book throughout its development.

Foreword

JavaSpaces™ technology is a new realization of the “tuple spaces” that were first described in 1982, in the context of a programming language called “Linda.” My first description of the idea was aimed mainly at “distributed” versus “parallel” programming—at software ensembles, that is, that were built out of many simultaneously active programs scattered over the physically dispersed machines of a computer network.

Tuple spaces seemed like the right tool for distributed programming because they allowed processes to communicate even if each was wholly ignorant of the others. One process (say “alpha”) could get information to another (“beta”) by releasing a heterogeneous bunch of values (a “tuple”) into tuple space; eventually, beta could read the tuple or grab it, and the communication act would be complete. Alpha didn’t need to know beta’s name, address, or anything else; alpha couldn’t care less whether beta read its tuple, or no process read it, or a million processes did. And beta didn’t need to know anything about alpha. The two processes could be running on different types of machines. Alpha could die before beta was born. The computer that housed alpha could be unplugged, taken apart, and recycled into beer cans and spare sand fifteen years before ground was broken for the office building that would eventually be torn down to make room for the factory in which beta’s computer would be assembled. This is what I meant by the claim that tuple space supported “uncoupled communication,” and allowed processes to communicate through space *or* time using the same simple operations.

In the mid-1980s, our focus shifted to parallel applications—particularly scientific and numerical ones. Nowadays, distributed systems are once again our center of attention and the hottest topic in computing, and JavaSpaces technology seems like a system with exactly the right characteristics for today’s most interesting and important software problems.

When we first described tuple spaces, they were generally regarded as cute (or even elegant or beautiful), but laughably impractical. Nick Carriero and I began work on the first serious implementation in 1984, using a custom multi-processor

at Bell Labs. Carriero's successful implementation (first described in 1985) established a fact that seemed improbable and was regarded as distinctly surprising at the time—that tuple-space communication could be just as efficient as low-level alternatives like message passing. Carriero's implementation, with significant extensions by Robert Bjornson, became the basis of a commercial Linda that is today one of the most widely-used vehicles for parallel programming in the world. (The network-parallel version of the scientific code Gaussian 95, for example, is built using Linda.)

Doctors Freeman and Hupfer were part of the Linda inner circle at Yale, and they know more about tuple spaces than nearly anyone. They are the obvious people to write a book about Java language plus tuple spaces.

Susanne Hupfer's early graduate research focussed on a problem that looks easy but turns out to be very hard, adding multiple tuple spaces to Linda. She went on to write a thesis that centered on "Turingware," a form of Linda ensemble in which processes and people shared a single coordination framework. Eric Freeman's first research project as a Yale graduate student was another avant garde tuple-spaces project in connection with the "adaptive parallelism" system called Piranha; a program that has adaptive parallelism can grow and shrink as it runs. He transferred the Piranha idea to a parallel supercomputer.

Ken Arnold comes from a different world, and many worlds make a good book. The JavaSpaces project at Sun Microsystems originated with Bill Joy; Ken Arnold was in charge of it. He is responsible (along with Jim Waldo, Ann Wollrath and Bob Scheiffler) for the actual JavaSpaces design.

JavaSpaces technology is a project that has seemed natural and important to Linda researchers from the start. We have been network enthusiasts for a long time. A book of mine called *Mirror Worlds* (1991) is said to have partly inspired the development of the Java language, and is thought by some people to have forecast the Web. A web site is in essence a type of "mirror world"; the book claimed that mirror worlds would constitute, collectively, "the new public square."

The book claimed that "The software revolution hasn't begun yet, but it will soon." The revolution began on schedule; the Java language has been a key ingredient in the rapidly-spreading influence and importance of the Web, and of computer networks generally. There are obvious philosophical connections between the Java project's goals and Linda's. Both systems aim for maximum power at minimum cost. Portability is a key to the Java technology's success, and has always been one of Linda's biggest selling points. (One of our '80s papers about tuple spaces was called "Linda: the Portable Parallel.") The Java language plus tuple spaces is a natural and powerful combination.

Where does it all lead? Computers have become terribly significant. Our fundamental problem today is to make them insignificant. I've argued many times that computers would become devices not to look at but to look through; I still

believe that, and still think it's important. Information is stored nowadays on some particular computer. To get the information, you connect to the right computer. Before long, information will escape from computers and take off on its own.

You won't care any more about the identity of your own "personal" computer; you won't care about a Web site's address. Information will be (for all intents and purposes) free floating, and you will deal with it directly. Using a TV you can tune in CBS; you don't care where CBS is; at any rate, it's not inside the TV. Using any telescope you can look at Jupiter—but no telescope is the "Jupiter server," and Jupiter doesn't live inside any telescope anywhere. Jupiter is an object and each telescope is an object, and they are all separate objects. By approximately the same token, you'll be able to tune in information (your own and everyone else's) using any computer that's handy.

How will we achieve this free-floating information? These free-floating data objects? Merely drop your information into some world-wide tuple space. Sun Microsystem's Jini is a step in this direction. Jini allows mutually anonymous processes to exchange information (so that a laptop, for example, can use a printer it has never met or been formally introduced to) by means of tuple-like information drops into a tuple-space-like communication medium. This seems like a powerful and flexible way to achieve coordination.

When you have a world-wide tuple space, you'll be able to tune it in from any computer anywhere—or from any quasi-computer: any cell phone, any TV, any toaster. World-wide tuple spaces will be achieved by fast computers wired together in fast networks, but the details won't matter to the users. The users will switch on their machines and information will flow, the way they turn on their faucets and get water (more or less). To the typical city-water user, the elaborate and sophisticated network that implements the water system is of critical importance and zero interest. The technology is there, and that's what matters. For worldwide tuple spaces, the technology is nearly there. Tuple spaces were invented (in technology terms) a long time ago, in the Middle Bronze Age of software, but projects like JavaSpaces technology are on the critical path to the future.

David Gelernter

Introduction

*A language that doesn't affect the way you think about
programming, is not worth knowing.*
—Alan Perlis, **Epigrams in Programming**

THE Java™ programming language is arguably the most popular programming language in computing history—never before has a language so quickly achieved widespread use among computing professionals, students, and hobbyists worldwide. The Java programming language owes its success in part to its clean syntax, object-oriented nature, and platform independence, but also to its unique position as the first general programming language expressly designed to work over networks, in particular the Internet. As a result, many programmers are now being exposed to network programming for the first time. Indeed, with the computer industry moving toward increasingly network-centric systems, no programmer's toolbox will be complete without the means and know-how to design and construct distributed systems.

The JavaSpaces™ technology is a new tool for building distributed systems. By providing a high-level coordination mechanism for Java, it significantly eases the burden of creating such systems. JavaSpaces technology is first and foremost designed to be *simple*: space-based programming requires learning only a handful of operations. At the same time, it is *expressive*: throughout this book we will see that a large class of distributed problems can be approached using this simple framework. The benefit for you, the developer, is that the combination of these two features can significantly reduce the design effort and code needed to create collaborative and distributed applications.

Before getting into the details of the JavaSpaces technology, let's take a look at why you might want to build your next application as a distributed one, as well as some of the trouble spots you might encounter along the way.

1.1 Benefits of Distributed Computing

The 1980s slogan of Sun Microsystems, Inc., “The Network is the Computer™,” seems truly prophetic in light of the changes in the Internet and intranets over the last several years. By early in the new millennium, a large class of computational devices—from desktop machines to small appliances and portable devices—will be network-enabled. This trend not only impacts the way we use computers, but also changes the way we create applications for them: distributed applications are becoming the natural way to build software. “Distributed computing” is all about designing and building applications as a set of processes that are distributed across a network of machines and work together as an ensemble to solve a common problem. There are many compelling reasons for building applications this way.

Performance: There is a limit to how many cycles you can squeeze out of one CPU. When you’ve optimized your application and still need better performance, there is only one thing left to do: add another computer. Fortunately, many problems can be decomposed into a number of smaller ones. Once decomposed, we can distribute them over one or more computers to be computed in parallel. In principle, the more computers we add, the faster the job gets done. In reality, adding processors rarely results in perfect speedup (often the overhead of communication gets in our way). Nevertheless, for a large class of problems, adding more machines to the computation can significantly reduce its running time. This class of problems is limited to those problems in which the time spent on communicating tasks and results is small compared to the time spent on computing tasks (in other words the computation/communication ratio is high). We will return to this topic in Chapter 6.

Scalability: When we write a standalone application, our computational ability is limited to the power and resources of a single machine. If instead we design the application to work over any number of processors, we not only improve performance, but we also create an application that scales: If the problem is too much work for the team of computers to handle, we simply add another machine to the mix, without having to redesign our application. Our “distributed computing engine” can grow (or shrink) to match the size of the problem.

Resource sharing: Data and resources are distributed, just as people are. Some computational resources are expensive (such as supercomputers or sophisticated telescopes) or difficult to redistribute (such as large or proprietary data sets); it isn’t feasible for each end user to have local access. With a distributed system, however, we can support and coordinate remote access to such data and services. We could, for instance, build a distributed application that continually collects data from a telescope in California, pipes it to a supercomputer in New York for number crunching, adds the processed data to a large astronomical data set in New Mexico, and at the same time graphs the data on our workstation monitor in Connecticut.

Fault tolerance and availability: Nondistributed systems typically have little tolerance for failure; if a standalone application fails, it terminates and remains unavailable until it is restarted. Distributed systems, on the other hand, can tolerate a limited amount of failure, since they are built from multiple, independent processes—if some fail, others can continue. By designing a distributed application carefully, we can reduce “down time” and maximize its availability.

high
2 months
if
faulty
networked

Elegance: For many problems, software solutions are most naturally and easily expressed as distributed systems. Solutions often resemble the dynamics of an organization (many processes working asynchronously and coordinating) more than the following of a recipe (one process following step-by-step instructions). This shouldn't be surprising, since the world at large, along with most of its organizations, is a distributed system. Instructing a single worker to sequentially assemble a car or run a government is the wrong approach; the worker would be overly complex and hard to maintain. These activities are better carried out by specialists that can handle specific parts of the larger job. In general, it is often simpler and more elegant to specify a design as a set of relatively independent services that individual processes can provide—in other words, as a distributed system.

1.2 Challenges of Distributed Computing

Despite their benefits, distributed applications can be notoriously difficult to design, build, and debug. The distributed environment introduces many complexities that aren't concerns when writing standalone applications. Perhaps the most obvious complexity is the variety of machine architectures and software platforms over which a distributed application must commonly execute. In the past, this heterogeneity problem has thwarted the development and proliferation of distributed applications: developing an application entailed porting it to every platform it would run on, as well as managing the distribution of platform-specific code to each machine. More recently, the Java virtual machine has eased this burden by providing automatic loading of class files across a network, along with a common virtual machine that runs on most platforms and allows applications to achieve “Write once, Run anywhere™” status.

The realities of a networked environment present many challenges beyond heterogeneity. By their very nature, distributed applications are built from multiple (potentially faulty) components that communicate over (potentially slow and unreliable) network links. These characteristics force us to address issues such as *latency*, *synchronization*, and *partial failure* that simply don't occur in standalone applications. These issues have a significant impact on distributed application design and development. Let's take a closer look at each one:

Latency: In order to collaborate, processes in a distributed application need to communicate. Unfortunately, over networks, communication can take a long time relative to the speed of processors. This time lag, called latency, is typically several orders of magnitude greater than communication time between local processes on the same machine. As much as we'd like to sweep this disparity under the rug, ignoring it is likely to lead to poor application performance. As a designer, you must account for latency into order to write efficient applications.

Synchronization: To cooperate with each other, processes in a distributed application need not only to communicate, but also to synchronize their actions. For example, a distributed algorithm might require processes to work in lock step—all need to complete one phase of an algorithm before proceeding to the next phase. Processes also need to synchronize (essentially, wait their turn) in accessing and updating shared data. Synchronizing distributed processes is challenging, since the processes are truly asynchronous—running independently at their own pace and communicating, without any centralized controller. Synchronization is an important consideration in distributed application design.

Partial failure: Perhaps the greatest challenge you will face when developing distributed systems is partial failure: the longer an application runs and the more processes it includes, the more likely it is that one or more components will fail or become disconnected from the execution (due to machine crashes or network problems). From the perspective of other participants in a distributed computation, a failed process is simply “missing in action,” and the reasons for failure can’t be determined. Of course, in the case of a standalone application, partial failure is not an issue—if a single component fails, then the entire computation fails, and we either restart the application or reboot the machine. A distributed system, on the other hand, must be able to adapt gracefully in the face of partial failure, and it is your job as the designer to ensure that an application maintains a consistent global state (a tricky business).

These challenges are often difficult to overcome and can consume a significant amount of time in any distributed programming project. These difficulties extend beyond design and initial development; they can plague a project with bugs that are difficult to diagnose. We’ll spend a fair amount of time in this book discussing features and techniques the JavaSpaces technology gives us for approaching these challenges, but first we need to lay a bit of groundwork.

1.3 What Is JavaSpaces Technology?

JavaSpaces technology is a high-level coordination tool for gluing processes together into a distributed application. It is a departure from conventional distributed tools, which rely on passing messages between processes or invoking meth-

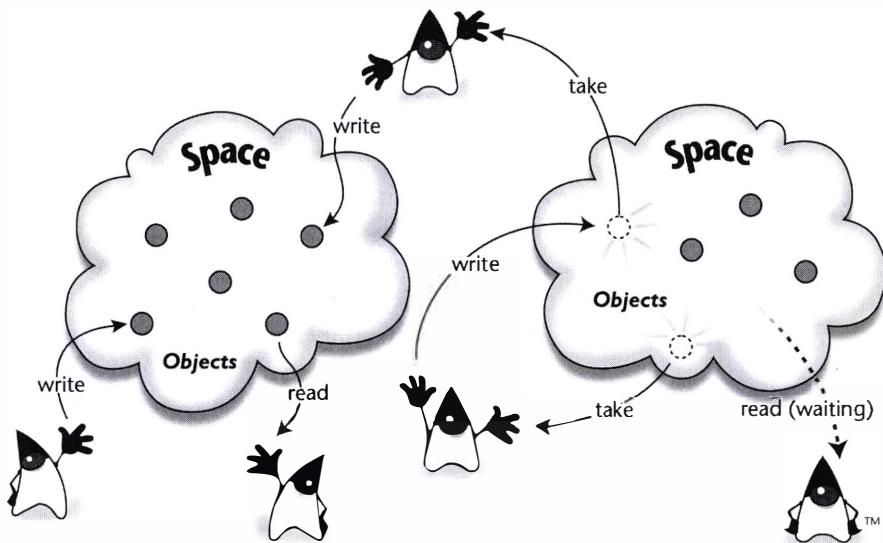


Figure 1.1 Processes use spaces and simple operations to coordinate.

ods on remote objects. JavaSpaces technology provides a fundamentally different programming model that views an application as a collection of processes cooperating via the flow of objects into and out of one or more *spaces*. This space-based model of distributed computing has its roots in the Linda coordination language developed by Dr. David Gelernter at Yale University. We provide several references to this work in Chapter 12.

A *space* is a shared, network-accessible repository for objects. Processes use the repository as a persistent object storage and exchange mechanism; instead of communicating directly, they coordinate by exchanging objects through spaces. As shown in Figure 1.1, processes perform simple operations to *write* new objects into a space, *take* objects from a space, or *read* (make a copy of) objects in a space. When taking or reading objects, processes use a simple value-matching lookup to find the objects that matter to them. If a matching object isn't found immediately, then a process can wait until one arrives. Unlike conventional object stores, processes don't modify objects in the space or invoke their methods directly—while there, objects are just passive data. To modify an object, a process must explicitly remove it, update it, and reinsert it into the space.

To build space-based applications, we design *distributed data structures* and *distributed protocols* that operate over them. A distributed data structure is made up of multiple objects that are stored in one or more spaces. For example, an ordered list of items might be represented by a set of objects, each of which holds

the value and position of a single list item. Representing data as a collection of objects in a shared space allows multiple processes to concurrently access and modify the data structure.

Distributed protocols define the way participants in an application share and modify these data structures in a coordinated way. For example, if our ordered list represents a queue of printing tasks for multiple printers, then our protocol must specify the way printers coordinate with each other to avoid duplicating efforts. Our protocol must also handle errors: otherwise a jammed printer, for example, could cause many users to wait unnecessarily for jobs to complete, even though other printers may be available. While this is a simple example, it is representative of many of the issues that crop up in more advanced distributed protocols.

Distributed protocols written using spaces have the advantage of being *loosely coupled*: because processes interact indirectly through a space (and not directly with other processes), data senders and receivers aren't required to know each other's identities or even to be active at the same time. Conventional network tools require that all messages be sent to a particular process (who), on a particular machine (where), at a particular time (when). Instead, using a JavaSpaces system, we can write an object into a space with the expectation that someone, somewhere, at some time, will take the object and make use of it according to the distributed protocol. Uncoupling senders and receivers leads to protocols that are simple, flexible, and reliable. For instance, in our printing example, we can drop printing requests into the space without specifying a particular printer or worrying about which printers are up and running, since any free printer can pick up a task.

The JavaSpaces technology's shared, persistent object store encourages the use of distributed data structures, and its loosely coupled nature simplifies the development of distributed protocols. These topics form the major theme of this book—before diving in and building our first space-based application, let's get a better idea of the key features of the technology and how spaces can be used for a variety of distributed and collaborative applications.

1.3.1 Key Features

The JavaSpaces programming interface is simple, to the point of being minimal: applications interact with a space through a handful of operations. On the one hand, this is good—it minimizes the number of operations you need to learn before writing real applications. On the other hand, it begs the question: how can we do such powerful things with only a few operations? The answer lies in the space itself, which provides a unique set of key features:

Spaces are shared: Spaces are network-accessible “shared memories” that many remote processes can interact with concurrently. A space itself handles the details of concurrent access, leaving you to focus on the design of your clients and

the protocols between them. The “shared memory” also allows multiple processes to simultaneously build and access distributed data structures, using objects as building blocks. Distributed data structures will be a major theme of Chapter 3.

Spaces are persistent: Spaces provide reliable storage for objects. Once stored in the space, an object will remain there until a process explicitly removes it. Processes can also specify a “lease” time for an object, after which it will be automatically destroyed and removed from the space (we will cover leases in detail in Chapter 7).

Because objects are persistent, they may outlive the processes that created them, remaining in the space even after the processes have terminated. This property is significant and necessary for supporting uncoupled protocols between processes. Persistence allows processes to communicate even if they run at non-overlapping times. For example, we can build a distributed “chat” application that stores messages as persistent objects in the space and allows processes to carry on a conversation even if they are never around at the same time (similar to email or voice mail). Object persistence can also be used to store preference information for an application between invocations—even if the application is run from a different location on the network each time.

Spaces are associative: Objects in a space are located via *associative lookup*, rather than by memory location or by identifier. Associative lookup provides a simple means of finding the objects you’re interested in according to their content, without having to know what the object is called, who has it, who created it, or where it is stored. To look up an object, we create a template (an object with some or all of its fields set to specific values, and the others left as `null` to act as wildcards). An object in the space matches a template if it matches the template’s specified fields exactly. We’ll see that with associative lookup, we can easily express queries for objects such as: “Are there any tasks to compute?” or “Are there any answers to the prime factor I asked for?” We will cover the details of matching in the next chapter.

Spaces are transactionally secure: The JavaSpaces technology provides a transaction model that ensures that an operation on a space is atomic (either the operation is applied, or it isn’t). Transactions are supported for single operations on a single space, as well as multiple operations over one or more spaces (either *all* the operations are applied, or *none* are). As we will see in Chapter 9, transactions are an important way to deal with partial failure.

Spaces allow us to exchange executable content: While in the space, objects are just passive data—we can’t modify them or invoke their methods. However, when we read or take an object from a space, a local copy of the object is created. Like any other local object we can modify its public fields as well as invoke its methods, even if we’ve never seen an object like it before. This capability gives us a powerful mechanism for extending the behavior of our applications through a space.

1.3.2 JavaSpaces Technology in Context

To give you a sense of how distributed applications can be modeled as objects flowing into and out of spaces, let's look at a few simple use scenarios. Consider a space that has been set up to act as an “auction room” through which buyers and sellers interact. Sellers deposit for-sale items with descriptions and asking prices (in the form of objects) into the space. Buyers monitor the space for items that interest them, and whenever they find some, they write bid objects into the space. In turn, sellers monitor the space for bids on their offerings and keep track of the highest bidders; when an item’s sale period expires, the seller marks the object as “sold” and writes it back into the space (or perhaps into the winning buyer’s space) to close the sale.

Now consider a computer animation production house. To produce an animation sequence, computer artists create a model that must then be rendered for every frame of a scene (a compute-intensive job). The rendering is often performed by a network of expensive graphics workstations. Using the JavaSpaces technology, a series of tasks—for instance, one task per frame that needs to be rendered—are written into the space. Each participating graphics workstation searches the space for a rendering task, removes it, executes it, drops the result back into the space and continues looking for more tasks. This approach scales transparently: it works the same way whether there are ten graphics workstations available or a thousand. Furthermore, the approach “load balances” dynamically: each worker picks up exactly as much work as it can handle, and if new tasks get added to the space (say another animator deposits tasks), workers will begin to compute tasks from both animation sequences.

Last, consider a simple multiuser chat system. A space can serve as a “chat area” that holds all the messages making up a discussion. To “talk,” a participant deposits message objects into the space. All chat members wait for new message objects to appear, read them, and display their contents. The list of attendees can also be kept in the space and gets updated whenever someone joins or leaves the conversation. Late arrivals can examine the existing message objects in the space to review previous discussion. In fact, since the space is persistent, a new participant can view the discussion long after everyone else has gone away, and participants can even come back much later to pick up the conversation where they left off.

These examples illustrate some of the possible uses of spaces, from workflow systems, to parallel compute servers, to collaborative systems. While they leave lots of details to the imagination (such as how we achieve ordering on chat messages) we'll fill them in later in the book.

1.4 JavaSpaces Technology Overview

Now we are going to dive into our first example by building the obligatory “Hello World” application. Our aim here is to introduce you to the JavaSpaces programming interface, but we will save the nitty-gritty details for the next chapter. We are going to step through the construction of the application piece by piece, and then, once it is all together, make it a little more interesting.

1.4.1 Entries and Operations

A space stores *entries*. An entry is a collection of typed objects that implements the `Entry` interface. Here is an example “message” entry, which contains one field—the content of the message:

```
public class Message implements Entry {  
    public String content;  
  
    public Message() {  
    }  
}
```

We can instantiate a `Message` entry and set its content to “Hello World” like this:

```
Message msg = new Message();  
msg.content = "Hello World";
```

With an entry in hand, we can interact with a space using a few basic operations: `write`, `read` and `take` (and a couple others that we will get to in the next chapter). The `write` method places one copy of an entry into a space. If we call `write` multiple times with the same entry, then multiple copies of the entry are placed into the space. Let’s obtain a space object and then invoke its `write` method to place one copy of the entry into the space:

```
JavaSpace space = SpaceAccessor.getSpace();  
space.write(msg, null, Lease.FOREVER);
```

Here we call the `getSpace` method of the `SpaceAccessor` class, which returns an instance of an object that implements the `JavaSpace` interface (which we will refer to as a “space object” or “space” throughout this book). We then call `write` on the space object, which places one copy of the entry into the space. We will define a `SpaceAccessor` class and cover the details of the `write` method in the next chapter; for now, it’s enough to know that `getSpace` returns a `JavaSpace` object and that `write` places the entry into the space.

Now that our entry exists in the space, any process with access to the space can read it. To read an entry we use a *template*, which is an entry that may have one or more of its fields set to `null`. An entry matches a template if the entry has the same type as the template (or is a subtype), and if, for every specified (non-`null`) field in the template, their fields match exactly. The `null` fields act as wildcards and match any value. We will get back to the details of matching in the next chapter, but for now let's create a template:

```
Message template = new Message();
```

That was easy. It is important to point out that the `content` field of the template is by default set to `null` (as Java does with all noninitialized object fields upon creation). Now let's use our template to perform a `read` on the space:

```
Message result =
    (Message)space.read(template, null, Long.MAX_VALUE);
```

Because the template's `content` field is a wildcard (`null`), the template will match any `Message` entry in the space (regardless of its contents). We're assuming here that this space is our own private space and that other processes are not writing or reading `Message` entries. So, when we execute our `read` operation on the space, it matches and returns a copy of the entry we wrote there previously and assigns it to `result`. Now that we have our entry back, let's print its content:

```
System.out.println(result.content);
```

Sure enough, we get:

```
Hello World
```

For the sake of completeness, the `take` operation is just like `read`, except that it withdraws the matching entry from the space. In our code example, suppose we issue a `take` instead of a `read`:

```
Message result =
    (Message)space.take(template, null, Long.MAX_VALUE);
```

We would see the same output as before. However, in this case, the entry would have been removed from the space.

So, in just a few steps we've written a basic space-based "Hello World" program. Let's pull all these code fragments together into a complete application:

```
public class HelloWorld {
    public static void main(String[] args) {
        try {
            Message msg = new Message();
            msg.content = "Hello World";
```

```
JavaSpace space = SpaceAccessor.getSpace();
space.write(msg, null, Lease.FOREVER);

Message template = new Message();
Message result =
    (Message)space.read(template, null, Long.MAX_VALUE);
System.out.println(result.content);
} catch (Exception e) {
    e.printStackTrace();
}
}
```

In this code we've kept things simple by wrapping the code in a `try/catch` statement that catches all exceptions. We also left out the implementation of the `SpaceAccessor`. We will return to both of these topics in the next chapter. However, note that you can find the complete source for each example at the book's web site <http://java.sun.com/docs/books/jini/javaspaces>.

Before moving on, let's step back a bit—with a small bit of simple code, we've managed to send a message using spaces. Our `HelloWorld` places a message into a space, in effect broadcasting a “Hello World” message to anyone who will listen (everyone with access to the space who is looking for `Message` entries). Right now, `HelloWorld` is by design the only listener; it reads the entry and prints out its contents. In the next section we will change that.

1.4.2 Going Further

Let's take our example and make it a little more interesting. In doing so, you'll get a glimpse of the key features that make the JavaSpaces technology an ideal tool for building distributed applications.

We'll begin by modifying the `Message` class to hold not only a message but also a count of how many times it has been read. So far, our `HelloWorld` application is the only process reading the message entry, so we'll also create a `HelloWorldClient` to read the entry. We will also enhance our `HelloWorld` application so that it can monitor the entry's popularity by keeping track of how many times it has been read. Let's start with the new `Message` entry:

```
public class Message implements Entry {
    public String content;
    public Integer counter;

    public Message() {
    }
```

```

public Message(String content, int initVal) {
    this.content = content;
    counter = new Integer(initVal);
}

public String toString() {
    return content + " read " + counter + " times.";
}

public void increment() {
    counter = new Integer(counter.intValue() + 1);
}
}

```

We've added an `Integer` field called `counter`, and a new constructor that sets the `content` and `counter` fields to values passed in as parameters. We've also added a `toString` method, which prints the values of the fields, and a method called `increment`, which increments the counter by one.

Note that in all our examples, we've been violating a common practice of object-oriented programming by declaring our entry fields to be public. In fact, fields of an entry *must* be public in order to be useful; if they are instead declared private or protected, then processes that take or read the entry from a space won't be able to access their values. We'll return to this subject in the next chapter and explain it more thoroughly.

Now let's modify the `HelloWorld` class to keep track of the number of times the `Message` entry has been read by other processes:

```

public class HelloWorld {
    public static void main(String[] args) {
        try {
            Message msg = new Message("Hello World", 0);

            JavaSpace space = SpaceAccessor.getSpace();
            space.write(msg, null, Lease.FOREVER);

            Message template = new Message();
            for (;;) {
                Message result = (Message)
                    space.read(template, null, Long.MAX_VALUE);
                System.out.println(result);
                Thread.sleep(1000);
            }
        }
    }
}

```

```
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Following along in our `main` method, we first make use of our `Message` entry's new constructor, which takes a `String` parameter and an initial counter value and assigns them to the `content` field and the `counter` field respectively. Next we obtain a space object and write the `Message` entry into the space. As in the previous version of `HelloWorld`, we then create a template (with `null` fields) to match the entry.

Now things become more interesting: we enter a `for` loop that continually reads the message entry from the space using `template`. Each time we read a `Message` entry, we print out the value of its counter by calling `println`, which implicitly calls the message's `toString` method. The loop then takes a short breather by sleeping for one second before continuing. If we now run this version, it will continually print a counter value of zero because we are waiting for other processes to read the entry and, so far, there are none.

So, let's write a `HelloWorldClient` that will take `Message` entries, increment their counters and place them back in the space:

```
public class HelloWorldClient {
    public static void main(String[] args) {
        try {
            JavaSpace space = SpaceAccessor.getSpace();

            Message template = new Message();
            for (;;) {
                Message result = (Message)
                    space.take(template, null, Long.MAX_VALUE);
                result.increment();
                space.write(result, null, Lease.FOREVER);
                Thread.sleep(1000);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Just as in the `HelloWorld` application, `HelloWorldClient` first creates a template using the default constructor (both fields are set to `null` and act as wildcards). Rather than reading (as in `HelloWorld`) a message from the space, we take it out of the space and assign it to `result`. We then call `result`'s `increment` method, which increments the `counter` by one, and write the result back into the space. Like `HelloWorld`, we then sleep for one second and repeat the entire process.

So let's now run `HelloWorld` and then start up a few `HelloWorldClients`. The output for a typical run might look something like:

```
Hello World read 0 times.  
Hello World read 1 times.  
Hello World read 5 times.  
Hello World read 10 times.  
. . .
```

Let's trace through the whole scenario to understand exactly what has happened. First, we started up `HelloWorld`, which deposits its `Message` entry into the space and enters a loop that reads the entry and prints the value of the counter. The first time through the loop, the `counter` field's value is still zero (no other processes have yet updated the counter). We also start up several `HelloWorldClient` applications, which each begin searching the space using a template of type `Message` (with both its fields set to `null` to act as wildcards). Since the system is asynchronous, the `HelloWorldClients` access the `Message` entry in an unpredictable order. If a client tries to take the entry but it is has already been removed, then the client simply blocks and waits until the entry shows up in the space. Once a client manages to take an entry, it calls the entry's `increment` method to update the counter, and then it returns the modified entry to the space.

Our output indicates that, by the second time `HelloWorld` reads the counter, one client process has accessed the entry and incremented its counter. By the third time, four more clients have managed to access the entry. Finally, by the fourth time, five more clients have accessed it. In general, the more clients we add, the faster `counter` gets incremented (although only so many processes can take and write the entry in the one-second interval).

1.5 Putting It All Together

Even though our “Hello World” example is simple, it demonstrates the key features of space-based programming and ties together many of the topics we've cov-

ered in this chapter. JavaSpaces technology is simple and expressive: with very little code (and only four lines that contain JavaSpace operations) we've implemented a simple distributed application that provides concurrent access to a shared resource (in this case a shared object). Because spaces provide a high-level coordination mechanism, we didn't need to worry about multithreaded server implementation, low-level synchronization issues, or network communication protocols—usual requirements of distributed application design. Instead, our example concretely illustrates what we said earlier in this chapter—that we build space-based applications by designing distributed data structures along with distributed protocols that operate over them. `HelloWorld` uses a very simple distributed data structure: a shared object that acts as a counter. It also uses a simple protocol: clients `take` the `Message` entry to gain exclusive access to it, increment the counter, and then `write` the entry back to the space to share it once again. This protocol deserves a closer look.

Note that the protocol is loosely coupled—`HelloWorld` writes an entry into the space without worrying about the specifics of which clients will access it, how many, from where, or when. Likewise, the `HelloWorldClients` don't care who generated the entry, where, or when; they simply use associative lookup to find it. They don't even care if the entry exists in the space or not, but are content to wait until it shows up. Because the entry is persistent, clients can even show up at much later times (possibly even after `HelloWorld` has terminated) to retrieve it.

In our example, processes use entries to exchange not only data (a counter) but also behavior. Processes that create entries also supply proper methods of dealing with them, removing that burden from the processes that look up the entries. When a `HelloWorldClient` retrieves a `Message` entry, it simply calls the object's `increment` method (without needing to know how it works).

Our distributed protocol also relies on synchronization. Without coordinated access to a shared resource—in this case, the counter—there would be no way to ensure that only one process at a time has access to it, and processes could inadvertently corrupt it by overwriting each other's changes. Here, to alter an entry, a process must remove it, modify it, and then return it to the space. While the process holds the entry locally, no other processes can access or update it. Transactional security of spaces also plays a key part in guaranteeing this exclusive access: If a process succeeds at a `take` operation, the entry is removed and returned atomically, and the process is guaranteed to have the only copy of the entry.

This isn't to say our simple example covers everything. Although we can trust a single space operation to be transactionally secure (either it completes or it doesn't), there is nothing in our current example to prevent the `Message` entry from being irretrievably lost if a client crashes or gets cut off from the network after taking the message entry from the space (as often happens in the presence of

partial failure). In cases like this, to ensure the integrity of our applications, we will need to group multiple space operations into a *transaction* to ensure that either *all* operations complete (in our example, the entry gets removed *and* returned to the space) or *none* occur (the entry still exists in the space). We'll revisit the topic of transactions in greater detail in Chapter 9.

1.6 Advantages of JavaSpaces Technologies

We hope that in this introduction you've gained a sense for why you might want to build your next distributed application using spaces. If your application can be modeled as a flow of objects into and out of spaces (as many can), then the JavaSpaces technology offers a number of compelling advantages over other network-based software tools and libraries:

It is simple. The technology doesn't require learning a complex programming interface; it consists of a handful of simple operations.

It is expressive. Using a small set of operations, we can build a large class of distributed applications without writing a lot of code.

It supports loosely coupled protocols. By uncoupling senders and receivers, spaces support protocols that are simple, flexible, and reliable. Uncoupling facilitates the composition of large applications (we can easily add components without redesigning the entire application), supports global analysis (we can examine local computation and remote coordination separately), and enhances software reuse (we can replace any component with another, as long as they abide by the same protocol).

It eases the burden of writing client/server systems. When writing a server, features such as concurrent access by multiple clients, persistent storage, and transactions are reinvented time and time again. JavaSpaces technology provides these functionalities for free; in most cases, we only need to write client code, and the rest is handled by the space itself.

The beauty of JavaSpaces technology is that it can be grasped easily and used in powerful ways. In comparison to other distributed programming tools, space-based programming will, in many cases, ease design, reduce coding and debugging time, and result in applications that are more robust, easier to maintain, and easier to integrate with other applications.

1.7 Chapter Preview

This book is about building distributed and collaborative applications with the JavaSpaces technology. As with any programming methodology, a number of general principles and patterns have emerged from the use of spaces, and we will spend the bulk of this book covering them. Our aim is to help you explore new ways of thinking about, designing, and building distributed applications with spaces (and in short order so that you can quickly begin to create your own distributed applications). The following is a roadmap to what you'll find as you make your way through this book:

Chapters

Chapter 2—*JavaSpaces Application Basics*—lays the foundation you will need to understand and experiment with the examples in the rest of the book. In a tutorial style, we cover the mechanics of creating a space-based application and introduce the syntax and semantics of the JavaSpaces API and class library.

Chapter 3—*Building Blocks*—presents basic “distributed data structures” that recur in space-based applications, and describes common paradigms for using them. Code segments are given to illustrate the examples, which include shared variables, bags, and indexed structures. This chapter lays the foundation for the next two: Synchronization and Communication.

Chapter 4—*Synchronization*—builds upon Chapter 3 and describes techniques for synchronizing the actions of multiple processes. We start with the simple idea of a space-based semaphore and incrementally present more complex examples of synchronization, from sharing resources fairly, to controlling a group in lockstep, to managing multiple readers and writers.

Chapter 5—*Communication*—also builds upon Chapter 3 and describes common communication patterns that can be created using distributed data structures. We first introduce space-based message passing and then explore the principles behind space-based communication (which provides a number of advantages over conventional communication libraries). We then present a “channel” as a basic distributed data structure that can be used for many common communication patterns.

Chapter 6—*Application Patterns*—introduces several common application patterns that are used in space-based programming, including the replicated-worker pattern, the command pattern, and the marketplace pattern. In each case, we develop a simple example application that makes use of the pattern. We also provide a general discussion of more *ad hoc* patterns.

Chapter 7—*Leases*—begins the book’s coverage of more advanced topics. Spaces use leases as a means of allocating resources for a fixed period of time.

This chapter explores how to manipulate and manage the leases created from writing entries into a space. The techniques covered for managing leases are also applicable to distributed events and transactions, which are covered in the next two chapters.

Chapter 8—*Distributed Events*—introduces the Jini distributed event model and shows how applications can make use of remote events in conjunction with spaces.

Chapter 9—*Transactions*—introduces the idea of a transaction as a tool for counteracting the effects of partial failure in distributed applications. This chapter covers the mechanics as well as the semantics of using transactions.

Chapter 10—*A Collaborative Application*—explores the creation of a distributed interactive messenger service using spaces. This collaborative application makes use of the full JavaSpaces API, and also some of the advanced topics encountered in previous chapters, namely leases, events, and transactions.

Chapter 11—*A Parallel Application*—explores parallel computing with spaces. We first build a simple compute server and then a parallel application that runs on top of it. Both are used to explore issues that arise when developing space-based parallel applications. Like the collaborative application, in this chapter we make full use of the JavaSpaces API and its advanced features.

Chapter 12—*Further Exploration*—provides a set of references (historical and current) that you can use as a basis for further exploration.

Appendices A, B, and C—contain the official *Jini™ Entry Specification*, *Jini™ Entry Utilities Specification*, and *JavaSpaces™ Specification* written by the Jini product team at Sun Microsystems, Inc.

Online Supplement

The online supplement to this book can be accessed at the World Wide Web site <http://java.sun.com/docs/books/jini/javaspaces>. The supplement includes the following:

- ◆ Full source code to examples
- ◆ Links to sites with related information, including links to specifications, white-papers, and demonstration programs
- ◆ Corrections, supplements, and commentary generated after this book went to press

1.8 Exercises

Exercise 1.1 List all the applications you can think of that can't be represented as the flow of objects into and out of a space. Review your list. Are you sure?

Exercise 1.2 Rerun the second version of HelloWorld; this time run several HelloWorldClients along with two HelloWorld applications. What happens? What is the cause of this output?

Exercise 1.3 Change HelloWorld and HelloWorldClient so that multiple instances of the application can be run independently by different users at the same time, within the same space. Was it difficult?

Exercise 1.4 Write a program using the standard Java API networking classes (from the `java.net` package) that has the same functionality as our first version of HelloWorld. Compare the two.

JavaSpaces Application Basics

"Think simple" as my old master used to say—meaning reduce the whole of its parts into the simplest terms, getting back to first principles.
—Frank Lloyd Wright

IN this chapter we are going to cover the basics of the JavaSpaces API—examining its core interfaces and classes for dealing with entries and spaces. We will also look at how it all works and explain some of the mechanics of creating space-based applications. Our aim is to give you the foundation you need to understand and experiment with the example programs in the next several chapters and to start building your own space-based applications.

Our discussion in this chapter will center around the `net.jini.javaspace` and `net.jini.core.entry` packages, which are at the heart of JavaSpaces programming. Both of these packages are part of a larger set of Jini (pronounced genie) packages; Jini is an ambitious distributed computing environment upon which JavaSpaces is built. You can find references to information about Jini in Chapter 12.

We will also spend a fair amount of time in this book covering the `net.jini.core.lease`, `net.jini.core.event`, and `net.jini.core.transactions` packages. The `lease` package allows us to control the lifetime of entries that we place into a space. The `event` package naturally extends Java's event model to the distributed environment. Using this distributed event model in combination with spaces, we can write programs that react to the arrival of entries in a space. The `transaction` package gives us a way to deal with partial failure by providing transactional security for operations across multiple operations and spaces. We will use transactions to write space-based programs that behave correctly even in the face of failing machines and networks.

Together, these packages provide a powerful set of tools for creating advanced distributed applications. However, to begin creating space-based applications we don't have to understand all of them; in fact, we will cover a lot of ground with just the `net.jini.javaspace` and `net.jini.core.entry` packages. You'll need to be aware of the other packages and their role in space-based programming in this chapter, but you won't have to understand their details until later in the book; you'll find these other APIs first covered in Chapter 7 (Leases), Chapter 8 (Distributed Events), and Chapter 9 (Transactions).

2.1 Entries

Entries are the common currency of all space-based applications; by exchanging entries, processes can communicate, synchronize and coordinate their activities.

An entry is just an object that follows a few conventions to make it safe for travel through a space. Let's define an entry:

```
import net.jini.core.entry.*;  
  
public class SpaceShip implements Entry {  
    // a no-arg constructor  
    public SpaceShip() {  
    }  
}
```

Here we've defined an entry called `SpaceShip`. We first import the `net.jini.core.entry` package, which contains the `Entry` interface. For a class to be an entry it needs to implement the `Entry` interface. You can also create an entry by extending a class that already implements the `Entry` interface, which we will see later in this chapter.

Next, you'll notice that we've defined a `public` "no-arg constructor" in `SpaceShip`. A no-arg constructor is just what it sounds like: a constructor that has no arguments. For an entry to make its way into and out of spaces, it needs a `public` no-arg constructor; we'll see why in Section 2.6. For now, just know that you should always add a `public` no-arg constructor to your entries.

2.1.1 The Entry Interface

Recall that an interface in the Java programming language supplies a list of unimplemented methods. If your class implements a particular interface, then it must pro-

vide an implementation for each method in that interface. Let's take a look at the `Entry` interface as it is defined in the `net.jini.core.entry` package:

```
public interface Entry extends java.io.Serializable {  
    // this interface is empty  
}
```

The `Entry` interface is empty—it has no methods that have to be implemented. Empty interfaces are often referred to as “marker” interfaces because they are used to mark a class as suitable for some role. That is exactly what the `Entry` interface is used for—its sole purpose is to mark a class appropriate for use within a space. That is good news for you, because to implement the interface you don't need to implement any methods, but just need to append “`implements Entry`” to your class definitions (and follow a few conventions we will cover shortly).

2.1.2 Instantiating an Entry

As with any other object, you use the `new` operator to instantiate an entry:

```
SpaceShip enterprise = new SpaceShip();
```

Here we've created an instance of `SpaceShip` (assigned to the variable `enterprise`) that can be written into a space. Other than that, `enterprise` is just a normal object; we can invoke its methods and alter the values of its public fields. Right now, the `SpaceShip` entry doesn't have any fields or methods, so let's make it a little more interesting before we launch our ship into space.

2.1.3 Adding Fields and Methods

Let's add a few fields and methods to our `SpaceShip` entry:

```
public class SpaceShip implements Entry {  
    public String name;  
    public Integer score;  
  
    // a no-arg constructor  
    public SpaceShip() {  
    }  
  
    public SpaceShip(String name, int score) {  
        this.name = name;  
        this.score = new Integer(score);  
    }  
}
```

```

        public void decreaseScore() {
            int val = score.intValue() - 1;
            score = new Integer(val);
        }

        public void increaseScore() {
            int val = score.intValue() + 1;
            score = new Integer(val);
        }
    }
}

```

Here we've added two fields, `name` and `score`. The `name` field will be used to identify the name of the ship, while `score` will be used to hold a numeric score, as in a game.

It is important to point out that both fields are declared to be publicly accessible. If you are already familiar with object-oriented programming, this might strike you as a bit strange, since one typically declares most fields to be private in order to provide encapsulation. Here is a short explanation of why this is the case: As you saw in Chapter 1, *associative lookup* is the way space-based programs locate entries in a space. We specify templates to locate matching entries based on the content of their fields. If you declare entry fields to be private, the space has no way to compare against them (since it doesn't have access to a class's private fields). By declaring entry fields to be public, you are allowing other processes to find your entries based on their field values.

There is one other important point to make regarding our field definitions. The fields of an entry must contain references to objects and not primitive types (such as `int`, `boolean`, `float`, `double`). If you'd like to include primitive types in an entry, then you need to use their corresponding wrapper classes (`Integer`, `Boolean`, `Float`, `Double`). The reason for this will become clear when we cover matching in Section 2.4.

Besides the two new fields, we've also added a new constructor and two new methods. The new constructor takes an initial name and score for our spaceship. This constructor is just for our convenience, to make creating spaceships easier. Here is an example of its use:

```
SpaceShip enterprise = new SpaceShip("enterprise", 10);
```

We've just created a spaceship with the name "enterprise" and an initial score of ten points.

Now let's look at the new method, `decreaseScore`, that we've added to our `SpaceShip` class. The method simply decrements the ship's score by one. Since `score` is an `Integer` object and not a primitive integer type, we must first extract

its integer value, decrement it, and wrap it back in an `Integer` object. Here's how we would reduce the score of our newly created `enterprise` ship by one:

```
enterprise.decreaseScore();
System.out.println("score is " + enterprise.score);
```

We've also printed the resulting score, which in this case is 9.

The other new method, `increaseScore`, is implemented in the same way as `decreaseScore`, except that it adds one to the ship's score.

Adding methods to entries points out one of the powerful features of the Java-Spaces technology: It allows you to pass around behavior by passing around entries. If you implement an entry with a set of public methods, any process that eventually reads or takes your entry from a space will be able to invoke those methods, even if the process has never encountered the entry's class (in this case `SpaceShip`) before.

2.2 Building an Application

Before going any further with our spaceship example, we are going to build an application framework so that you can run and experiment with the code. We are going to do this by creating a `SpaceGame` class:

```
public class SpaceGame {
    private JavaSpace space;

    public SpaceGame(JavaSpace space) {
        this.space = space;
    }

    public static void main(String[] args) {
        JavaSpace space = SpaceAccessor.getSpace();
        SpaceGame game = new SpaceGame(space);
    }
}
```

This class contains a constructor that takes one parameter, an object that implements the `JavaSpace` interface, and assigns it to the local variable `space`. It also contains a static `main` method.

When we run our application, the static `main` method is executed, which first makes use of the `SpaceAccessor` class to obtain a reference to an object that implements the `JavaSpace` interface. It then instantiates a `SpaceGame` object, passing the `space` to its constructor. We will make our application more interesting shortly.

This is the way we will write many of the examples in this book: We develop a class that encapsulates particular behavior, and then add a static `main` method to make the class into an application that can be executed.

2.2.1 The SpaceAccessor

Before moving on, let's take a look at the `SpaceAccessor` class, which as we mentioned in Chapter 1, is a utility class of our own creation that allows us to gain access to a space. The reason we have abstracted this code into a utility class is that there are many possible ways to gain access to a space. For instance, a space might register as a Jini lookup service that can be queried to obtain a remote reference to the space or, a space might register with a RMI registry. Both of these methods can be used with the reference implementation of JavaSpaces technology from Sun Microsystems; other vendors may provide other means of accessing spaces. Given this, we have created one utility class that uses either Jini or RMI to look up spaces, but if you prefer or need another means, you can easily plug in your own implementation.

We cover this class for completeness; understanding the inner workings of it is not necessary to write space programs, but some familiarity with it will help you understand many of the details of running space programs.

Let's take a look at our version:

```
public class SpaceAccessor {
    public static JavaSpace getSpace(String name) {
        try {
            if (System.getSecurityManager() == null) {
                System.setSecurityManager(
                    new RMISecurityManager());
            }

            if (System.getProperty("com.sun.jini.use.registry")
                == null)
            {
                Locator locator =
                    new com.sun.jini.outrigger.DiscoveryLocator();
                Finder finder =
                    new com.sun.jini.outrigger.LookupFinder();
                return (JavaSpace)finder.find(locator, name);
            } else {
                RefHolder rh = (RefHolder)Naming.lookup(name);
                return (JavaSpace)rh.proxy();
            }
        }
```

```
        } catch (Exception e) {
            System.err.println(e.getMessage());
        }
        return null;
    }

    public static JavaSpace getSpace() {
        return getSpace("JavaSpaces");
    }
}
```

The SpaceAccessor class provides a static method `getSpace`, which returns an object that implements the `JavaSpace` interface. Two forms of this method are supplied. The first version takes one argument, a `String`, that is the name of a space you'd like to access. The second version, which takes no arguments, calls `getSpace` with a default space name of "JavaSpaces," which is the default space name that is used with the examples that ship with the `JavaSpaces` reference implementation from Sun Microsystems.

Let's step through the first `getSpace` method to see how it works: the first thing we do is set a security manager, to ensure that any code that is downloaded into your Java virtual machine (JVM) will go through a set of security checks. For instance, if you retrieve an entry from a space that is not defined (or has references to objects that are not defined) in your JVM, your application will have to download one or more class files. In this code we've set the security manager to the `RMISecurityManager`, which is supplied by the `java.rmi` package and allows a restricted set of permissions for the downloaded code. You can tailor these permissions with the use of policy files that allow you to grant more permissions to downloaded code.

Note: This section brings up many issues involved in the setup, configuration, and runtime parameters needed to properly run examples. Many of these issues are dependent on the implementation and version of the `JavaSpaces` technology you happen to be using. The reference implementation of `JavaSpaces` technology from Sun Microsystems is assumed in this book, and you can find detailed instructions for setting up, configuring and running examples in this environment on the book's web site at <http://java.sun.com/docs/books/jini/javaspaces>.

Next we get the system property `com.sun.jini.use.registry`. This property determines whether we use Jini or RMI to look up the space. If `com.sun.jini.use.registry` is not defined (that is, if it is equal to `null`), we use the Jini lookup service, otherwise, we use the RMI registry.

If we are using the Jini lookup service, then we make use of two classes from the `com.sun.jini.outrigger` package that encapsulate much of the code

needed to perform simple Jini lookups. The first class, `DiscoveryLocator`, knows how to locate a Jini lookup service. The second class, `LookupFinder`, provides a `find` method that takes a locator and a service name, and returns an interface to that service. In the case of the JavaSpace service, this interface is a local (nonremote) proxy object that implements the `JavaSpace` interface and uses a private protocol to communicate with the remote space.

If we are locating the space using the RMI registry, then we call the `Naming.lookup static` method from the `java.rmi` package to look for the requested name in the registry. If one is found, then a reference to a remote object is returned. In the case of the reference implementation from Sun Microsystems, this remote reference is of type `com.sun.jini.mahout.binder.RefHolder`, not a space proxy object. To obtain a proxy to the space, we need to call the `proxy` method on the `RefHolder` object, which returns a local proxy object that implements the `JavaSpace` interface.

In both cases (Jini or RMI lookup) we then return the proxy object as a result of the `getSpace` call. We will return to the proxy object a little later in this chapter and discuss some behind-the-scenes aspects of its operation, but the most important aspect of this object is that it implements the `JavaSpace` interface and can be used to interact with remote spaces.

Now that we know how to construct a `SpaceShip` entry and how to build an application that can interact with spaces, isn't it about time we write some code to launch our ship into space?

2.3 Writing Entries into a Space

To launch our spaceship, we need to write it into a space. Entries are written to spaces so they can be shared with other processes taking part in a distributed computation. If you don't write your entries into a space, they remain local objects of no use to other processes. We will see over the next several chapters that a great many things can be accomplished by sharing entries in a space, from synchronizing processes, to controlling access to resources, to implementing sophisticated forms of communication. For now we are going to write our spaceship entry into a space to travel around (in other words, for the fun of it).

Assuming we have a space object at hand, which is referenced by the variable `space`, here's a method we can call when we want to finally launch a ship into a space:

```
public void writeShip(SpaceShip ship) {  
    try {  
        space.write(ship, null, Lease.FOREVER);  
    } catch (Exception e) {  
        System.out.println("Error writing ship: " + e);  
    }  
}
```

```
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

The `writeShip` method takes a `SpaceShip` entry as an argument and writes it into the space. Specifically, the method calls the space's `write` method, which is wrapped in a `try/catch` clause. The `try/catch` clause will catch any exceptions that might occur during this operation (we will cover the exceptions that `write` throws a little later in the chapter). Assuming all goes well, the space's `write` method places a copy of the `ship` in the remote space. We can also call `write` multiple times on the same entry, which would result in multiple copies of the entry being placed into the space.

Once a copy of the entry is stored remotely in the space, we can no longer invoke the entry's methods or directly examine the value of its fields without first obtaining a local copy of the entry (with `read` or `take`) from the space. This is an important point, because many distributed object models store "active" objects remotely, such that their methods can be called from remote processes. This is a different model—in effect, an entry in a space is a passive data object that can't be changed or altered unless it is first retrieved from the space. As we will see, this distinction has a powerful effect when developing distributed algorithms.

Here's how we would create our "enterprise" spaceship and write it to a space within the context of our application:

```
public class SpaceGame {
    private JavaSpace space;

    public SpaceGame(JavaSpace space) {
        this.space = space;
    }

    public static void main(String[] args) {
        JavaSpace space = SpaceAccessor.getSpace();
        SpaceGame game = new SpaceGame(space);

        SpaceShip enterprise = new SpaceShip("enterprise", 10);
        game.writeShip(enterprise);
        System.out.println(enterprise.name +
            " written into space");
    }

    // ... writeShip method definition goes here
}
```

This should all look familiar. In `main`, we first locate a space and then instantiate the game object. We then create a spaceship entry `enterprise`—with its name set to “enterprise” and its score set to ten—and pass it to our `writeShip` method to write it into the space. If you compile and run this application, you will see the following:

```
enterprise written into space
```

2.3.1 A Closer Look at `write`

We gave only a brief explanation of the space’s `write` operation above. Let’s take a closer look by reviewing the syntax of the `write` operation as defined in the JavaSpaces specification:

```
Lease write(Entry e, Transaction txn, long lease)
    throws RemoteException, TransactionException;
```

The `write` method takes an entry, which we’re already familiar with, and two additional arguments: a *transaction* and a *lease time*. The topic of transactions is important and deserves a whole chapter (Chapter 9), but for now recall from Chapter 1 that transactions are used to provide transactional security for a group of space operations. This means that the group of operations is atomic: Either all the operations complete, or, if any fails, all are aborted as if they never occurred.

For the next several chapters we will be writing applications without supplying an explicit transaction. Instead, we use a `null` value for the transaction. When we invoke space operations and supply `null` for the `txn` argument we say that the operation is operating under a “`null` transaction.” A `null` transaction is simply a transaction with one operation in it (which either completes or fails).

The last argument to `write` is the requested *lease time* of the entry. A lease time specifies how long we’d like the entry to be stored in the space before the space removes it. Using leases allows a space to perform its own version of garbage collection and clean up entries when they’ve been around longer than their creator intended. In our `writeShip` method, the `write` operation requests a lease time of `Lease.FOREVER`, which specifies that the entry should be stored in the space indefinitely. You can also request a duration in milliseconds; for instance, a lease time of 10,000 specifies that the entry should be stored for ten seconds before being removed. The actual lease granted by a space is returned as a result of the `write` method call. We will cover leases in detail in Chapter 7.

The `write` method throws two kinds of exceptions: `RemoteException` and `TransactionException`. A `RemoteException` is thrown if a communication-related problem occurs between your process and the remote space, or if any exceptions occur in the remote space while receiving and storing your entry.

Remote exceptions contains a field, called `detail`, that may contain the exception on the remote space, which cause the `RemoteException` to be thrown.

A `TransactionException` can occur when `write` is operating under a transaction; this exception is thrown when the transaction used in the call is invalid (it has aborted or otherwise cannot be used). We will talk more specifically about this exception in Chapter 7.

Now that we know the details of the `write` operation, and have even written our spaceship into a space, what other interesting things can we do with the ship? Let's see.

2.4 Reading and Taking Entries

The `JavaSpace` interface provides `read` and `take` methods that allow you to retrieve a copy of an entry in a space (`read`) or to retrieve a copy and at the same time remove an entry (`take`). To read or take an entry we first have to locate it within the space, and this is done via associative lookup. Rather than specifying the address of the entry or an identifier that points to the entry, we create a *template* that can be used to match entries in a space based on content.

As we will see in later chapters, this content-based method of retrieving entries allows us to uncouple distributed components and to support anonymous forms of communication. Both of these qualities lead to more robust, flexible, and scalable code.

2.4.1 Associative Lookup

A template is an entry that specifies enough content to identify the kind of entry we're looking for. To create a template, we instantiate an entry and selectively specify values for some fields and leave others as `null`. An entry and a template match if every specified field in the template “matches” the corresponding field in the entry. All `null` fields in the template act as wildcards and match any value in the corresponding field of an entry.

For instance, to look for any `SpaceShip` entry in a space, we first create a template out of a `SpaceShip` entry as follows:

```
SpaceShip anyShipTemplate = new SpaceShip();
anyShipTemplate.name = null;
anyShipTemplate.score = null;
```

Due to the two `null` fields, the `anyShipTemplate` will match any spaceship in the space, regardless of its name and score.

If we'd like to create a template to find our "enterprise" ship and not just any old spaceship, we need to specify the name "enterprise" in our template and leave the score field as a wildcard:

```
SpaceShip enterpriseTemplate = new SpaceShip();
enterpriseTemplate.name = "enterprise";
enterpriseTemplate.score = null;
```

Now this template will match any ship in the space that has a name of "enterprise," independent of its score.

As another example, we could set up a template that looks for any ship with a score of zero, regardless of its name:

```
SpaceShip zeroScoreTemplate = new SpaceShip();
zeroScoreTemplate.name = null;
zeroScoreTemplate.score = new Integer(0);
```

Now that we know how to create templates, let's do something with them.

2.4.2 The Basics of `read` and `take`

Let's create a new method in our application called `getScore`, which takes a ship's name as an argument and returns that ship's score. To accomplish this, we need to read a copy of a `SpaceShip` entry from the space, and then access its `score` field. The method looks like this:

```
public int getScore(String name) {
    SpaceShip template = new SpaceShip();
    template.name = name;

    try {
        SpaceShip ship = (SpaceShip)
            space.read(template, null, Long.MAX_VALUE);
        return ship.score.intValue();
    } catch (Exception e) {
        e.printStackTrace();
        return -1;
    }
}
```

In `getScore`, we first create a template by instantiating a `SpaceShip` entry and setting its `name` field to the name of the ship we are looking for (the one passed into the method). Unlike our template examples from the last section, note that we don't set the `score` field to `null`. It turns out we don't have to explicitly set object

fields to `null` to act as wildcards, since they are set to `null` by default when an object is created. We can just set the fields that we want to match against (in this case just the `name` field).

Wrapped inside a `try/catch` clause, you will see that we call the `read` method on our space object. The `read` method takes a template, a transaction (we again supply `null`), and a time-out value (specified in milliseconds). If the `read` does not immediately find a match for a template, it will wait up to the given number of milliseconds for a matching entry to be added to the space. By convention, a time-out value of `Long.MAX_VALUE` means that the `read` will wait indefinitely.

When a matching entry is found, `read` returns a copy of the entry, which we assign to the variable `ship`. Since `read` returns an object of type `Entry`, we have to explicitly cast the object to `ShapeShip` in the assignment.

Let's return to the `main` method of our application and add one line of code to make use of our new `getScore` method:

```
public static void main(String[] args) {  
    JavaSpace space = SpaceAccessor.getSpace();  
    SpaceGame game = new SpaceGame(space);  
  
    SpaceShip enterprise = new SpaceShip("enterprise", 10);  
    game.writeShip(enterprise);  
    System.out.println(enterprise.name +  
        " written into space");  
    System.out.println("The " + "enterprise's score is " +  
        game.getScore("enterprise"));  
}
```

Here we've added a call to `getScore` within the second call to `println` to print the current score of the “enterprise” ship. Running our applet now gives us:

```
enterprise written into space  
The enterprise's score is 10
```

2.4.3 The Rules of Matching

Now that we've informally seen how template matching works, let's take a look at the definitive rules of matching.

A template matches an entry if these two rules hold true:

1. The template's type is the same as the entry's, or is a supertype of the entry's.

2. Every field in the template *matches* its corresponding field in the entry, where:
 - If a template's field has a wildcard (`null`) value then it matches the entry's corresponding field.
 - If a template's field is specified (non-`null`) then it matches the entry's corresponding field if the two have same value.

The first rule says that a template and an entry can potentially match only if they are from the same class. Alternately, the entry may be from a class that was subclassed from the template's class. Let's look at some examples to make the typing rule clear. Consider the following classes:

```
// fields and no-arg constructors omitted below

public class Vegetable implements Entry {
}

public class Fruit implements Entry {
}

public class Apple extends Fruit {
}

public class Orange extends Fruit {
```

Here we have a class `Vegetable`, with no relation to the other classes, and a class `Fruit`, which has two subclasses, `Apple` and `Orange`. If we place a `Vegetable` entry into a space, only a template created from the `Vegetable` class can potentially match it. The same is true of a `Fruit` entry; we can only match it with a `Fruit` template (a `Vegetable` template, for instance, will never match it).

However, if we place an `Apple` entry into the space, then either an `Apple` template or a `Fruit` (`Apple`'s superclass) template could be used to match it. An `Orange` entry is similar in that both `Fruit` and `Orange` templates have the potential to match it.

If you think about matching a `Fruit` template against an `Apple` entry, you might wonder how matching is handled for the fields in `Apple` that are not present in the `Fruit` superclass. For instance, the `Apple` subclass might have a boolean field `worm`, which indicates whether or not the apple has a worm in it. In this case, when the matching of an entry (of some class) and a template (of a superclass) is attempted, any additional fields in the entry are treated as if they were wildcards in the template.

Returning to the second rule of matching, given an entry and a template that satisfy the typing conditions of rule 1, the two will match if, for all fields that aren't wildcards in the template, each field has the same value as its corresponding field in the entry. The meaning of "same value" is closely tied to lower level details of how spaces operate, and we'll return to the topic briefly in Section 2.6.

2.4.4 Dealing with `null`-valued Fields in Entries

Entries generally have fields with non-`null` values, but you can certainly write an entry with `null`-valued fields into a space. The problem is that you can't match directly on the `null` values. For many fields, this limitation is not a big issue, since many fields just hold data and are not used in matching.

For fields that will be used in matching, and for which you want to be able to match on `null` values, there is a workaround to the limitation. Suppose you want to create a template that matches explicitly on the existence of a `null` value in a particular field of an entry; the obvious thought would be to create a template that has `null` for that field. The two fields will certainly match, but not because they are both `null`; rather, they will match because a `null` field in a template acts as a wildcard that will match *any* field value in an entry, `null` or otherwise.

There is no way of creating a template that can distinguish between an entry that has a `null` value in a certain field and one that doesn't. This is a slight limitation of the spaces system: the designers had to select some value to represent a wildcard, and `null`—which works for all object types—was the obvious choice. However, since `null` is used as the wildcard value, it ceases to be a value that we're able to match against.

Fortunately, this limitation is simple enough to work around. If you need to design an entry field such that you can distinguish between the case when it is `null` and when it isn't, then all you need to do is add a boolean field as an indicator of whether the field is set to `null`. As an example, let's take a look at the following class:

```
public class NodePtr implements Entry {  
    public Boolean ptrIsNull;  
    public Node ptr;  
}
```

Let's assume you have a cache of `NodePtr` entries in a space and you need one that isn't pointing to any Nodes yet—whose `ptr` field is still set to `null`. You

can construct a template that will match only `NodePtr` entries with `ptr` set to `null` like this:

```
NodePtr template = new NodePtr();
template.ptrIsNull = new Boolean(true);
template.ptr = null; // for completeness; null by default
```

2.4.5 Primitive Fields

As we have seen, only object fields are used in entry matching, and primitive types are not. The reason for this has to do with wildcards—as we saw in the last section, the designers of JavaSpaces technology chose to use `null` as a wildcard value for matching against fields with object references. The choice of a wildcard for primitive types is less obvious. For instance, for an integer we might choose `0`, or `-1` or `Integer.MIN_VALUE` as reasonable wildcards. However, what wildcard value would we use for a primitive such as a `boolean`? Would we use `true` or `false`? Obviously, neither is acceptable.

For this reason, the designers of JavaSpaces technology made the decision that all fields of entries should be object references—in the case where you are inclined to use a primitive field, you must instead use the primitive’s corresponding wrapper class within an entry.

2.4.6 A Closer Look at `read` and `take`

Now that you have a better idea of how matching works, let’s return to the `read` and `take` operations and look at them a little more closely.

The `JavaSpace` interface actually provides two versions of the `read` operation: `read` and `readIfExists`. Here are their definitions from the JavaSpaces specification:

```
Entry read(Entry tmpl, Transaction txn, long timeout)
    throws TransactionException, UnusableEntryException,
           RemoteException, InterruptedException;

Entry readIfExists(Entry tmpl, Transaction txn, long timeout)
    throws TransactionException, UnusableEntryException,
           RemoteException, InterruptedException;
```

The `read` and `readIfExists` methods both take a template, a transaction, and a time-out value. As we’ve already seen, the template is just an entry with some fields specified and others left as `null`. Note that you can also use `null` as a template, which will match *any* entry in the space.

Just as we saw in the `write` method, the second parameter is a transaction. For the next several chapters, we will continue to supply a `null` transaction in our examples. The last parameter is a time-out parameter specified in milliseconds.

Both `read` and `readIfExists` look for and return a copy of a matching entry from the space (if such an entry exists). In the case where there are many entries in the space that match the template, just one entry will be returned. You should assume that the space makes an arbitrary choice among multiple matching entries—in other words, you shouldn’t write code that depends on any ordering of entries. For example, even if you write a particular entry first, there’s no guarantee it will be the first one retrieved. Even if you repeatedly use the same template in a `read` operation, there is no guarantee that you’ll get the same entry back repeatedly.

The two variants of `read` differ in that `read` is a “blocking” operation, while `readIfExists` is not. If no matching entry exists in the space, a `read` will wait for `timeout` milliseconds until a matching entry is found. Two time-out values have special meaning: `Long.MAX_VALUE` means to wait indefinitely (the `read` will block until a matching entry is available), while the constant `JavaSpaces.NO_WAIT` means to return immediately if no entry matches the template. If `read` returns without finding a matching entry, then `null` is returned by the method.

The `readIfExists` method, on the other hand, is a “nonblocking” operation that will return immediately if no matching entry is found. The time-out parameter comes into play only when the operation occurs under a transaction; we’ll see the details in Chapter 9. For now, you need to know that, under the `null` transaction, `readIfExists` is equivalent to calling `read` with a time-out of `NO_WAIT`.

Both methods can throw the following exceptions: `TransactionException`, `UnusableEntryException`, `RemoteException` and `InterruptedException`. The `TransactionException` is thrown for the same reasons it is thrown in the `write` operation (that is, if the supplied transaction is invalid). The `RemoteException` occurs if the network or remote space fails. The `UnusableEntryException` is thrown when an entry is retrieved from the space that cannot be properly serialized by your process (we will talk about this exception and serialization in Section 2.6). Finally, `InterruptedException` is thrown when the thread executing the `read` method is interrupted in some way, typically by an invocation of the `interrupt` method on the thread.

Like `read`, `take` has both blocking and non-blocking forms:

```
Entry take(Entry tmpl, Transaction txn, long timeout)
throws TransactionException, UnusableEntryException,
RemoteException, InterruptedException;
```

```
Entry takeIfExists(Entry tmpl, Transaction txn, long timeout)
    throws TransactionException, UnusableEntryException,
        RemoteException, InterruptedException;
```

Each `take` method performs just like the corresponding `read` method, with the difference that `take` operations actually *remove* the matching entry from the space. If `take` returns a non-null value and no exceptions occur, then we're guaranteed that the entry has been removed from the space, and no other space operation—by this or any other process—will return the same entry.

Both forms of `take` throw the same exceptions as `read` and `readIfExists`. In the case of the `take` operation, if a `TransactionException`, `UnusableEntryException` or `InterruptedException` occurs, then no entry is taken from the space. However, with a `RemoteException`, an entry might be removed from a space and yet never returned to the proxy that performed the `take`. This results in the loss of an entry. In code where this is unacceptable, the `take` can be performed under a transaction to guarantee that the entry is either taken and returned to the proxy, or not taken from the space at all. We will see how this can be accomplished in Chapter 9.

2.5 Going Further with the Example

At this point we've covered the basic syntax and semantics of the JavaSpaces API. Other than the more advanced topics of leases, remote events, and transactions, which we will cover in later parts of the book, that's all there is—it is a small and simple API. What we haven't yet covered are the principles, patterns, and common practices of space-based programming, and that topic will consume much of the remainder of the book.

As a first step on the journey, we are going to take our spaceship example a bit further. Our goal in this section is to get across the nature of entries as they flow into and out of spaces—while in a space, entries are passive objects that can be read or taken from the space (but we can't invoke their methods). Once an entry has been read or taken from a space, a process can do whatever it likes with the object. For instance, we've already seen one example of reading an entry (in our `getScore` method) where the purpose was to obtain the entry's state. In the example just ahead, we will take an object from the space, alter its state, and then return it to the space, where the changes can be seen by all processes in the distributed application. We will also cover a few other topics such as subclassing entries.

To start, we are going to have some fun with our spaceship example by creating a new form of spaceship called `BattleSpaceShip`—based on the standard `SpaceShip`—that holds “ammunition.” In the end, we'll have a very simple dis-

tributed application that behaves a bit like a game: we'll have one standard spaceship and one battle spaceship in a space, and the battle spaceship will have the ability to "shoot at" the other. When the battle spaceship shoots, its ammunition decreases, its score increases, and the target ship's score decreases. The important point of this game is that to implement this behavior (and make appropriate state changes to the spaceship entries) we have to remove the objects from the space and simulate the firing within a process before returning them to the space. Our intention is to get across the flavor of working with entries, not to suggest strategies for developing real games.

Let's get on with the example.

2.5.1 Subclassing an Entry

We can create a specialized version of our SpaceShip class by subclassing it. Here is a new entry class called BattleSpaceShip that we've created in this way:

```
public class BattleSpaceShip extends SpaceShip {  
    public Integer ammunition;  
  
    public BattleSpaceShip() {  
    }  
  
    public BattleSpaceShip(String name, int score, int ammunition)  
    {  
        super(name, score);  
        this.ammunition = new Integer(ammunition);  
    }  
  
    public void decreaseAmmo() {  
        ammunition = new Integer(ammunition.intValue() - 1);  
    }  
}
```

Our subclass contains a new field, `ammunition`, that holds a count of how many rounds of ammunition the ship has (battleships carry ammunition, while regular spaceships don't). Next we supply the mandatory no-arg constructor and then a new convenience constructor, which takes a ship name, an initial score and an initial count of ammunition. In the constructor, we first call the parent (`SpaceShip`) class's constructor to set the initial values of `name` and `score` for us, and then we set the `ammunition` field to its initial value.

Our `BattleSpaceShip` class also has a new method called `decreaseAmmo`, which takes no arguments and decreases the ship's ammunition count by one.

We'll call the `decreaseAmmo` method to simulate the use of ammunition when the ship is firing its weapon.

Now that we have a subclass of `SpaceShip`, let's add a few new methods to our application that make use of it.

2.5.2 Adding a Few More Methods

Here we'll add three new methods to our application: `getAmmo` to determine the ammunition level of a ship, `retrieveShip` to remove a ship from the space (the opposite of `writeShip`), and `shoot` to simulate firing ammunition at another ship.

The `getAmmo` method takes the name of a ship as an argument and returns the ship's current ammunition count. You'll see that the code follows the same pattern as the `getScore` method we defined previously, except that it uses a `BattleSpaceShip` template, instead of a standard `SpaceShip` template, to find a ship in the space:

```
public int getAmmo(String name) {
    BattleSpaceShip template = new BattleSpaceShip();
    template.name = name;

    try {
        BattleSpaceShip ship = (BattleSpaceShip)
            space.read(template, null, Long.MAX_VALUE);
        return ship.ammunition.intValue();
    } catch (Exception e) {
        e.printStackTrace();
        return -1;
    }
}
```

If we pass `getAmmo` the name of a regular ship (say, “enterprise”), instead of a `BattleSpaceShip`, the method won't be able to find a battle spaceship by that name, and the `read` operation will block indefinitely.

Next, we are going to add a method `retrieveShip` that makes use of the `take` method. This method does the opposite of `writeShip`—rather than writing a ship into a space, it takes a ship name and removes a matching ship from the space:

```
public SpaceShip retrieveShip(String name) {
    SpaceShip template = new SpaceShip();
    template.name = name;
```

```
try {
    return (SpaceShip)
        space.take(template, null, Long.MAX_VALUE);
} catch (Exception e) {
    e.printStackTrace();
    return null;
}
}
```

In `retrieveShip`, we first create a template to match the requested ship. Next we call the space's `take` method, supplying our template, a `null` transaction, and a time-out value of `Long.MAX_VALUE`, which means to wait indefinitely. When a ship is returned from the `take` operation, we return it as the result of the method. Note that by using a `SpaceShip` template, `retrieveShip` is able to retrieve both standard spaceships and battle spaceships from the space (according to our rule 1 of matching).

Our third and final new method is called `shoot`, which takes source and target ship names, and simulates the source ship firing at and hitting the target. Here is the code:

```
public void shoot(String source, String target) {
    // error checking omitted

    BattleSpaceShip sourceShip =
        (BattleSpaceShip)retrieveShip(source);

    if (sourceShip.ammunition.intValue() <= 0) {
        System.out.println(source + " can't fire, it has no ammo");
        writeShip(sourceShip);
        return;
    }
    SpaceShip targetShip = retrieveShip(target);
    System.out.println(source + " firing at " + target);
    sourceShip.decreaseAmmo();
    targetShip.decreaseScore();
    sourceShip.increaseScore();

    writeShip(sourceShip);
    writeShip(targetShip);
}
```

Let's step through this code. We first try to retrieve the source ship, using the `retrieveShip` method. Recall that this method was defined to look for a ship using a `SpaceShip` template, which means it can find matching `SpaceShip` entries, or `BattleSpaceShip` entries (or any other entry subclassed from `SpaceShip`). Once the `retrieveShip` method returns a ship, as type `SpaceShip`, we cast the returned entry into a `BattleSpaceShip`.

There is a bit of danger in this casting, because we could potentially retrieve a ship from the space that's a matching `SpaceShip` (or of some other subclass of `SpaceShip` that isn't `BattleSpaceShip`), and then casting would cause a runtime exception. In this example, we will be controlling the space, so this shouldn't happen. To eliminate the danger in a more open environment, we could do some checking on the returned entry, or we could write a new version of `retrieveShip` that takes and returns exactly the type of entry we're seeking.

Anyway, let's continue with the code. For simplicity, some error checking code has been omitted (refer to the source code). Once `retrieveShip` returns our source ship, we first make sure it has some ammunition so it can shoot, and if it doesn't we return it to the space. If the source ship does have ammunition, then we retrieve the target ship. Once we have both source and target ship entries, we call `decreaseAmmo` on the source ship to simulate the firing of ammunition. We make an assumption that every call to the `shoot` method results in a strike on the target ship: to simulate the "hit," we decrease the score of the target ship and increase the score of the shooting ship. Finally, we write both ships back into the space.

2.5.3 Trying the Game

With these new methods, let's return to our `main` method and add some more functionality:

```
public static void main(String[] args) {
    JavaSpace space = SpaceAccessor.getSpace();
    SpaceGame game = new SpaceGame(space);

    SpaceShip enterprise = new SpaceShip("enterprise", 10);
    game.writeShip(enterprise);
    System.out.println(enterprise.name +
        " written into space");
    System.out.println("The " + "enterprise's score is " +
        game.getScore("enterprise"));

    BattleSpaceShip klingon =
        new BattleSpaceShip("klingon", 10, 10);
```

```
        game.writeShip(klingon);
        System.out.println(klingon.name + " written into space");
        System.out.println("The " + "klingon's score is " +
            game.getScore("klingon"));
        System.out.println("The " + "klingon's ammo count is " +
            game.getAmmo("klingon"));

        game.shoot("klingon", "enterprise");
        game.shoot("klingon", "enterprise");
        game.shoot("klingon", "enterprise");

        System.out.println("The " + "enterprise's score is " +
            game.getScore("enterprise"));
        System.out.println("The " + "klingon's score is " +
            game.getScore("klingon"));
        System.out.println("The " + "klingon's ammo count is " +
            game.getAmmo("klingon"));
    }
```

The first several lines of `main` are the same as before. Beyond that, we create a new `BattleSpaceShip` called “`klingon`” and write it into the space. We then call `getScore` and `getAmmo` within `println` to report the current score and ammunition count of the “`klingon`” ship. We then simulate the `klingon` battle spaceship shooting and hitting the “`enterprise`” ship three times (since the `enterprise` isn’t defined as a battle spaceship, it can’t shoot back). Last, we print out the ending scores of both ships and the ammunition count of the “`klingon`” ship.

When you run this example, you will see:

```
enterprise written into space
The enterprise's score is 10
klingon written into space
The klingon's score is 10
The klingon's ammo count is 10
klingon firing at enterprise
klingon firing at enterprise
klingon firing at enterprise
The enterprise's score is 7
The klingon's score is 13
The klingon's ammo count is 7
```

2.5.4 Post-Game Analysis

We probably would not ever implement a game like this for real, but it has been a fun way to explore the basics of working with entries and space operations. Along the way, we've learned quite a few important points about space-based programming. We've seen how to define an entry by implementing the `Entry` interface, or by subclassing an existing entry. We've seen that to get the state of an entry, we simply read the entry and leave it in the space. We've also seen that, to change the state of an entry, we first need to pull it out of the space, update it, and then put it back—we can't change an entry while it's in the space. We've simulated one ship shooting another, but it is all done outside of the space. We've also seen how entries are a means of shipping behavior through a space: a process that reads or takes a ship entry from the space can invoke the methods that come along with it.

We are now going to take a look at a low-level detail of a space: how entries are sent, stored and retrieved from a remote space. There are several reasons for doing so. First, there are several subtleties involved in using entries that are likely to cause unexpected behavior if you aren't aware of them. Second, many of the requirements and conventions for using entries and spaces will become clearer (and easier to remember) when you understand these details. In addition, as with any technology, the more you know about spaces, the more powerful things you can do with them.

2.6 Serialization and Its Effects

Each `JavaSpace` object is a local object that acts as a proxy to the remote space implementation—it is not a reference to a remote object. As a result, all space operations go through the intermediate proxy and then on to the remote space (see Figure 2.1).

To transfer an entry to or from the remote space, the proxy makes use of *serialization*. Serialization provides a way to turn an object into a stream of bytes that can be stored in a file or transmitted over a network and later reconstituted into a copy of the original object.

When an entry is written into a space, the proxy first serializes its fields and then transmits it to the space. While in the space, an entry is stored in its serialized form. When you read or take an entry, it is transmitted back to the proxy in serialized form and then the fields are deserialized into an entry before it is returned by the `read` or `take` methods.

Recall from earlier in the chapter that the `read` and `take` operations may throw an `UnusableEntryException` when an entry is retrieved from the space but cannot be fully deserialized by the proxy. The `UnusableEntryException`

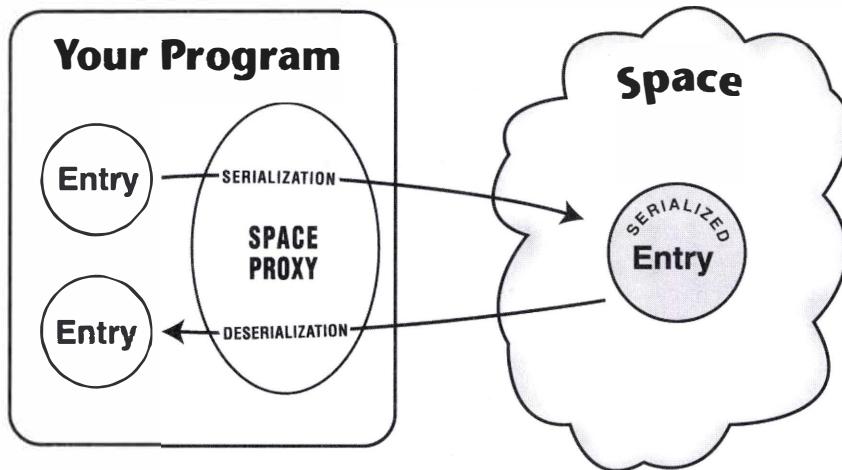


Figure 2.1 Entries are serialized by a local proxy object and transmitted and stored in the remote space in serialized form.

contains information about how the deserialization failed and any fields that could be deserialized. You can find out more about this exception in the official *Java-Spaces™ Specification* from Sun Microsystems, which is provided in Appendix C.

2.6.1 Entry Serialization

The way that `write` serializes entries is a bit different from normal object serialization. When `write` serializes an entry object, the proxy first writes the class information for the entry into the stream and then serializes each public field and writes its serialization into the stream. In this way each public field is serialized independently of the others. Private fields are not written into the serialized stream.

This manner of serializing has some unexpected consequences: if two fields of the entry happen to refer to the same object, the serialized form of the entry that is written into the space will have two separate copies of that object. When you read or take the entry from a space, the returned entry will be structured differently from the one you put into the space: the two fields, which before both referenced the same object, now point to different objects. Consider the following entry:

```
public class TestEntry implements Entry {
    public Integer obj1;
    public Integer obj2;
```

```
public TestEntry() {
}
}
```

`TestEntry` is a simple entry that just contains two `Integer` fields. Now, let's look at a code snippet that makes use of the `TestEntry`:

```
Integer obj = new Integer(0);

TestEntry entry = new TestEntry();
entry.obj1 = obj;
entry.obj2 = obj;

System.out.println("obj1 and obj2 are the same object? " +
    (entry.obj1 == entry.obj2));

System.out.println("obj1 and obj2 are equivalent? " +
    (entry.obj1.equals(entry.obj2))));
```

Here we create an `Integer` object and assign the same object to both fields of a `TestEntry` object. We then print out a comparison of the two fields in the entry and get:

```
obj1 and obj2 are the same object? true
obj1 and obj2 are equivalent? true
```

Next we write the entry into a space and then take it back from the space:

```
space.write(entry, null, Lease.FOREVER);

TestEntry template = new TestEntry();
TestEntry transferredEntry =
    (TestEntry)space.take(template, null, Long.MAX_VALUE);

System.out.println("obj1 and obj2 are the same object? " +
    (transferredEntry.obj1 == transferredEntry.obj2));

System.out.println("obj1 and obj2 are equivalent? " +
    (transferredEntry.obj1.equals(transferredEntry.obj2))));
```

Again we print out the comparison of the two fields, and this time we see:

```
obj1 and obj2 are the same object? false
obj1 and obj2 are equivalent? true
```

This potentially confusing result is due to the separate serialization of fields that we just mentioned. The entry stored in the space was serialized with separate copies of the integer obj and as a result the two fields are different objects but have equivalent values.

You should expect to see the same behavior anywhere you have two fields that reference the same object, whether directly or indirectly via an intermediate object. For instance, if an entry field references a tree data structure and another field points into that structure to some node in the tree, then the structure that the two fields have in common will be replicated in any entries obtained from a space.

As we have already discussed, private fields are not used in entry matching, and they are also not serialized along with the public fields. As a result, private fields will lose their values once they are written into a space. When an entry is deserialized by the proxy, the private fields will contain their default values, unless their no-arg constructor assigns values to them (we will return to the no-arg constructor shortly).

2.6.2 Matching and Equality

Serialization also affects matching semantics. As we mentioned previously, corresponding fields in a template and entry match if the two have the *same value*. What precisely do we mean by “same value”?

An entry is stored in a space in its serialized form. When a template arrives at the space for matching, it is also in serialized form. Matching is done by comparing the serialized versions field-by-field. Two fields match if the bytes of their serialized forms match (however, some bytes of the serialized field include annotations that describe where the class came from, and these bytes are ignored in the comparison), and has nothing to do with type-specific value matching such as `Object.equals`.

2.6.3 The No-arg Constructor

When we first stated the requirement that all entries have a no-arg constructor, we promised we'd explain why. The reason has to do with the way entries are returned as the result of a `read` or `take` operation. To return an entry, the space proxy does the following:

1. Retrieves a serialized copy of the entry
2. Reads the entry's type from the serialized stream
3. Instantiates an object of that type, using its no-arg constructor
4. Reads and deserializes each public field from the stream
5. Assigns the deserialized value to each field of the object

So, to recreate an entry from its serialized form, the proxy first needs to create an instance of that object; in order to do that, the proxy needs a constructor, so it uses the no-arg constructor. Once the entry is instantiated, the proxy can then assign a value to each field from the serialized stream of field values.

2.6.4 The Entry Interface Revisited

When we first presented the `Entry` interface, we skipped over the fact that it extends the `Serializable` interface. The `Serializable` interface is another marker interface that contains no methods and serves to mark a class as appropriate for serialization. Here is the `Entry` interface again:

```
public interface Entry extends java.io.Serializable {  
    // this interface is empty  
}
```

The fact that the `Entry` interface extends the `Serializable` interface causes a bit of confusion. The proxy assumes that each field of an `Entry` is serializable because each field has to be serialized and transmitted to the remote space, but it is not technically necessary that `Entry` itself be (because the entry itself is not serialized by the proxy, only its fields). The designers of the JavaSpaces technology defined `Entry` to extend `Serializable` out of common sense. Each field of the entry is serializable, so the entry itself might as well be serializable too. It's important to note that if an entire entry object is serialized—for example, passed as a parameter in an RMI call—it will be serialized in the normal fashion, not field-by-field. It is only the `JavaSpace` `write` method that uses field-by-field serialization of `Entry` objects.

2.6.5 Improving Entry Serialization Using `snapshot`

In cases where you repeatedly use the same entry over and over without modification, you will incur unnecessary overhead, because the entry is reserialized every time it is used. For instance, it is common to use the same template multiple times in `take` or `read` operations, say within a loop.

In these cases the JavaSpaces API provides a way to avoid this overhead, with the `snapshot` method:

```
Entry snapshot(Entry e) throws RemoteException;
```

The `snapshot` method takes an entry and returns a “snapshotted” version, which can be used in space operations to avoid unnecessary serializations. Obtaining a snapshot may require communication with a space, so `snapshot` throws `RemoteException` like all other space operations. Once you have a snapshot of an entry, any changes to the original entry will not affect the snapshot.

A snapshot is only guaranteed to work with the space that created it. You should not count on a snapshot working across multiple spaces. Let's look at a short example and then we will explore the use of `snapshot` further in Chapter 10 and Chapter 11.

In the following code the `MyEntry` template is used multiple times in the call to `read`. Each time `read` is called, it passes the template to the proxy, which serializes it and sends it to the space to be compared against existing entries.

```
MyEntry template = new MyEntry();
for (int i = 0; i < 100; i++) {
    MyEntry result = (MyEntry)
        space.read(template, null, Long.MAX_VALUE);
}
```

Because the template never changes, clearly the serialization on each iteration isn't optimal. Here is a more efficient version that uses `snapshot`:

```
MyEntry template = new MyEntry();
Entry snapTemplate = space.snapshot(template);
for (int i = 0; i < 100; i++) {
    MyEntry result = (MyEntry)
        space.read(snapTemplate, null, Long.MAX_VALUE);
}
```

Here we've created a snapshotted version of `template` that is assigned to the variable `snapTemplate`. When `snapTemplate` is used in the `read`, the entry is not reserialized by the proxy. So, this version serializes the template just once, not 100 times.

2.7 Summary

We have covered a lot of ground in this chapter, taking a close look at the two primary objects used in space-based programming: the space and the entry. We covered the basic operations in tutorial style, giving some simple examples along with explanations of how to use the JavaSpace interface.

We've also spent a fair amount of time in this chapter discussing the various requirements and conventions you need to follow when creating entries. Implementing the `Entry` interface is a requirement, and so is supplying a no-arg constructor in your entries. If you don't follow these requirements, you'll encounter compile-time and run-time errors. If you don't follow other conventions—namely, making sure the fields of your entries are public object references—you're likely to face run-time errors and non-obvious run-time behavior.

Below is a summary of the conventions you should follow when you define an entry.

- ◆ Declare your entries as `public` classes.
- ◆ Implement the `Entry` interface, extend a class that does, or implement an interface that already implements the `Entry` interface.
- ◆ Provide a `public` no-arg constructor.
- ◆ Define `public` fields as object references (not primitive types).
- ◆ Avoid non-`public` fields (except `static`, `transient`, or `final` fields).
- ◆ Ensure that all fields are serializable.

The first two chapters should have provided you with a good foundation upon which you can begin learning how to write distributed applications with Java-Spaces technology. We also encourage you to peruse the official *JavaSpaces™ Specification*, which appears in Appendix C. Over the next few chapters, we will be showing you basic techniques for building and using space-based data structures to provide synchronization and sophisticated communication in your applications.

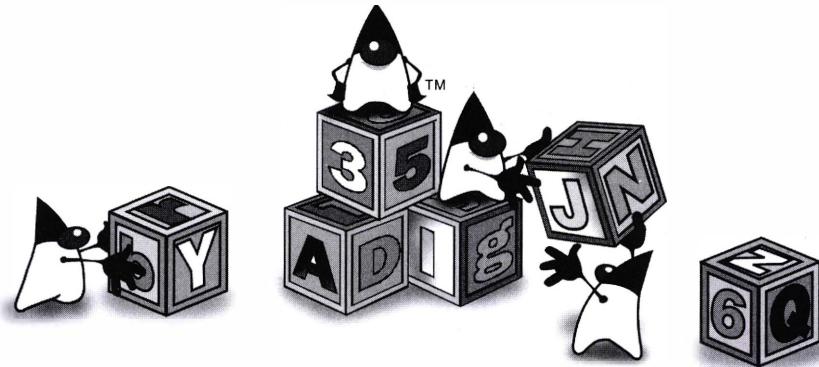
Building Blocks

Show me your code and conceal your data structures, and I shall continue to be mystified.

Show me your data structures, and I won't usually need your code; it'll be obvious.

—Eric Raymond (paraphrasing Fred Brooks), **The Cathedral and the Bazaar**

NOW that we've taken a quick tour of the JavaSpaces API, we have the tools in hand to begin exploring space-based programming in more depth. As with any new programming model or paradigm, learning to write programs using JavaSpaces technology involves both changing the way you think about programming and mastering a set of new techniques and patterns for your programming repertoire.



Our aim over the next several chapters is to show you a new way of programming based on passing around (writing, reading, and taking) entries through spaces. The techniques we will show you are straightforward and surprisingly simple, yet they form the basis of many of the most sophisticated space-based applications. This is the beauty of JavaSpaces technology. Using a simple API, we can create powerful programs yet avoid both the complexity of lower-level

approaches and many of the pitfalls that crop up when developing distributed applications.

In this chapter we are going to concentrate on a common set of building blocks that are used in practically every space-based program. You can think of these building blocks as data structures—in fact, they are *distributed data structures*. With these simple data structures under our belt, we'll use them in the next two chapters to achieve two different goals: synchronization and communication (the glue that ties processes in a distributed program together). Then, armed with techniques for sophisticated communication and synchronization, in Chapter 6 we will explore the most common application patterns used in space-based programming.

3.1 Introduction to Distributed Data Structures

Space-based applications are typically designed around *distributed data structures*. Conceptually a distributed data structure is a data structure that can be accessed and manipulated by multiple processes at the same time. In most distributed computing models, distributed data structures are hard to achieve. Message passing and remote method invocation systems provide a good example of the difficulty. As Figure 3.1 illustrates, these systems tend to barricade data structures behind one central manager process, and processes that want to perform work on

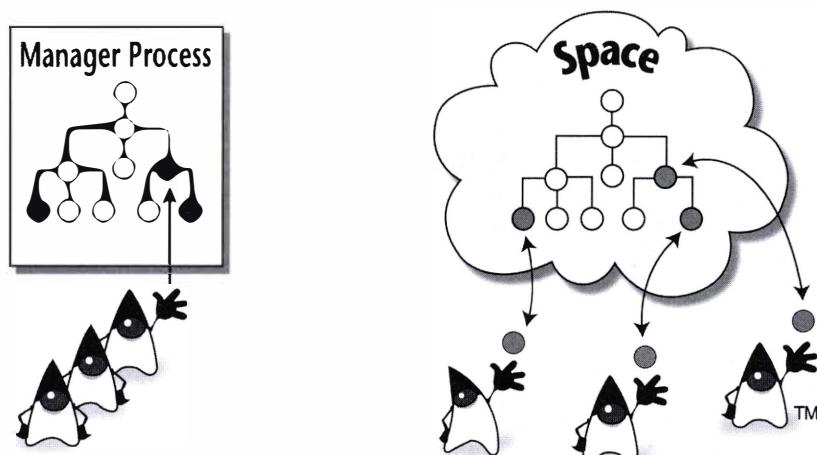


Figure 3.1 Most computing environments hide data behind a centralized manager process, whereas JavaSpaces technology easily supports distributed data structures.

the data structure must “wait in line” to ask the manager process to access or alter a piece of data on their behalf. Attempts to parallelize or distribute a computation across more than one machine face a bottleneck: Since the data are so tightly controlled by the one manager process, multiple processes can’t truly access the data structure concurrently.

Distributed data structures provide an entirely different approach where we uncouple the data from any particular process. Instead of hiding data structures behind a manager process, we represent data structures as *collections of objects that can be independently accessed and altered by remote processes in a concurrent manner*. With distributed data structures, processes no longer wait in line—they can work at the same time on independent pieces of the data structure as long as they don’t step on each other’s toes. This last point is important—it is the “stepping on toes” that has always been difficult to avoid when building distributed systems with conventional network tools. However, as we will see over the next several chapters, this is the area where JavaSpaces technology really shines.

There are two fundamental pieces needed for any distributed data structure: a *representation* of the data structure, along with a *protocol* that processes follow when accessing and altering the data structure. The protocol typically ensures that operations on the data structure occur in a safe and fair manner in the presence of other processes accessing the structure. Let’s see how we can represent distributed data structures with entries, and how we can implement distributed protocols using space operations.

3.1.1 Building Distributed Data Structures with Entries

Let’s say we want to store an array of values in a space so that multiple processes can access and alter it concurrently. This array might hold the scores of the members of a distributed game, the values of a matrix that is being multiplied, or the pixels of an image that is being ray traced.

The simplest and most obvious approach is to create an entry that holds an array (let’s say scores from an online game), like this:

```
public class Scores implements Entry {  
    public Integer[] values;  
  
    public Scores() {  
    }  
}
```

To read or alter values in the array, we need to read or remove the entire array from the space. If many processes are trying to access the array at the same time, then the entry itself becomes a point of contention—just as in a manager process model, we’ve created a bottleneck.

Since it isn't possible for multiple processes to access and alter the `values` array within the entry concurrently, our space-based `Scores` entry isn't a truly distributed array. Let's try again and create a distributed version this time, by breaking the array up into a collection of entries, each of which represents one array element. We can represent an element of an array like this:

```
public class Score implements Entry {
    public Integer index;
    public Integer value;

    public Score() {
    }

    public Score(int index, int value) {
        this.index = new Integer(index);
        this.value = new Integer(value);
    }
}
```

Each `Score` entry holds an `index` (the element's position in the array) and a `value` (the value of that array element). To initialize an array of length ten we can do the following (see the complete source code for error checking):

```
for (int i = 0; i < 10 ; i++) {
    Score score = new Score(i, 0);
    space.write(score, null, Lease.FOREVER);
}
```

Iterating through this loop, we add ten `Score` entries to the space, each of which is initialized with an index and the value zero. To access a score, say the fourth one, we use a `read` operation like this:

```
// create a template to retrieve the element
Score template = new Score();
template.index = new Integer(3);

// read it from the space
Score score = (Score)
    space.read(template, null, Long.MAX_VALUE);
```

To update the value of a score, we use `take` (with the template we created above) and then a `write` as follows:

```
// remove the element so we can alter its value
Score score = null;
score = (Score)
    space.take(template, null, Long.MAX_VALUE);

// alter the value
score.value = new Integer(1000);

// return the element to the space
space.write(score, null, Lease.FOREVER);
```

There you have the basics of creating a simple distributed data structure. We've take a common data structure (an array) and created a distributed version by using multiple entries to represent it in a space. Once we've "distributed" the data structure into multiple entries it becomes concurrently available for remote processes to read and alter. If a process wants to update element five of the array, it can do so without holding up another process reading element three. This kind of concurrent access to the structure wasn't supported by our first naive attempt at a space-based array, nor by manager process computing models. Granted, this particular data structure leaves a lot to be desired—we have no way of knowing the length of the distributed array, for instance—but this simple example is a good start to get you thinking about how to turn ordinary data structures into their distributed representations.

Along with designing a distributed data structure, we've developed a simple protocol that processes must follow when dealing with the distributed array. To read an array value, we just read its current value from the space. To alter an array value, we first remove the entry with `take`, update the `value` field, and then return the entry to the space with `write`. This is about as simple as space-based protocols get, but you could easily extend the protocol to store the size of the array, to copy a value from one array element to another, to delete an element, or to perhaps sort the array.

Now that we've seen a simple example that provides the two key pieces needed for any distributed data structure—a space-based representation and a protocol—let's get on with our exploration of the most basic and common distributed data structures. Some of these distributed data structures (like the array example) have analogs in the sequential programming world and will be familiar to you (although their representation in the space will be new), while others have no analog in the sequential world and will be brand-new.

3.2 Shared Variables

One of the most common data structures in space-based distributed programming is the shared variable. A space-based shared variable is the simplest of distributed data structures, since it is made up of a single entry. Like any variable, a shared variable provides storage for some value or object; however, a space-based shared variable allows multiple processes to easily access and modify its value in an atomic manner. We will return to the topic of atomic modification shortly, but let's first build a space-based shared variable.

Here is a definition of a simple shared variable:

```
public class SharedVar implements Entry {
    public String name;
    public Integer value;

    public SharedVar() {
    }

    public SharedVar(String name) {
        this.name = name;
    }

    public SharedVar(String name, int value) {
        this.name = name;
        this.value = new Integer(value);
    }
}
```

Here we've defined a `SharedVar`, which is really just an entry with a value (in this case an `Integer`). We've also provided a `name` field, which gives us a way to uniquely identify a particular shared variable without having to subclass `SharedVar` for each shared variable we need to create. To create a shared variable, we simply pass a unique name and initial value to the `SharedVar` constructor, and write the result to a space:

```
SharedVar myvar = new SharedVar("duke's counter", 0);
space.write(myvar, null, Lease.FOREVER);
```

The `SharedVar` class can be used to instantiate multiple shared variables and write them to the same space, as long as we make sure each shared variable is created with a unique name.

3.2.1 Atomic Modification

Shared variables typically support two operations, “read” and “update.” A “read” returns the current value of the variable, while “update” stores a new value. A crucial requirement of a shared variable is that its modification must be *atomic*—in other words, must behave as an indivisible unit.

Without an atomic means of accessing and altering shared variables, having multiple processes share access to data can lead to race conditions and corrupt values. This scenario demonstrates the problem: Suppose two bank tellers are making deposits to the same bank account at the same time. Bank teller #1 looks up the bank balance, let’s say \$200, and is about to deposit \$100. At the same time, bank teller #2 also looks up the balance (also \$200 since teller #1 hasn’t completed the deposit yet) and is about to deposit \$50. Next, bank teller #1 performs the deposit of \$100 ($\$200 + \$100 = \300) and records a balance of \$300 for the account. Just a moment later, bank teller #2 deposits \$50 ($\$200 + \$50 = \250) and records a balance of \$250 for the account. Unfortunately, the bank balance has been corrupted: the account has \$250 when it should have \$350.

The problem occurs because updating the bank balance is not an atomic operation, but involves the three separate operations of looking up, modifying, and setting the account’s value. Concurrent bank tellers can carry out those operations in an interleaved way, interfering with each other and corrupting values. To solve the problem, updating a shared value needs to be atomic—performed as one indivisible step.

With JavaSpaces technology we get this atomicity “for free” as a key feature of the space. To see what we mean by this, let’s take a look at how we atomically update a space-based shared variable:

```
SharedVar template = new SharedVar("duke's counter");
SharedVar result = (SharedVar)
    space.take(template, null, Long.MAX_VALUE);
result.value = new Integer(result.value.intValue() + 5);
space.write(result, null, Lease.FOREVER);
```

Here, we’ve added five to the current value of the shared variable named “duke’s counter.” We didn’t need to do anything special to cause the update to happen atomically; the space-based operations themselves naturally provide atomic access to entries. Any update to an entry (here, a shared variable) requires a process to “physically” remove the entry, alter its value locally, and place a copy back into a space. This means that only one process can hold the entry locally and update it at any given time, but many processes can read the entry while it is in the space.

The race condition we encountered in our bank teller example can't occur here, because no two tellers can "own" the bank balance entry at the same time. Furthermore, while the entry is being updated, the entry is not available to other processes trying to read or take it. Any process that wants to read or alter the shared variable after we've removed it must wait patiently until it is returned.

3.2.2 Additional Operations

Beyond the typical read and update operations on shared variables, other operations are sometimes defined on them, such as "increment" and "decrement." Implementing these operations is straightforward. Let's see how `increment`, for instance, can be implemented.

First we'll expand our `SharedVar` class to include a new `increment` method:

```
public class SharedVar implements Entry {
    public String name;
    public Integer value;

    public SharedVar() {
    }

    public SharedVar(String name) {
        this.name = name;
    }

    public SharedVar(String name, int value) {
        this.name = name;
        this.value = new Integer(value);
    }

    public Integer increment() {
        value = new Integer(value.intValue() + 1);
        return value;
    }
}
```

The `increment` method gives us an easy way to increase the value of the shared variable by one.

Incrementing a shared variable is really just a special case of updating it: We first remove the shared variable from the space, increment its value by calling its `increment` method, and then return it to the space:

```
SharedVar template = new SharedVar("duke's counter");
SharedVar result = null;
result = (SharedVar)
    space.take(template, null, Long.MAX_VALUE);
result.increment();
space.write(result, null, Lease.FOREVER);
```

Again, the update of the shared variable happens atomically here, thanks to the nature of space operations. The entry has to be removed from the space before we can call its `increment` method, so only one process can increment the shared variable at any given time.

3.2.3 Creating a Web Counter

Let's put our knowledge of shared variables to use. We are going to implement a web page counter that tracks the number of visits to a web page. Web page counters typically work like this: Each time a web page is loaded in a browser, a counter for that page is incremented and displayed within the page with a message such as: "You are the 100th visitor to our site." On the web server, this is often accomplished by a CGI script that does the following:

1. Locks a file containing the counter
2. Opens the file
3. Reads the counter from the file
4. Increments the counter
5. Re-saves the counter to the file
6. Closes the file
7. Unlocks the file
8. Outputs the counter value, which is then loaded in the browser's page

Now let's try to do the same with a space-based approach. Our web counter is going to be implemented with a small applet that takes a counter from a space, increments it, returns it to the space, and displays its value. To implement the counter, we use a shared variable. The `name` field of the shared variable will hold the URL of a web page, and the `value` field will hold the number of web page visits.

For our counter to work, each web page needs to have a counter, initialized to zero, that is placed in the space. We've created an "installer" application to do this, which is included in the complete source code; for now we will just give the code fragment that creates an initial counter for an URL that's passed in as a parameter to the application:

```
String url = args[0];
SharedVar webCounter = new SharedVar(url, 0);
try {
    space.write(webCounter, null, Lease.FOREVER);
} catch (TransactionException e) {
    e.printStackTrace();
} catch (RemoteException e) {
    e.printStackTrace();
}
```

This code fragment instantiates a shared variable called `webCounter`, setting its name field to the URL of the web page we wish to track and its initial value to zero, and then writes it into the space.

Now let's look at the code for the actual counter applet that we can embed into any web page that we want to track:

```
public class WebCounterClient extends Applet {
    private SharedVar counter;

    public void init() {
        JavaSpace space = SpaceAccessor.getSpace();

        String url = getDocumentBase().toString();
        System.out.println(url);

        SharedVar template = new SharedVar(url);
        try {
            counter = (SharedVar)
                space.take(template, null, Long.MAX_VALUE);
            counter.increment();
            space.write(counter, null, Lease.FOREVER);
        } catch (RemoteException e) {
            e.printStackTrace();
        } catch (TransactionException e) {
            e.printStackTrace();
        }
    }
}
```

```

        } catch (UnusableEntryException e) {
            e.printStackTrace();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        repaint();
        setLayout(null);
        setSize(430, 270);
    }

    public void paint(java.awt.Graphics gc) {
        if (counter != null) {
            Integer count = counter.value;
            gc.drawString("Page visited " + count +
                " times before.", 20, 20);
        }
    }
}

```

This is a fairly simple applet. All the action takes place in the `init` method. Recall that an applet's `init` method is invoked when an applet is initially loaded, so our code will execute the first time a web page is displayed and won't be re-invoked if the user browses back to the page. This is good, because we only want to count the page hit once, even if the user visits it multiple times in a browsing session.

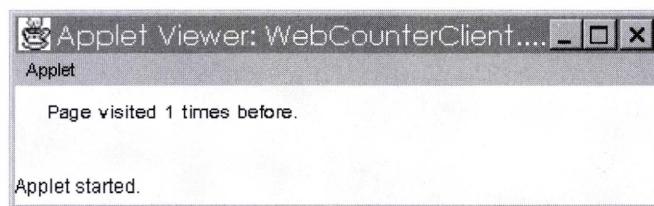
Let's step through the code. First we obtain a handle to a space object by calling the `getSpace` method of `SpaceAccessor`. Next we obtain the URL of the page in which our applet is embedded (the page that contains the `<applet>` tag for this applet) by calling the `getDocumentBase` method. If you are testing your code by retrieving a page from a web server, the path would be something like `http://www.domain.com/path/mydocument.html`. In our example, since we are most likely using `appletviewer`, `getDocumentBase` will return the pathname of the HTML file that contains our applet. The important point here is that the name of the URL returned from `getDocumentBase` should be the same as the name that was passed to our installer application and used to initialize the counter in the space.

Now that we have the URL of the web page, we can create a template to retrieve its corresponding counter from the space. We do this by passing the URL to `SharedVar`'s convenience constructor that assigns it to the entry's `name` field.

Using this template, we take the matching counter from the space—waiting until one exists—and assign it to the applet variable `counter`. We then increment `counter` and write it back into the space. The display of the counter is handled by

the `paint` method, which displays the counter value each time it is called (as long as the value is not `null`, which it is before the counter is retrieved from the space).

Let's run our applet and see how it works:



Here we've run the appletviewer once, and the applet reports that the page has been visited one time. Each time you run the applet, you will see the page count increase by one.

3.2.4 Stepping Back

Before moving on, let's take one more look at how this applet works in an environment where it is embedded in a web page that multiple browser clients access concurrently.

Our file-based CGI scenario enforces synchronization of the counter by explicitly locking and unlocking a file. A key feature of the space is that it enforces synchronized, exclusive access to an entry, because only a single process can hold it locally and modify it at one time (we will revisit synchronization in the next chapter in great detail). When a client issues a `take`, if the web counter entry exists in the space, it is removed, incremented, and returned. If the web counter has already been “checked out” by another client accessing the same web page, the client will have to wait patiently until the entry is written back into the space.

Note that the space also provides persistence for us. In our CGI script example, a file is used to store the counter; in our space-based web counter, the space manages the persistence for us.

Unfortunately, our applet isn't yet ready to be sold to millions yet. To make it industrial-strength, we need to address a few issues that arise in the presence of partial failure. Consider this: What would happen if our `WebCounterClient` removed the counter entry from the space and then failed to return it because the applet crashed or the network went down? The answer is that the counter entry would be lost forever. We'd lose track of page hits, and our applet wouldn't work correctly, because it would have to wait for a entry that didn't exist.

Don't worry, there are several things we can do to improve our code—we will return to the issues of partial failure in Chapter 9. But for now, we've seen that

implementing a shared variable is quite easy. A shared variable is a fundamental distributed data structure that you are likely to use over and over again in your space-based programs.

3.3 Unordered Structures

Unordered data structures don't occur often in nondistributed programming, but they are widely used in space-based programming. An unordered distributed data structure consists of a collection of objects. Unlike an ordered collection, such as an array where we can address a particular element and read its value or change it, an unordered structure allows us to typically perform two operations: "put" and "get." The "put" operation allows you to add a new object to a collection, while "get" returns an arbitrary object from the collection. Because these data structures have no internal order, there is no criteria for returning any one object over any other, so an arbitrary one is returned.

Typically most unordered data structures are called *bags* because they resemble a bag of objects. A bag of apples is a good example. If you have an apple and want to store it in a bag, you just throw it in. You don't need to place the apple in a particular spot, since there is no need to order the apples. When you are hungry and want an apple, you just reach in and grab an apple; apples are pretty much the same, so any apple you first lay your hands on is good enough.

In general bags have many uses, and their use in space-based programming reflects the loosely coupled nature of space-based programs. We talked a bit about loose coupling in Chapter 1, and we'll return to the subject over the next several chapters. The connection here is that bags provide a distributed data structure that can be used by processes to request and provide services without having to know the specifics of who they are requesting a service from or providing a service to. In this way the components of the distributed application are loosely coupled, because there is no direct link from one component to another.

Bags are exceedingly simple to create in space-based programming: to create a bag, you only need to define an entry class and deposit as many instances of that entry into a space as you like. While the entry may have fields that can be used to order the items (for instance, an apple may be marked with the date it was picked from the tree) those fields are usually not important to the processes that make use of the bag (in this case, the dates aren't important to eating, unless some of the apples are really old). When a process needs an item from a bag, it grabs one, since any one will do.

Let's take a look at bags and see how we might use them in space-based programming.

3.3.1 Bags

Let's start with a simple example: many online services provide a means of picking an anonymous game partner from a pool of other users that have registered interest in playing a game (say chess). To implement such a service, we first need to create an entry that represents a user's interest. In the case of someone interested in chess games, the entry might look like:

```
public class ChessPartner implements Entry {
    public String username;
}
```

For every person interested in playing chess, we instantiate a `ChessPartner` entry, filling in that person's username, and write the entry to a space.

To find a chess partner, we simply create a template and fish one out:

```
ChessPartner template = new ChessPartner();
ChessPartner partner = (ChessPartner)
    space.take(template, null, Long.MAX_VALUE);
```

Here we first create a template (with its `username` field set to `null` by default) that will match any `ChessPartner` entry in the space. We then take one such entry out of the space, waiting if necessary until one enters the space. Once we have an entry, we obviously need to contact the partner and begin the game, but we will leave those details to the online service.

Bags need not consist of a collection of all entries of a specific type—we can treat a subset of entries as a bag as well. For instance, we could define our chess partner entry like this:

```
public class ChessPartner implements Entry {
    public String username;
    public String skillLevel;
}
```

Here, we've added a `skillLevel` field that can be used to describe the desired skill level of the opponent. Now you can look for an opponent that might give you a more challenging game:

```
ChessPartner template = new ChessPartner();
template.skillLevel = "expert";
ChessPartner partner = (ChessPartner)
    space.take(template, null, Long.MAX_VALUE);
```

Here we are still choosing an arbitrary partner from a pool of potential candidates, but we are dividing the bag up into smaller bags that are still unordered.

3.3.2 Task Bags and Result Bags

A common method of providing space-based services is through task bags and result bags. Conceptually a service works as follows: a process looking for a service places a task entry into a task bag, and a service looking for “orders to fill” removes the task from the bag and provides a service of some kind. When the service order is complete, the service places a result entry into a result bag, from which the original process can retrieve it.

While this scenario sounds like it is tailored to electronic commerce, task and result bags have wider application and are often used in distributed and parallel computations. In the remainder of this section we are going to take a quick peek at using task and result bags for parallel computation, which we will discuss in more depth in Chapter 6.

Replicated worker computing (sometimes called master-worker) is built on the observation that many computational problems can be broken into smaller pieces that can be computed by one or more processes in parallel. A great many computations fall into this class of problems, including ray tracing, weather model simulations, and parallel searches (to name a few).

If we look at the “skeleton” of these computations, many tend to look something like:

```
for (int i = 0; i < len; i++) {
    //... compute intensive code
}
```

That is, the computations are fairly simple, consisting of a loop over a common, usually compute-intensive, region of code. Often the size of the loop itself (dictated here by the variable `len`) can be quite long. In a ray tracing application, a computation might have to iterate over millions of pixels in an image, each of which takes a fair number of seconds to compute. These kinds of computations lend themselves well to being decomposed into smaller pieces that can be computed in parallel.

Typically in replicated worker computing, a *master* process takes the work performed in the `for` loop and divides it up into a number of tasks that it deposits into a task bag. One or more processes, known as *workers*, grab these tasks, compute them, and place the results back into a result bag. The master process collects the results as they are computed and combines them into something meaningful, such as a ray-traced image.

Here is what the skeleton for a typical master process might look like:

```
public class Master {
    for (int i = 0; i < totalTasks; i++) {
        Task task = new Task(...);
        space.write(task, ...);
    }
}
```

```

        for (int i = 0; i < totalTasks; i++) {
            Result template = new Result(...);
            Result result = (Result)space.take(template, ...);
            //... combine results
        }
    }
}

```

Here we first deposit `totalTasks` requests into a bag of tasks, and then wait for `totalTasks` results from another bag. We don't care about the order in which the tasks are selected to be computed, as long as someone computes them and returns their results. Likewise, we don't care about the order in which results are retrieved from the results bag; we are happy to retrieve any results that have been computed. As we collect the results, we combine them into some meaningful form.

Here is what the skeleton for a typical worker process might look like:

```

public class Worker {
    for (;;) {
        Task template = new Task(...);
        Task task = (Task)space.take(template, ...);
        Result result = compute(task);
        space.write(result, ...);
    }
}

```

Here the worker continually looks for new tasks in the space. Whenever it finds and takes one, it computes the task (which returns some result) and then writes the result back into the space for the master to pick up.

One of the things that makes this simple pattern so powerful is that it is loosely coupled. The master adds tasks to a space, but doesn't worry about who will ultimately receive the tasks and compute them. Similarly, the workers themselves just pick up tasks from the bag, without worrying about who is asking for the work to be done. Because this pattern is loosely coupled, we have the beginnings of a scalable architecture: as we add more workers, the computation is generally completed more quickly. The pattern also naturally load balances, because the workers compute tasks at their own pace. We will return to this topic and explore it more fully in Chapter 6.

3.4 Ordered Structures

One of the most commonly asked questions from those seeing the JavaSpace interface for the first time is “Why can't I get a list of all the entries in a space?” It

is a good question, as it is often necessary to iterate through a set of entries. The answer is, quite simply, that the interface doesn't need to provide a way of listing entries, because enumeration is easily accomplished using simple programming techniques.

We already got a taste of these techniques earlier in the chapter when we took a brief look at how a distributed array might be implemented. The basic technique is this: We create lists (or more generally, ordered distributed data structures) by including an index or “position” field in entries. Used in its simplest form, this technique provides a way of maintaining lists and accessing their elements. However, as we will see in Chapter 5, these same distributed data structures can be used as the basis for powerful communication patterns.

In this section we are going to explore a simple example of creating and maintaining a distributed array of values. This example should provide a good foundation on which to build any ordered distributed data structure in the rest of the book.

3.4.1 Distributed Arrays Revisited

To create a distributed array, we define two classes of entries—entries that hold meta-information about the array (its size, next open position, and so on) and entries that hold the data elements of the array. We already saw one example of an entry that holds data elements in our simple array example. The following entry builds upon that code:

```
public class Element implements Entry {  
    public String name;  
    public Integer index;  
    public Object data;  
  
    public Element() {}  
  
    public Element(String name, int index, Object data) {  
        this.name = name;  
        this.index = new Integer(index);  
        this.data = data;  
    }  
}
```

Here we've added a new field called `name`, which is used to identify an element as belonging to a particular array. For instance, we might give the array a name of “duke's array,” which would be stamped on all its elements and its meta-

information in order that it not be confused with other arrays in the space. This is a common technique: we mark all the entries belonging to a distributed data structure with the same unique identifier. For this example (and the examples in this book), we will use `String` names to identify elements, because they are readable and easy to work with. However, more sophisticated schemes can be used to uniquely identify entries, such as the use of objects that contain several pieces of information (timestamps, machine addresses, digital signatures) that together are unique.

To define an entry that will be used for describing meta-information about the array, we are going to start with an abstract entry called `Index`:

```
abstract public class Index implements Entry {
    public String name;
    public Integer position;

    public int increment() {
        int pos = position.intValue();
        position = new Integer(position.intValue() + 1);
        return pos;
    }

    public int decrement() {
        int pos = position.intValue();
        position = new Integer(position.intValue() - 1);
        return pos;
    }
}
```

`Index` contains a `name` field, which associates it with a particular array, and also a `position` field, which will be used to point to a specific position within the array. The entry also contains two methods: `increment` and `decrement`. The `increment` method adds one to the `position` field and returns the `position` field's value before it was incremented. Likewise, `decrement` behaves similarly, except that it decrements the `position` field by one.

Now let's extend `Index` to create two entries that mark the beginning and ending positions of the array:

```
public class Start extends Index {
    public Start() {
    }
}
```

```
public class End extends Index {
    public End() {
    }
}
```

The Start entry will be used to point to the beginning of the array, and the End entry will point to the end.

With these entries in hand, let's develop the protocols needed to create an array, read its elements, append elements to it, and determine its size. Here's the skeleton of a distributed array class:

```
public class DistribArray {
    private JavaSpace space;
    private String name;

    public DistribArray(JavaSpace space, String name) {
        this.space = space;
        this.name = name;
    }

    //... methods go here
}
```

Here the constructor takes a space and an array name and assigns them to private fields.

Now let's see how the methods are implemented; each of them will deal with the space and array specified in the class's fields. Here's how we create an array:

```
public void create()
    throws RemoteException, TransactionException
{
    Start start = new Start();
    start.name = name;
    start.position = new Integer(0);

    End end = new End();
    end.name = name;
    end.position = new Integer(0);

    space.write(start, null, Lease.FOREVER);
    space.write(end, null, Lease.FOREVER);
}
```

To create a distributed array we need to create Start and End entries in the space, each of which hold the name of the array and position values of zero. This, in effect, creates an empty distributed array.

Now that we have a method for creating an array in the space, let's write an append method, which takes an object to append to the array:

```
public int append(Object obj)
    throws RemoteException, TransactionException,
           UnusableEntryException, InterruptedException
{
    End template = new End();
    template.name = name;

    End end = (End) space.take(template, null, 0);
    int position = end.increment();
    space.write(end, null, Lease.FOREVER);

    Element element = new Element(name, position, obj);
    space.write(element, null, Lease.FOREVER);

    return position;
}
```

In this method, we first need to retrieve the End entry for the array from the space so that we can allocate a new position in the array. We retrieve the entry in the usual way, creating a template, filling in its name field, and leaving the position field as a wildcard.

After retrieving the End entry, we call the `increment` method (defined in the `Index` superclass). The method adds one to the end position's value (which results in a value of one the first time it is called) and returns the value of the end position before it was incremented (which will be zero the first time it is called). Now that we've incremented the end position, we write the End entry back to the space. In effect, we've allocated a space in the array for the object.

Now all that is left to do is to add our object, `obj`, to the array. To do this we create a new `Element` entry with the space's name, the pre-allocated position, and the object to be appended. Then we write the new element into the space.

Let's review how the `append` method works. The first time the method is called, we retrieve the End entry and call `increment`, which increments the position to the value one and returns a value of zero. So, zero is the position in the array where our first element will be added. We then immediately return the End entry to the space. Now that we've got a spot for our element, we add it to the

space. This might strike you as a little strange—even before we added our element to the space, we allocated a position for it and wrote an End entry that indicates that such an element exists. There's no need to worry: Given the blocking semantics of both take and read, if a process sees that an element has been allocated and tries to read or remove it from the space, then it will block until the element shows up.

Now that we can add elements to the array, what about determining the current size of the array? Here is a method that accomplishes this:

```
public int size()
    throws RemoteException, TransactionException,
           UnusableEntryException, InterruptedException
{
    Start startTemplate = new Start();
    startTemplate.name = name;

    End endTemplate = new End();
    endTemplate.name = name;

    Start start = (Start)
        space.read(startTemplate, null, Long.MAX_VALUE);
    End end = (End)
        space.read(endTemplate, null, Long.MAX_VALUE);

    return (end.position.intValue() -
            start.position.intValue());
}
```

To obtain the size of the array, we first read the Start and End entries from the space and assign them to local variables. We compute the size of the array by using our local copies to subtract the start position from the end position.

It's sufficient in this case to read (rather than take and return) the Start entry—since no elements are ever removed from the array and the start position won't change after we read it. It is also sufficient to read (rather than take) the End entry because it gives us a snapshot of how big the array is at a given time. If elements were being removed from the array, however, reading the entries wouldn't give us a consistent view of the array: after we read the Start entry, it might change in the space even before we manage to read the End entry. In other words, the start and end positions we get by using read may not represent an actual state of the array at any point in time. In a situation like this, we would use take to remove the Start and End entries (obtaining both the start and end values before other processes are allowed to change them), get an accurate measure of

the array's size, and then write the entries back. We'll see examples along these lines in Chapter 5.

What about reading an element of the array? We can do this just as we did earlier in the chapter with our simple array example:

```
public Object readElement(int pos)
    throws RemoteException, TransactionException,
           UnusableEntryException, InterruptedException
{
    Element template = new Element(name, pos, null);

    Element element = (Element)
        space.read(template, null, Long.MAX_VALUE);
    return element.data;
}
```

This method takes the integer position of the array element we want to read, and creates an `Element` template with the array name and the position. It then uses the template to read the array element from the space.

Finally, let's exercise our methods a bit with the following example code:

```
public class DistribArray {
    private JavaSpace space;
    private String name;

    // ... constructor/create/append/size/readElement here

    public static void main(String[] args) {
        JavaSpace space = SpaceAccessor.getSpace();

        try {
            String name = "duke's array";
            DistribArray array = new DistribArray(space, name);

            array.create();

            for (int i = 0; i < 10; i++) {
                array.append(new Integer(i));
            }

            System.out.println("Size of array is " + array.size());
        }
    }
}
```

```

        for (int i = 0; i < 10; i++) {
            Integer elem = (Integer)array.readElement(i);
            System.out.println("Elem " + i + " is " + elem);
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}

```

The `DistribArray` class contains the implementations of the `create`, `append`, `size`, and `readElement` methods that we've just seen.

After obtaining a handle to a space object, the application instantiates a `DistribArray`, passing it the reference to the space and a name “duke’s array,” and assigns the result to `array`. Next we call `create` to build the initial distributed array in the space, and then we loop, appending integer values 0 through 9 to the array. At that point, we print out the result of calling `size` on the array. Finally, we loop again, reading elements 0 through 9 of the array and printing out their values. When we run this application, we’ll see the following output:

```

Size of array is 10
Elem 0 is 0
Elem 1 is 1
Elem 2 is 2
Elem 3 is 3
Elem 4 is 4
Elem 5 is 5
Elem 6 is 6
Elem 7 is 7
Elem 8 is 8
Elem 9 is 9

```

3.5 Summary

In this chapter we’ve seen how to take ordinary data structures and create distributed versions that can be concurrently accessed and manipulated by multiple processes. We started with the most basic example: an individual entry that can act as a shared variable and provide important atomic properties that allow us to safely update its values without explicit locking. These properties follow directly from the key features of JavaSpaces technology. From there we looked at slightly more

complicated data structures; while most of these data structures have conventional analogs (arrays) others (like the unordered bag) may have been new to you. There is no limit to the kinds and complexity of distributed data structures you can build using entries: distributed linked lists, hierarchical tree structures, graphs, and so on.

Now that we've gotten the basic building blocks out of the way, we are going to explore their use in two different, but related, domains: synchronization and communication.

3.6 Exercises

Exercise 3.1 Extend the distributed array example to create a multidimensional array.

Exercise 3.2 Extend the distributed array example to support copying of one array element to another. Make sure it is done safely, such that no other process can alter the value of either element as the copying takes place.

Exercise 3.3 Extend the distributed array example by implementing a prepend method. The method should take an Object and should add the object to the beginning of the distributed array in the space.

Synchronization

Never practice two vices at once.

—Tallulah Bankhead

SYNCHRONIZATION plays a crucial role in the design of any distributed application. Inevitably, to accomplish a common task or goal, the processes in a distributed system need to coordinate with one another and avoid stepping on each other's toes. For instance, distributed applications often need to mediate access to limited shared resources, ensure fair access to resources, or prevent a set of processes from coming to a standstill because of competing requests for the same resources.

Synchronizing distributed processes can be challenging. In a nondistributed system in which multiple threads are running, the operating system serves as a centralized manager of all synchronization. In a distributed environment, on the other hand, asynchronous processes run independently at their own pace, and there is no central controller that can manage their activities and interactions. To synchronize processes, we have to rely on designing and building distributed means of cooperation, which can be tricky and error-prone.

Creating your distributed applications with spaces can significantly ease the burden of synchronizing processes. This is largely because synchronization is already built into the space operations themselves. You've already seen the basics of how synchronization works in space-based programming: Multiple processes can read an entry in a space at any time, but when a process wants to update an entry, it has to remove it from the space and thereby gain exclusive access to it first. In other words, coordinated access to entries is enforced by the `read`, `take`, and `write` operations themselves.

Entries and the operations on them give us everything we need to build more complex synchronization schemes. In this chapter, we'll explore useful synchronization techniques that can be implemented using some basic distributed data structures.

4.1 Semaphores

Processes often need to coordinate access to a set of shared resources. A *semaphore* is a synchronization construct that was first used to solve concurrency problems in operating systems back in the 1960s. Today, they are commonly found in multithreaded programming languages, but are more difficult to achieve in distributed systems.

The conventional definition of a semaphore is an integer counter that represents the count of available resources at any given time. A semaphore is controlled by two operations:

- ◆ down: checks to see if the counter is greater than zero, and if so, decrements its value by one. If the counter is zero, then the calling process blocks until the value is greater than zero. This operation is atomic: by calling down, a process tests and decrements the counter in an indivisible operation.
- ◆ up: increments the counter value. If one or more processes are waiting in the down operation, then one is chosen at random to proceed.

Semaphores are used as follows: A semaphore is initially created with its counter set to the total number of resources available. Whenever a process wants to access a resource, it must down the semaphore before using it. When it's done using a resource, the process calls up on the semaphore and releases it for use by others. The semaphore counter value will go up and down, depending how many processes are currently using the resources, and how many remain free. When a semaphore's count goes to zero, any process that tries to access a resource will block when calling down on the semaphore until a resource becomes available.

Semaphores are typically implemented as an integer counter that requires special language or hardware support to ensure the atomic properties of the up and down operations. Using spaces we could easily implement a semaphore as a shared variable that holds an integer counter. However we are going to take a distributed data structure approach to semaphores and implement a more scalable solution. Our implementation strategy is as follows: Instead of maintaining an integer counter, we'll make use of the bag distributed data structure and put one entry into the bag for each available resource. In effect we will have a bag of resources. When a process wants to down a semaphore, it must take an entry out of the bag first. When it's finished with the resource, to up the semaphore it must write a semaphore entry back to the space.

You can think of the semaphore entries as a pool of available resources: If n entries exist in the space at a given time, that means n resources are currently unused and available. That is, in our space-based implementation we are replacing

an integer counter (that has a maximum value of n), with a pool of entries (that at any time contains at most n entries). Note that a “semaphore entry” is not a semaphore; it takes an entire bag of semaphore entries to make up a semaphore.

4.1.1 Implementing a Semaphore

To implement a space-based semaphore we first need to define an entry that will represent one resource in the pool:

```
public class SemaphoreEntry implements Entry {
    public String resource;

    public SemaphoreEntry() {
    }

    public SemaphoreEntry(String resource) {
        this.resource = resource;
    }
}
```

The single field, `resource`, is a string that uniquely identifies the resource. We can have many semaphores in the space as long as each group of `SemaphoreEntry` entries is identified by a unique resource name. We can also add more fields to describe properties of the specific resource this entry represents (such as the address of a specific printer).

Now that we have an entry to work with, we can implement the semaphore itself. To implement the distributed semaphore we are going to define a class `DistributedSemaphore`. Here is the code:

```
public class DistributedSemaphore {
    private JavaSpace space;
    private String resource;

    public DistributedSemaphore(JavaSpace space, String resource) {
        this.space = space;
        this.resource = resource;
    }

    public void create(int num) {
        for (int i = 0; i < num; i++) {
            SemaphoreEntry semaphoreEntry =
                new SemaphoreEntry(resource);
        }
    }
}
```

```

        try {
            space.write(semaphoreEntry, null, Lease.FOREVER);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public void down() {
        SemaphoreEntry template = new SemaphoreEntry(resource);
        try {
            space.take(template, null, Long.MAX_VALUE);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public void up() {
        SemaphoreEntry semaphoreEntry =
            new SemaphoreEntry(resource);
        try {
            space.write(semaphoreEntry, null, Lease.FOREVER);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

Here we provide a constructor that takes a space and a resource name. The combination of these two uniquely identifies a specific semaphore. The constructor assigns these values to the private fields `space` and `resource`, respectively.

Once a semaphore object has been instantiated, you can call three methods: `create`, `down`, and `up`. The `create` method is called by one process to allocate a certain number of resources. Processes that need to obtain the resources never call `create`. The `create` method takes an integer `num`, which specifies the number of resources to create in the semaphore, and writes `num` entries into the space. For instance, specifying the number five would result in five `SemaphoreEntry` entries being added to the space.

The `down` method creates a `SemaphoreEntry` template with its `resource` field set to the resource name of the `DistributedSemaphore` object—and calls `take` to remove a `SemaphoreEntry` for the resource from the space. If no such entries

are available, then no such resources are available and `take` will wait indefinitely until one shows up. Once a `SemaphoreEntry` is taken from the space, the `down` method returns to the calling process, which can proceed knowing it has the resource.

The `up` method creates a new `SemaphoreEntry` and writes it into the space. Once it is returned, the resource can be used by other processes; in fact, some may already be waiting.

Now that we've implemented a distributed semaphore, let's put it to use in an example application: a license manager for a networked software product.

4.1.2 Implementing a License Manager

Networked software products are commonly sold with a license for a fixed number of simultaneous users. Using our distributed semaphore we are going to implement a license manager library that ensures only a limited number of clients can run at the same time. Our library isn't designed to be bulletproof, but it does demonstrate how we can implement a license service using semaphores.

The library consists of two distinct parts: a trivial installer component that is used when installing the product, and a runtime component that is embedded in a client application to ensure that a license resource is available before the application runs. Let's see how the components work.

4.1.3 The License Manager Installer

Our installer is a simple application that takes a product name and number of seats as command line parameters. For instance, a command line to execute the installer looks something like this:

```
java jsbook.chapter4.license.LicenseInstall JSGame 3
```

In this example, the application parameters specify that the software product is called "JSGame" and comes with a three-seat license that restricts its use to just three simultaneous users. This isn't very safe—anyone installing the product could change the parameters—but our goal here is to demonstrate semaphores, not build a secure installer.

Here's the `LicenseInstall` code:

```
public class LicenseInstall {  
    public static void main(String[] args) {  
        JavaSpace space = SpaceAccessor.getSpace();  
  
        String product = args[0];  
        int seats = Integer.parseInt(args[1]);  
    }  
}
```

```

        DistributedSemaphore license =
            new DistributedSemaphore(space, product);
        license.create(seats);
        System.out.println("License created for " + seats +
            " seats");
    }
}

```

When the `LicenseInstall` application starts, its `main` method is called, which first obtains access to the space. The application then processes the command-line parameters to obtain the product name and number of seats. Next, the application creates a `DistributedSemaphore` initialized with the current space and product name (in this case “JSGame”). Then the `create` method is called to create a semaphore with the specified number of seats (in this case three). Once the `LicenseInstall` execution is complete, the license is installed and clients can make use of it.

4.1.4 The License Manager Client Library

The license manager is implemented as a simple object that the client-side application instantiates as the application starts up. To obtain a license to run a product the application calls the license manager’s `getLicense` method. Before the application terminates it calls the license manager’s `returnLicense` method.

Here is the code for the `LicenseManager`:

```

public class LicenseManager {
    private DistributedSemaphore semaphore;

    public LicenseManager(JavaSpace space, String product) {
        semaphore = new DistributedSemaphore(space, product);
    }

    public void getLicense() {
        System.out.println("getting key");
        semaphore.down();
    }

    public void returnLicense() {
        System.out.println("returning key");
        semaphore.up();
    }
}

```

Here the constructor takes a space and a product name and uses them to instantiate a private `DistributedSemaphore` object. The `getLicense` method simply calls a down on the distributed semaphore, which has the effect of obtaining a license; while the `returnLicense` method calls up, which has the effect of releasing the license.

Now any class can make use of this license manager by adding a small code fragment to its startup code; here is an example:

```
LicenseManager licenseMgr = new LicenseManager(space, product);
licenseMgr.getLicense();

// application's main loop here

licenseMgr.returnLicense();
```

Assuming that `space` and `product` variables have been initialized, we instantiate a `LicenseManager` and call `getLicense`. If a license is available, then the call returns immediately and we can use the application, otherwise we wait until one is available. When we are finished with the application we return the license.

The semaphore technique for controlling access to licensed software works equally well no matter how many seats the license allows, and no matter whether we have two users sharing access to the software or hundreds. If all the license resources are taken, other users will wait until a license becomes available before they can start running the software. Of course, it makes sense for the number of seats to comfortably accommodate usage demands. If there are 100 users who want to use the application frequently, the three-seat license of our example will severely limit users' access to the software.

As we mentioned at the outset, this isn't an industrial-strength license manager quite yet. For one thing, we've sidestepped the issue of partial failure for the time being. If a process takes a license semaphore entry from the space and then crashes before writing it back, the entry is lost for good and our resource pool is reduced by one. For our semaphore code to work in the presence of failure, we will have to introduce transactions; we'll return to the topic of transactions in Chapter 9. The second drawback of our license manager is that it is very easy to forge license resources and override the semaphore: a piece of code could take a license from the space and then write it back not just once, but as many times as it pleases.

4.2 Using Multiple Semaphores

In many distributed systems, synchronization needs to go far beyond what can be handled by a single semaphore. Often, many different kinds of resources come

into play, and the system must control access to each kind, whether through semaphores or other techniques. But even when semaphores are in place to coordinate access to all resources, that isn't always enough by itself to prevent synchronization problems.

Consider the following example:

```
public aThenB() {  
    //... obtain resource A  
    //... obtain resource B  
    //... compute  
}  
  
public bThenA() {  
    //... obtain resource B  
    //... obtain resource A  
    //... compute  
}
```

Suppose Process 1 executes the `aThenB` method, while Process 2 executes `bThenA`. The following interleaved execution might commonly occur:

```
Process 1 obtains resource A.  
Process 2 obtains resource B.  
Process 1 waits for resource B.  
Process 2 waits for resource A.
```

In this scenario each processes is blocked and waiting for the other to release a resource. Neither can proceed—the processes are deadlocked. A *deadlock* occurs when each process in a collection is waiting for an action or response that only another process in the same collection can generate.

There are no general-purpose techniques for preventing deadlock, other than careful algorithm design. However, these types of problems can often be addressed by using additional semaphores to coordinate access to multiple resources. Let's look at an example of how this can be accomplished.

4.2.1 The Dining Philosophers

Our example comes out of a classical coordination problem from computer science known as the *dining philosophers problem*. The dining philosophers problem is often used to measure the expressivity of a particular coordination language or tool. Here is the story of the dining philosophers from computer science folklore:



Figure 4.1 The dining philosophers.

According to the story, five philosophers sit around a table as shown in Figure 4.1. Each has a chopstick to his left. An everlasting bowl of rice is placed in the middle of the table. A philosopher spends most of his time thinking, but whenever he is hungry he picks up the chopstick to his left and the chopstick to his right. When attempting to pick up either chopstick, if he finds it already in use he simply waits until it becomes available again. When he has finished eating he puts down both chopsticks and continues to think.

We can easily simulate this scenario by creating one semaphore per chopstick, but there is a serious flaw in the system: suppose all five philosophers enter the room and pick up their left chopsticks at once; this leaves no right chopsticks available, and no one is able to eat. They are deadlocked.

We can solve this problem by allowing only four philosophers to enter the room at a time—ensuring that at least one will have access to both his left and right chopsticks and be able to eat. We can control access to the room using four ticket resources represented by a semaphore: a philosopher can enter the room only after obtaining a ticket first.

Here is a space-based implementation of a dining philosopher:

```
public class Philosopher {
    private JavaSpace space;
    private int id;
    private int num;
    private DistributedSemaphore ticket;
    private DistributedSemaphore left;
    private DistributedSemaphore right;
```

```
public static void main(String[] args) {
    JavaSpace space = SpaceAccessor.getSpace();
    int id = Integer.parseInt(args[0]);
    int num = Integer.parseInt(args[1]);

    Philosopher philosopher = new Philosopher(space, id, num);
    philosopher.work();
}

public Philosopher(JavaSpace space, int id, int num) {
    this.space = space;
    this.id = id;
    this.num = num;

    if (id == 0) {
        initSpace();
    }

    ticket = new DistributedSemaphore(space, "ticket");
    left = new DistributedSemaphore(space,
        "chopstick" + id);
    right = new DistributedSemaphore(space,
        "chopstick" + ((id + 1) % num));
}

public void work() {

    for (;;) {
        try {
            think();

            ticket.down();
            left.down();
            right.down();

            eat();

            right.up();
            left.up();
            ticket.up();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```
        }
    }
}

private void initSpace() {
    DistributedSemaphore ticket =
        new DistributedSemaphore(space, "ticket");
    ticket.create(num - 1);

    for (int i = 0; i < num; i++) {
        DistributedSemaphore chopstick =
            new DistributedSemaphore(space, "chopstick" + i);
        chopstick.create(1);
    }
}

private void think() {
    System.out.println("Philosopher " + id + " thinking.");

    try {
        Thread.sleep(500);
    } catch (Exception e) {
        e.printStackTrace();
    }
}

private void eat() {
    System.out.println("Philosopher " + id + " eating.");

    try {
        Thread.sleep(500);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Let's look carefully at what this code does. The `main` method expects two command line arguments: An integer identifier that is used to identify a particular philosopher (philosophers are labelled 0 through `num-1`) and an integer that represents the total number of philosophers. After gaining access to a space, the `main` method assigns these command line arguments to `id` and `num`, respectively.

The method then instantiates a `Philosopher` object, passing it the space, identifier and total number of philosophers, and calls its `work` method.

Let's now take a look at the constructor for the `Philosopher` class. The constructor first assigns each of its parameters to local fields. The philosopher with identifier 0 then has the special task of initializing the space by calling `initSpace`, which writes `num-1` ticket entries into the space and then `num` chopstick entries (identified as "chopstick0", "chopstick1," and so on).

Next, every philosopher creates a `DistributedSemaphore` object for the ticket, and its left and right chopsticks. The left chopstick is given the philosopher's own identifier, and the right chopstick is given an identifier one greater than the philosopher's (modulo the number of chopsticks).

Now let's look at the `work` method, which allows the philosopher to do what all dining philosophers do: repeatedly think and eat. First the philosopher thinks for a while, and then he tries to down the ticket semaphore, waiting until one is found. Once he has obtained a ticket to the dining room, he picks up the chopstick to his left (by calling `down` on that chopstick's semaphore), and then picks up the chopstick to his right (also by calling `down` on that chopstick's semaphore). Note that the method names here are a little confusing: To pick "up" a chopstick we "down" the chopstick's semaphore. But if you think about the semantics of the semaphore operations that we discussed in Section 4.1, it makes sense.

Once the philosopher has both chopsticks in hand, he eats for a while. When he's done eating, he puts down the right chopstick, then the left chopstick, and then gives back his ticket so another hungry philosopher can have a chance to eat. The `eat` and `think` methods just print an appropriate message and sleep for half a second.

To run five dining philosophers you need to execute five commands like the following (note that these should be run simultaneously, not one after the other):

```
java jsbook.chapter4.dining.Philosopher 0 5
java jsbook.chapter4.dining.Philosopher 1 5
java jsbook.chapter4.dining.Philosopher 2 5
java jsbook.chapter4.dining.Philosopher 3 5
java jsbook.chapter4.dining.Philosopher 4 5
```

Note that the `id` parameter's value ranges from 0 through 4, to identify each philosopher. When we start up the five philosophers, their interleaved output looks something like this:

```
Philosopher 2 thinking.
Philosopher 4 thinking.
Philosopher 3 thinking.
Philosopher 1 thinking.
```

```
Philosopher 0 thinking.  
Philosopher 4 eating.  
Philosopher 3 eating.  
Philosopher 4 thinking.  
Philosopher 2 eating.  
Philosopher 3 thinking.  
Philosopher 1 eating.  
Philosopher 2 thinking.  
Philosopher 0 eating.  
Philosopher 1 thinking.  
Philosopher 4 eating.  
Philosopher 0 thinking.  
Philosopher 3 eating.  
Philosopher 4 thinking.  
.  
.  
.
```

Of course, the dining philosophers problem is abstract, but it illustrates important concepts that carry over any time you’re building a distributed application with complex synchronization needs. In the dining philosophers example, we’ve seen how multiple processes can coordinate access to various limited resources: each of the chopsticks is a distinct resource that two processes must share. In order to avoid a situation where all processes are deadlocked, we introduced an additional ticket semaphore that limits access to the room.

4.3 Fairly Sharing a Resource

Our dining philosophers implementation was successful in that it synchronizes the actions of multiple processes competing for limited resources and prevents them from entering into a deadlocked state. However, there is one potential problem it doesn’t make any attempt to solve—starvation. *Starvation* occurs when a process can’t access an available resource, because other processes are beating it to the mark—either they are faster, or on a better network connection, or have some other competitive advantage. The dining philosophers code makes no guarantee that every philosopher will eventually get a turn to eat, or that philosophers will eat with the same frequency. Your fellow philosophers might repeatedly beat you to any tickets that become available, preventing you from entering the dining room and eating.

We can address starvation by enforcing fair access to shared resources in a number of ways. One simple technique is to force processes to take turns. Let's see how.

4.3.1 Using a Queue to Take Turns

Consider what happens when you want to be waited on, say at a deli. You take a numbered ticket from a dispenser, in effect placing yourself at the end of the current line of waiting customers. To keep things simple, we'll assume that just one customer can be served at a time. There is typically a message board indicating the "service number" of the customer currently being served. You wait patiently until your number is displayed, and then it's your turn for service. When you're done, the service number gets incremented by one.

Under this first-come, first-served scheme, everyone who requests service will eventually get a turn (as long as each service request has an upper time limit). In other words, customers are treated fairly and won't be made to wait for service indefinitely.

Of course, we are just describing a queue, but our reason for introducing the deli theme is that an implementation of a space-based distributed queue is similar to the working of a ticket dispenser. Let's see how this works.

To implement a queue we need a ticket dispenser and a service number. Both can be represented with our shared variable class. Here is how we can create a queue:

```
SharedVar dispenser = new SharedVar("Ticket Dispenser", 0);
SharedVar serviceNum = new SharedVar("Service Number", 0);

try {
    space.write(dispenser, null, Lease.FOREVER);
    space.write(serviceNum, null, Lease.FOREVER);
    System.out.println("Dispenser ready to dispense ticket " +
        dispenser.value);
    System.out.println("Ready to service " +
        serviceNum.value);
} catch (Exception e) {
    e.printStackTrace();
}
```

Here we simply instantiate two shared variables with the appropriate names and write them into the space. You'll find an installer application that does this in the complete source code.

Once our queue is created, to enter the queue a customer needs to take a number and wait its turn. Here's the code for `takeANumber`:

```
private int takeANumber() {
    int myNum = 0;
    try {
        SharedVar template = new SharedVar("Ticket Dispenser");
        SharedVar dispenser = (SharedVar)
            space.take(template, null, Long.MAX_VALUE);
        myNum = dispenser.value.intValue();
        dispenser.increment();
        space.write(dispenser, null, Lease.FOREVER);
    } catch (Exception e) {
        e.printStackTrace();
    }
    return myNum;
}
```

The `takeANumber` method first sets up a template to retrieve the “Ticket Dispenser” shared variable. It then takes the shared variable from the space (waiting if necessary), stashes its value in `myNum`, and calls the dispenser’s `increment` method. We then write the dispenser back into the space and return `myNum`. This mirrors what happens when you take a number from a mechanical ticket dispenser: you take (and keep) a number from the dispenser and at the same time advance the dispenser’s ticket value.

Once a customer has retrieved a number from the dispenser, it has to wait for its turn. Here is the code for `waitForTurn`:

```
public SharedVar waitForTurn(int myNum) {
    try {
        System.out.println("Customer " + myNum +
            " waits for turn");
        SharedVar template =
            new SharedVar("Service Number", myNum);
        SharedVar serviceNum = (SharedVar)
            space.take(template, null, Long.MAX_VALUE);
        return serviceNum;
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}
```

To wait for a turn, we first create a SharedVar template, using its constructor to set the name field to “Service Number” and the value field to myNum. We then perform a take with the template, which will cause the process to wait until the service number reaches the number myNum. In effect, the customer is waiting until all the customers with lower numbers are serviced.

When through with its turn, a customer calls incrementServiceNumber to increment the service number and return the entry to the space so the next customer in line can have its turn. Here’s the code that does this:

```
public void incrementServiceNumber(SharedVar serviceNum) {
    serviceNum.increment();
    try {
        space.write(serviceNum, null, Lease.FOREVER);
        System.out.println("Ready to service " +
                           serviceNum.value);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

That is all there is to implementing a distributed queue. Let’s put all this together into a simple test application:

```
public class Customer {
    private JavaSpace space;

    public Customer(JavaSpace space) {
        this.space = space;
    }

    public static void main(String[] args) {
        JavaSpace space = SpaceAccessor.getSpace();
        Customer customer = new Customer(space);

        // get a numbered ticket from the dispenser
        int myNum = customer.takeANumber();

        // wait your turn
        SharedVar serviceNum = customer.waitForTurn(myNum);

        // take your turn
        System.out.println("Customer " + myNum + " takes its turn");
    }
}
```

```
try {
    Thread.sleep(1000);
} catch (Exception e) {
    e.printStackTrace();
    return;
}

// increment the service number
customer.incrementServiceNumber(serviceNum);
}

// ... other method definitions go here
}
```

Let's step through this code. The `main` method of the application obtains a handle to a space and then instantiates a `Customer` object, which takes a number from the ticket dispenser (`takeANumber`), waits in line for its turn (`waitForTurn`), and then takes its turn (in this case, all the customer does is sleep for a while to simulate the time it takes to be serviced). In this example, only one customer is served at a time. So, when the customer is done with its turn, it increments the service number (`incrementServiceNumber`) so that the next customer in line can proceed. You can imagine a different scenario where there are multiple workers servicing multiple customers at once; in that case, the workers would increment the service number every time they finish with a customer.

Now it is time to make a test run of our turn-taking application. Here is sample output of running five customer applications:

```
Dispenser ready to dispense ticket 0
Ready to service 0
Customer 0 waits for turn
Customer 0 takes its turn
Customer 1 waits for turn
Customer 2 waits for turn
Customer 3 waits for turn
Customer 4 waits for turn
Ready to service 1
Customer 1 takes its turn
Ready to service 2
Customer 2 takes its turn
Ready to service 3
Customer 3 takes its turn
Ready to service 4
Customer 4 takes its turn
Ready to service 5
```

Here all five customers show up, take a ticket and wait for their turn. More customers can come along later and pick up where these left off. The next customer to show up takes number five from the ticket dispenser and can take its turn without waiting, since the service number is already up to five.

4.3.2 Round-Robin Synchronization

In the previous example, there is a potentially unlimited line of customers waiting for service. As soon as a customer has had a turn at the resource or service, it's done; to take another turn, the customer has to start all over again by taking a new ticket from the dispenser.

Round-robin synchronization provides another method of sharing a resource in which a set of processes accesses a resource repeatedly. With round-robin, we arrange processes in a ring and grant the resource to the processes one after another, going around the ring. Once each process has had a turn, the ring is traversed again, giving each process another turn. This circular resource sharing can either continue indefinitely or for a specified number of rounds.

Here is how we can use the round-robin technique to coordinate n participants: Each participant is given a unique numeric identifier between 0 and $n-1$, and the service number is started at 0. Each process waits for the service number to reach its own identifier, and then takes its turn. When finished with its turn, it increments the service number. In this manner, participant i can't take its turn until the participant $i-1$ has finished. Once every member of the circle has taken a turn, the "last" process in the ring updates the service number to point to the "first" process by resetting the service number to 0, and another round begins.

Using our code from the last example as a starting point, we first need to rework the `SharedVar` a bit for use with the round-robin technique. Here we define a `RoundRobinSharedVar` that extends `SharedVar`:

```
public class RoundRobinSharedVar extends SharedVar {
    public Integer numParticipants;

    public RoundRobinSharedVar() {
    }

    public RoundRobinSharedVar(String name,
                               int value)
    {
        this.name = name;
        this.value = new Integer(value);
    }
}
```

```

public RoundRobinSharedVar(String name,
    int value, int numParticipants)
{
    this.name = name;
    this.value = new Integer(value);
    this.numParticipants = new Integer(numParticipants);
}

public Integer increment() {
    int curValue = value.intValue();
    int numParts = numParticipants.intValue();
    value = new Integer(
        curValue == (numParts - 1) ? 0 : (curValue + 1));
    return value;
}
}

```

We've added a new field called `numParticipants`, which holds the number of participating processes, and also a new constructor that takes an extra integer argument and assigns its value to the `numParticipants` field. As usual, we supply the required no-arg constructor.

We also override the `increment` method, which must update the `value` field to represent the next participant in the circle. It's no longer correct for `increment` to simply add 1 to the current value, since participants are now arranged in a ring, not a line. If `value` holds the value of the last process in the ring (the one numbered one less than the number of participants), the method sets `value` to 0, otherwise it just adds 1 to the current value as before.

Instead of a `Customer` class, we define a `RoundRobinParticipant` class, with a few small differences. Here is its `main` method, which contains the only changes:

```

public static void main(String[] args) {
    JavaSpace space = SpaceAccessor.getSpace();
    RoundRobinParticipant participant =
        new RoundRobinParticipant(space);

    int myNum = Integer.parseInt(args[0]);
    int rounds = Integer.parseInt(args[1]);
    int numParticipants = Integer.parseInt(args[2]);

    if (myNum == 0) {
        participant.initSpace(numParticipants);
    }
}

```

```

for (int i = 0; i < rounds; i++) {
    // wait your turn in this round
    RoundRobinSharedVar serviceNum =
        participant.waitForTurn(myNum);

    // take your turn
    System.out.println("Round " + i + ": Process " +
        myNum + " takes its turn");
    try {
        Thread.sleep(1000);
    } catch (Exception e) {
        e.printStackTrace();
    }

    // increment the service number
    participant.incrementServiceNumber(serviceNum);
}
}

```

There are a few changes from the previous *Customer* code. The application now takes three command-line arguments: an identifying number (`myNum`), a number of rounds (`rounds`) and the total number of participants (`numParticipants`). Because we are assigning an identifying number to each process, the participant no longer uses a ticket dispenser to obtain a number. The process now loops `rounds` times; each time through the loop, the participant waits for its turn, takes its turn, and increments the service number, just as *Customer* did. If the process is the last participant in the ring (its number is `numParticipants - 1`), then our new implementation of `increment` resets the service number to 0, the number of the first participant.

If we start up five processes that will run for two rounds, here is what the output will look like:

```

Round 0: Process 0 takes its turn
Round 0: Process 1 takes its turn
Round 0: Process 2 takes its turn
Round 0: Process 3 takes its turn
Round 0: Process 4 takes its turn
Round 1: Process 0 takes its turn
Round 1: Process 1 takes its turn
Round 1: Process 2 takes its turn
Round 1: Process 3 takes its turn
Round 1: Process 4 takes its turn

```

4.4 Barrier Synchronization

Barriers are a common and simple space-based technique used to synchronize a group of processes. A *barrier* is a particular point in a distributed computation that every process in a group must reach before any process can proceed further. For instance, a distributed application may start many “worker” processes and have them wait until some initial conditions hold before they are allowed to start processing. Some distributed computations may also proceed in phases, where all processes need to complete phase one before proceeding as a group to phase two.

Barriers are easy to implement with a shared variable: If we have n processes that need to reach a certain point in the computation before any processes can continue, then to create a barrier we write a shared variable to a space with the value zero. As each process reaches the barrier, it takes the barrier entry, increments its value, and returns it to the space. Each process then reads the barrier entry with a template that specifies a counter value of n , which causes the process to wait until all other processes have reached the barrier.

Let’s look at a simple demonstration of a barrier using the `SharedVar` from Chapter 3. Here is a worker class that makes use of a barrier:

```
public class Worker {  
    public static void main(String[] args) {  
        JavaSpace space = SpaceAccessor.getSpace();  
  
        int id = Integer.parseInt(args[0]);  
        int numWorkers = Integer.parseInt(args[1]);  
  
        try {  
            // let worker zero create the barrier  
            if (id == 0) {  
                SharedVar barrier =  
                    new SharedVar("barrier", 0);  
                space.write(barrier, null, Lease.FOREVER);  
            }  
  
            System.out.println("Worker " + id +  
                " before the barrier");  
  
            // barrier point - take the barrier entry and  
            // increment its counter  
            SharedVar template = new SharedVar("barrier");  
            SharedVar result = (SharedVar)  
                space.take(template, null, Long.MAX_VALUE);  
        }  
    }  
}
```

```

        result.increment();
        space.write(result, null, Lease.FOREVER);

        // wait until barrier entry has been updated by
        // all workers
        SharedVar barrier =
            new SharedVar("barrier", numWorkers);
        space.read(barrier, null, Lease.FOREVER);

        // proceed past the barrier
        System.out.println("Worker " +
            id + " past the barrier");

    } catch (Exception e) {
        e.printStackTrace();
    }
}
}

```

The worker's `main` method expects two command line parameters: a unique integer identifier for the process (a number between 0 and `n-1`) and the total number of workers participating; these parameters are assigned to `id` and `numWorkers` fields, respectively.

Following along, the worker then enters into the barrier part of the code. If the process has an `id` of zero, then it is responsible for writing the barrier entry into the space with a value of zero. First we do some work before reaching the barrier; in this example the work is just a `println` statement that prints:

Worker 0 before the barrier

Once we reach the barrier we create a template, remove the matching barrier entry from the space, increment its value, and return the entry to the space. As each process reaches this fragment of code, the number in the barrier entry will approach `numWorkers`.

The worker then creates a barrier template, specifying a counter value of `numWorkers`, and calls `read`. This `read` will block until all processes have reached the barrier and incremented the barrier entry. Once that occurs, the value of the barrier is at `numWorkers`, and all processes complete their `read` operation and proceed to do post-barrier work, which in this example is a `println` statement that prints:

Worker 0 past the barrier

That is really all there is to a barrier. Give this example a try by running several workers.

Note that we have presented one method of creating a barrier, but there are many variations on this theme: barriers can be initially set to `numWorkers` and decremented each time a worker reaches the barrier. Other times, barriers aren't dependent on the number of processes, but rather some external condition (such as the total tasks computed by a set of processes), and a master process places an entry into the space when it is okay to move past the barrier. We will see an example of this in Chapter 11.

4.5 Advanced Synchronization: The Readers/Writers Problem

Even complex synchronization problems can be handled with variants of the techniques you have already encountered in this chapter. As an example, we'll look at a classical synchronization problem known as the *readers/writers problem*.

The readers/writers problem goes like this: A collection of processes need to share access to some resource. Some processes are "readers" that need to read the current state of the resource, while others are "writers" that need to update its state. Two rules must be followed:

1. The resource may be accessed by multiple readers or by a single writer at a given time, but not both.
2. No read request or write request may be postponed indefinitely.

According to the first rule, if readers are already active and additional processes show up that wish to read, we can allow them to read as well. If a writer shows up, on the other hand, it must wait till all the readers have finished. If a writer is active and a new write (or read) request arrives, the new process must wait until the writer finishes.

According to the second rule, readers and writers must be treated fairly. If readers are active and new read and write requests arrive, for instance, we can't perpetually allow the new readers to proceed (since, after all, multiple simultaneous readers are permitted) and ignore the waiting writers. In other words, a continual stream of read requests can't indefinitely put off a write request, and vice-versa.

Although readers/writers is stated as a general problem, we can apply its solution to practical scenarios. The solution is useful any time you want to restrict access to a resource to a single writer, or to multiple readers, at one time. For instance, consider an airline reservations system that allows access by multiple

travel agents at any given time. The resource in this case is a shared data structure that stores information about seat availability on a particular flight. To reserve or relinquish seats, a travel agent needs exclusive access to the flight resource. It makes sense to prohibit readers from reading the resource as it's being updated, because they may get inconsistent information. When no writer is updating the resource, it's reasonable to allow any number of readers to read the flight resource.

Implementing the readers/writers protocol is a matter of enforcing both of the conditions above. Doing so requires a bit of thought but is not a large conceptual leap from the previous examples in this chapter. Before diving into the code details, let's get a high-level overview of the algorithm we'll use to solve the readers/writers problem.

4.5.1 Implementing a Readers/Writers Solution

Conceptually, our solution to the readers/writers problem is simple. Readers and writers line up to take turns on a first-come, first-served basis. Whenever a reader or writer shows up, it takes a numbered spot at the end of a line and waits until its number gets called. When a reader gets its turn, it calls the number of the next process in the line, and then it goes ahead and reads. In essence, a reader immediately says "it's your turn" to the next process, and if that process is also a reader, it will say "it's your turn" to the next process, and so on—any lined up readers will be allowed to read. When a writer is encountered in the line, it has to wait until there are no more active readers before it can begin to write. It calls the number of the next process in line only when it's done writing—everyone else in line behind the writer will be kept waiting while the writer is active. This simple algorithm enforces both readers/writers rules: No read/write request will be postponed indefinitely, and just one writer, or multiple readers, can be active at any time.

To implement this scheme we are going to use three objects: a ticket dispenser, a counter of active readers, and a counter that indicates whose turn it is. When a reader or writer needs to access the resource, it first obtains a numbered ticket from the ticket dispenser. When it's the process's turn (as indicated by the turn counter), if the process is a reader, it increments the reader counter, sets the turn counter to the next process in line, and then begins reading. When a reader finishes, it decrements the reader counter. If the process is a writer, on the other hand, it waits until the counter of readers is down to zero, and then begins to write. When done writing, the process finally increments the turn counter to the next value. As long as readers and writers finish their work in a finite amount of time, every process that shows up will eventually get a turn.

4.5.2 Implementing a Counter

Now let's get into the implementation details. You've seen that a ticket dispenser, turn counter, and reader counter all play a role in the algorithm. We can implement each of these as a `Counter` object that provides methods that are handy for manipulating the shared variable in the space. These methods are used to create a shared variable in the space, to update a shared variable by writing it back to the space, to increment or decrement its value by one, and to wait for it to reach a particular value (and then read or remove the variable). Here's the class definition of `Counter`:

```
public class Counter {  
    private JavaSpace space;  
    private String name;  
  
    public Counter(JavaSpace space, String name) {  
        this.space = space;  
        this.name = name;  
    }  
  
    public void create(int count) {  
        SharedVar sharedVar = new SharedVar(name, count);  
        try {  
            space.write(sharedVar, null, Lease.FOREVER);  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
  
    public void update(int count) {  
        SharedVar sharedVar = new SharedVar(name, count);  
        try {  
            space.write(sharedVar, null, Lease.FOREVER);  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
  
    public Integer increment() {  
        try {  
            SharedVar template = new SharedVar(name);  
            SharedVar count = (SharedVar)  
                space.take(template, null, Long.MAX_VALUE);  
            if (count != null) {  
                int currentCount = count.getValue();  
                if (currentCount < Integer.MAX_VALUE) {  
                    count.setValue(currentCount + 1);  
                    space.write(count, null, Lease.FOREVER);  
                }  
            }  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
        return null;  
    }  
}
```

```

        Integer num = count.increment();
        space.write(count, null, Lease.FOREVER);
        return num;
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}

public Integer decrement() {
    try {
        SharedVar template = new SharedVar(name);
        SharedVar count = (SharedVar)
            space.take(template, null, Long.MAX_VALUE);
        Integer num = count.decrement();
        space.write(count, null, Lease.FOREVER);
        return num;
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}

public void waitFor(int num, String operation) {
    try {
        SharedVar template = new SharedVar(name, num);
        if (operation.equals("take")) {
            space.take(template, null, Long.MAX_VALUE);
        } else {
            space.read(template, null, Long.MAX_VALUE);
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}

```

You'll notice that the constructor takes two parameters, `space` and `name`, and sets the object's corresponding private variables to those values. Every `Counter` object is uniquely identified by its `space` and `name`.

The `create` method is called whenever we want to create a new shared variable in the space. It constructs a shared variable with the counter’s name, and with `count` (an integer parameter) as its starting value, and writes it to the space. The `update` method is used whenever we want to write an updated shared variable to the space; it’s implemented the same way as `create`.

The `Counter` class provides an `increment` method for increasing the value of a counter shared variable in the space by one. The method constructs a template and uses it to remove the shared variable from the space. After incrementing the shared variable, the method returns the entry to the space. The `Counter` class provides a corresponding `decrement` method as well.

Finally, the `Counter` class supplies a `waitFor` method that makes the calling process wait until the shared variable in the space reaches the value given by the first parameter, `num`. The second parameter, `operation`, is either “take” or “read”—to specify whether the variable should be removed from the space or simply read when it reaches the crucial value. We’ll see shortly that both operations have their uses in our readers/writers solution.

4.5.3 Implementing a Space-based Readers/Writers Application

Now we are ready to implement the readers and the writers. To do this we are going to create a `ReaderWriter` class:

```
public class ReaderWriter {  
    private JavaSpace space;  
    private int rounds;  
    private int id;  
    private Counter ticketDispenser;  
    private Counter turnCounter;  
    private Counter readerCounter;  
  
    public ReaderWriter(JavaSpace space, int rounds, int id) {  
        this.space = space;  
        this.rounds = rounds;  
        this.id = id;  
  
        ticketDispenser = new Counter(space, "ticket number");  
        turnCounter = new Counter(space, "turn number");  
        readerCounter = new Counter(space, "reader count");  
    }  
}
```

```
if (id == 0) {
    ticketDispenser.create(-1);
    turnCounter.create(0);
    readerCounter.create(0);
}
}

public static void main(String[] args) {
    JavaSpace space = SpaceAccessor.getSpace();

    int rounds = Integer.parseInt(args[0]);
    int id = Integer.parseInt(args[1]);

    ReaderWriter readerWriter =
        new ReaderWriter(space, rounds, id);

    readerWriter.readOrWrite();
}

private void readOrWrite() {
    for (int i = 0; i < rounds; i++) {
        boolean isReader = ((id % 2) == 0);

        if (isReader) {
            reader();
        } else {
            writer();
        }
    }
    System.out.println(id + " DONE");
}

private void reader() {
    waitToRead();
    System.out.println("reader " +
        id + " working");
    try {
        Thread.sleep(2000);
    } catch (Exception e) {
        e.printStackTrace();
    }
    doneReading();
}
```

```
private void waitToRead() {
    int myTicketNum = ticketDispenser.increment().intValue();
    System.out.println("reader " +
        id + " waiting for ticket " + myTicketNum);
    turnCounter.waitFor(myTicketNum, "take");
    readerCounter.increment();
    turnCounter.update(myTicketNum + 1);
}

private void doneReading() {
    System.out.println("reader " +
        id + " done");
    readerCounter.decrement();
}

private void writer() {
    int ticketNum = waitToWrite();
    System.out.println("writer " +
        id + " working");
    try {
        Thread.sleep(2000);
    } catch (Exception e) {
        e.printStackTrace();
    }
    doneWriting(ticketNum);
}

private int waitToWrite() {
    int myTicketNum = ticketDispenser.increment().intValue();
    System.out.println("writer " +
        id + " waiting for ticket " + myTicketNum);
    turnCounter.waitFor(myTicketNum, "take");
    readerCounter.waitFor(0, "read");
    return myTicketNum;
}

private void doneWriting(int ticketNum) {
    System.out.println("writer " +
        id + " done");
    turnCounter.update(ticketNum + 1);
}
```

You'll note that the `main` method starts as usual, by obtaining the space and a couple command line arguments (an integer identifier for the process and a number of rounds). The `main` method then instantiates a `ReaderWriter`, passing the space, the number of rounds, and the identifier to its constructor. The constructor instantiates three `Counter` objects; as we mentioned earlier, our readers/writers solution uses three counters: a ticket dispenser, a turn counter, and a reader counter.

The application with identifier 0 is given the responsibility of writing these counter shared variables to the space for the first time. The starting value of the ticket dispenser is -1, since a process will increment the ticket dispenser before taking the number. In other words, the first process to access the ticket dispenser increments the value to 0 and walks off with number 0, the second process increments the value to 1 and takes number 1, and so on. We start off the turn counter at 0, so that the process holding ticket 0 will get the first turn. The reader counter is also started at 0, since initially there are no readers.

After the `ReaderWriter` has been instantiated, the `main` method calls `readOrWrite`, which is where the real fun begins. In the `readOrWrite` method, we first determine whether the object will be a reader or a writer. If its `id` is even, it will be a reader, otherwise it will be a writer (of course, we could decide randomly, but our simple scheme suffices). The method then iterates `rounds` times, each time calling either the `reader` or `writer` method according to its role.

The reader acts just as we described in the overview of the algorithm. The reader first calls `waitToRead` to wait until the circumstances are right for it to read: It increments the ticket dispenser's value, waits until the turn counter has that same value, and then removes the turn counter variable from the space. The reader increments the count of active readers and then writes the incremented turn counter variable back to the space. After `waitToRead`, the process proceeds to do its "reading" (in this case, printing a message and sleeping for a bit), and finally calls `doneReading` to decrement the reader counter.

Like a reader, the writer must first wait for its turn by incrementing the ticket dispenser, waiting for the turn counter to have the same value, and then removing the turn counter variable from the space. Next, the writer must wait until the reader counter has a value of zero. Then the writer proceeds to write, and when it's finished, it writes the incremented turn counter variable back to the space so another process can have a turn.

There you have it: a space-based solution to the readers/writers problem. Though readers/writers is a complex coordination problem, we've managed to build a rather elegant solution that uses two elements we've used in the past: a fair access protocol based on tickets, and a shared variable data structure used to implement counters. With basic building blocks in hand, your imagination is the only limit to the complex coordination schemes you can develop using simple distributed data structures and protocols that operate over them.

4.6 Summary

In this chapter, we've seen a range of useful techniques for achieving synchronization in space-based programs. We've implemented and used semaphores to control access to a set of limited resources, and also to avoid deadlock. We've designed a turn-taking scheme that ensures fair access to a collection of resources and prevents starvation. We've seen one way—barriers—of allowing distributed computations to occur in distinct phases. Beyond that, we've explored a more advanced synchronization example that enforces rules about what kinds of access (reading or writing) there can be to a resource at a given time, and that also ensures fair access to the resource. In each case, we designed one or more distributed data structures to achieve the synchronization.

The techniques we've seen are really just the tip of the iceberg. Different distributed applications will have unique (and sometimes very complicated) synchronization requirements. But, regardless how complex the coordination gets, you now have the tools in hand to design complex synchronization schemes for your space-based programs. Distributed data structures and the protocols that operate over them are really the key ingredient.

4.7 Exercises

Exercise 4.1 Our round-robin code only works for a fixed number of participants. How would you make it dynamic, such that participants can be added? Added and deleted?

Exercise 4.2 Rewrite the deli queue example such that multiple workers behind the counter can service the queue of customers. Just as at a typical deli counter, a worker looks at the service number on the display board, takes a request from the customer with the matching ticket number, and services the request. Multiple customers can be serviced at once. Where should the service number now be incremented? Make sure the customer receives an acknowledgment when the request has been completed.

Communication

When your tail happens to be missing, someone will try to convince you that you left it somewhere. But you'll know that somebody must have taken it.

—Eeyore, **Eeyore's Gloomy Little Instruction Book**

IN the last chapter we used distributed data structures for synchronization, and in this chapter we are going to explore their use in communication. Together, communication and synchronization are the glue that holds processes together in any distributed application. If you master the basics of each, you will be well on your way to building sophisticated applications with spaces.

Space-based communication is a little different from the message passing or remote method invocation packages you may have used in the past. Space-based communication loosens the ties between senders and receivers of information and promotes a *loosely coupled* communication style in which senders and receivers don't interact directly, but instead indirectly through a space. This is a significant difference, because loosely coupled applications tend to be more flexible, scalable, and reliable than their tightly coupled counterparts, and also easier to build.

This chapter focuses on building loosely coupled *communication patterns*. By this, we mean the many forms of communication (beyond simple message passing) that often occur in distributed and collaborative applications. For instance, it is common for collaborative applications to broadcast a message to a group of processes. Other distributed applications need to use “anonymous” forms of communication, such as sending a service request to “any” service that knows how to help with a computation. In fact, the landscape of distributed applications makes use of an entire spectrum of communication patterns among processes, including communication from one sender to one receiver, from one or more senders to many receivers, or from many senders to just one receiver.

In this chapter, you will learn how to build useful communication patterns with spaces, and will see first-hand the benefits of building protocols that are loosely coupled. Our journey will start with the simplest pattern—message passing—and proceed through the more complex patterns. By the end of the chapter,

you should have a good understanding of how to accomplish complex communication with simple operations over entries in a space.

5.1 Basic Message Passing

It is easy to perform message passing in space-based programming by passing an entry from one process to another. To do so, we first need a message to pass. Let's begin by defining a `Message` entry with two fields: one that identifies the receiver, and another that holds the content of the message:

```
public class Message implements Entry {
    public String receiver;
    public String content;

    public Message() {
    }
}
```

To send a message, a process creates a `Message` entry and writes it into a space:

```
Message msg = new Message();
msg.receiver = "duke@sun.com";
msg.content = "Hello";

space.write(msg, null, Lease.FOREVER);
```

You'll note that we've used the string “duke@sun.com” as a way to identify the receiver, but any agreed-upon convention will do. We've also chosen a string to represent the content of the message, but obviously any object could be sent as the content.

To receive a message, a process creates a `Message` template that matches any messages meant for it—we do this by filling in the `receiver` field and leaving the `content` field `null` to act as a wildcard:

```
Message msg = new Message();
msg.receiver = "duke@sun.com";
```

Once the receiver has a template, it can be used to retrieve one or more messages from the space using `take`:

```
Message receivedMsg = (Message)
    space.take(msg, null, Long.MAX_VALUE);
```

That's really all there is to simple message passing in space-based programs. To send a message, you wrap its content in an entry—earmarked for a particular receiver—and deposit it into a space. To receive a message, you create a template that matches messages meant for you, and then take the messages from the space.

Of course, we might want to try an example that is a little more interesting. Let's take a look at a more complete example of message passing that is an old stand-by in space-based programming.

5.1.1 Playing Ping-Pong

This example is inspired by the 1970s arcade game Pong, in which two players paddle a “ball” back and forth to each other. In our space-based version, two processes—named “Ping” and “Pong”—are going to paddle an entry back and forth through the space (as shown in Figure 5.1).

Before Ping and Pong can get started, we need to define a `Ball` entry:

```
public class Ball implements Entry {  
    public String receiver;  
  
    public Ball() {}  
  
    public Ball(String receiver) {  
        this.receiver = receiver;  
    }  
}
```

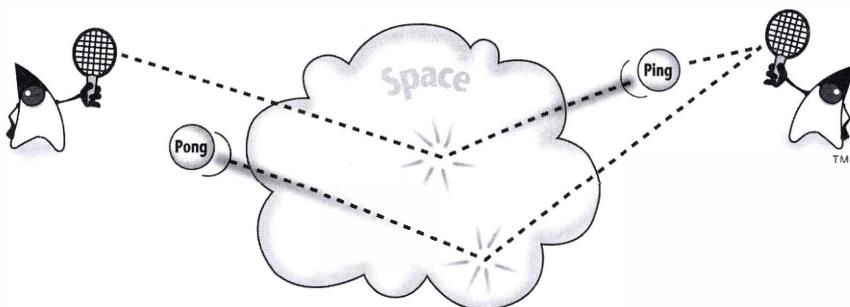


Figure 5.1 Ping and Pong paddling an entry back and forth through the space.

Here we've just defined a `receiver` field; we won't be sending any content back and forth, but just the ball itself.

To create Ping and Pong, we are going to write just one piece of code and use it to instantiate both players, since they do practically the same thing. Here is the code for the class `PingPong`, from which both players are instantiated:

```
public class PingPong {  
    private JavaSpace space;  
    private String myName;  
    private String yourName;  
  
    public PingPong(JavaSpace space, String myName) {  
        this.space = space;  
        this.myName = myName;  
  
        if (myName.equals("Ping")) {  
            yourName = "Pong";  
        } else {  
            yourName = "Ping";  
        }  
    }  
  
    public static void main(String[] args) {  
        String myName = args[0];  
        int rounds = Integer.parseInt(args[1]);  
  
        JavaSpace space = SpaceAccessor.getSpace();  
  
        new PingPong(space, myName).play(rounds);  
    }  
  
    private void play(int rounds) {  
        for (int i = 0; i < rounds; i++) {  
            if (myName.equals("Ping")) {  
                // Ping throws then catches  
                throwBall(yourName, i);  
                Ball b = (Ball)catchBall(myName, i);  
            } else {  
                // Pong catches then throws  
                Ball b = (Ball)catchBall(myName, i);  
                throwBall(yourName, i);  
            }  
        }  
    }  
}
```

```

        }
    }
}

private void throwBall(String receiver, int i) {
    Ball ball = new Ball(receiver);

    try {
        space.write(ball, null, Lease.FOREVER);
    } catch (Exception e) {
        e.printStackTrace();
    }
    System.out.println(myName + " threw " + i);
}

private Ball catchBall(String receiver, int i) {
    Ball template = new Ball(receiver);

    Ball ball = null;
    try {
        ball = (Ball)
            space.take(template, null, Long.MAX_VALUE);
        System.out.println(myName + " caught " + i);
    } catch (Exception e) {
        e.printStackTrace();
    }
    return ball;
}
}

```

The `PingPong` class has three private fields. The `space` field holds the handle to the space in which the ping-pong game takes place. To distinguish between the two players, the field `myName` specifies the player's name ("Ping" or "Pong"), and the field `yourName` specifies the opponent's name.

Let's step through the code. In `main`, we assign the value of the first parameter passed to the application to `myName`, and the value of the second parameter to `rounds`, specifying the number of roundtrips the ball will make between the players. So, to start up a Ping player and a Pong player that will play five rounds (in other words, receive and return the ball five times), we need to run the `PingPong` application twice, once with the command-line parameters "Ping" and 5, and once with the parameters "Pong" and 5.

Next in `main`, we call the `getSpace` method of the `SpaceAccessor` class to obtain a reference to a space, which we assign to the `space` field. Finally we instantiate a `PingPong` object and call its `play` method, passing the number of rounds to play.

The `play` method consists of a `for` loop that loops for `rounds` iterations. Each time through, the loop does the following: if the player is Ping, it throws a ball addressed to Pong and then catches a ball addressed to itself. If the player isn't Ping (it must be Pong) then it does the opposite: Pong first catches a ball addressed to itself and then throws a ball addressed for Ping.

Ball throwing and catching are handled by the `throwBall` and `catchBall` methods. The `throwBall` method instantiates a `Ball` entry, setting its receiver to the name of the player who should catch the ball, and then drops the ball into the space. The `catchBall` method instantiates a `Ball` template, sets its receiver to the name of the player who should catch the ball, and then tries to take a matching entry from the space, waiting if necessary until one shows up.

Let's get a more concrete idea of how players Ping and Pong play the game. If we look at the sequence of `write` and `take` operations that occur in each loop iteration, the Ping player performs the following operations (written in a shorthand form):

```
write("Pong");
take("Ping");
```

while the Pong player performs:

```
take("Pong");
write("Ping");
```

It's important to understand how this back-and-forth proceeds in a synchronized manner. In the scenario above, when a player does a `take`, it must wait for a ball addressed to itself. So, the Pong player must wait until it receives a ball from Ping before it can generate a ball to bounce back to Ping. Similarly, once Ping serves a ball to Pong, it must wait until it receives a ball from Pong before it can loop again and serve another ball. This back-and-forth synchronization using the ball "messages" continues until all the rounds have been played. In effect, the players bounce a single ping-pong message back and forth.

Let the games begin! Here's what the output looks like for several rounds of play:

```
Ping threw 0
Pong caught 0
Pong threw 0
Ping caught 0
Ping threw 1
```

```
Pong caught 1
Pong threw 1
Ping caught 1
Ping threw 2
Pong caught 2
Pong threw 2
Ping caught 2

.
.
.
```

Ping-Pong has remained an old-standby example in space-based programming because it presents a nice example of two processes iteratively passing a message between themselves. Of course, here our message has no real content, but we could obviously include some—for instance, we could add a timestamp to the ball entry, so that a process can report roundtrip times and gauge message passing performance. There are other uses for this simple pattern as well; for instance the ball entry could be used to pass chess moves back and forth between the two players in an online chess game.

5.2 Characteristics of Space-based Communication

So far, we have seen a couple examples of fairly standard message passing between two processes. These space-based “message passing” examples behave like conventional message passing: one process sends out a message, and another process receives it. But if we scratch the surface a bit, we’ll see that the space-based examples illustrate the loosely coupled style of communication we mentioned earlier.

We will fully explore the loosely coupled style of communication shortly—it will be helpful as we begin to explore more sophisticated communication patterns. Let’s first take a look at the alternative: tightly coupled communication.

5.2.1 Tightly Coupled Communication

Conventional message passing—supplied by most low-level communication packages—forces senders and receivers to “rendezvous” in order to exchange a message. A rendezvous requires that the sender and receiver

- ◆ Know each other’s identity (say, a process identifier)
- ◆ Know each other’s precise location (say, a machine’s network address)
- ◆ Exist at the same time

We can call these three properties the *who*, *where*, and *when* of communication. With conventional message passing, all three must be specified explicitly in order for a message to be delivered. This requirement tightly couples components (senders and receivers) in a distributed application: in order for a sender and receiver to communicate successfully, they must know each other's identity and location, and must both be running at the same time. If a participant is late, or is unsure of the other's identity, or ends up at the wrong location, then the message exchange will fail to occur. This tight coupling of components leads to inflexible applications that are particularly difficult to build and debug, and more fragile than necessary.

5.2.2 Loosely Coupled Communication

With tightly coupled systems, the communication questions of “who?”, “where?” and “when?” must be answered with “that process,” “that machine,” “right now.” Space-based communication dramatically eases this requirement and allows the question to be answered with “anyone,” “anywhere,” at “anytime.” Let’s look at what each of these answers means:

Anyone: Senders and receivers don’t need to know each other’s identities, but can instead communicate *anonymously*. In our space-based messaging examples, processes that pick up messages don’t know who sent them. We can go even one step further and omit the `receiver` field from our `Message` entry, meaning that a sender doesn’t direct messages to anyone in particular. Since neither party knows or cares who the other is, this would be truly “anonymous” communication.

Anonymous communication is powerful: it allows you to send a message to “anyone” that knows what to do with the message. You don’t care exactly who gets it, but you assume that some process that knows how to handle the message will eventually pick it up. Conversely, the process that picks up your message doesn’t necessarily care who sent it. Think of a space where you submit requests for computation, say a calculation of your income tax. You drop a “help wanted” request into the space. Eventually, an available worker process with the right know-how picks up the request, performs the calculation, and posts the result to the space. At some point, you’ll retrieve the result. You care only that your job gets done, and you don’t need to direct your task to a specific receiver. Workers just pick up work from the space, not from any particular sender.

Anywhere: Senders and receivers can be located anywhere, as long as they have access to an agreed-upon space for exchanging messages. The sender doesn't need to know the receiver's location—it just drops an entry into the space. Conversely, the receiver picks up the message from the space using associative lookup, and has no need to know where the sender is. Even if the sender and receiver roam from machine to machine, the communication code used by your space-based programs doesn't need to change.

Anytime: With space-based communication, senders and receivers are able to communicate even if they don't exist at the same time, because message entries *persist* in the space. Persistence enables us to build flexible, "time-uncoupled" communication patterns. Suppose the sender exists before the receiver and leaves a message in the space; when the receiver comes along, it can pick up the message without waiting. Even if you start the receiver long after the sender has terminated, the persistent nature of the space ensures that the message is still there, ready to be removed. On the other hand, if the receiver exists before the sender and the receiver tries to perform a `take`, for example, it will have to wait until the sender shows up to drop the message into the space. Even if you start the sender long after the receiver, the receiver is ready and waiting to pick up the message once it arrives.

Of course, not every communication in a space-based program is specified as anyone/anywhere/anytime. You might send a message that is meant for "this receiver"/anywhere/anytime, or "any receiver that knows how to compute pi"/anywhere/"by next Tuesday," and so on.

5.2.3 Benefits of Loose Coupling

By being able to break the tight coupling inherent in other communication approaches, we can build systems that are more flexible, adaptable, and reliable. Throughout the examples in this chapter, you'll see the benefits of loosely coupled communication. Take `PingPong`, for instance. Because the `PingPong` players don't need to know each other's location to communicate, they can exist anywhere, and can even migrate from machine to machine, without changing how they communicate. Because communication is time-uncoupled, the players can continue to play the game whether they're both active at the same time, or taking occasional breaks to do other things.

Besides enabling flexible communication, loose coupling facilitates the composition of large applications by letting programmers easily add components without redesigning the entire application. If we want multiple Ping and Pong players on each side, for instance, we simply start up more Ping and Pong applications, in equal numbers (if the numbers of players are uneven, some will end up waiting

forever for balls that will never come). Each ball will still be bounced back by just one player on a side at a given time, and the code for the players can remain unchanged.

Loose coupling also promotes the reuse of software components. Even if a player is removed from the game through partial failure (a player’s machine crashes or is “cut off” by a network failure), we can plug in a replacement player that follows the same protocol with the space, and the system continues merrily on its way. Thanks to loosely coupled communication, it doesn’t matter who the stand-in is, where it lives, or when it was plugged in—as long as it adheres to the protocol.

5.3 Beyond Message Passing

Now that we’ve gotten a taste of simple space-based communication and have learned a little about loose coupling and its benefits, we are going to explore some more advanced forms of communication through spaces, starting with a distributed data structure known as a *channel*. A channel is often referred to as a “stream,” but given that the Java application environment already has a well-defined notion of a stream, we will stick to the name channel.

Think of a channel as an information conduit that takes a series of messages at one end and delivers them at the other end, while maintaining their ordering. At the input end, a channel may have one or many processes sending messages. At the output end, there may be one or many processes retrieving messages. The channel is capable of exhibiting many different behaviors, depending on the protocols that the senders and receivers of the channel obey. In this chapter you’ll see that the channel distributed data structure is versatile, and can be used to implement a variety of communication patterns.

We’ll start by implementing a basic channel distributed data structure. As you’ll see, the implementation is straightforward and uses techniques you’ve already encountered in previous chapters.

5.4 A Basic Channel

To implement a channel, we use an ordered collection of `Message` entries (similar to the distributed data structure we used in Chapter 3 to implement a distributed array)—one for each message in the channel. Each `Message` entry contains its position number in the channel. For instance, the fourth message to be added to the channel has a position of four (assuming that the message sequence starts from one). Each channel also has a `Tail` entry that marks the end of the channel. For

example, if there are four messages in the channel, the tail entry contains the value four to represent the end position.

Using the channel is straightforward. To send a message into the channel, a “channel writer” appends a new Message entry to the end of the channel by

1. Taking the tail
2. Obtaining the position number of the tail
3. Incrementing the tail and placing it back into the space
4. Setting the new message’s position number to the tail position
5. Placing the message into the space

To read a message in the channel, a “channel reader” simply uses the read method to retrieve the appropriate message. A reader might begin reading at the first message (though the starting point could be anywhere in the channel) and continue through the final one. When the reader reaches the end, it blocks and waits for the next message to be added to the channel.

Let’s see then, piece by piece, how to build a simple channel.

5.4.1 The Channel Message

Channels are made up of Message entries. Here’s our definition of a Message entry:

```
public class Message implements Entry {  
    public String channel;  
    public Integer position;  
    public String content;  
  
    public Message() {}  
  
    public Message(String channel, Integer pos, String msg) {  
        this.channel = channel;  
        this.position = pos;  
        this.content = msg;  
    }  
}
```

Each `Message` entry contains three fields: `channel`, `position`, and `content`. The `channel` field identifies the name of the channel the message belongs to, the `position` field specifies the integer position of the message in the channel, and the `content` field contains the content of our message (in this case just a string).

5.4.2 The Channel Tail

The channel's tail entry marks the "tail" end of a channel: it provides a way to track the size of the channel and marks the position of the last message in the channel. Here is our definition of the `Tail` entry:

```
public class Tail implements Entry {  
    public String channel;  
    public Integer position;  
  
    public Tail() {  
    }  
  
    public Tail(String channel) {  
        this.channel = channel;  
    }  
  
    public Integer getPosition() {  
        return position;  
    }  
  
    public void increment() {  
        position = new Integer(position.intValue() + 1);  
    }  
}
```

The `Tail` entry has two fields: a `channel` field that contains the name of the channel, and a `position` field that contains the position number of the last message in the channel. We also provide two convenience methods: `getPosition`, which returns the tail's position, and `increment`, which increments the position by one.

Now that we have entries for messages and the tail, we're going to spend the next few pages implementing methods that will be used by a `ChannelWriter` class.

5.4.3 Creating a Channel

First, let's define a method that creates a channel. Each channel, even when empty, is associated with exactly one `Tail` entry. To create a new channel, we create a

new Tail entry—with the channel's name and a starting position of zero—and drop the entry into a space. The following method takes a channel name and creates a new channel:

```
private void createChannel(String channel) {
    Tail newTail = new Tail(channel);
    newTail.position = new Integer(0);

    showStatus("Creating new channel " + channel);
    try {
        space.write(newTail, null, Lease.FOREVER);
    } catch (Exception e) {
        showStatus("Error occurred creating the channel.");
        e.printStackTrace();
        return;
    }
    showStatus("Channel " + channel + " created.");
}
```

Here we create a tail entry by passing the channel name to the Tail constructor, and then we set the tail's position number to zero. At that point, we write the tail entry into a space. We'll see shortly where `createChannel` is used.

5.4.4 Appending a Message to a Channel

Once a channel exists, we are free to add messages to it. Let's implement an append method that adds a message to the tail end of a channel: the method will obtain the position number of the tail, increment it, and stamp the new message with the number.

First we'll define a `getMessageNumber` method that obtains a position number for the new message:

```
private Integer getMessageNumber(String channel) {
    try {
        Tail template = new Tail(channel);

        Tail tail = (Tail)
            space.take(template, null, 2000);
        if (tail == null) {
            createChannel(channel);
            tail = (Tail)
                space.take(template, null, Long.MAX_VALUE);
```

```

        }
        tail.increment();
        space.write(tail, null, Lease.FOREVER);

        return tail.getPosition();
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}

```

Stepping through the method, we first create a `Tail` template, specifying a channel name and leaving the position number as a wildcard. We then call `take` to find a tail entry for that channel. Note that we specify a time-out of 2000 milliseconds in the call to `take`, so if we haven't found the tail for this channel within two seconds, the call returns the value `null`.

What happens next is important. If our attempt to retrieve the tail for the channel fails, then we assume that the channel doesn't exist; in this case, we call `createChannel` to create the channel and again try to take the tail entry from the space (this time waiting indefinitely, since we know for certain the tail exists). In general, we can't really count on this technique as a sure way to determine the existence of a channel. For instance, starvation (discussed in Chapter 4) could keep us from retrieving a tail entry, or it could take us more than two seconds to retrieve a tail from a remote space, due to very slow network connections. However, within the context of running the examples in the book, we can be pretty sure that if we don't see the tail entry within two seconds, it doesn't exist. The benefit here is that, by adding this piece of code, we don't have to create a separate applet to create channels for us *a priori* (although we will build a "channel creator" applet later in the chapter).

Once we've retrieved the tail entry for the channel, we increment the tail and write it back to the space. We return the tail's new position number to the caller, to be used as the position number for the new message. Since the tail entry for a channel starts out at position zero, the first time we call `getMessageNumber` for a channel, the position one will be returned to use for the first message.

Now let's use the `getMessageNumber` method to implement `append`; here is the code:

```

private void append(String channel, String msg) {
    Integer messageNum = getMessageNumber(channel);

    Message message = new Message(channel, messageNum, msg);
    showStatus("Sending message " + messageNum +
               " to channel " + channel + "...");
}

```

```
try {
    space.write(message, null, Lease.FOREVER);
} catch (Exception e) {
    showStatus("An error occurred writing to the channel.");
    e.printStackTrace();
    return;
}
showStatus("Sending message " + messageNum +
           " to channel " + channel + "... Done.");
}
```

The append method takes a channel name and a message in the form of a `String`. We first call `getMessageNumber`, passing it the channel name, to increment the tail and obtain a position number for the new message in the channel. Next, we create a new `Message` entry with the given channel name, position number, and message string, and write it into the space.

We now have most of the foundation in place to create a channel reader and channel writer. Let's create the writer first.

5.4.5 Implementing a Channel Writer

Using the methods and entries just defined, we are going to develop a channel writer applet that can send messages to any channel. To make the writer easier to experiment with, let's first develop a user interface using the Java AWT package.

The channel writer's user interface is shown in Figure 5.2. You'll see a text field labelled "Channel" where you can type the name of the channel to which

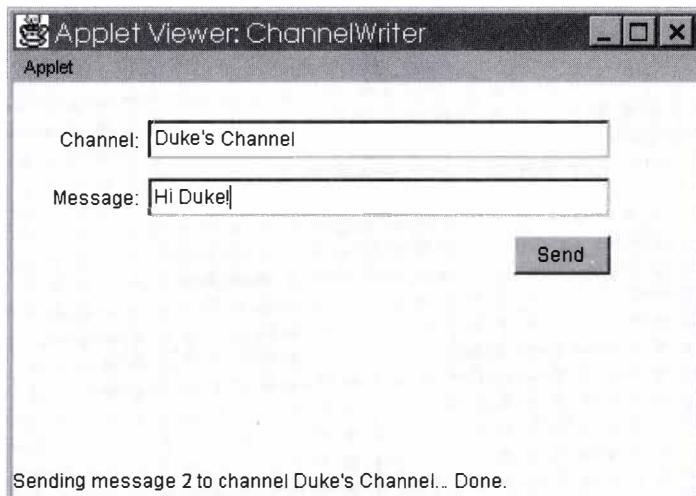


Figure 5.2 The ChannelWriter user interface.

messages are sent. A second text field labeled “Message” is where you enter messages. We also make use of the applet’s status area at the bottom to display status information. When a user presses “Enter” in the message field or clicks on the “Send” button, the channel writer appends the supplied message to the channel.

While we won’t exhaustively explain the implementation of this simple user interface, we will spend a little time studying a skeleton of its code, since its interaction model directly affects how the channel writer calls our channel methods.

The skeleton of our `ChannelWriter` applet is shown below:

```
public class ChannelWriter extends Applet
    implements ActionListener
{
    private JavaSpace space;
    private String channel;

    //... variables for user interface components go here

    public void init() {
        space = SpaceAccessor.getSpace();

        //... creation of user interface components

        channelTextField = new TextField();
        messageTextField = new TextField();
        sendButton = new java.awt.Button();

        messageTextField.addActionListener(this);
        sendButton.addActionListener(this);
        showStatus("Type message & hit Enter or press Send");
    }

    // "Enter" pressed in Message field
    // or "Send" button clicked
    public void actionPerformed(ActionEvent event) {
        String channel = channelTextField.getText();
        String message = messageTextField.getText();

        if (message.equals("") || channel.equals("")) {
            showStatus("Enter both channel & message fields.");
            return;
        }
    }
}
```

```
        }
        append(channel, message);
        messageTextField.setText("");
    }

//... createChannel, append, getMessageNumber
}
```

The `init` method of the applet first calls the `SpaceAccessor`'s `getSpace` method to obtain a reference to a space, and assigns it to the field `space`. From there, `init` creates a number of interface components (labels, text fields, and the button) and adds them to the applet. Three of these are important to point out: `channelTextField`, `messageTextField`, and `sendButton`. The `channelTextField` holds the channel name, the `messageTextField` holds the message, and the `sendButton` is used to send the message (pressing “Enter” in the `messageTextField` will also send the message). Finally, the `init` method has the following:

```
messageTextField.addActionListener(this);
sendButton.addActionListener(this);
```

This code adds action listeners for the message text field and the “Send” button. If the button is clicked or “Enter” is pressed in the message field, then the listener (which in this case is `this`, meaning the channel writer itself) is notified. To refresh your memory of the AWT package, that notification occurs by calling the applet’s `actionPerformed` method.

In the `actionPerformed` method, we first extract the message and channel strings out of the appropriate text fields and check to make sure that a channel and message have been entered. If not, we print an error message in the status bar; if both were entered we call `append` to send the message and then clear the message field.

5.4.6 Implementing a Channel Reader

Now that we have a channel writer, let’s turn our attention to building a channel reader. Our channel reader also has a user interface, which is shown in Figure 5.3. In the interface you’ll find a text field labeled “Channel” where you can type in the name of the channel you want to “listen” to. Whenever you press “Enter” in the channel field or click on the “Read” button, the Channel Reader begins to read the messages in the channel. As messages are read, they are displayed in succession in the text area labeled “Messages.”

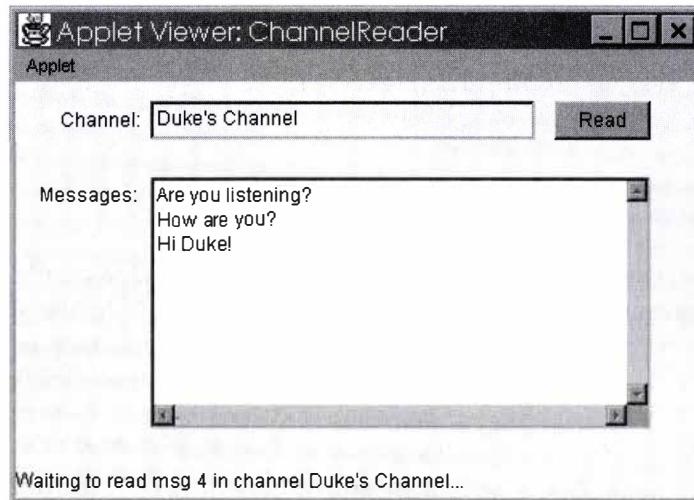


Figure 5.3 The ChannelReader user interface.

Let's take a look at the `ChannelReader` class:

```
public class ChannelReader extends Applet
    implements ActionListener, Runnable
{
    private JavaSpace space;
    private String channel;
    private Thread channelReader;

    //... variables for user interface components go here

    public void init() {
        space = SpaceAccessor.getSpace();

        //... creation of user interface components

        channelTextField = new TextField();
        readButton = new Button();
        channelOutput = new TextArea();

        channelTextField.addActionListener(this);
        readButton.addActionListener(this);
    }
}
```

```
// "Enter" pressed in channel field or "Read" button clicked
public void actionPerformed(ActionEvent event) {
    channel = channelTextField.getText();
    if (channel.equals("")) {
        showStatus("Please enter a channel name.");
        return;
    }

    channelTextField.setEditable(false);

    if (channelReader == null) {
        channelReader = new Thread(this);
        channelReader.start();
    }
}

public void run() {
    String newline = System.getProperty("line.separator");
    int position = 1;
    Message msg = null;

    Message template = new Message();
    template.channel = channel;

    while (true) {
        showStatus("Waiting to read msg " + position +
                  " in channel " + channel + "...");
        template.position = new Integer(position++);
        try {
            msg = (Message)
                space.read(template, null, Long.MAX_VALUE);
        } catch (Exception e) {
            e.printStackTrace();
        }

        String msgText = msg.content;
        channelOutput.append(msgText + newline);
    }
}
```

After obtaining a reference to a space, the `init` method creates the interface components, including `channelTextField` where we type the channel name, a `readButton`, and the `channelOutput` text area where messages will be displayed. We then make the applet a listener for action events on the `channelTextField` and the `readButton`.

The applet works as follows: when you enter a channel name and press “Enter” or click on the “Read” button, the `actionPerformed` method is called. In this method we first check to make sure you entered a channel name. If not, we print an error message in the status area and return. If a channel name was entered, then we first set the `channelTextField` so that it is no longer editable. This means that once we enter a channel name, the channel reader reads only from that channel and cannot be switched to another. Next, we create and start a new thread to continually read messages from the channel (without tying up the user interface’s thread).

Let’s step through the `run` method that the thread executes: we declare a local integer variable called `position`, which is used to track the message we’re currently reading in the channel. The `position` is initialized to one, so that we’ll start off by reading the first message. We also create a message template and set its `channel` field to the name we specified in the user interface. Then we enter a `while` loop, where we continually set the template’s `position` number (leaving the `content` field as a wildcard), call `read` to retrieve a matching message, and display its content in the `channelOutput` text area. If there is no message matching the template, then `read` blocks until one arrives in the space. Each time through the loop, the `position` is incremented by one. So, the loop starts at position one, reads through all the messages in the channel, and waits for the next message to arrive.

5.4.7 Demonstrating the Channel Writer and Reader

Now that we have a channel writer and a channel reader, let’s give them a try. We’ll begin by starting a channel writer. Enter a channel name such as “Duke’s Channel” and a message such as “Hi Duke!” Now press the “Send” button. You’ll notice that the message text entry box clears, the status message displays “`Sending message 1 to channel Duke’s Channel...`” and then there is a pause as `append` is called. Recall that `append` calls `getMessageNumber`, which attempts to take the tail entry, and if it isn’t found in two seconds creates one. After two seconds you’ll see the message “`Sending message 1 to channel Duke’s Channel... Done.`” appear in the status area to indicate that the message has been sent.

Let’s now start a channel reader. You should type “Duke’s Channel” again as the channel name and then click on the “Read” button. You should see “Hi

Duke!” appear in the channel output area. Now type another message into the channel writer and hit the “Send” button. You should see the message show up in the reader’s channel output area not long after.

Now let’s get more adventurous. Start another channel writer, fill in the channel name with “Duke’s Channel” and send a few messages, and you should see these appear in the channel reader also. You can still send messages from your original channel writer applet as well. Last, start a second channel reader, and you’ll see all the messages you previously sent from both channel writers appear in order in the output area. Sending another message from either channel writer will cause the message to appear in both channel readers’ output areas.

This example demonstrates a communication pattern that resembles a conventional broadcast that supports one or more writers and one or more readers. However, our channel example really goes far beyond conventional broadcast (in much the same way that space-based message passing is more flexible than typical message passing). The added flexibility is once again due to the loosely coupled nature of space-based programming. In conventional broadcast libraries, the sender and receiver of broadcast messages are tightly coupled: if a process isn’t around at the time of the broadcast, it will miss the message. With space-based broadcasting, since the messages exist as persistent entries in the space, they can be read at any time at any speed. If a reader retrieves messages faster than the channel writer generates them, it will stop reading at the end of the channel and wait until the next message is delivered. If the channel writer generates messages faster than a reader can keep up, the reader will continue reading the messages at its own pace. A reader can even come along long after the channel writer has terminated and scan the channel from the beginning to get caught up.

5.5 Building a Chat Application with Channels

In this section we are going to take the `ChannelWriter` and `ChannelReader` applets one step further and create a simple “chat” applet by combining their functionality. Doing so requires only a few changes to our existing code.

5.5.1 The Graphical User Interface

We are going to call our new applet `ChannelChat`. The `ChannelChat` applet’s user interface, shown in Figure 5.4, is a combination of the `ChannelReader` and `ChannelWriter` user interfaces. `ChannelChat` provides text fields for entering a channel and a message, a “Send” button for adding messages to the channel, and a “Read” button to start reading the channel. It also provides an output area where channel messages are displayed.

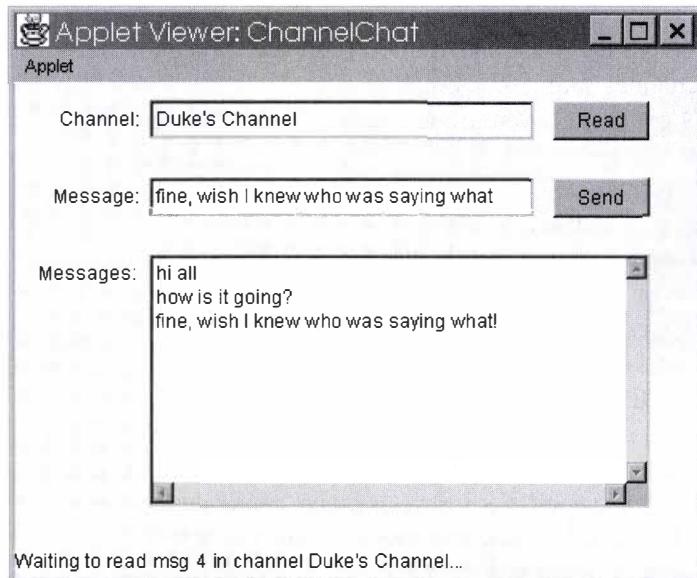


Figure 5.4 The ChannelChat interface: a hybrid of ChannelWriter and ChannelReader.

5.5.2 Combining Channel Writing and Reading

For the chat application, we reuse our previous Message and Tail classes. Here is the code for the ChannelChat applet, which combines code from our previous ChannelWriter and ChannelReader applets:

```
public class ChannelChat extends Applet
    implements ActionListener, Runnable
{
    private JavaSpace space;
    private String channel;
    private Thread channelReader;

    //... variables for user interface components go here

    public void init() {
        space = SpaceAccessor.getSpace();

        //... creation of user interface components
    }
```

```
// "Enter" pressed in a field, or a button was clicked
public void actionPerformed(ActionEvent event) {
    channel = channelTextField.getText();
    String message = messageTextField.getText();
    Object object = event.getSource();

    if ((object == messageTextField) ||
        (object == sendButton)) {
        // "Enter" pressed in msg field or "Send" clicked
        if (message.equals("") || channel.equals("")){
            showStatus("Enter both channel & msg fields.");
            return;
        }
        channelTextField.setEditable(false);
        messageTextField.setText("");
        append(channel, message);
    } else {
        // "Enter" pressed in channel field or "Read" clicked
        if (channel.equals("")) {
            showStatus("Enter a channel name.");
            return;
        }
        channelTextField.setEditable(false);

        if (channelReader == null) {
            channelReader = new Thread(this);
            channelReader.start();
        }
    }
}

private void createChannel(String channel) {
    //... code from ChannelWriter applet
}

private void append(String channel, String msg) {
    //... code from ChannelWriter applet
}
```

```
private Integer getMessageNumber(String channel) {
    //... code from ChannelWriter applet
}

public void run() {
    //... code from ChannelReader applet
}
}
```

In this applet, the `append`, `getMessageNumber`, and `createChannel` methods are taken directly from our previous `ChannelWriter` applet, while `run` is reused from our `ChannelReader` applet. The `init` method is based on the one in `ChannelReader`, except that there are additional user interface components for sending messages.

The only substantially reworked method is `actionPerformed`. The method first determines which component (“Send” button or “Read” button) generated the action event. If “Send” was clicked (or “Enter” was pressed in the message text field), `actionPerformed` calls `append` to send the message out. On the other hand, if “Read” was clicked (or “Enter” was pressed in the channel text field), `actionPerformed` starts a new listener thread that executes the `run` method (just as it did in `ChannelReader`).

By combining channel writing and reading functionality into a single applet, we’ve just built a simple multiuser chat application. You should now be able to start up several copies of `ChannelChat` and carry on a conversation.

We could certainly improve our application by adding features (such as being able to see who sent a message) and making it robust in the face of failure (as you’ll explore in a Chapter 9 exercise). However, even in its current form, we’ve managed to build a fairly sophisticated collaborative application on the basis of a simple distributed data structure.

5.6 A Consumer Channel

We’re now going to explore a variant of the channel, in which processes consume messages as they retrieve them. We call this kind of channel a *consumer channel*. One common use of a consumer channel is to provide a task queue. With a task queue, writers add tasks (requests for services) into the channel and worker processes remove and process the tasks (computing them or providing some other service). Since each task only needs to be processed once, a consumer channel provides a natural way to store the requests. At the same time, it allows requests to be processed in a first-come, first-served manner.

5.6.1 Implementing a Pager Service

Rather than implement a task queue, we are going to explore consumer channels with an example that is related to our previous chat channel—a simple “pager” like the one shown in Figure 5.5. Here is how the pager service works: Each customer is assigned a unique pager account name, such as “duke@pager.sun.com,” which is created by an administrative application. As messages arrive for a particular pager account, they are added to a consumer channel that represents that account. A customer uses the pager to read and display the messages, which are removed from the consumer channel.

To support the pager service, we are going to implement three different applets: a `ChannelCreator` that creates pager accounts, a `PageWriter` that sends messages to a pager, and a `PageTaker` that removes messages from the pager channel.

The interface for the `ChannelCreator` is shown in Figure 5.6. To use the applet we enter the name of a pager account and click on “Create,” which causes a channel for the pager to be created in the space. We will get to the implementation details shortly.

Once your pager is in service, people can send you messages using the `PageWriter` applet, shown in Figure 5.7. When a user presses “Send,” the specified message is appended to the end of the pager’s channel.



Figure 5.5 A pager that uses the JavaSpaces technology.

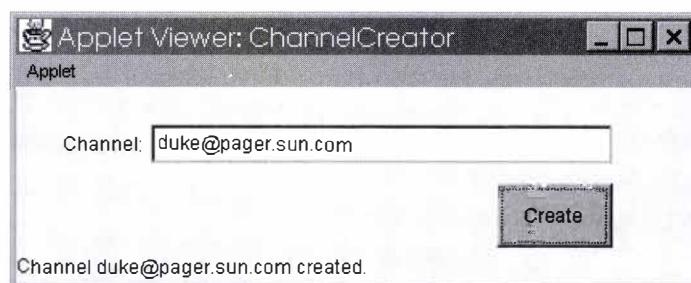


Figure 5.6 An interface for creating a pager's communication channel.

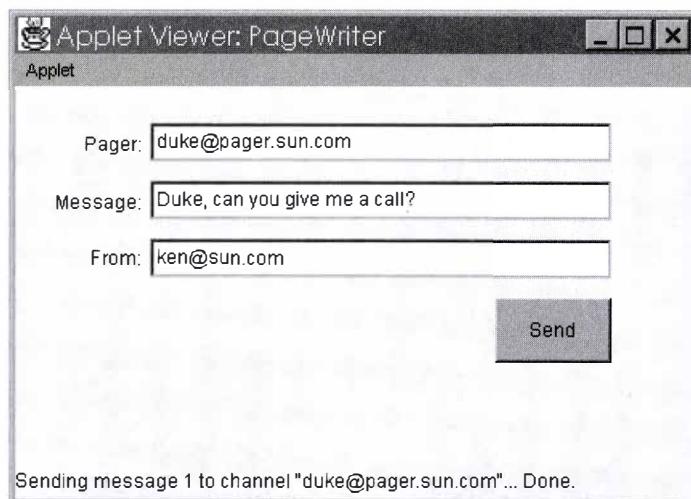


Figure 5.7 The PageWriter applet sending a message to duke@pager.sun.com.

Whenever you want to retrieve your messages, you just click on your pager, shown in Figure 5.5. When clicked, the pager removes a message (the oldest one in the channel) and displays the sender, the time the message was sent, and the message itself in the display panel. You can keep clicking as much as you'd like; if there are messages in the channel they will be removed and displayed in order, but if there are no messages the pager will display "No messages."

Now that we have a feel for how the application works, let's focus on the implementation details.

5.6.2 The Pager Message Entry

The PagerMessage entry is similar to the messages we've seen before. We use `pageNumber` instead of the `channel` field, and we've added `from` and `timestamp` fields to our entry:

```
public class PagerMessage implements Entry {  
    public String pageNumber;      // recipient of the page  
    public Integer position;      // position # of msg in channel  
    public String content;        // message content  
    public String from;           // who sent the page  
    public Date timestamp;        // when it was sent  
  
    public PagerMessage() {       // the no-arg constructor  
    }  
  
    public PagerMessage(String pageNumber) {  
        this.pageNumber = pageNumber;  
    }  
  
    public PagerMessage(String pageNumber, Integer position) {  
        this.pageNumber = pageNumber;  
        this.position = position;  
    }  
  
    public PagerMessage(String pageNumber, Integer position,  
                        String content, String from)  
    {  
        this.pageNumber = pageNumber;  
        this.position = position;  
        this.content = content;  
        this.from = from;  
        timestamp = new Date();  
    }  
}
```

5.6.3 Tracking the Start and End of a Channel

As in previous examples, we'll continue to use a Tail entry to keep track of the tail end of a channel. Recall that the tail holds the position number of the last message in the channel, and gets incremented before we append a new message.

This time we'll need to keep track of the front or "head" end of the channel as well, since the pager removes the message at the front of the channel whenever it is clicked. A head entry holds the position of the first message in the channel. Every time a message is removed, the head's position number is incremented to point to the next message.

Messages are added to the tail end of the channel and taken away from the head end. If messages are removed faster than new ones are added, at some point there will be just one message in the channel, with both head and tail pointing to it. Now if you click the pager, the message is removed and head is incremented; head's position number is now greater than tail's, which indicates an empty channel. As long as tail is greater than or equal to head, we know there are messages in the channel.

5.6.4 The Index Entry

We could keep our old Tail entry and define a new Head entry. However, since head and tail provide similar functions, we'll define a new Index entry (based on the Tail entry) that we'll conveniently use for both:

```
public class Index implements Entry {  
    public String type;          // head or tail  
    public String channel;  
    public Integer position;  
  
    public Index() {  
    }  
  
    public Index(String type, String channel) {  
        this.type = type;  
        this.channel = channel;  
    }  
  
    public Index(String type, String channel, Integer position) {  
        this.type = type;  
        this.channel = channel;  
        this.position = position;  
    }  
  
    public Integer getPosition() {  
        return position;  
    }
```

```

public void increment() {
    position = new Integer(position.intValue() + 1);
}
}
}

```

The `Index` entry is just like our previous `Tail` entry, except that we have added a `String` field named `type`, to identify the type of index (whether the entry points to the “head” or “tail” of the channel). The `Index` entry works like this: Say a channel holds four messages, numbered 1 through 4; the head holds a position number of 1, while the tail holds a position number of 4.

5.6.5 Creating a Consumer Channel

Now let’s see how to create a communication channel for a pager. An administrator runs a `ChannelCreator` applet and sees the user interface shown in Figure 5.6. Here’s the code for the `ChannelCreator`:

```

public class ChannelCreator extends Applet
    implements ActionListener
{
    private JavaSpace space;

    //... variables for user interface components go here

    public void init() {
        space = SpaceAccessor.getSpace();

        //... creation of user interface components
    }

    // "Create" button pressed
    public void actionPerformed(ActionEvent event) {
        String channel = channelTextField.getText();

        if (channel.equals("")) {
            showStatus("Enter channel field.");
            return;
        }
        createChannel(channel);
    }
}

```

```

private void createChannel(String channel) {
    Index head =
        new Index("head", channel, new Integer(1));
    Index tail =
        new Index("tail", channel, new Integer(0));

    showStatus("Creating new channel " + channel);
    try {
        space.write(head, null, Lease.FOREVER);
        space.write(tail, null, Lease.FOREVER);
    } catch (Exception e) {
        showStatus("Error creating the channel.");
        e.printStackTrace();
        return;
    }
    showStatus("Channel " + channel + " created.");
}
}

```

Whenever the administrator presses the “Create” button in the Channel-Creator interface, the applet’s `actionPerformed` method is called. This method obtains the channel name from the text field (making sure it isn’t empty) and calls the `createChannel` method. The job of `createChannel` is simple: it sets up two `Index` entries to mark the head and tail of the channel and writes them to the space. Since a new channel is initially empty, `createChannel` sets up the tail entry with an index of 0 and the head entry with an index of 1 (recall that the tail being less than the head indicates an empty channel).

5.6.6 Sending Messages to the Channel

To send messages to a pager, you run a `PageWriter` applet, whose user interface is shown in Figure 5.7. To send a message you fill in the “Pager,” “Message,” and “From” fields and click on the “Send” button, which causes the `PageWriter` to append the message to the appropriate pager channel.

The `PageWriter` is almost identical to our previous `ChannelWriter` applet. A small difference is that `PageWriter` must deal with the additional “From” label and text field. And unlike `ChannelWriter`, this applet doesn’t include a `createChannel` method but instead leaves that functionality to the `ChannelCreator` applet we’ve already described. Here’s the code skeleton for the writer:

```
public class PageWriter extends Applet
    implements ActionListener
{
    //... variables for user interface components go here
    private JavaSpace space;

    public void init() {
        space = SpaceAccessor.getSpace();

        //... creation of user interface components
    }

    // "Enter" pressed in message field or "Send" button clicked
    public void actionPerformed(ActionEvent event) {
        //... obtain strings from text fields here
        append(channel, message, from);
    }

    private void append(String channel, String msg, String from) {
        showStatus("Looking for channel " + channel + "...");
        Integer messageNum = getMessageNumber(channel);
        PagerMessage message =
            new PagerMessage(channel, messageNum, msg, from);

        showStatus("Sending message " + messageNum +
                  " to channel \\" + channel + "\\...");
        try {
            space.write(message, null, Lease.FOREVER);
        } catch (Exception e) {
            showStatus("Error occurred writing to the channel.");
            e.printStackTrace();
            return;
        }
        showStatus("Sending message " + messageNum +
                  " to channel \\" + channel + "\\... Done.");
    }

    private Integer getMessageNumber(String channel) {
        try {
            Index template = new Index("tail", channel);
            Index tail = (Index)
                space.take(template, null, Long.MAX_VALUE);
        }
    }
}
```

```
        tail.increment();
        space.write(tail, null, Lease.FOREVER);
        return tail.getPosition();
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}
}
```

The other difference you'll notice from previous examples is that `get(MessageNumber)` now works with `Index` entries, rather than `Tail` entries, and doesn't call a `createChannel` method. The method still behaves similarly though: it waits as long as necessary to find an entry in the space that matches a tail entry for this channel, increments the tail, and writes it back to the space. For instance, when the applet sends messages into a new, empty channel, `get(MessageNumber)` updates the tail's position number from zero to one (and the first message is written into the channel at position one).

5.6.7 Reading and Removing Messages from the Channel

Now we know how messages get appended to a pager's channel. But how do they get removed from the channel? The pager shown in Figure 5.5 reads and removes messages by using the `PageTaker` applet defined as follows:

```
public class PageTaker extends Applet
    implements MouseListener, Runnable
{
    //... variables for user interface components go here

    private JavaSpace space;
    private String channel;
    private Thread consumer;

    public void init() {
        space = SpaceAccessor.getSpace();
        channel = getParameter("myaddress");

        //... creation of user interface components
        this.addMouseListener(this);
    }
}
```

```

// mouse was pressed & released in the interface
public void mouseReleased(MouseEvent e) {
    if (consumer == null) {
        display("Getting message...");
        consumer = new Thread(this);
        consumer.start();
    } else {
        display("Please wait, retrieving message...");
    }
}

public void run() {
    displayNextMessage(channel);
    consumer = null;
}

//... other methods go here (details will follow):
//... displayNextMessage - method to display next pager msg
//... removeMessage - method that takes msg from channel
//... removeIndex - method that takes a channel Index
//... readIndex - method that reads a channel Index
//... writeIndex - method that writes a channel Index

//... other methods go here (see book source code)
//... additional methods required of MouseListeners
//... paint - paints pager on the screen
//... display - displays message info in pager
}

```

The applet behaves as follows: The `init` method gains access to the space, gets the pager's name from an applet parameter, and performs the setup that results in the user interface shown in Figure 5.5. The applet also registers to listen to any mouse events that occur in its interface.

Whenever the mouse is clicked and released anywhere in the pager interface, `mouseReleased` gets called. This method checks to see if a "message consumer" thread already exists. If `mouseReleased` finds the consumer thread to be non-null, it will inform the user that there's already a process checking for new messages, otherwise it will create a new consumer thread that executes `run`. The `run` method calls `displayNextMessage` to read, remove, and display the next message in the pager's channel. Then `run` finishes by setting the consumer back to `null`, and the consumer thread dies as the method returns. In other words, the consumer thread doesn't run continually (`run` doesn't loop), but is instead recreated every time the user clicks on the pager to retrieve a message.

Now let's take a look at the `displayNextMessage` method, which takes a channel name as a parameter, and tries to read and remove the next message in the channel:

```
private void displayNextMessage(String channel) {
    Index tail = readIndex("tail", channel);
    Index head = removeIndex("head", channel);

    if (tail.position.intValue() < head.position.intValue()) {
        display("No messages.");
        writeIndex(head);
        return;
    }

    // get the next message & increment the head:
    PagerMessage message = removeMessage(channel, head.position);
    head.increment();
    writeIndex(head);

    if (message != null) {
        display(message.content, message.from,
               message.timestamp);
    } else {
        display("Communication error.");
    }
}
```

The method first reads the tail index for this channel (but doesn't remove it), then removes the head index, and compares the two. If the tail's position number is less than the head's, the channel is empty and there's nothing to display. In this case, the method prints "No messages." to the pager's screen, writes the head index back to the space unchanged, and returns. On the other hand, if there are messages in the channel, `displayNextMessage` calls `removeMessage` to remove the message at the head, increments the head `Index` entry, and writes the head back to the space. Then the message is displayed on the pager's screen.

Here is the code for `removeMessage`:

```
private PagerMessage removeMessage(String channel,
                                   Integer position)
{
    PagerMessage template = new PagerMessage(channel, position);
    PagerMessage message = null;
```

```

try {
    message = (PagerMessage)
        space.take(template, null, Long.MAX_VALUE);
    return message;
} catch (Exception e) {
    e.printStackTrace();
    return null;
}
}

```

Here we simply take a `PagerMessage` entry, with the given channel and position, from the space and return the entry.

You've no doubt noticed that `displayNextMessage` makes use of the methods `removeIndex`, `readIndex`, and `writeIndex`. These are simple methods that do the work of taking, reading, and writing Index entries. You can find their definitions in the complete source code.

5.6.8 Demonstrating the Pager

Now let's put our pager into service. Start up the `ChannelCreator` applet and create a pager channel with any name you like. Then start up a `PageWriter` applet and enter the same channel name you just created. Send a few messages to the channel. Now start up the `PageTaker` applet and click on the forward button (actually anywhere in the interface will do). You should see the first message appear. Click a few more times to retrieve all the messages you sent. After retrieving them all you should see "No messages." Now type a few more messages into the writer, and then click on the pager repeatedly and once again they should appear. You can also start up a few more channel writers and append messages to the pager's channel.

5.7 Bounded Channels

Our assumption in the channel examples has been that the space itself can store as much data in a channel as we can throw at it. Obviously that approach might be infeasible, or at least unfair, to other users of the space. In many cases, a better approach is to place an upper bound on the number of messages a channel can hold and have writers obey those bounds. For instance, in our pager example, the pager service could place a limit on the number of messages a pager channel can hold.

Before adding bounded channels to our pager, let's step back and get a clearer picture of the general problem. We can view our channels as conduits from pro-

ducers of information to consumers (or readers) of information. Producers need to add elements, while consumers need to remove them. If the channel is full, the producer must wait until there is room for a new element. If the channel is empty, the consumer must wait until new elements arrive.

This “producer/consumer problem”—creating a system where a producer and consumer share a bounded (fixed-size) channel—proves to be difficult to solve in concurrent systems, because a number of race conditions can occur that cause incorrect behavior. Like the dining philosophers problem, which we covered in Chapter 4, the producer/consumer problem is often used to show how elegant and expressive a concurrent programming language or system is. With space-based programming, we can easily implement a bounded channel for managing producers and consumers that avoids the race conditions. In fact, we can build one by making a handful of changes to the consumer channel we built for our pager example. Let’s see how.

5.7.1 The Status Entry

We’ve already seen how head and tail entries mark the beginning and end of a channel. To implement a fixed-size channel, we are going to introduce a “status” entry that keeps track of the maximum size of a channel, the current number of elements in it, and whether it’s currently empty or full. Here’s what our new Status entry looks like:

```
public class Status implements Entry {  
    public String channel;  
    public Integer maxSize;  
    public Integer curSize;  
    public Boolean empty;  
    public Boolean full;  
  
    public Status() {  
    }  
  
    public Status(String channel, Integer maxSize) {  
        this.channel = channel;  
        this.maxSize = maxSize;  
        curSize = new Integer(0);  
        empty = new Boolean(true);  
        full = new Boolean(this.curSize == this.maxSize);  
    }  
}
```



Figure 5.8 The ChannelCreator applet lets an administrator specify the maximum size of the channel.

The entry has several fields that are important to the status of the channel. The `channel` field associates the entry with a particular channel (as we've seen before). The next field, `maxSize`, holds the upper limit on the number of messages that the channel can hold, which is in effect the “buffer size” of the channel. The `curSize` field holds the current number of messages in the channel. Next there are two Boolean fields: `empty`, which is `true` when the channel has no messages, and `full`, which is `true` when the channel has `maxSize` messages.

You may be wondering why the `empty` and `full` fields are needed, given that the entry has a field `curSize`. The reason stems from the fact that spaces only handle exact matching on fields, and cannot do inexact (other than wildcard) or range matching. For instance, if we wish to retrieve a `Status` entry representing a non-empty channel, we can't construct a template that will match entries where `curSize` is greater than zero. Instead, we construct a template with the value `false` for the `empty` field. Likewise, if we want to retrieve a `Status` entry that represents a channel that's not full, we can't construct a template that will match entries for which `curSize` is less than `maxSize`. Instead, we construct a template that has its `full` field set to `false`.

5.7.2 Channel Creation Revisited

We've revised the `ChannelCreator` applet so that it provides a text field where the administrator can enter the maximum channel size, as shown in Figure 5.8.

Whenever `ChannelCreator` creates a channel, in addition to creating head and tail entries, it also sets up an initial status entry by passing the `Status` constructor a channel name and size limit:

```
    Status status = new Status(channel, maxSize);
```

In the two-argument `Status` constructor shown in Section 5.7.1, the `channel` field is set to the channel name, the `maxSize` field is set to the maximum size of the channel, the `empty` field is set to true (since the channel is empty at creation time), and the `full` field is set to true only if the size limit of the channel is zero, else it is set to false. After constructing the status entry, the `ChannelCreator` writes it into the space (refer to the complete source code for the class definition).

5.7.3 Writing to a Bounded Channel

In our nonbounded consumer channel, the writer was free to append new entries to the channel whenever it wanted. Now, before appending, the writer first needs to ensure that there is room available in the channel. If there isn't, the writer will wait until a spot frees up. Here is the revised `append` method of `PageWriter`:

```
private void append(String channel, String msg, String from) {
    showStatus("Looking for empty spot in channel " +
               channel + "...");

    // wait till the channel is not full:
    Status statusTemplate = new Status();
    statusTemplate.channel = channel;
    statusTemplate.full = new Boolean(false);
    Status statusResult = null;
    try {
        statusResult = (Status)
            space.take(statusTemplate, null, Long.MAX_VALUE);
    } catch (Exception e) {
        showStatus(
            "Error occurred looking for status entry.");
        e.printStackTrace();
        return;
    }

    //... same as before: get the next message number,
    //... construct message, & write it into the space

    // update the status entry & write it back to the space:
    int curSize = statusResult.curSize.intValue();
```

```

        curSize++;
        int maxSize = statusResult.maxSize.intValue();
        statusResult.curSize = new Integer(curSize);
        statusResult.empty = new Boolean(false);
        statusResult.full = new Boolean(curSize == maxSize);
        try {
            space.write(statusResult, null, Lease.FOREVER);
        } catch (Exception e) {
            showStatus("Error occurred writing to the channel.");
            e.printStackTrace();
            return;
        }

        showStatus("Sending message " + messageNum +
            " to channel \\" + channel + "\\... Done.");
    }
}

```

The append method first fills in a `Status` template with the name of the channel and `full` set to `false`, and then it waits until it is able to take a matching status entry from the space. In other words, the process will wait until the channel is no longer full before the call to `take` completes. At that point, `append` is free to construct the message and write it to the channel. We can be sure the channel won't fill up before the message is added, because the process is holding the `Status` entry, which any other writer needs before appending.

Once a message has been added, `append` updates the `Status` entry: It increments the size of the channel, sets `empty` to `false` (since it just added a message), and sets `full` to `true` if the size is now at the maximum, otherwise sets it to `false`. Then `append` writes the updated entry back to the space.

There is one other revision we need to make, to account for the possibility a writer could wait indefinitely for a status entry that indicates a nonfull channel. If the writer is blocked in a waiting mode, the process won't be able to continue repainting the user interface as it should. To address this problem, we revise `PageWriter` so that every call to `append` runs within its own thread, as follows:

```

public class PageWriter extends Applet
    implements ActionListener, Runnable
{
    private Thread writer;
    //...

    // "Enter" hit in message field or "Send" button pressed
    public void actionPerformed(ActionEvent event) {
        channel = channelTextField.getText();

```

```

        message = messageTextField.getText();
        from = fromTextField.getText();
        //...
        if (writer == null) {
            writer = new Thread(this);
            writer.start();
        } else {
            return;
        }
    }

    public void run() {
        append(channel, message, from);
        writer = null;
    }

    //...
}

```

Here, each time the “Send” button is pressed, we start a new thread to append the message (if a thread isn’t already running).

5.7.4 Taking from a Bounded Channel

We also need to revise our `PageTaker` code to wait until the channel is nonempty, and to update the status entry as it removes messages. Recall that in our previous `PageTaker` code (Section 5.6.7), the `run` method called `displayNextMessage` to display the next message in the channel. If the channel was empty (determined by comparing its head and tail values), “No messages.” was printed. Every mouse click on the interface resulted in the next message being retrieved (if there was one).

Here we’ll revise our `PageTaker` to demonstrate blocking on empty channels; it will continually remove and display messages until the channel is empty, and then wait for a new message to arrive. To do this, the applet’s `run` method calls a `displayMessages` method that loops: Each time through the loop, it waits for the status entry to be nonempty and then removes one message. Let’s take a closer look:

```

public class PageTaker extends Applet
    implements MouseListener, Runnable
{
    //...

```

```
public void run() {
    displayMessages(channel);
    consumer = null;
}

private void displayMessages(String channel) {
    Status statusTemplate = new Status();
    statusTemplate.channel = channel;
    statusTemplate.empty = new Boolean(false);
    Status statusResult = null;

    for (;;) {
        // wait till the channel is not empty:
        try {
            statusResult = (Status)
                space.take(statusTemplate, null,
                           Long.MAX_VALUE);
        } catch (Exception e) {
            showStatus("Error looking for status entry.");
            e.printStackTrace();
            return;
        }

        //... no comparison of "head" and "tail" needed:
        //... remove head index, remove message at head, &
        //... return incremented head index to the space

        //... display the message in the pager

        // update status entry & write it back to the space:
        int curSize = statusResult.curSize.intValue();
        curSize--;
        statusResult.curSize = new Integer(curSize);
        statusResult.empty = new Boolean(curSize == 0);
        statusResult.full = new Boolean(false);
        try {
            space.write(statusResult, null, Lease.FOREVER);
        } catch (Exception e) {
            showStatus("Error writing to the channel.");
            e.printStackTrace();
            return;
        }
    }
}
```

```

        }
    }
}

//...
}

```

The `displayMessages` method first fills in a `Status` template with the name of the channel and `empty` set to `false`. The method then enters a loop, where it waits until it's able to take a matching `Status` entry from the space. In other words, the process must wait until the channel is nonempty before proceeding; unlike the `displayNextMessage` of Section 5.6.7, this method doesn't compare the channel's head and tail but instead uses the `Status` entry to determine non-emptiness. As soon as there's a message in the channel, the method takes the head entry, removes the message at the head, and returns an incremented head to the space.

Next, `displayMessages` updates the `Status` entry. It decrements the size of the channel, sets `full` to `false` (since it just removed a message), and sets `empty` to `true` if the size of the channel is now zero, else to `false`. Then it writes the updated `Status` entry back to the space and begins its loop all over again.

5.7.5 Demonstrating the Bounded Channel

By making a few revisions to our original pager application, we've created an example of a bounded channel. When you run the new pager example, you'll get a sense for how the bounded channel mediates the coordination of producers and consumers. After starting a `ChannelCreator` applet and creating a channel of a particular size, run the `PageWriter` and append the maximum number of messages, to fill the channel. If you try to add another, the `PageWriter` must wait until the channel is no longer full before it's able to append it.

The only way to remove messages, of course, is to start up the `PageTaker` applet and click on the interface; the applet consumes the messages in the channel, displaying them one by one, till the channel is empty. At that point, the `PageTaker` must wait until the channel is nonempty before it's able to remove a message.

Of course you can start up as many writers to the same pager channel as you want, and although the pager is intended to be used by one person, you can also experiment by starting up as many pagers for the channel as you want. However, since each pager is consuming messages, each one will only retrieve and display a subset of the messages sent to the pager channel.

5.8 Summary

In this chapter, we've seen how to implement a range of useful communication techniques using distributed data structures and simple protocols that operate over them. We started off by emulating basic message passing, and we saw that space-based programming results in a type of message passing that goes beyond what conventional message passing libraries can provide. Namely, the JavaSpaces technology naturally supports loosely coupled communication, which lends flexibility to all the communication patterns you'll ever build with spaces.

After basic message passing, we turned our attention to a versatile distributed data structure—a channel—as a means of supporting more sophisticated communication patterns. We explored many variants of channels, each with different features. We started off by building a simple channel with broadcast-like semantics. From there, we combined channel writing and reading functionality into a single applet, and managed to build a multiuser, real-time chat application with a small amount of communication code. We also saw how to implement first-in, first-out consumer channels, in which one or more writers add messages to one end, and one or more consumers remove messages at the other end. Last, we saw how to implement bounded channels that have a size limit.

Taken together, these examples showed us how space-based programming is a high-level, powerful means of implementing a wide spectrum of communication techniques. In each case, we saw how the example could be implemented with a small set of entries, along with simple protocols. We also saw how loose coupling allows us to build flexible programs that support “anyone, anywhere, anytime” communication.

In this chapter and the last, we've seen how to achieve useful communication and synchronization with space-based programming—the “coordination glue” that holds processes in a distributed system together. Using these basic coordination techniques as our foundation, we'll spend the next few chapters exploring more interesting ways of using JavaSpaces technology for developing richer application frameworks.

5.9 Exercises

Exercise 5.1 Try a loose coupling experiment with the PingPong example: start Ping before Pong. What happens? Now try starting Pong before Ping. What happens?

- Exercise 5.2** In the PingPong implementation, add a field to the Ball entry to hold a timestamp. Modify the PingPong code such that, if the player is Ping, a report of the roundtrip time of the message is printed each time a Ball is received from Pong.
- Exercise 5.3** How would our PingPong example behave differently if Pong were changed to throw a ball first and then catch a ball (as Pong does)?
- Exercise 5.4** In our basic channel implementation of Section 5.4, we noted some drawbacks in how we check for the existence of a channel. Our solution to creating channels ignores another potential problem: what happens if two processes execute `createChannel` at the same time? Implement a proper way to test for the existence of a channel that will avoid these problems: Implement a channel registry in the space that can be used to see if a channel already exists.
- Exercise 5.5** In the basic channel of Section 5.4 and chat channel of Section 5.5, the readers of messages are locked into a particular channel and can't switch to others (once a channel is chosen, the `channelTextField` of the user interface is set to be noneditable). How could you modify the code to allow "surfing" to other channels at will? One issue is that our readers start reading with message one, so if a new reader thread were created whenever a user switched channels, it would start from the beginning of that channel. Modify the basic channel implementation so that it supports channel surfing and allows a reader to "pick up where it left off" whenever it switches to another channel.
- Exercise 5.6** One limitation of the `ChannelCreator` applet we built in Section 5.6.5 is that we don't ensure that the "Create" button isn't pressed more than once for the same channel. If it is, then the pager will behave unpredictably due to multiple head and tail entries for the channel. Fix this problem by making use of the channel registry you created in Exercise 5.4.
- Exercise 5.7** Build a FILO (first-in, last-out) channel, in which messages are delivered in reverse order (the newest message is always delivered first, and so on).

Exercise 5.8 Implement a workflow system in which messages are taken from one channel and routed to other channels.

Application Patterns

Simplicity does not precede complexity, but follows it.
—Alan J. Perlis, **Epigrams in Programming**

OVER the last few chapters we have implemented useful communication and synchronization patterns, using distributed data structures, that allow remote processes to coordinate their activities. Often, frameworks for entire space-based applications express themselves in a few tried-and-true patterns as well.

In this chapter we will cover some simple, yet powerful, application frameworks, built from distributed data structures we're already familiar with. You'll find that you may be able to model your spaced-based applications on one of these patterns, or some combination or variant of the patterns. And, given what you learn in this chapter, you'll no doubt start inventing your own customized application frameworks.

Our first two examples present useful parallel application patterns—frameworks for solving compute-intensive problems in terms of smaller tasks that can be computed concurrently. As a third example, we'll introduce a framework for producers and consumers of resources. Then we'll discuss further application patterns you may wish to explore.

6.1 The Replicated-Worker Pattern

We are going to start with one of the most common application patterns, the *replicated-worker pattern*—a simple, yet powerful, application framework that is sometimes referred to as the “master-worker pattern.” Recall from our quick peek at the pattern in Chapter 3 that it’s well suited to solving a computational problem that can be decomposed into a number of smaller, independent tasks, all of which turn out to be nearly identical. The replicated-worker pattern is often used for *parallel computing*, in which multiple machines take on the tasks to solve the problem faster.

Typically the replicated-worker pattern involves one *master* process along with any number of *workers*. The master takes a problem, divides it up into smaller tasks, and deposits them into a space. The workers spend their lives waiting for tasks, removing them from the space, computing them, and writing results back into the space. The master's job is to collect the task results and combine them into a meaningful overall solution.

Let's look at a simple pseudocode skeleton that applies to most replicated-worker applications. Here is the pseudocode for the master process:

```
public class Master {
    for (int i = 0; i < totalTasks; i++) {
        Task task = new Task(...);
        space.write(task, ...);
    }

    for (int i = 0; i < totalTasks; i++) {
        Result template = new Result(...);
        Result result = (Result)space.take(template, ...);
        //... combine results
    }
}
```

Here, the master iterates through all the tasks that need to be computed, creating a task entry for each and writing it into the space. The master then iterates again, this time removing as many result entries from the space as there were tasks and combining them into some meaningful form.

Now let's look at the pseudocode for each worker process:

```
public class Worker {
    for (;;) {
        Task template = new Task(...);
        Task task = (Task)space.take(template, ...);
        Result result = compute(task);
        space.write(result, ...);
    }
}
```

A worker repeatedly removes a task from the space, computes it, and writes the result of the computation back into the space, where it will be picked up by the master process. There are many subtle improvements we can make to this code (and we will do so in Chapter 11), but this basic structure represents almost all replicated-worker applications.

It's worth pointing out a couple important characteristics of the replicated-worker pattern. First, each worker process may compute many tasks, one after another. As soon as one task is complete, a worker can look for another task in the space and compute it. In this way, replicated-worker patterns naturally *load balance*: workers stay busy and compute tasks in relation to their availability and ability to do the work. While one worker is occupied with a big task, for instance, another worker might be picking up and completing several smaller tasks. Second, the class of applications that fit into the replicated-worker pattern scales naturally—we can add more workers running on more machines without rewriting our code, and the computation generally speeds up. In general, the more machines, the faster we can compute.

As an example of using the replicated-worker pattern, suppose that you want to ray trace an image (a popular technique for producing realistically rendered scenes). Typically you'd start with a model of the scene, and an image plane in front of the model that is divided into pixels. Rendering an image involves iterating through all the pixels in the plane and computing a color value for each. The task of computing the color value of each pixel involves tracing the rays of light that pass from a viewpoint (such as your eye) through the pixel in the image plane, and to the model. The computation is identical for each pixel, except that the parameters describing the pixel's position differ. Image ray tracing is a perfect candidate for the replicated-worker pattern, since it is made of up a number of independent tasks that are computationally identical.

In the case of ray tracing, the master process might generate tasks that describe collections of pixels to compute (for instance, scan lines of pixels running across an image). Each worker computes a few scan lines (a possibly lengthy process) and returns a result. The master collects the results and combines them to form an image. To produce computer animations, the master might iterate through lots of images and combine the results into an animation sequence.

6.1.1 Computing the Mandelbrot Set

In this section we are going to look at a concrete example that is in the same spirit as ray tracing but a little less complex (and still a lot of fun): computing a Mandelbrot set. Iterating through a particular formula results in values that can be mapped to colors and used to draw an image of the Mandelbrot set, as shown in Figure 6.1. It isn't necessary that you understand the mathematical details of computing a Mandelbrot image; the only thing you need to know is that given an (x, y) coordinate of the screen image, we can compute an integer value for that pixel and map it to a color. (See the sidebar below on "How to compute the Mandelbrot set" if you are interested in the mathematical details.)

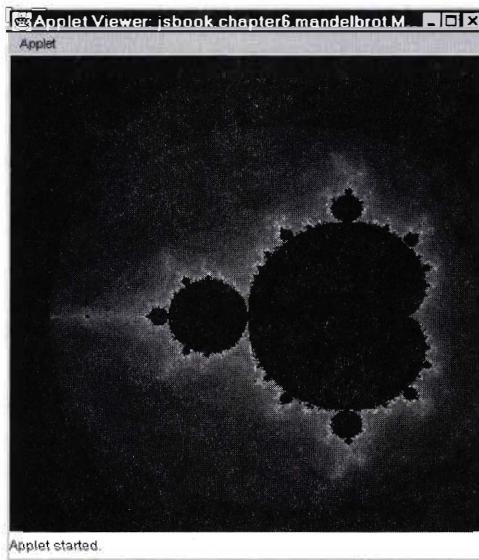


Figure 6.1 The Mandelbrot set.

How to compute the Mandelbrot set. The image of the Mandelbrot set shown in Figure 6.1 is a mapping from (x,y) pixel coordinates to color values. Each (x,y) point represents a complex number C , where:

$$C = x + yi$$

In order to calculate the color at that point, we iterate through the following sequence:

$$Z_0 = 0$$

$$Z_{n+1} = Z_n^2 + C$$

Depending on the value of C , the absolute value of Z approaches 0 or infinity at different rates. We iterate until the absolute value of Z exceeds 2 (if it gets that big, we know it will go to infinity), or until a maximum number of iterations (say 100) has been reached. The number of iterations determines the color to paint the pixel. The Mandelbrot set is essentially a mapping from C values (in other words, (x,y) coordinates) to the associated rate at which the absolute value of Z approaches infinity.

As with ray tracing, when we compute and draw a Mandelbrot image, each pixel can be computed independently of all the rest. Unlike ray tracing, however, each pixel's color value can be computed fairly quickly, because the computation

isn't very complex. In fact, because of the overhead of communicating task and result entries to and from the space, a replicated-worker version of a Mandelbrot computation may actually compute more slowly than a nonparallel version (refer to the sidebar below on “The Computation/Communication Ratio” for further explanation). But our aim here in building a Mandelbrot viewer isn't to build a fast parallel application, but to gain experience with the replicated-worker framework (and Mandelbrot computations are simpler than ray tracing).

In building a replicated-worker version of a Mandelbrot set viewer, we will create a master process that iterates through all the scan lines that need to be computed and writes out tasks (each representing multiple scan lines). The master then collects the results (in whatever order it finds them) and draws the results to the screen. We'll also create worker processes, each of which repeatedly removes a task from the space, computes the color values for the scan lines represented by the task, and writes the result back to the space for the master to pick up. Let's start building our Mandelbrot viewer by first defining entries for tasks and results.

The Computation/Communication Ratio. The replicated-worker pattern is typically used to compute problems more quickly than a standalone implementation would allow. In general, the more machines tackle a problem, the faster it can be solved. However, some problems (such as the Mandelbrot computation) can be computed at least as fast in a standalone version as they can be in a distributed version. The reason has to do with the computation/communication ratio.

A problem is typically worth breaking up into a parallel version if the time workers spend on communication (passing entries to and from the space) is a small fraction of the time they spend computing. If that is the case, then adding processors typically results in faster computations. However, if the time spent communicating is significant with respect to the computation time, then a parallel version may actually take longer to compute than a standalone version. This is the case with a parallel Mandelbrot solution: since the time to compute a few scan lines is very quick, the communication cost is comparatively high and will increase the solution time. In the case of ray tracing, or other compute-intensive examples, the computation/communication ratio is large, and the solution time can be greatly shortened by decomposing and distributing the pieces of the computation among multiple processors.

6.1.2 Task and Result Entries

As we mentioned in Chapter 3, we often use a *task bag* and a *result bag* in replicated-worker computations. The task bag holds all the tasks a master process

needs to have computed, while the result bag holds the results as they are completed. In most cases, bags are all we need to hold tasks and results. In cases where an ordering needs to be imposed on tasks, then we'll use a task queue, as we described in Section 5.6. For our Mandelbrot viewer, we don't care about the order in which the tasks are computed, so bags will suffice. The master picks up and draws whichever results it finds, in no particular order, until the entire image has been rendered.

First let's define a task entry that can be placed into the task bag. A task entry holds the coordinates of the region (representing a set of complex numbers) for which the application is collectively computing the Mandelbrot set, and also specifies the particular "slice" of that region that a worker must compute:

```
public class Task implements Entry {
    public Long jobId;          // id for Mandelbrot computation

    // region for which application is computing Mandelbrot
    public Double x1;           // starting x
    public Double y1;           // starting y
    public Double x2;           // ending x
    public Double y2;           // ending y

    // slice of the region this task computes
    public Integer start;       // starting scan line
    public Integer width;        // width of scan line in image
    public Integer height;       // number of scan lines in image
    public Integer lines;        // number of scan lines per task

    public Task() {
    }
}
```

You'll notice that we have a field called `jobId`, which provides a unique name for this particular Mandelbrot computation in the space. We'll assign the current date and time to this field in the form of a long integer value, which for our purposes should be unique. Next the task entry has `x1`, `y1`, `x2`, and `y2` fields, specifying the region of complex numbers for which the Mandelbrot set is being collectively computed.

Next, the task entry has fields that are used to communicate to the worker how to compute a slice of that region. The `start` field indicates the first scan line to compute, `width` specifies how many pixels wide the scan lines are, `height` holds the number of scan lines in the overall region being computed by the application, and `lines` specifies the number of scan lines to compute.

Now let's look at the result entry, which holds the result of computing a task described by a task entry:

```
public class Result implements Entry {
    public Long jobId;          // id for Mandelbrot computation
    public Integer start;        // starting scan line of results
    public byte[][] points;     // pixel values as byte integers

    public Result() {
    }

    public Result(Long jobId, Integer start, byte[][] points) {
        this.jobId = jobId;
        this.start = start;
        this.points = points;
    }
}
```

Each result contains a `jobId` field, which is the same as the `jobId` supplied in the task entry, a `start` field that indicates the starting scan line for the pixels in this result, and a `points` field that holds a set of pixel values.

Now that we've seen the task and result entries that will populate the task and result bags, let's take a look at the code for the master process.

6.1.3 The Master

Our master process is responsible for breaking the Mandelbrot computation into tasks, collecting the results, and also displaying the Mandelbrot image. In addition, the master process allows the user to interact with the image. A user can “zoom into” an area of the image by clicking on it and dragging a small rectangle. When the user releases the mouse button, the region bounded by the rectangle becomes the basis for a new Mandelbrot computation, and a new set of tasks is generated to be computed by the workers.

Minus some of the interface and graphics code, here is the skeleton of the `Master` class:

```
public class Master extends Applet
    implements Runnable, MouseListener, MouseMotionListener
{
    private JavaSpace space;
    private int xsize;           // dimensions of window
    private int ysize;
```

```
private Long jobId;      // id for this computation
private int tasks;        // number of tasks to generate
private int lines = 0;    // # of scan lines per task
private Thread master;

// initial region for which Mandelbrot is being computed
private double x1 = -2.25;
private double x2 =  3.0;
private double y1 = -1.8;
private double y2 =  3.3;

private boolean done = false; // computation finished?
private int progress;        // number of scan lines
//... other variables go here

public void init() {
    space = SpaceAccessor.getSpace();
    xsize = getSize().width;
    ysize = getSize().height;

    String taskStr = getParameter("tasks");
    if (taskStr == null) {
        tasks = 20;
    } else {
        tasks = Integer.parseInt(taskStr);
    }
    lines = ysize / tasks; // scan lines per task

    //... code for offscreen graphics buffer goes here

    // set up listeners
    this.addMouseListener(this);
    this.addMouseMotionListener(this);

    // spawn thread to handle computations
    if (master == null) {
        master = new Thread(this);
        master.start();
    }
}
```

```
public void run() {
    generateTasks();
    collectResults();
}

//... generateTasks method goes here

//... collectResults method goes here

//... user interface and graphics methods go here
}
```

Note the `x1`, `y1`, `x2`, and `y2` coordinates initialized in the variable declarations section; these values specify the starting region of the complex numbers for which the Mandelbrot set will be computed. In the `init` method, we perform setup for the Mandelbrot computation: We obtain a reference to a `space` object, the dimensions of the applet window in pixels, and the number of tasks to break the image into (obtained from the applet parameter `tasks`, if one exists, otherwise set to twenty). Given the image height and number of tasks, we compute the number of scan lines to be computed per task. Then we register the applet to be a listener for mouse events, which will occur when the user zooms into a region of the Mandelbrot image.

Finally, the master spawns a new thread that generates tasks and then collects results, as you see in the `run` method. Of course, some replicated-worker applications may have more advanced designs—such as allowing the task generation and the result collection to run concurrently, or performing multiple iterations of a computation. However, in this example we are trying to demonstrate a simple pattern.

Let's take a look at the details of the `generateTasks` method:

```
private void generateTasks() {
    Task task = new Task();

    long millis = System.currentTimeMillis();
    jobId = new Long(millis);
    task.jobId = jobId;

    task.x1 = new Double(x1);
    task.x2 = new Double(x2);
    task.y1 = new Double(y1);
    task.y2 = new Double(y2);
```

```
task.width = new Integer(xsize);
task.height = new Integer(ysize);
task.lines = new Integer(lines);

for (int i = 0; i < ysize; i += lines) {
    task.start = new Integer(i);
    try {
        System.out.println("Master writing task " +
                           task.start + " for job " + task.jobId);
        space.write(task, null, Lease.FOREVER);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

This method creates a new `Task` entry and fills in its `jobId` field (the identifier for the overall Mandelbrot computation) with the current time in milliseconds. We then fill in the values that are identical for every task we will generate: the coordinates of the region the application is computing, the width and height of the entire image, and the number of scan lines per task. Finally, to generate tasks, we iterate through the scan lines in the image. Each task's `start` field is assigned the starting line for the slice of the image it will compute, and the task is written into the space.

When `generateTasks` is finished, the `run` method calls `collectResults`, which is defined as follows:

```
private void collectResults() {
    while (true) {
        while (done) {
            try {
                System.out.println(
                    "Master done; waiting for new job.");
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

Result template = new Result();
template.jobId = jobId;
Result result = null;
```

```

        try {
            result = (Result)
                space.take(template, null, Long.MAX_VALUE);
            System.out.println("Master collected result " +
                result.start + " for job " + result.jobId);
            display(result);
            progress = progress + lines;
        } catch (Exception e) {
            e.printStackTrace();
        }

        if (progress == ysize) {
            done = true;
            repaint();
        }
    }
}

```

The `collectResults` method consists of a `while` loop: each time through the loop we check to see if the current computation is done (signified by the `done` variable being equal to `true`). If it is done, then we loop until `done` becomes `false` (which happens when the user zooms into a new region), sleeping for half a second (500 milliseconds) on each iteration.

If the computation isn't done, then we must collect task results. To do this, we first create a `Result` template, leaving all fields as wildcards except for `jobId`. This template allows us to retrieve any results computed for the Mandelbrot, regardless of the order in which the tasks were generated or computed. Once we have a template, we take a result from the space and pass it to the `display` method, which draws the scan lines contained in it. We also update the `progress` variable; when `progress` is equal to the number of scan lines we are computing, we know that all results have been retrieved, and we set the `done` variable to `true`.

The `generateTasks` and `collectResults` methods cover the major points of how the master process works (leaving aside the details of the graphics display). The only other aspect we need to discuss is what happens when the user zooms into a finished Mandelbrot image. When the user clicks, drags, and releases the mouse, the `mouseReleased` method is called:

```

public void mouseReleased(MouseEvent e) {
    int x = e.getX();
    int y = e.getY();
    if (done) {
        x1 += ((float)x / (float)xsize) * x2;

```

```

        y1 += ((float)y / (float)ysize) * y2;
        x2 /= 2.0;
        y2 /= 2.0;
        x1 -= x2 / 2.0;
        y1 -= y2 / 2.0;
        done = false;
        drag = false;
        offg.setColor(Color.black);
        offg.fillRect(0, 0, xsize, ysize);
        progress = 0;
        repaint();
        generateTasks();
    }
}

```

Note that the `mouseReleased` method does nothing if `done` is `false`—the user can only start exploring a new region once the current Mandelbrot image has been fully computed. This method determines the new area of the image the user wants to explore, and it resets the master’s `x1`, `y1`, `x2`, and `y2` variables to that region. It’s also important to point out that the method sets `done` to `false` (to indicate that a new Mandelbrot computation is underway) and `progress` for the new computation to zero. It then calls `generateTasks`, which creates a new set of task entries for the new Mandelbrot computation.

While `generateTasks` occurs in the master’s main thread as a result of the `mouseReleased` event, our `collectResults` method is still running in the thread spawned by the master and will wake up within half a second to begin looking for new results.

6.1.4 The Worker

Now that we’ve seen how the master process generates tasks and collects results, let’s look at the code for the worker processes that compute the tasks:

```

public class Worker {
    public static void main(String[] args) {
        JavaSpace space = SpaceAccessor.getSpace();

        Task template = new Task();
        Task task;

        for (;;) {
            byte[][][] points;

```

```

try {
    task = (Task)
        space.take(template, null, Long.MAX_VALUE);
    System.out.println("Worker got task "
        + task.start +
        " for job " + task.jobId);

    points = calculateMandelbrot(task);
    Result result = new Result(task.jobId,
        task.start, points);

    System.out.println(
        "Worker writing result for task " +
        result.start + " for job " + result.jobId);
    space.write(result, null, Lease.FOREVER);
} catch (Exception e) {
    e.printStackTrace();
}
}

//... calculateMandelbrot method goes here
}
}

```

When we run a worker application, its `main` method is executed. In `main`, the worker obtains a handle to a space and then creates a Task template, leaving all fields as wildcards so that it can match any task in the bag. Then `main` enters a `for` loop that continually looks for tasks. Using the template, we take a task from the space and pass it to the `calculateMandelbrot` method, which calculates the pixel color values for the scan lines described in the task and returns them as a byte array (the `calculateMandelbrot` and zooming code were originally written by Robin Edwards).

We construct a new `Result` entry that holds the task's job identifier, the starting scan line that was computed, and the array of bytes, and then write the result into the space. At that point, the worker begins the loop anew and looks for another task to compute.

At this point you should experiment with the Mandelbrot viewer. Start up any number of workers and then start the master process (of course, given the loosely coupled nature of our application, you could reverse the ordering). Watch the Mandelbrot image fill in, click to zoom into a region of interest, and watch as the workers compute the new area. Add or remove workers as you wish.

6.2 The Command Pattern

The second pattern we will explore is the *command pattern*. The command pattern is simple; to implement it, an object needs to implement the following interface:

```
public interface Command {  
    public void execute();  
}
```

In other words, any object that implements the command pattern provides an `execute` method. What the `execute` method does depends on the context and application in which it is being used. The important point is, if you have an object that implements the command pattern, you know you can invoke its `execute` method. While this pattern is simple, its power stems from our ability to drop an object into a space and have another process pick it up and invoke methods on it, even if the remote process has never seen the object before. In other words, we can pass object *behavior* through spaces, as well as object data, which is a very powerful technique.

You might be asking yourself why this is useful. Let's reconsider our Mandelbrot example: our application required that we run a number of worker processes that wait for Mandelbrot tasks and compute them. But what if we want to compute Julia sets (a slightly different kind of fractal image), or ray trace images, or multiply matrices? Our Mandelbrot worker can't perform these new kinds of tasks. For each kind of computation, we'd have to create and run a new worker applet or application that knows how to service those kinds of tasks. It would be far more elegant (not to mention less work for us) to create one generic worker application that accepts task requests of many kinds, where the code to perform the task is bundled with the task object itself, in an `execute` method.

So in other words, using the command pattern, we can create a generic application that can service the requests of many master processes that need tasks computed. Suppose we have a network of fast graphics workstations that handle requests for ray-traced images from several workgroups working on animations. Using the command pattern, each group can supply its own specialized code to compute the images, while using the same servers to compute them. The workgroups only need to know and maintain one interface to the group of compute servers.

The command pattern isn't specific to space-based programming; in fact, it is a well known pattern that has been used in a variety of domains to allow an object to specify its own behavior in response to certain events. The command pattern has a wide range of uses, from implementing actions in user interface code, to providing "undo objects" that can undo state changes in a system. For more detailed information on this pattern, refer to *Design Patterns: Elements of Reusable Object-Oriented Software*, by Erich Gamma, et al.

6.2.1 Implementing a Compute Server

Let's implement a simple compute server based on the command pattern. The compute server consists of a collection of worker processes. Each worker continually takes a task from the space (which was placed there by some master process), computes it by calling its `execute` method, and then continues on to the next task. The tasks themselves can perform space operations; typically they write the results of the worker's computation into a result entry, as in the Mandelbrot example.

We'll start by defining a generic task entry class that implements the `Command` interface and that will serve as the base class for all other (more specialized) task entries:

```
public class TaskEntry implements Entry, Command {  
    public ResultEntry execute() {  
        throw new RuntimeException(  
            "TaskEntry.execute() not implemented");  
    }  
}
```

The `TaskEntry` class provides an `execute` method that throws an exception if it isn't overridden by a subclass. Our reason for doing this is to force any subclass to implement this method. The conventional method of enforcing a subclass to implement a method is to make the class `abstract`, but we have avoided doing that here. If `TaskEntry` were defined to be `abstract`, then we would not be able to instantiate it to act as a template to retrieve entries belonging to any of its subclasses—a major drawback. Note that `execute` returns a `ResultEntry` object; we will see why shortly.

6.2.2 The Generic Worker

Now that we have a task entry, let's look at a generic worker that processes task entries. Here is the skeleton for the generic worker:

```
public class GenericWorker {  
    JavaSpace space;  
  
    public static void main(String[] args) {  
        GenericWorker worker = new GenericWorker();  
        worker.startWork();  
    }  
}
```

```
public GenericWorker() {
    space = SpaceAccessor.getSpace();
}

public void startWork() {
    TaskEntry taskTmpl = new TaskEntry();

    while (true) {
        System.out.println("getting task");
        try {
            TaskEntry task = (TaskEntry)
                space.take(taskTmpl, null, Lease.FOREVER);
            ResultEntry result = task.execute();
            if (result != null) {
                space.write(result, null, Lease.FOREVER);
            }
        } catch (RemoteException e) {
            System.out.println(e);
        } catch (TransactionException e) {
            System.out.println(e);
        } catch (UnusableEntryException e) {
            System.out.println(e);
        } catch (InterruptedException e) {
            System.out.println("Task cancelled");
        }
    }
}
```

The worker runs as an application and executes the `main` method. The method first instantiates a `GenericWorker` object; the constructor simply obtains a reference to a space object. Then `main` calls the worker's `startWork` method.

The `startWork` method first creates a task entry to be used as a template to match any available task entry in the space. Then the method enters a loop, in which it waits for a task entry in the space. When one is available, a task is taken from the space and assigned to the variable `task`, and its `execute` method is called. Note that the worker process doesn't need to know what kind of task it has picked up or the details of how to compute it—it just calls the task's `execute` method. If successful, `execute` returns a `ResultEntry` object, which is then written into the space. In this manner, the results of a task computation are communicated back to the process that created the task.

A `ResultEntry` is an object that implements the `Entry` interface. Once again, we make it a class rather than an abstract class or interface, because we want to be able to use it as a template:

```
public class ResultEntry implements Entry {  
}
```

6.2.3 The Generic Master

The job of the master process is to generate tasks—which will be computed by workers monitoring the space—and then to collect the results. First we'll create an abstract generic master class:

```
public abstract class GenericMaster extends Applet  
    implements Runnable {  
  
    private JavaSpace space;  
    private Thread thread;  
  
    public void init() {  
        space = SpaceAccessor.getSpace();  
  
        thread = new Thread(this);  
        thread.start();  
    }  
  
    public synchronized void run() {  
        generateTasks();  
        collectResults();  
    }  
  
    protected abstract void generateTasks();  
  
    protected abstract void collectResults();  
  
    protected void writeTask(TaskEntry task) {  
        try {  
            space.write(task, null, Lease.FOREVER);  
        } catch (RemoteException e) {  
            e.printStackTrace();  
        } catch (TransactionException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

```
protected ResultEntry takeResult(ResultEntry template) {
    try {
        ResultEntry result = (ResultEntry)
            space.take(template, null, Long.MAX_VALUE);
        return result;
    } catch (RemoteException e) {
        e.printStackTrace();
    } catch (TransactionException e) {
        e.printStackTrace();
    } catch (UnusableEntryException e) {
        e.printStackTrace();
    } catch (InterruptedException e) {
        System.out.println("Task cancelled");
    }
    return null;
}
```

}

The `GenericMaster` is an applet; its `init` method first obtains a handle to a `space` object, and then instantiates and starts a new thread. The `run` method follows a familiar pattern: it calls the methods `generateTasks` and `collectResults`. These two abstract methods are meant to create a set of tasks for the workers and to retrieve a set of results.

In the `GenericMaster` we've also provided two convenience methods: one to write a task to the space, and another to take a result from the space.

6.2.4 Creating Specialized Tasks and Results

The `TaskEntry`, `ResultEntry`, `GenericWorker`, and `GenericMaster` classes make up the foundation of our compute server. Before we can use the compute server, though, we first need to subclass the `TaskEntry` and `ResultEntry` classes, and also to create a specialized master process that generates specific kinds of tasks and collects the results of computing them. Then we can start up the specialized master process and also start up as many instances of the worker on as many machines as we like, and our compute server will be in full gear.

To demonstrate the compute server, we are going to create two simple subclasses of `TaskEntry` and a master class that generates tasks from both sub-

classes. Rather than implementing complex, compute-intensive tasks, we'll create two simple (and quickly computed) subclasses that compute factorials and Fibonacci numbers. Our aim here is to demonstrate how the compute server works, not to concentrate on creating an application that shows good parallel performance (recall the discussion of the computation/communication ratio in Section 6.1). When we get to Chapter 11, we will create a more sophisticated compute server and explore its use in breaking password encryption, which is a highly compute-intensive problem.

Let's start with our factorial task. The factorial function is defined as follows:

$$\begin{aligned}\text{factorial}(0) &= 1 \\ \text{factorial}(n) &= n * \text{factorial}(n-1)\end{aligned}$$

So, given an integer n , we can compute its factorial by multiplying $n * (n-1) * (n-2) * (n-3) * \dots * 1$. Here is a subclass of `TaskEntry` that implements the factorial task:

```
public class FactorialTask extends TaskEntry {
    public Integer n;

    public FactorialTask() {
    }

    public FactorialTask(int n) {
        this.n = new Integer(n);
    }

    public ResultEntry execute() {
        System.out.println(
            "Computing Factorial of " + n);
        int num = n.intValue();
        return new FactorialResult(num, factorial(num));
    }

    public int factorial(int i) {
        if (i == 0) {
            return 1;
        } else {
            return i * factorial(i - 1);
        }
    }
}
```

As you can see, `FactorialTask` is a subclass of `TaskEntry`. The class has one field, an integer `n` that indicates which factorial to compute, and the usual no-arg and convenience constructors. In addition, we have overridden the parent class's `execute` method. The new `execute` method converts the field `n` from an `Integer` into an `int`, and then passes the integer to the `factorial` method, which recursively computes the factorial and returns the final result.

Once the factorial is returned, the `execute` method creates and returns a `FactorialResult` object, which extends `ResultEntry` and has fields that hold the integer and its factorial:

```
public class FactorialResult extends ResultEntry {  
    public Integer n;  
    public Integer factorial;  
  
    public FactorialResult() {  
    }  
  
    public FactorialResult(int n, int factorial) {  
        this.n = new Integer(n);  
        this.factorial = new Integer(factorial);  
    }  
}
```

The `FactorialResult` object is returned by a call to `execute`, and as a result will be written into the space by a worker process.

Our Fibonacci task is another simple mathematical task that computes a number in the Fibonacci sequence: 1 1 2 3 5 8 13 21 34 55 89.... The Fibonacci function is defined recursively as follows:

$$\begin{aligned} \text{fibonacci}(0) &= 1 \\ \text{fibonacci}(1) &= 1 \\ \text{fibonacci}(n) &= \text{fibonacci}(n-2) + \text{fibonacci}(n-1) \end{aligned}$$

In other words, a number in this sequence is computed by adding the previous two numbers in the sequence (for instance, $55 = 21 + 34$). Here is the subclass of `TaskEntry` that implements our Fibonacci task:

```
public class FibonacciTask extends TaskEntry {  
    public Integer n;  
  
    public FibonacciTask() {  
    }
```

```
public FibonacciTask(int n) {
    this.n = new Integer(n);
}

public ResultEntry execute() {
    System.out.println(
        "Computing Fibonacci of " + n);
    int num = n.intValue();
    return new FibonacciResult(num, fibonacci(num));
}

public int fibonacci(int n) {
    int lo = 1;
    int hi = 1;

    for (int i = 1; i < n; i++) {
        hi = lo + hi;
        lo = hi - lo;
    }
    return hi;
}
}
```

As you can see, this task entry mirrors the factorial task, except that it computes a Fibonacci number and returns a `FibonacciResult` (which, as you can see in the complete source code, mirrors `FactorialResult`).

6.2.5 Creating a Specialized Master

We've already seen the code for our generic master process, but recall that the `generateTasks` and `collectResults` methods are abstract. We need to create a subclass of our generic master, with its own methods to generate specific tasks, and then collect the results of computing them.

Let's create a subclass to generate both factorial and Fibonacci tasks and retrieve their results:

```
public class FibFactMaster extends GenericMaster {
    public void generateTasks() {
        for (int i = 0; i < 10; i++) {
            FactorialTask task = new FactorialTask(i);
            writeTask(task);
        }
    }
}
```

```

        for (int i = 0; i < 10; i++) {
            FibonacciTask task = new FibonacciTask(i);
            writeTask(task);
        }
    }

    public void collectResults() {
        ResultEntry template = new ResultEntry();
        for (int i = 0; i < 20; i++) {
            ResultEntry result = takeResult(template);

            if (FactorialResult.class.isInstance(result)) {
                FactorialResult fact = (FactorialResult)result;
                System.out.println("Factorial of " + fact.n +
                    " is " + fact.factorial);
            } else if (FibonacciResult.class.isInstance(result)) {
                FibonacciResult fib = (FibonacciResult)result;
                System.out.println("Fibonacci of " +
                    fib.n + " is " + fib.fibonacci);
            } else {
                System.out.println("Result class not found");
            }
        }
    }
}

```

Here in `FibFactMaster` we've subclassed the `generateTasks` and `collectResults` methods. The `generateTasks` method creates ten factorial tasks (to compute factorials for the numbers 0 through 9) and calls the `writeTask` convenience method to write the tasks to the space. Then ten Fibonacci tasks are created and written to the space.

The `collectResults` method first creates a `ResultEntry` template and then loops for twenty iterations (the number of result entries that will be computed and placed in the space). Each time through the loop, we take a result entry from the space. Note that we are using the generic `ResultEntry` to take a result from the space, rather than one of the derived classes `FactorialResult` or `FibonacciResult`—recall that by the semantics of template matching, the `ResultEntry` template can potentially match any subclass. Because the `ResultEntry` template has no defined fields, it will match any other `ResultEntry` object, or any subclassed objects.

Once we've retrieved an entry from the space, we then need to know what type of result it is, factorial or Fibonacci. We make use of the `isInstance` method (from the core Java API) to determine which it is. In either case, we print an appropriate result message.

6.2.6 Running the Compute Server

At this point, all that's left to do is run the example. Start up any number of workers and then a master process (or, thanks to loose coupling and persistence, feel free to run the master first and then the workers). The output of a worker process will look something like the following:

```
getting task
computing Fibonacci of 8
getting task
computing Fibonacci of 9
getting task
Computing Factorial of 2
getting task
Computing Factorial of 3
getting task
Computing Factorial of 4
getting task
Computing Factorial of 5
```

The output of the master process will look something like this:

```
Factorial of 0 is 1
Fibonacci of 0 is 1
Fibonacci of 1 is 1
Fibonacci of 2 is 2
Fibonacci of 3 is 3
Fibonacci of 4 is 5
Factorial of 1 is 1
Fibonacci of 5 is 8
Fibonacci of 6 is 13
Fibonacci of 7 is 21
Fibonacci of 8 is 34
Fibonacci of 9 is 55
Factorial of 2 is 2
Factorial of 3 is 6
Factorial of 4 is 24
```

```
Factorial of 5 is 120
Factorial of 6 is 720
Factorial of 7 is 5040
Factorial of 8 is 40320
Factorial of 9 is 362880
```

Note that the results come back in an arbitrary order, which is a direct result of the bag data structures our application uses. The set of tasks we generate resides in a bag data structure, from which the workers can pick out tasks in any order. Likewise, the result pool is a bag of entries, from which results can be retrieved and displayed in any order.

We will return to the topic of compute servers in Chapter 11, where we will build a more sophisticated server and use it to solve an interesting, compute-intensive decryption problem.

6.3 The Marketplace Pattern

As a third example application framework, we'll explore the *marketplace pattern*. The marketplace pattern represents a framework in which producers and consumers of resources can interact with one another to find the best deal.

A resource may be any product or service that can be bought and sold—whether it be bandwidth, computing cycles, regional electric power, compact discs, automobiles, antiques, or something else. Buyers request resources with certain characteristics, and sellers with products that match or approximate those characteristics issue “bids.” By evaluating bids and choosing the “best,” buyers are able to obtain resources that provide the characteristics they care about most—the resource might be the least expensive, the highest quality, the most quickly available, or any combination of factors. The marketplace pattern provides a framework for these types of bidding applications to occur.

6.3.1 An Automobile Marketplace

To illustrate the pattern, we're going to build a space-based automobile marketplace. A automobile marketplace might work as follows: We have a set of car buyers and a set of car sellers. Car buyers are interested in finding the optimal car that matches their buying criteria. They might be most interested in the make and model, the year, the price, how soon the car can be delivered, the color, or any combination of automobile attributes you can imagine. Car sellers want to sell cars. In a competitive market they have to keep their prices low enough that they sell a few, but they take many factors into consideration in making offers to buyers

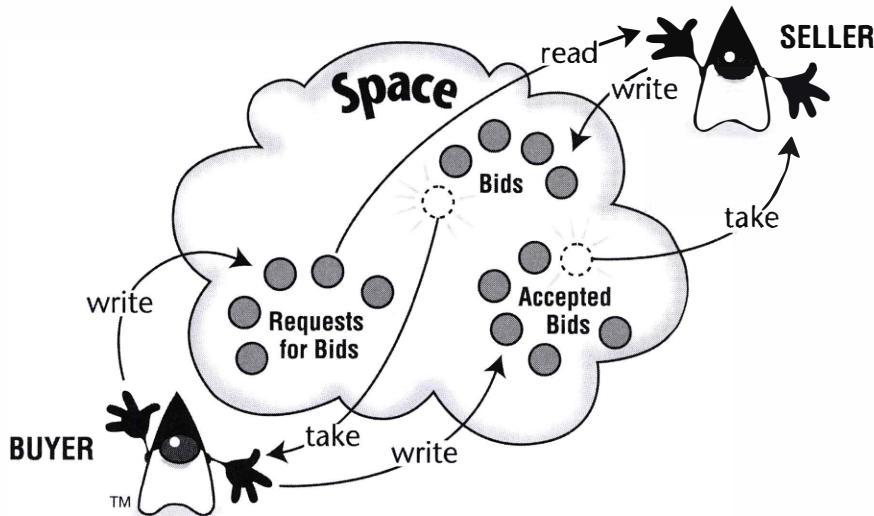


Figure 6.2 The Marketplace Pattern: Buyers and sellers interact by exchanging requests for bids, bids, and accepted bids.

(how in-demand a particular car is, how many cars they must sell to meet a quota, and so on).

While our example is specific to car markets, it represents the more general and useful marketplace pattern. You can use the same pattern in other domains—wherever there is a buyer/seller scenario in which buyers need resources and there are sellers competing to provide them.

As Figure 6.2 illustrates, in our space-based marketplace, buyers and sellers use a space to exchange three kinds of entries—"requests for bids," "bids," and "accepted bids." We'll see how to use these entries, along with a couple familiar distributed data structures, to implement a marketplace. But first, let's take a look at the buyer and seller interfaces and get a feel for how the participants interact.

6.3.2 Interaction in the Marketplace

A potential car buyer sees the interface shown in Figure 6.3. The buyer fills out the "request for bid" form fields at the top, specifying the desired make, model, year, and colors of the car, as well as the price limit. The buyer must also give a "duration" for the request—the amount of time the request will exist (here it means minutes, but in a real-world system, duration would typically be given in days). After the time limit is up, the request will expire. In the interface shown in the figure, the buyer duke@sun.com wants to buy a Volkswagen New Beetle, 1999 model, with red exterior and tan interior, for no more than \$14,000. The request for bids will be available in the space for 30 minutes.

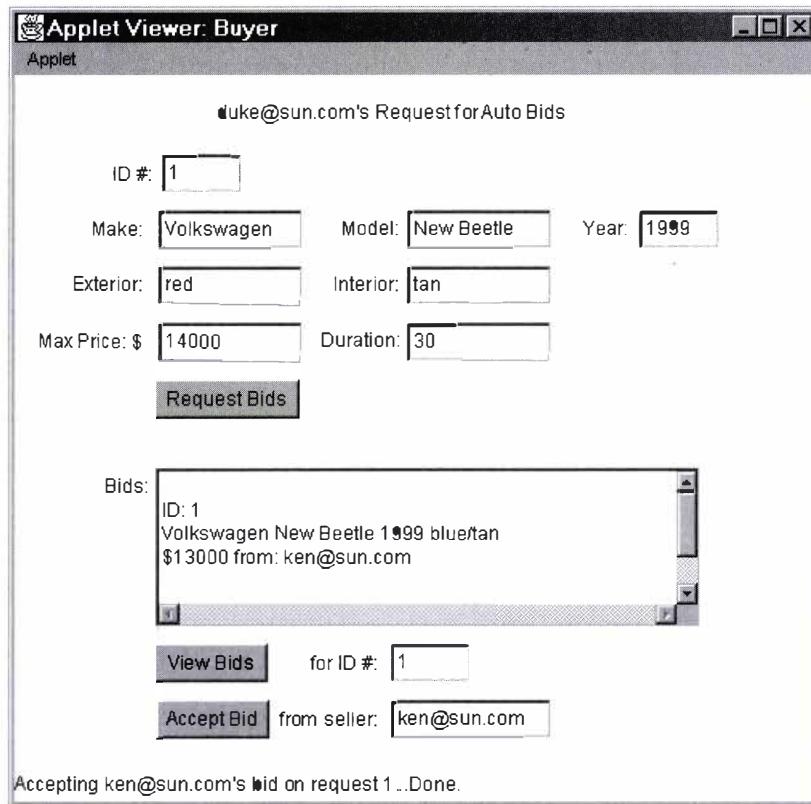


Figure 6.3 A car buyer's interface.

When the form is fully filled in, the buyer presses the “Request Bids” button to deposit a “request for bids” entry into the space. The system gives each request in the automobile marketplace a unique ID, which is automatically displayed in the “ID #” field. There is no limit on the number of requests the same buyer can issue.

In the meantime, car sellers (which could be dealerships or individuals) fill out a similar form, shown at the top of Figure 6.4, to specify the kinds of cars they’re interested in selling. At any time, the seller may press the “View Requests” button to read and display all requests in the marketplace that meet the criteria. One important point to note is that the seller doesn’t need to fill in all the fields for make, model, year, exterior, and interior, but is free to make the specification very general. For instance, if the seller specifies only the make “Ford” and no other fields, all requests for Fords will be retrieved. The seller can even leave all fields blank, which means that all existing car requests in the marketplace will be printed. In this example, the seller specified “Volkswagen” and one matching

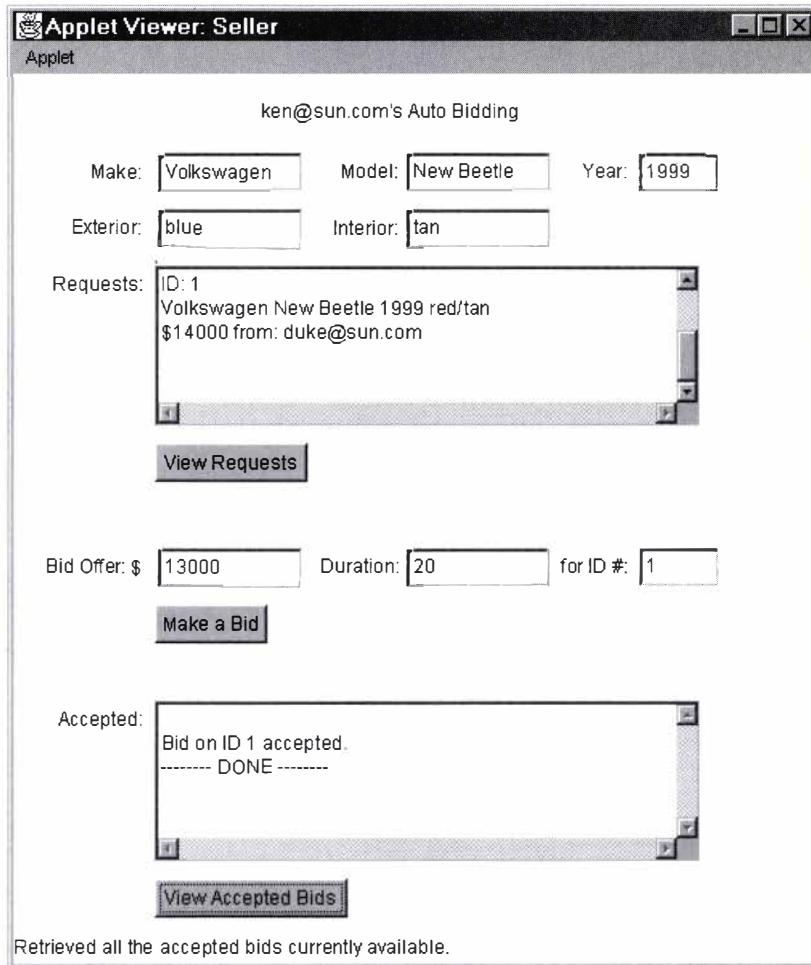


Figure 6.4 A car seller's interface.

request was displayed in the “Requests” window. Every time a seller presses “View Requests,” only those items that have been added since the last time the button was pressed will be displayed.

To respond to a request, a seller prepares a “bid” by filling in full details of the make, model, year, and colors of the car for sale, and by specifying a bid price, bid duration, and the request ID. In Figure 6.4, the seller has filled in all these fields and wishes to make a bid on request ID #1. By pressing the “Make a Bid” button, a bid entry—with contact information for the seller—is written into the space.

At any time, a buyer can press the “View Bids” button to pick up and display all bids in the space that match a particular request. The buyer of Figure 6.3 has

done this and sees the bid that was made by seller ken@sun.com. To accept a particular bid, the buyer fills in the “ID #” and “seller” text fields and presses “Accept Bid,” which launches an “accepted bid” entry into the space. At any time, a seller can press its “View Accepted Bids” button to pick up and display its “accepted bid” entries, as shown in Figure 6.4.

6.3.3 The Bid Entry

Now that we’ve seen the buyer and seller interfaces and know how buyers and sellers interact to make deals on automobiles, we’re going to get into the nitty-gritty of how the marketplace is implemented. We’ve seen that three kinds of messages—requests for bids, bids, and accepted bids—play a role in the interaction. It turns out that just one entry type, which we call `BidEntry`, will suffice to implement all three kinds of messages. Here’s the definition:

```
public class BidEntry implements Entry {  
    public String label;  
    public Integer idNum;  
    public String participant;  
    public Integer year;  
    public String make;  
    public String model;  
    public String exteriorColor;  
    public String interiorColor;  
    public Integer price;  
  
    public BidEntry() {  
    }  
  
    public BidEntry(String label, Integer idNum,  
        String participant, Integer year,  
        String make, String model,  
        String exteriorColor, String interiorColor, Integer price)  
    {  
        this.label = label;  
        this.idNum = idNum;  
        this.participant = participant;  
        this.year = year;  
        this.make = make;  
        this.model = model;  
        this.exteriorColor = exteriorColor;  
    }  
}
```

```

        this.interiorColor = interiorColor;
        this.price = price;
    }

    //... a print method to display the entry fields goes here
}

```

The `label` field is given the value “request,” “bid,” or “accepted-bid,” depending on the purpose of a particular instance of the `BidEntry`. The `idNum` field represents the unique integer ID of a car request (whether the entry represents the request itself or is a “bid” or “accepted-bid” response to a request). The `participant` field holds information about the name of the buyer (if the entry is a request) or the seller (if the entry is a bid or accepted bid).

Accepted bid entries don’t use any of the remaining fields; just request and bid entries do. There are several fields that are used to store information about some common criteria used to choose a car: `make`, `model`, `year`, `exteriorColor`, and `interiorColor`. Of course, there are many other aspects of car buying we might want to include in the `BidEntry`, but the ones we have chosen are enough to illustrate the concept. The `price` field indicates either the maximum price the buyer is willing to pay (if the entry is a request) or the price the seller is asking (if the entry is a bid).

6.3.4 The Application Framework

Now let’s take a look at the application framework that makes this pattern work. Since we can expect to have many buyers submitting requests and many potential sellers interested in reading those requests, we need a many-to-many data structure. We’ll use the channel structure from the last chapter: a “requests” channel will hold `BidEntry` elements that represent automobile requests. Buyers append requests to the end of the channel, and sellers press a button to read and display those requests in the channel that match their interest. Every time a seller presses the button, reading picks up where it left off with the previous button press.

When a seller makes a bid in response to a particular request, that bid is destined for a single recipient (the buyer who made the request). A buyer can press a button at any time to view all bids that have arrived for a certain request. Since the order of the bids isn’t important, we store them in a bag data structure, and since no one else needs to view them, the buyer will remove them from the bag.

Similarly, when a buyer accepts a bid, the acceptance message is destined just for the seller and no one else. The seller can press a button at any time to see all the bids that have been accepted. Again, we use a bag data structure to store bid acceptances, and the seller removes them from the bag when viewing them.

Two bags and a channel distributed data structure form the basis of the application and support all the necessary buyer/seller interaction. Let's look at the details of the buyer and seller applet code.

6.3.5 The Buyer

Here is the code for our `Buyer` applet. It sets up a user interface for the buyer and allows a buyer to request bids from sellers, view bids, and accept bids:

```
public class Buyer extends Applet
    implements ActionListener, Runnable
{
    //... variables for user interface components go here
    //... variables for car & bid info. go here

    private Thread displayerThread;
    private JavaSpace space;

    public void init() {
        space = SpaceAccessor.getSpace();
        buyerName = getParameter("name");
        displayerThread = null;
        setupUserInterface();
    }

    private void setupUserInterface() {
        //... code to set up user interface
    }

    // A button was pressed:
    public void actionPerformed(ActionEvent event) {
        Object object = event.getSource();

        // "Request Bids" button pressed:
        if (object == requestButton) {
            action = REQUEST_BIDS;
            if (processFormFields() == SUCCESS) {
                BidEntry item = new BidEntry("requests",
                    null, buyerName, yearNum, make, model,
                    exteriorColor, interiorColor, maxPriceNum);
                append("requests", item, durationNum);
            }
        }
        return;
    }
}
```

```
    }

    // "View Bids" button pressed:
    if (object == viewButton) {
        action = VIEW_BIDS;
        if (processFormFields() == SUCCESS) {
            if (displayerThread == null) {
                displayerThread = new Thread(this);
                displayerThread.start();
            }
        }
        return;
    }

    // "Accept Bid" button pressed:
    if (object == acceptButton) {
        action = ACCEPT_BID;
        if (processFormFields() == SUCCESS) {
            BidEntry item = new BidEntry();
            item.label = "accepted-bid";
            item.idNum = new Integer(idNum);
            item.participant = sellerName;
            acceptBid(item, sellerName, idNum);
        }
        return;
    }
    showStatus("Invalid button press.");
}

private int processFormFields() {
    //... makes sure user has filled in required fields
    //... then extracts info & assigns it
    //... to variables
}

public void run() {
    showStatus("running");
    takeAndDisplayBids(idNum);
    showStatus("DONE.");
    displayerThread = null;
}
```

```
private void append(String channel, BidEntry item,
                   int durationMinutes)
{
    Integer itemNum = getItemNumber(channel);
    item.idNum = itemNum;
    int duration = durationMinutes * 60000; // in ms

    showStatus("Submitting item " + itemNum +
               " for sale to channel \\" + channel + "\\...");
    try {
        space.write(item, null, duration);
    } catch (Exception e) {
        showStatus("Error writing to the channel.");
        e.printStackTrace();
        return;
    }
    idTextField.setText(itemNum.toString());
    showStatus("Submitting request " + itemNum + " for bid " +
               " to channel \\" + channel + "\\... Done.");
}

private Integer getItemNumber(String channel) {
    //... return sequence # of next item to go in channel
}

private void takeAndDisplayBids(int idNum) {
    BidEntry item = null;
    BidEntry bidTemplate = new BidEntry();
    bidTemplate.label = "bids";
    bidTemplate.idNum = new Integer(idNum);
    String displayText = "";
    String newline = System.getProperty("line.separator");

    showStatus("Looking up bids for request ID # " +
               idNum + "\\...");
    bidOutput.append(newline);

    // take & display bids for this idNum, until no more found:
    do {
        try {
            item = (BidEntry)space.takeIfExists(bidTemplate,
                                                 null, 0);
            if (item != null) {
                displayText += item.toString() + newline;
            }
        } catch (Exception e) {
            showStatus("Error reading bids for ID " + idNum);
            e.printStackTrace();
            return;
        }
    } while (item != null);
    bidOutput.append(displayText);
}
```

```

        if (item != null) {
            displayText = item.print();
            bidOutput.append(displayText + newline);
        }
    } catch (Exception e) {
        showStatus("Error occurred obtaining item " +
                   idNum);
        e.printStackTrace();
    }
} while (item != null);
bidOutput.append("----- DONE -----" + newline);
showStatus("Retrieved all the bids currently available.");
}

private void acceptBid(BidEntry item, String seller,
                      int idNum)
{
    showStatus("Accepting " + seller
              + "'s bid on request " + idNum + "...");
    try {
        space.write(item, null, Lease.FOREVER);
    } catch (Exception e) {
        showStatus("An error occurred writing the bid.");
        e.printStackTrace();
        return;
    }
    showStatus("Accepting " + seller +
              "'s bid on request " + idNum + "...Done.");
}
}
}

```

As usual, the `init` method gains access to a space and retrieves information from applet parameters (in this case, the buyer's name), and calls a `setupUserInterface` method that takes care of creating and displaying the buyer's user interface.

The `actionPerformed` method is called whenever the buyer presses the “Request Bids,” “View Bids,” or “Accept Bid” buttons. In each case, we record the kind of button press in the `action` variable and call `processFormFields`, a method that determines if all the required fields for that action have been filled in and are appropriate (for example, a car's “year” must be an integer) and prints an error message if anything is amiss. If the fields pass the tests, `processForm-`

Fields returns the constant SUCCESS, and actionPerformed continues to service the button press.

If the buyer is requesting a bid, a new BidEntry is constructed—with “requests” in the label field, the buyer’s name in the participant field, and other fields filled in with car information from the buyer’s form—and passed to an append method, along with the duration (in minutes) for this bid. The append method calls getItemNum to determine the sequence number of the next item to be added to the tail of the “requests” channel, sets the idNum field of the bid entry to that number, and then writes the entry into the channel. Note the lease time given in the third argument to write; it specifies the number of milliseconds the bid will exist in the space before automatically expiring. By using lease times, buyers can conveniently have their old requests disappear after a certain amount of time.

If the buyer wishes to view bids on a particular request, actionPerformed starts up a new displayThread that executes the run method, which in turn calls takeAndDisplayBids. Since retrieving and viewing bids can potentially be time-consuming, it’s best for the retrieval and display to run within its own thread (then other parts of the interface will still be able to respond to user interaction). The takeAndDisplayBids method sets up a BidEntry template, filling in the label field with “bids” and the idNum field with the number identifying the request. Then the method loops, removing matching bids using takeIfExists and displaying them, until no more bids can be found for the given request.

Finally, if the buyer pressed a button to accept a bid on a particular request, a BidEntry is constructed, this time with “accepted-bid” in the label field, the request number in the idNum field, and the seller’s name in the participant field. Then the acceptBid method is called, which writes the acceptance entry to the space. As we’ll see shortly, the seller may look for its bids that have been accepted.

6.3.6 The Seller

The code for our Seller applet sets up a user interface for the seller and allows a seller to view requests from buyers, make bids, and view bids that have been accepted. You’ll notice that the overall structure of the code is very similar to the Buyer applet:

```
public class Seller extends Applet
    implements ActionListener, Runnable
{
    //... variables for user interface components go here
    //... variables for car & bid info. go here
```

```
private Thread displayThread;
private JavaSpace space;
private int lastRead = 0;

public void init() {
    space = SpaceAccessor.getSpace();
    sellerName = getParameter("name");
    displayThread = null;
    setupUserInterface();
}

private void setupUserInterface() {
    //... code to set up user interface
}

// A button was pressed
public void actionPerformed(ActionEvent event) {
    Object object = event.getSource();

    // "View Requests" button pressed
    if (object == viewRequestsButton) {
        action = VIEW_REQUESTS;
        if (processFormFields() == SUCCESS) {
            if (displayThread == null) {
                displayThread = new Thread(this);
                displayThread.start();
            }
        }
        return;
    }

    // "Make a Bid" button pressed
    if (object == bidButton) {
        action = MAKE_BID;
        if (processFormFields() == SUCCESS) {
            BidEntry item = new BidEntry("bids",
                new Integer(idNum), sellerName,
                yearNum, make, model,
                exteriorColor, interiorColor, bidNum);
            addBid(item, idNum, durationNum);
        }
    }
}
```

```
    }

    // "View Accepted Bids" button pressed
    if (object == acceptedBidsButton) {
        action = VIEW_ACCEPTED;
        if (displayThread == null) {
            displayThread = new Thread(this);
            displayThread.start();
        }
        return;
    }
    showStatus("Invalid button press.");
}

private int processFormFields() {
    //... makes sure user has filled in required fields
    //... then extracts info & assigns it
    //... to variables
}

public void run() {
    showStatus("running");

    if (action == VIEW_REQUESTS) {
        readAndDisplayRequests();
    } else { // action is VIEW_ACCEPTED
        takeAndDisplayAcceptedBids();
    }

    showStatus("DONE.");
    displayThread = null;
}

private void readAndDisplayRequests() {
    String newline =
        System.getProperty("line.separator");
    BidEntry item = null;
    BidEntry template = new BidEntry("requests", null, null,
        yearNum, make, model, exteriorColor, interiorColor,
        null);
```

```
// traverse lastRead to tail messages in the channel
int tail = readTailNumber("requests").intValue();
for (int sequence = lastRead; sequence <= tail;
     sequence++)
{
    showStatus("Looking for request " + sequence + " ...");

    try {
        template.idNum = new Integer(sequence);
        // see if next entry in channel matches
        item = (BidEntry)
            space.readIfExists(template, null, 0);
    } catch (Exception e) {
        e.printStackTrace();
    }

    // append text to the requests text area
    if (item != null) {
        String displayText = item.print();
        requestsOutput.append(displayText + newline);
    }
}
lastRead = tail + 1;
}

private Integer readTailNumber(String channel) {
    //... code that returns the sequence # of the last item in
    //... the channel (-1 if the channel is empty)
}

private void takeAndDisplayAcceptedBids() {
    BidEntry item = null;
    BidEntry bidTemplate = new BidEntry();
    bidTemplate.label = "accepted-bid";
    bidTemplate.participant = sellerName;
    String displayText = "";
    String newline = System.getProperty("line.separator");

    showStatus("Looking up my accepted bids...");
    acceptedBidsOutput.append(newline);
```

```

// take & display your accepted bids, until no more found:
do {
    try {
        item = (BidEntry)
            space.takeIfExists(bidTemplate, null, 0);
        if (item != null) {
            acceptedBidsOutput.append("Bid on ID " +
                item.idNum + " accepted." + newline);
        }
    } catch (Exception e) {
        showStatus("Error getting accepted bid.");
        e.printStackTrace();
    }
} while (item != null);
acceptedBidsOutput.append("----- DONE -----" +
    newline);
showStatus("Retrieved all the accepted bids currently " +
    "available.");
}

private void addBid(BidEntry item, int idNum,
    int durationMinutes)
{
    showStatus("Submitting bid on request " + idNum + "...");
    int duration = durationMinutes * 60000;

    try {
        space.write(item, null, duration);
    } catch (Exception e) {
        showStatus("An error occurred writing the bid.");
        e.printStackTrace();
        return;
    }
    showStatus("Submitting bid on request " +
        idNum + "... Done.");
}
}

```

The `init` method is identical to the buyer's `init` method, except it extracts the seller's name from the applet parameters. Again, the `actionPerformed` method responds to button presses ("View Requests," "Make a Bid," or "View Accepted

Bids”), recording the kind of button press in the `action` variable and calling `processFormFields` to error-check input fields.

If the seller wishes to view requests or accepted bids (which could take arbitrarily long to retrieve and print, depending on how many there are), a new `displayThread` is constructed and the `run` method is called. The `run` method calls either `readAndDisplayRequests` or `takeAndDisplayAcceptedBids`, depending on the button pressed.

The `readAndDisplayRequests` code is a variant on channel reading code we’ve seen before. The method sets up a `BidEntry`, setting the `label` field to “requests” and the car information fields to whatever the seller specified in the form (`null` if they were left blank). The `idNum`, `participant`, and `price` fields of the template are assigned `null`. Then the method calls `readTailNumber` to determine the sequence number of the last item in the “requests” channel and assigns it to `tail`. Next, `readAndDisplayRequests` enters a loop that tries to read matching entries in the channel: Each time through the loop, the `idNum` field of the template is set to the next number in the sequence, from `lastRead` (where we stopped reading after the last “View Requests” button press) to `tail`. Since `readIfExists` is used, if the next entry in the sequence does not match the kind of car specified by the seller, we won’t retrieve or display it.

The `takeAndDisplayAcceptedBids` methods is similar to the buyer’s `takeAndDisplayBids`: it sets up a `BidEntry` template, filling in the `label` field with “accepted-bids” and the `participant` field with the seller’s name. Then the method loops, removing matching bids using `takeIfExists` and displaying them, until no more accepted bids can be found for this seller.

Finally, if the seller wishes to make a bid on a particular request, a `BidEntry` is constructed with the `label` “bids,” `idNum` set to the number of the request, `participant` set to the seller’s name, and all car details and bid price fields filled in. Then, `addBid` is called to write the bid entry to the space, for a specified lease time. Just as buyers do with requests, sellers can limit the amount of time a bid is valid; after the lease time it simply disappears from the space for good.

6.3.7 Running the Marketplace

At this point, you’re in good shape to set up and experiment with the automobile marketplace. You must run a `CreateChannels` applet to initialize the “requests” channel. You can start up any number of buyers and sellers:

```
<applet code="Buyer" width=500 height=430>
    <param name="name" value="duke@sun.com">
</applet>
```

```
<applet code="Seller" width=500 height=530>
    <param name="name" value="ken@sun.com">
</applet>
```

As you experiment, you will begin to see that the marketplace pattern is generalizable to any situation in which goods or services are exchanged. The useful variants of the pattern are limited only by your imagination and the requirements of the particular trading environment.

6.4 Other Patterns

So far in this chapter we've explored three common patterns, each of which was simple, powerful, and quite general. The replicated-worker pattern isn't geared to any particular problem domain, but is applicable to a large class of compute-intensive problems. Likewise, the command pattern can be used as the basis for a generic compute server, ready to take on any kind of problem we throw at it. The marketplace pattern is suitable for any domain in which there's a resource being bought and sold and parties are trying to find the best deal.

These three very general patterns are just a starting point to the application patterns you can devise. Other patterns tend to be more ad hoc or specialized than the ones we've already seen, and demonstrating interesting examples of how they can be used in particular domains would require a great deal of code. So, in this section, we'll stick to descriptions and leave the implementation for your own experimentation. Here are some ideas to get you started.

6.4.1 Specialist Patterns

The *specialist pattern* is in many ways the opposite of the replicated-worker pattern. Rather than using a group of workers that are exactly the same, the specialist pattern uses workers that are each specialized to perform a particular task. Consider building a house using a replicated-worker pattern. You'd assemble a set of workers and each would pitch in to do whatever was needed. All workers would collectively pour the foundation, then frame the house, and finally put on the roof. Houses are more commonly built using a specialist pattern, with each worker taking and performing tasks suited for his expertise: one set of workers pours the foundation, another set frames the house, expert roofers put on the roof, electricians do the wiring, and so on.

One type of specialist pattern is the *blackboard pattern*, based on the metaphor of a blackboard that contains data which can be added to, read, and erased.

With this pattern, a group of specialists (with expertise in different parts of the problem) continually observe the data repository for items of interest and “raise their hands” when they want to add, erase, or update data. Typically a “teacher” chooses an expert to make a modification, and then decides if the state of the blackboard represents a solution yet or not; if not, the process continues. The state of the blackboard keeps progressing until a solution is found. Working together, the experts arrive at a heuristic solution to a problem that none of them could solve single-handedly. For more in-depth discussion of the blackboard pattern, you can refer to Grady Booch’s *Object-Oriented Analysis and Design With Applications* or Buschmann, et al.’s *Pattern Oriented Software Architecture: A System of Patterns*.

Another type of specialist pattern is a *trellis pattern*, consisting of a hierarchical graph of processes. To use a trellis framework for solving a large problem, we need to break the problem into low-level and high-level pieces. Consider the problem of a company that sells many products through web advertising and wants a constant overview of how well the effort is going for various products and for various places the advertisements appear. Low-level processes might collect and track low-level data. For example, for every web page an advertisement appears on, low-level processes would track how many hits the page is getting, how many “click-throughs” to an online order form are occurring, and how many actual sales are being generated. High-level processes would concentrate on providing overviews of how well sales of a particular product are doing in a certain geographic area, or in response to a particular new advertisement, or to advertisements placed in certain sets of web pages. Mid-level processes would concentrate on recognizing trends, for instance that advertisements on a particular set of web pages have been decreasing in popularity.

Once we’ve broken a problem up into low-, mid-, and high-level pieces, we arrange the components into a hierarchical “information-flow” graph. Data flows into the low-level processes at the bottom of our graph, filters upward through the mid-level processes, and emerges as a birds-eye analysis by the high-level processes sitting at the top (with an advertising executive no doubt monitoring the results). Carriero and Gelernter’s *How to Write Parallel Programs: A First Course* presents the trellis architecture in depth.

6.4.2 Collaborative Patterns

Many applications can be described as *collaborative patterns*. The multiuser chat example from the last chapter was a simple example. We’ll spend all of Chapter 10 building a more advanced interactive “messenger service,” similar to many commercially available applications that allow you to monitor a group of online friends and chat with them. *Workflow patterns* are another specific kind of collaborative pattern: when one person finishes with a piece of information, it

must be copied to one or more other people, and when they finish their processing on it, they route it to others. Collaborative patterns tend to be highly specific to the domain and the organization in which they're used, so it's difficult to abstract out a general framework. Most collaborative patterns are custom-built for the use at hand.

6.5 Summary

In this chapter we've investigated some tried-and-true frameworks for space-based applications that are simple, powerful, and quite general. First, we studied the replicated-worker pattern, which is appropriate for solving computational problems that can be broken down into a number of smaller, independent, nearly identical sub-problems. A master generates tasks and collects results, while workers repeatedly pick up tasks and compute results. The framework is well suited to building parallel applications that speed up solutions to a large class of compute-intensive problems. The second pattern we explored, the command pattern, represents a powerful way of using spaces to pass around object behavior. The pattern allows us to create a generic compute engine that is ready to take on any kind of tasks we throw at it, without needing to know anything about their specifics. The final application pattern we investigated in detail was the marketplace pattern, which provided a general framework for producers and consumers of some resource to interact to find the best deal. The resource may be any product or service that can be bought and sold—computing cycles, electric power, cars, or anything else.

Beyond these three patterns, we gave a brief overview of some more specialized and ad hoc application patterns, such as blackboards, trellises, and collaborative patterns. The patterns of this chapter, while not exhaustive, give you a foundation for starting to invent and develop your own. As the JavaSpaces technology matures and programmers develop more real-world applications with it, the body of useful patterns will no doubt grow. Please check this book's web site at <http://java.sun.com/docs/books/jini/javaspaces> for patterns contributed by readers.

6.6 Exercises

Exercise 6.1 Find a compute-intensive computation that is appropriate for the replicated-worker pattern (such as ray tracing or computational physics) and implement it.

Exercise 6.2 Rewrite the Mandelbrot application so that it makes use of the command pattern.

Exercise 6.3 Our marketplace example could be “reversed” so that sellers make requests (they have a specific item to sell, and are requesting a minimum price for it), and interested buyers come along and make bids. How hard would it be to reverse the marketplace in this way?

Exercise 6.4 One limitation of the marketplace example is that it supports only a single bid from a given seller per bid request (since there is no way to distinguish between two bids for the same request when accepting a bid). As a result, a seller can’t give two options of cars for sale that are close to what a buyer wants. How could you augment the code to overcome this limitation?

Leases

The biggest difference between time and space is that you can't reuse time.

—Merrick Furst

LEASING provides a manner of allocating resources for a fixed period of time, after which the resource can be freed. This model is beneficial in the distributed environment, where partial failure can cause holders of resources to fail or become disconnected from the resources before they explicitly free them. In the absence of a leasing model, resources could grow without bound.

Spaces allocate resources that are tied to *leases*. For instance, when you write an entry into a space it is granted a lease that specifies a period of time for which the space guarantees (to the best of its abilities) that it will store the entry. The holder of the lease may renew or cancel the lease before it *expires*. If the lease holder does neither, the lease simply expires, and the space removes the entry from its store.

In this chapter we will learn how to manipulate and manage the leases created from writing entries into a space. The techniques you learn will be applicable to two other areas of space-based programming that make use of leases, namely distributed events and transactions, which we will cover in the next two chapters.

7.1 Leases on Entries

When we call `write`, we supply a `lease` parameter that specifies the amount of time we'd like the space to store the entry. Recall the signature of the `write` method, which we discussed in Chapter 2:

```
Lease write(Entry e, Transaction txn, long lease)
throws RemoteException, TransactionException;
```

The `write` method returns a `Lease` object that contains the actual lease time the entry was granted—the space may grant the full time requested, or a reduced time. In either case, when the lease time expires, the space will remove the entry from the space. If the space is unable to grant a lease at all, the `write` method will throw a `RemoteException`.

So far in this book, we've been supplying `Lease.FOREVER` as the lease time, which requests that an entry be stored in the space indefinitely. In practice, specifying that every entry should live indefinitely may be wasteful of resources, and in general we can't assume that spaces are willing to grant indefinite leases. As an alternative, we can specify a lease time of `Lease.ANY`, which leaves it up to the space to choose the length of the lease. Or, we can request a specific duration in milliseconds, as we do here:

```
try {
    long duration = 1000 * 60 * 5; // 5 minutes
    Lease lease = space.write(entry, null, duration);
} catch (Exception e) {
    //... handle exception here
}
```

Here we've written `entry` into the space and requested a lease time of five minutes. The actual lease is represented in the `Lease` object returned from the call.

You may be wondering why lease times are requested as a duration, rather than an absolute date and time. This is because clocks of computer systems often differ, even by as much as a few minutes. Using durations allows your process and the remote space to have close to the same idea of when a lease will expire. If absolute times were used instead, each participant's notion of when the lease expires might differ by as much as a few minutes. Using durations assumes that time passes at the same rate between computers; although there's no absolute guarantee of this, it's a better bet than using absolute times.

Communication lag between the computers requesting and granting a lease can also lead to different notions of a lease's expiration time. For instance, if a remote space grants a lease duration y , but it takes two seconds for the granted lease to make its way back to the calling process, then the caller thinks the lease has y milliseconds remaining, while there are really only $y - (1000 * 2)$ milliseconds remaining. It's important to note, though, that lease times in space-based programming are expected to be on the order of tens of seconds to hours (or more). In general, then, the time lag from a lease's creation to its arrival at the calling process will be small relative to the lease duration, and therefore shouldn't pose a problem in your programs.

7.2 The Lease Object

A Lease object must implement the Lease interface, presented in its entirety in Figure 7.1, which contains a set of methods for dealing with leases. In discussing the interface, we will ignore a few constants and methods that are not typically used in space-based programming; the full details of leasing can be found in the *Jini™ Distributed Leasing Specification*.

```
package net.jini.lease;
import java.rmi.RemoteException;

public interface Lease {
    long FOREVER = Long.MAX_VALUE;
    long ANY = -1;
    int DURATION = 1;
    int ABSOLUTE = 2;

    long getExpiration();

    void cancel()
        throws UnknownLeaseException,
               RemoteException;

    void renew(long duration)
        throws LeaseDeniedException,
               UnknownLeaseException,
               RemoteException;

    void setSerialFormat(int format);

    int getSerialFormat();

    LeaseMap createLeaseMap(long duration);

    boolean canBatch(Lease lease);
}
```

Figure 7.1 The Lease interface.

7.2.1 Lease Expiration

Every lease object provides a `getExpiration` method that you can use to determine the expiration time of the lease. For instance, consider the following code:

```
Lease lease = space.write(entry, null, Lease.ANY);
long expirationTime = lease.getExpiration();
```

Here, we first call `write` with a lease length of `Lease.ANY`, which means we let the space choose the lease duration. Next we call `getExpiration`, which returns the lease's expiration time (in the fashion of the Java platform, measured in milliseconds since midnight, January 1, 1970).

The time returned from `getExpiration` is measured according to the local clock of the calling process (not the remote space's clock). This means that we can determine the time remaining in a lease by calling the Java platform's `System.currentTimeMillis` method (which returns the current local machine time in milliseconds) and compare its result to the value returned from `getExpiration`. Here is an example:

```
long currentTime = System.currentTimeMillis();
long delta = expirationTime - currentTime;
```

Here, `delta` contains the number of milliseconds before the lease will expire.

7.2.2 Renewing a Lease

To extend a lease's expiration time, we call its `renew` method, which is defined as follows:

```
void renew(long duration)
throws LeaseDeniedException, UnknownLeaseException,
RemoteException;
```

The `renew` method takes one parameter that specifies the amount of time (in milliseconds) desired for the new lease. If the renewal is granted for that duration, the existing lease object's expiration time is updated to expire after `duration` milliseconds. For instance, if we request a duration of 60,000 milliseconds on a lease that has 10,000 milliseconds remaining, then the updated lease expiration will reflect 60,000 milliseconds remaining (not 10,000 plus 60,000 milliseconds). It's important to point out that there is no guarantee a renewal request will be granted at all. If a renewal is granted, the space determines the actual lease time, which may be shorter than the time we requested.

The `renew` method can throw three exceptions: `LeaseDeniedException`, `UnknownLeaseException`, and `RemoteException`. A `LeaseDeniedException`

is thrown if the space cannot renew the lease at all. The reasons a lease might be denied are dependent on the implementation and runtime characteristics of the space; for instance, a space might lack the resources (such as disk space) to renew the lease at that time. When a `LeaseDeniedException` occurs, the lease is left unchanged, and has the expiration time it had before the `renew` call. The `UnknownLeaseException` is thrown if the lease object isn't known to the space. This can happen if the lease has already expired, or has been cancelled (as we'll see in the next section) or if the entry has been taken from the space by another process. Finally, the `RemoteException` is thrown when a communication problem occurs between the process and the remote space.

Here is an example of how we would request a renewal of an existing lease on an entry:

```
try {
    long duration = 1000 * 60;      // 1 minute
    lease.renew(duration);
    System.out.println("lease now ends at " +
        new Date(lease.getExpiration()));
} catch (LeaseDeniedException e) {
    // handle exception here
} catch (UnknownLeaseException e2) {
    // handle exception here
} catch (RemoteException e3) {
    // handle exception here
}
```

Here, we request that a `lease` object be renewed for one minute, in effect asking to extend the entry's lifetime in the space. To see if the renewal request was granted for the full duration, we print the lease's new expiration time.

7.2.3 Cancelling a Lease

A lease can be terminated before the end of its lease time by calling its `cancel` method, which has the following signature:

```
void cancel() throws UnknownLeaseException, RemoteException;
```

The `cancel` method may throw two exceptions: `UnknownLeaseException` and `RemoteException`. The `UnknownLeaseException` is thrown when the lease has already been cancelled, or has expired. The `RemoteException` is thrown whenever there is a communication problem between the calling process and the remote space.

Here's how we would cancel an existing lease:

```

try {
    lease.cancel();
} catch (UnknownLeaseException e) {
    // handle exception here
} catch (RemoteException e2) {
    // handle exception here
}

```

In the case of a leased entry, calling `cancel` terminates the lease and results in the entry's removal from the space—just as if the lease had expired.

7.3 Lease Maps

Applications often need to manage many leases. *Lease maps* provide a convenient means of maintaining a collection of leases and renewing or cancelling all of them with one operation.

7.3.1 Creating a Lease Map

To create a `LeaseMap` object, you call the `createLeaseMap` method on a `Lease` object. Here is the signature of the method:

```
LeaseMap createLeaseMap(long duration);
```

This method creates and returns a new lease map—a container for a set of leases, each of which has an associated lease renewal time. The method takes the duration parameter, pairs it with the lease the method was called on, and adds the lease/duration pair to the new lease map.

For example, consider the following code:

```
long duration = 1000 * 60;      // 1 minute
LeaseMap myLeaseMap = lease.createLeaseMap(duration);
```

Here, we've created a new lease map, `myLeaseMap`, which initially contains `lease` with an associated duration of one minute. When a lease map is renewed, an attempt is made to renew all the leases it contains for their associated durations. If we choose to renew `myLeaseMap`, a renewal request of one minute will be made for the `lease` object.

To create a lease map, you can call the `createLeaseMap` method on any of the existing leases you wish to batch together—the lease and its associated duration become the first lease/duration pair in the map. Once you have instantiated a `LeaseMap`, you are free to add as many other lease/duration pairs as you'd like.

7.3.2 Adding and Removing Leases

Figure 7.2 shows the `LeaseMap` interface, which itself extends the `java.util.Map` interface from the core Java API. A Map is an object that maps keys to values (in this case, leases to durations). Lease maps support the Map interface's `put` method, which can be used to add a lease/duration pair to a lease map:

```
Long leaseDuration = new Long(1000 * 60 * 4); // 4 minutes
try {
    myLeaseMap.put(lease2, leaseDuration);
} catch (IllegalArgumentException e) {
    // handle exception here
}
```

Here, we pass the method `put` a lease `lease2` and a renewal duration of four minutes, and the method adds the lease/duration pair to the lease map. An `IllegalArgumentException` will be thrown if you try to put an object that's not a `Lease` or pass a duration that's not a `Long`.

A lease can be added to the lease map only if it can be renewed (or cancelled) with the other leases in the lease map. Leases granted from different services—different spaces, for instance—may not always be renewable as a unit, depending on the implementations of those services. Typically, leases in the same space can be batched together. The `put` method will throw an `IllegalArgumentException` if its lease parameter can't be batched with the other leases in the map.

To check if a lease can be batched in a lease map, we use the `canContainKey` method of the `LeaseMap` interface, shown in Figure 7.2. The method takes a lease as a parameter and returns true if the lease can be added to the `LeaseMap`, otherwise it returns false. Here's an example of doing this check:

```
if (myLeaseMap.canContainKey(lease2)) {
    //... try/catch clause omitted
    myLeaseMap.put(lease2, leaseDuration);
}
```

In this code fragment, we add `lease2` to `myLeaseMap` only if the call to `canContainKey` determines that `lease2` can be batched with the leases already in the map.

Alternately, the `canBatch` method, which operates on a lease and takes a second lease as a parameter, can be used to determine if leases can be batched together for renewals and cancellations:

```
if (lease.canBatch(lease2)) {
    //... try/catch clause omitted
    myLeaseMap.put(lease2, leaseDuration);
}
```

```

package net.jini.lease;
import java.rmi.RemoteException;

public interface LeaseMap extends java.util.Map {
    boolean canContainKey(Object key);

    void renewAll()
        throws LeaseMapException,
               RemoteException;

    void cancelAll()
        throws LeaseMapException,
               RemoteException;
}

```

Figure 7.2 The LeaseMap interface.

The `canBatch` method is an equivalent mechanism to `canContainKey`: The former determines only if two leases can be batched, while the latter determines if a lease can be batched with the (potentially many) leases already in a lease map.

Now that we know how to create lease maps and add leases to them, let's look at how we can renew or cancel all the leases in a map in a single operation.

7.3.3 Renewing Leases

Once we've placed one or more leases into a lease map, we can perform two operations on the map: `renewAll` and `cancelAll`, shown in Figure 7.2. The `renewAll` method attempts to renew each lease in the map for its associated duration. Here is an example of a call to `renewAll`:

```

try {
    myLeaseMap.renewAll();
} catch (LeaseMapException e) {
    // handle exception here
} catch (RemoteException e) {
    // handle exception here
}

```

If all the leases in the map are renewed, `renewAll` completes without throwing any exceptions, and each lease object reflects its new expiration time. If one or more of the leases can't be renewed, `renewAll` throws an exception of type

```
package net.jini.lease;
import java.util.Map;

public class LeaseMapException extends LeaseException {
    public Map exceptionMap;

    public LeaseMapException(String s, Map exceptionMap) {
        super(s);
        this.exceptionMap = exceptionMap;
    }
}
```

Figure 7.3 The LeaseMapException class.

LeaseMapException, defined in Figure 7.3. In addition, for any lease that can't be renewed, the lease is removed from the lease map and added to the exception's exceptionMap field. The exceptionMap object associates Lease objects (the ones that couldn't be renewed) to Exception objects (the reasons for each failure). For example, if a lease couldn't be renewed because it had already expired, it would be mapped to an UnknownLeaseException in the exceptionMap object.

Applications may process the exception map in order to take appropriate action for each renewal failure. For instance, it may make sense to try the renewal again if the failure was due to a LeaseDeniedException, but not if the failure was due to an UnknownLeaseException. We'll return to the idea of processing an exception map in Section 7.4.7.

The renewAll method might also throw a RemoteException, since lease renewals involve communication between the calling process and the remote space(s) that govern the leases.

7.3.4 Cancelling Leases

To cancel all the leases in a lease map, we call cancelAll, shown in the LeaseMap interface in Figure 7.2. Here's an example in which we call cancelAll to cancel the leases in myLeaseMap:

```
try {
    myLeaseMap.cancelAll();
} catch (LeaseMapException e) {
    // handle exception here
} catch (RemoteException e) {
    // handle exception here
}
```

The `cancelAll` method attempts to cancel each lease in the map. The method operates in the same manner as `renewAll`: If all the leases are cancelled successfully, `cancelAll` completes without throwing any exceptions. If one or more leases can't be cancelled, the method removes them all from the lease map and throws a `LeaseMapException` that contains those leases in its `exceptionMap` field. Once again, the `exceptionMap` object maps each `Lease` object that couldn't be cancelled to an `Exception` object that represents the reason for failure. The method will throw a `RemoteException` in the event there are communication problems with the remote space(s) during lease cancellations.

7.4 Automated Lease Renewal

Often a lease or a set of leases needs to be renewed repeatedly, either because you don't know *a priori* how long you'll need a particular resource, or because a space may not be willing to grant you a resource for the entire time you request. You can deal with both cases by requesting a lease for some initial length of time and then continually renewing the lease until you're finished with the resource.

This process can be automated by using a *lease renewal manager*. A lease renewal manager administers one or more leases, renewing them as necessary up to some specified stopping time. The lease manager pattern is common enough that the `com.sun.jini.lease.LeaseRenewalManager` class provides an implementation. In the remainder of this section, we are going to implement our own lease manager that is similar to (although a bit simpler than) the Jini technology lease manager. Going through the exercise of implementing a lease manager is a great way to tie together all of the concepts we've covered in this chapter, and it may also prove useful if you need to construct your own specialized lease manager.

7.4.1 The LeaseManager Interface

Our lease manager implements the following interface, which provides a set of public constructors and methods that closely mirror those of the `com.sun.jini.lease.LeaseRenewalManager`:

```
public interface LeaseManager {  
    public void renewUntil(Lease lease, long stopTime,  
                          LeaseListener listener);  
  
    public void renewFor(Lease lease, long duration,  
                        LeaseListener listener);
```

```
public void cancel(Lease lease)
    throws UnknownLeaseException, RemoteException;
}
```

Two methods are supplied to add leases to the manager: `renewUntil` takes a lease, an absolute stop time, and a listener and adds that lease to the manager. The manager will automatically keep renewing the lease up to the stop time and will notify the listener if any renewals fail. The `renewFor` method is similar, except that it takes a duration time rather than an absolute stop time. Last, the `cancel` method takes a lease object and asks the lease manager to cancel its lease. A call to `cancel` throws an `UnknownLeaseException` if the manager doesn't know about the lease, or if the grantor of the lease no longer holds the resource, and throws a `RemoteException` if a communication error occurs during the lease cancellation.

7.4.2 Implementing the Constructors

We are now going to define an `AutomatedLeaseManager` class that implements the `LeaseManager` interface. We'll build the class incrementally, starting with its private fields and two constructors:

```
public class AutomatedLeaseManager implements LeaseManager {
    private long granularity = 1000 * 60 * 5; // five minutes
    private Hashtable leases = new Hashtable();

    public AutomatedLeaseManager() {
        RenewThread renewer = new RenewThread();
        renewer.start();
    }

    public AutomatedLeaseManager(long granularity) {
        this();
        this.granularity = granularity;
    }

    //... methods go here
}
```

The lease manager has two private fields: `granularity`, which is set to five minutes by default, and a `leases` hashtable that holds the list of leases being managed. We will see how both are used throughout the lease manager code.

The lease manager provides two constructors. The first, a no-arg constructor, creates a lease manager that is ready to accept and manage any number of leases. It does so by instantiating a `RenewThread` object from an inner class and starting it; as we'll see in detail in Section 7.4.5, the thread takes care of renewing the leases in the hashtable at fixed time intervals. The second constructor takes a granularity—the time interval (in milliseconds) at which the lease manager regularly checks to see if any leases need to be renewed. This constructor first calls the no-arg constructor to create the `RenewThread`, and then assigns the granularity time to the `granularity` field.

7.4.3 Adding Leases

Once we've instantiated an `AutomatedLeaseManager`, we can place leases under its management by calling either the `renewUntil` or `renewFor` methods, which are implemented as follows:

```
public void renewUntil(Lease lease, long stopTime,
    LeaseListener listener)
{
    addLease(lease, stopTime, listener);
}

public void renewFor(Lease lease, long duration,
    LeaseListener listener)
{
    addLease(lease, duration +
        System.currentTimeMillis(), listener);
}
```

The `renewUntil` method takes a lease, an absolute stop time up to which the lease will be renewed, and a listener object to be informed if any renewal failures occur. The `renewFor` method is a slight variant: For its second parameter, it takes a duration (instead of an absolute stop time) for the lease. Both methods make use of a helper method called `addLease`, which does the actual work of adding the lease to the manager. The `renewUntil` method simply passes its parameters to the `addLease` method (they have the same signatures). The `renewFor` method first adds the current time to the duration parameter (thus translating it into an absolute stop time) before calling `addLease`. In effect, both renew methods add their lease parameter to a hashtable that is used to hold all the leases under the lease renewal manager's control.

Here is the implementation of the `addLease` method that both `renew` methods make use of:

```
private void addLease(Lease lease, long stopTime,
    LeaseListener listener)
{
    if (stopTime == Lease.ANY || stopTime == Lease.FOREVER) {
        stopTime = Long.MAX_VALUE;
    }
    leases.put(lease, new LeaseInfo(stopTime, listener));
}
```

The `addLease` method takes a lease, an absolute stop time, and a listener and adds this information to the `leases` hashtable (a private field in the lease manager) in the form of a key/value pair. To add the information, we first see if the stop time is either `Lease.ANY` or `Lease.FOREVER`, and in either case set the stop time to be the maximum value representable by a long (in other words, the maximum date and time representable). We then create a `LeaseInfo` object, which is just a wrapper object to hold the stop time and listener, and then we put this object into the hashtable with the lease as its key.

7.4.4 Cancelling a Lease

We can ask the lease manager to cancel a lease by calling the `cancel` method of the `LeaseManager` interface. Here's how we implement the method:

```
public void cancel(Lease lease)
    throws UnknownLeaseException, RemoteException
{
    LeaseInfo info = (LeaseInfo)leases.get(lease);
    if (info != null) {
        lease.cancel();
        leases.remove(lease);
        return;
    } else {
        throw new UnknownLeaseException();
    }
}
```

To cancel a lease, we first retrieve it from the `leases` hashtable by passing the lease to the hashtable's `get` method. If `get` finds a matching key in the hashtable, it returns a corresponding non-null `LeaseInfo` object; in that case we invoke the lease's `cancel` method and remove the lease from the hashtable. On the other

hand, if `get` can't find a matching key, then the lease manager is not managing this lease, and we throw an `UnknownLeaseException`.

7.4.5 Renewing Leases

The most interesting part of the lease manager implementation is the automated lease renewal process. Recall that the `AutomatedLeaseManager` constructors instantiate and start a renewal thread that renews the leases stored in the hashtable at fixed time intervals. As we've mentioned, this renewal thread is instantiated from the `RenewThread` inner class of `AutomatedLeaseManager`:

```
private class RenewThread extends Thread {
    public RenewThread() {
        super("Renew Manager");
    }

    public void run() {
        System.out.println("RenewManager Running");
        for (;;) {
            checkLeases();
            try {
                Thread.sleep(granularity);
            } catch (InterruptedException e) {
                return; // we've been asked to stop
            }
        }
    }
}
```

The `RenewThread` extends `java.lang.Thread` and provides a `run` method that continually calls the `AutomatedLeaseManager`'s `checkLeases` method, sleeping for `granularity` milliseconds after each call.

The `checkLeases` method steps through all the leases in the lease manager's hashtable. Each lease is examined to see if its stop time has passed (in which case it can be removed), or if it needs to be renewed, or if it has enough time remaining that renewal can wait until the next call to `checkLeases`. For each lease that needs to be renewed, we add the lease to a lease map, and then we renew them all in a batch. If errors occur in the batch renewal, we notify the respective listeners of the failed renewals. Now that we have an overview of how automated lease renewals happen, let's take a closer look at how `checkLeases` operates.

7.4.6 Checking Leases

The `checkLeases` method is defined as follows:

```
private void checkLeases() {
    LeaseMap leasesToRenew = null;
    long now = System.currentTimeMillis();

    Enumeration enum = leases.keys();
    while (enum.hasMoreElements()) {
        Lease lease = (Lease)enum.nextElement();
        LeaseInfo info = (LeaseInfo)leases.get(lease);

        // if lease's stop time has passed, remove it:
        if (info.stopTime < now) {
            leases.remove(lease);
            continue;
        }

        // if enough time left in lease, leave it alone:
        long expires = lease.getExpiration();
        if (expires > (now + (granularity * 2))) {
            continue;
        }

        // not enough time left, so add lease to leasesToRenew map:
        if (leasesToRenew == null) {
            long duration = granularity * 2;
            leasesToRenew = lease.createLeaseMap(duration);
        } else {
            if (leasesToRenew.containsKey(lease)) {
                Long duration = new Long(granularity * 2);
                leasesToRenew.put(lease, duration);
            } else {
                info.listener.notify(
                    new LeaseRenewalEvent(this,
                        lease, info.stopTime,
                        new LeaseException("Not batchable")));
                leases.remove(lease);
            }
        }
    }
}
```

```

    if (leasesToRenew != null) {
        try {
            leasesToRenew.renewAll();
        } catch (LeaseMapException e) {
            processExceptionMap(e.exceptionMap);
        } catch (RemoteException e) {
            ; // hope things improve next time around
        }
    }
}

```

Let's start at the top and work our way through this code. We first declare two local variables: `leasesToRenew`, a lease map that will be used to hold the leases we are going to renew, and now, a `long` integer initialized to the current time via the `java.lang.System` class's `currentTimeMillis` static method.

We then enumerate over each lease that is stored in the `leases` hashtable. For each lease, we obtain a reference to the lease and its corresponding `LeaseInfo` object. Then we determine what to do with the lease—remove it from the lease manager's control, or renew it for some duration, or defer doing anything till the next call to `checkLeases`—according to the following algorithm.

If the stop time of the lease—the time until which the lease manager should renew it—has passed, we remove the lease from the hashtable and thus remove the lease from the renewal manager's control. Then we proceed to examine the next lease in the hashtable.

If the stop time hasn't passed, then we call `getExpiration` on the lease to determine when the space will expire the lease. The lease may or may not have enough time left that renewal can safely wait till the next time the `LeaseRenew` thread checks the lease. To determine whether the lease needs to be renewed right away, we check if the remaining lease time is less than the current time plus twice the granularity. If so, then we need to renew the lease.

Let's examine what this formula means. The granularity of the lease manager controls how often leases are examined to determine which ones need to be renewed (recall that the renewal thread sleeps for `granularity` milliseconds in between calls to `checkLeases`). Here our formula for whether a lease needs to be renewed is a heuristic that says if there is less time left in the lease than two times the granularity, then we need to renew the lease (otherwise we risk that the lease will expire before our next opportunity to renew it).

If our formula determines that the lease needs to be renewed, we add the lease to the `leasesToRenew` lease map. If the lease map doesn't already exist, we call `createLeaseMap` on the lease, which creates a new map and adds the lease to it. If the lease map already exists, we first need to call `canContainKey` (see Section 7.3.2) to make sure the lease can be batched with the existing leases in the

map. In either case, if we add the lease to the map, we add it with a duration of two times the granularity (for instance, if the granularity is five minutes, then the lease time will be ten minutes). If the lease can't be batched with the leases already in the map, then we notify the listener of the problem by throwing a `LeaseRenewalEvent` (described in Section 7.4.7) and remove the lease from the hashtable.

Once we've finished enumerating through all the leases, we invoke `renewAll` on the lease map. If all leases in the map are renewed without errors, then the `checkLeases` method returns. If a `LeaseMapException` is thrown, then a subset of the leases may have failed, and we must notify their listeners and also remove the leases from the `leases` hashtable; the `processExceptionMap` method is called to take care of both.

7.4.7 Processing Renewal Failures

The `processExceptionMap` method takes a `Map` parameter—the exception map field of a `LeaseMapException` that is thrown when the manager tries to renew the leases (see Figure 7.3)—which contains those leases that experienced renewal failures. Here's the code for the method:

```
private void processExceptionMap(Map map) {
    if (map == null) {
        return;
    }

    Enumeration enum = leases.keys();
    while (enum.hasMoreElements()) {
        Lease lease = (Lease)enum.nextElement();
        if (map.containsKey(lease)) {
            LeaseInfo info = (LeaseInfo)leases.get(lease);
            info.listener.notify(
                new LeaseRenewalEvent(this,
                    lease, info.stopTime,
                    (LeaseException)map.get(lease)));
            leases.remove(lease);
        }
    }
}
```

Since the `Map` interface does not support enumeration, we instead enumerate over every lease in our `leases` hashtable; for each lease that we find in the exception map (determined by a call to `containsKey`), we call `get` to extract its `LeaseInfo`

object. We then notify the lease's listener that a renewal problem has occurred; the `notify` method of the listener will be invoked and passed a `LeaseRenewalEvent`. Finally, we remove the lease from the `leases` hashtable.

The `LeaseRenewalEvent` class is defined as follows:

```
public class LeaseRenewalEvent extends java.util.EventObject {  
    private Lease lease;  
    private long expiration;  
    private Exception ex;  
  
    public LeaseRenewalEvent(LeaseManager source,  
        Lease lease, long expiration, Exception ex)  
    {  
        super(source);  
        this.lease = lease;  
        this.ex = ex;  
    }  
  
    public Lease getLease() {  
        return lease;  
    }  
  
    public long getExpiration() {  
        return expiration;  
    }  
  
    public Exception getException() {  
        return ex;  
    }  
}
```

`LeaseRenewalEvent` is an event object that holds information about an exception that occurred. In the case of `processExceptionMap`, for instance, the `LeaseRenewalEvent` is constructed to hold the lease manager that generated the event, the lease that wasn't successfully renewed, the manager's stop time for that lease, and the exception thrown by the renewal attempt. Recall that `checkLeases` also makes use of a `LeaseRenewalEvent`, if it can't batch a particular lease with the other leases it needs to renew.

7.4.8 Putting It All Together

We've now seen the complete code for the automated lease renewal manager. There is a bit of an art to writing lease renewal managers, as the policy for when,

and for how long, to renew leases is a heuristic rather than a well known algorithm. We encourage you to take the time to do Exercise 7.1, since the Jini software implementation of a lease renewal manager from Sun Microsystems, Inc. provides more sophisticated scheduling than we've shown in our example. Instead of waking up at fixed intervals, the manager makes a best guess at when it should wake up and examine the leases. It also does not follow a blanket policy for updating lease times, but instead provides a sliding scale that is dependent on the lease time—whether minutes, hours, or days—of the lease that is being renewed.

We will put our lease manager to good use later in the book; it will be one component of the collaborative application we build in Chapter 10.

7.5 Summary

In this chapter, we've introduced the use of leases in space-based programming. We've seen how to manipulate lease objects, how to create and manage lease maps that allow multiple leases to be treated as a unit, and how to implement a lease renewal manager. So far, the leases we have worked with were the result of writing entries into spaces. In the next two chapters, we'll see that leases come into play in two other areas of space-based programming as well—distributed events and transactions. We'll see that we can apply the same techniques we've learned here.

7.6 Exercises

Exercise 7.1 Study the `com.sun.jini.lease.LeaseRenewalManager` implementation to see the use of a slightly more sophisticated heuristic for renewing leases.

Distributed Events

If a listener nods his head when you're explaining your program, wake him up.
—Alan Perlis, **Epigrams in Programming**

THE `read` and `take` methods encourage a programming style whereby we explicitly read (or take) entries from a space, blocking if no such entry is yet available. Until now, we've ignored the `JavaSpace` interface's `notify` method, which allows us to react to the arrival of entries as they are placed in a space. Although `notify` won't allow you to write programs that can't already be written in terms of `reads` and `takes`, `notify` will provide you with another tool that can be used if your application is simpler to design or build using a reactive programming style. In addition, `notify` can be used to allow *non-space-based* programs to react to the arrival of entries in a space.

In this chapter you will first learn about the distributed event model—how it differs from the standard Java programming language event model and what guarantees can be made of events in the distributed environment. We will then take a look at a simple example that uses `notify`, and we will introduce the various classes and interfaces that come into play. By the end of this chapter you should have a grasp on how distributed events can be used with space-based programs.

8.1 Events in the Distributed Environment

Java provides a simple but powerful event model based on *event sources*, *event listeners* and *event objects* (see Figure 8.1). An *event source* is any object that “fires” an event. Events may be fired based on any internal state change in an object. For instance, an object representing a graphical button may fire an event when a user clicks on the button.

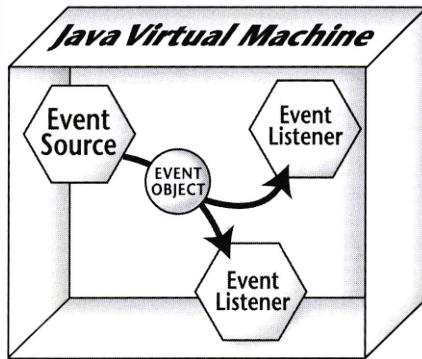


Figure 8.1 The Java programming language event model: event source, event object, and event listeners in a single JVM environment.

An *event listener* is an object that listens for events fired by an event source. Typically an event source provides a method whereby listeners can request to be added to a list of listeners. Whenever the event source fires an event, it notifies each of its registered listeners by calling a method on the listener object and passing it an *event object*.

Events in this model are delivered reliably and instantaneously. If a user clicks on “Button A” and then “Button B,” an object that’s registered to receive both button clicks will receive both events, and will never receive the second button click before the first. This guarantee is in large part due to the fact that events are occurring within one Java virtual machine (JVM).

Distributed events, on the other hand, must travel from one JVM to another JVM that may be running remotely over a network. Figure 8.2 illustrates how distributed events mirror the use of event sources, event listeners, and event objects. In the distributed environment the semantics of event delivery are not as straightforward because of the possibility of partial failure. Messages traveling from one JVM to another may be lost in transit, or may never reach their event listener. Likewise, an event may reach its listener more than once if the source can’t be sure the first event notification made it to the event listener. There is also a possibility two events may take different paths through the network, causing events to arrive out of order. In sum, it is difficult to make the same guarantees in the distributed environment that we take for granted in the single JVM model.

The lack of guarantees in the distributed event environment is present in the Jini™ Distributed Event model: Events may arrive multiple times, or may arrive out of order, or may not arrive at all. The distributed event API gives us a few tools to deal with these problems, but, as we will see, it will be your responsibility to ensure correctness in your applications.

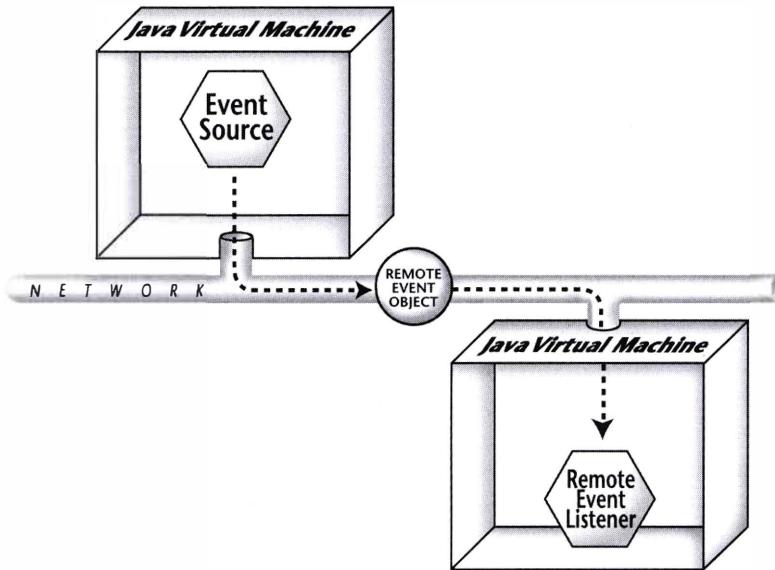


Figure 8.2 The Jini Distributed Event model: event source, remote event object, and remote event listener in a distributed, multi-JVM environment.

Space-based distributed events are built on top of the Jini Distributed Event model, which extends the Java platform's event model to allow events to be passed from event sources in one JVM to remote event listeners in different, possibly remote JVMs. When using space-based distributed events, the space is an event source that fires events when entries are written into the space that match templates. When the event fires, the space sends a remote event object to the listener.

The Jini Distributed Event model is incorporated into JavaSpaces technology as follows: The `JavaSpace` interface provides a `notify` method that allows processes to register an object's interest in the arrival of entries that match a specific template (which is passed as a parameter to `notify`). When an entry arrives that matches the template, an event is generated by the space and sent to the registered object in the form of a remote event object, by calling a `notify` method on the listener.

To avoid confusion, it's important to take note right away that two `notify` methods come into play in the distributed event model. One is a method in the `JavaSpace` interface, while the other is a method that listeners must implement. The fact that both have the same name is unfortunate, but their use should become clear as we progress through the chapter.

Before getting into these details, we are going to introduce space-based distributed events by building a simple example that uses *notify*. Then we'll dive down deeper by taking a look at the technical details surrounding event notification in JavaSpaces software development.

8.2 Hello World Using *notify*

For a simple example we are going to return to the “Hello World” application from Chapter 1. You will recall that we wrote a message entry into a space and then read it and printed its content, which happens to be the String “Hello World”. Now we are going to adapt that program to use the *notify* method instead of *read*.

To do so, our application will first register a listener that is interested in Message entries arriving in the space. Then, as in the original version, we will write a Message entry into the space. When the new entry arrives in the space, the space notifies the listener by calling the listener's *notify* method. The listener responds to the event by reading the Message entry from the space and printing its content field. In effect, the application behaves exactly as it did in Chapter 1, but instead of using a blocking *read* to retrieve the Message entry, it retrieves the entry as a result of being notified that one has been written into the space.

Here is the full code for our new reactive “Hello World” application:

```
public class HelloWorldNotify {
    public static void main(String[] args) {
        JavaSpace space = SpaceAccessor.getSpace();

        try {
            Listener listener = new Listener(space);
            Message template = new Message();
            space.notify(template, null, listener,
                         Lease.FOREVER, null);

            Message msg = new Message();
            msg.content = "Hello World";
            space.write(msg, null, Lease.FOREVER);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Let's step through the code in detail. As in the original code, the application first obtains a handle to a space. Then we instantiate a `Listener` object that will act as the listener for remote events from the space. Next we create a `Message` template, leaving the `content` field `null` to serve as a wildcard.

Next, the application calls the space's `notify` method:

```
space.notify(template, null, listener, Lease.FOREVER, null);
```

The first four parameters of the space's `notify` method are the template, a transaction (in this case a `null` transaction), the object that will receive notifications (in this case our `listener` object), and a lease time for the `notify`. We'll discuss the last parameter in Section 8.3.1; here we supply a `null` value. When called, the `notify` method registers the listener for notification whenever an entry is placed in the space that matches the template. This registration will remain in effect until the granted lease expires.

Finally, the application creates and writes a `Message` entry (with its `content` field set to "Hello World") to the space.

Now let's look at the details of the `Listener` inner class, which is defined as follows:

```
public class Listener implements RemoteEventListener {
    private JavaSpace space;

    public Listener(JavaSpace space) throws RemoteException {
        this.space = space;
        UnicastRemoteObject.exportObject(this);
    }

    public void notify(RemoteEvent ev) {
        Message template = new Message();

        try {
            Message result =
                (Message)space.read(template, null, Long.MAX_VALUE);
            System.out.println(result.content);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

The `Listener`'s constructor takes a space and assigns it to a local variable; it then calls the static `exportObject` method of the `UnicastRemoteObject` class. This

method is part of Remote Method Invocation package and is used by objects that support a remote interface. In effect, this method exports the listener as a remote object that can be called by the space when an event occurs for which a notification is needed. For more information on this topic you can refer to the *Java™ Remote Method Invocation Specification* from Sun Microsystems.

The `Listener` class implements the only method, `notify`, of the `RemoteEventListener` interface. The `notify` method is called remotely from the space whenever an entry matching the template is written to the space. When `notify` is called, the space passes it a remote event object, which we ignore (for now, the arrival of the event is enough to tell us a matching entry was written into the space). When the `notify` method is called by the remote space, it creates a message template, uses it to read a `Message` entry from the space, and then prints the entry's `content` field.

When we run our application, we see the following output:

```
Hello World
```

Now that we've seen a simple example of using notification, let's dive into the details of the API.

8.3 The Notification API

As we've already seen, several classes and interfaces play key roles in space-based distributed events, and we are going to examine the details of each in this section. Here is a brief overview:

- ◆ `JavaSpace`. Objects can register to be notified of new entries appearing in a space by calling the `JavaSpace` interface's `notify` method and supplying a template. The space will generate an event whenever an entry is written that matches the template.
- ◆ `EventRegistration`. When an object calls the `JavaSpace` interface's `notify` method, an `EventRegistration` object is returned. This object contains information about the registration, such as its lease time and a starting sequence number for events.
- ◆ `RemoteEventListener`. Any object that wishes to receive remote events from a space must implement the `RemoteEventListener` interface. This interface consists of one method, `notify`, that a space calls to deliver remote events to the object.

- ◆ **RemoteEvent**. A space passes a **RemoteEvent** object to the remote event listener's **notify** method when a space-based event occurs. This object contains details about the event, including its source and sequence number.

Now let's step through the details of each of these components.

8.3.1 The JavaSpace notify Method

The **JavaSpace** interface provides a **notify** method that is called to register interest in the arrival of entries into a space. Here is the method's signature from the *JavaSpaces Specification*:

```
EventRegistration notify(Entry tmpl,
                        Transaction txn,
                        RemoteEventListener listener,
                        long lease,
                        MarshalledObject handback)
    throws RemoteException, TransactionException;
```

The **notify** method takes five parameters and returns an **EventRegistration** object. The first parameter is an entry that acts as a template. As entries are added to the space, they are matched against this template (as well as all other registered templates), using the same matching rules as for **read** and **take** (see Section 2.4.3). If a match occurs, then a remote event is sent to the specified listener.

The second parameter to **notify** is a transaction. For now we will continue to supply a `null` transaction; we'll return to the topic of transactions in Chapter 9.

The third parameter is a listener that implements the **RemoteEventListener** interface. This is the object that receives remote events from the space. Since this object is a "remote" event listener it can be an object that exists locally or remotely on another JVM (even within a non-space-based program). We will describe the **RemoteEventListener** interface shortly (however, as we have seen, it contains only the **notify** method).

The next parameter is a lease time, which is a requested upper bound on the length of time the space should remember the registration. The syntax and semantics of leases are the same as those returned by the **write** method (as we saw in the last chapter). Lease times are requested in milliseconds and the space itself determines the actual length of the lease that it is willing to grant (as we will see, the lease object is returned as part of the **EventRegistration**). We can let the lease run its course, or we can use the lease object to cancel or renew the lease. Once the lease expires or is cancelled, the listener will no longer receive notifications.

The last parameter is a `MarshallledObject` called a “handback.” A handback is an object that the space will pass back to the remote event listener as part of an event notification; it is a wrapper around a serialized object. The space itself never instantiates the handback as an object; it simply keeps it around in serialized form and treats it as a byte array. When the remote event listener receives a handback as part of the event sent by the space, it can unpack the object and instantiate it.

Finally, the `notify` method can throw two exceptions: `TransactionException` and `RemoteException`. A `TransactionException` will be thrown when there is a problem with a non-null transaction parameter—for instance, if the transaction is unknown or has expired. As with other space-based operations, a `RemoteException` is thrown when communication problems occur between the object calling `notify` and the space, or when an exception occurs in the remote space in the process of registering the `notify`.

Now that we’ve taken a look at the `notify` call, let’s look at the event registration object that is returned from this call.

8.3.2 The EventRegistration Class

When a notification is successfully registered with a space, the space returns an `EventRegistration` object that encapsulates information about the registration.

The `EventRegistration` provides the following set of public methods:

```
public class EventRegistration implements java.io.Serializable {  
    public long getID();  
    public Object getSource();  
    public long getSequenceNumber();  
    public Lease getLease();  
}
```

The first two methods, `getID` and `getSource`, identify the event registration. The `getID` method returns an identifier assigned to a registration by the space (there is no standard convention for this identifier, and every space implementation is free to formulate its own identification scheme for the event registrations). The `getSource` method returns an object representing the event source (the space) with which interest in the event notification was registered. Together these two properties act to uniquely identify the event registration.

The `getSequenceNumber` method supplies an initial sequence number for events generated from a `notify` registration. Each successive event generated from the registration will contain a sequence number that is one greater than the previous sequence number. As we will see, the `getSequenceNumber` method gives the listener a way to deal with missing, out of order, and duplicate events generated from the registration.

Finally, the `getLease` method returns the lease granted to this event registration. If the client wishes to extend or cancel its registration, it can issue `renew` or `cancel` methods on the `Lease` object.

8.3.3 The `RemoteEventListener` Interface

In the context of JavaSpaces systems, remote event listeners are objects that receive remote events generated from the space. This object may reside in your space-based program or it can be an external third party object running remotely in another JVM. In order to act as a listener, an object must implement the `RemoteEventListener` interface:

```
public interface RemoteEventListener
    extends Remote, java.util.EventListener
{
    void notify(RemoteEvent theEvent)
        throws UnknownEventException, RemoteException;
}
```

The `RemoteEventListener` interface contains one method called `notify`, which is called whenever an entry is written into the space that matches the template supplied at event registration time. When the space calls a listener's `notify` method, it passes along a `RemoteEvent` object that provides specific information about the event. We will examine the `RemoteEvent` object in the next section.

The `notify` call is synchronous—when a space calls `notify` on a listener it waits until the call finishes. This is important: Since the space can't proceed until the call finishes, it is a good policy for the `notify` method to return as quickly as possible. In the case where you must do lengthy event processing, you should create a new thread for that purpose, and then return.

The `notify` method can throw two exceptions: `RemoteException` and `UnknownEventException`. The `RemoteException` occurs when there are communication problems between the space and the listener. The `UnknownEventException` can be thrown if the listener determines it has received an event it shouldn't have received, which allows the space the opportunity to cancel the particular notification.

8.3.4 The `RemoteEvent` Object

Each time a listener's `notify` method is called, a `RemoteEvent` object is passed to it. The `RemoteEvent` class is an extension of the `java.util.EventObject`, which is the event object used in Java's standard event model. The `RemoteEvent`

class provides the following public methods, which you will find closely related to the `EventRegistration` class in Section 8.3.2:

```
public class RemoteEvent extends EventObject
{
    public long getID();
    public Object getSource();
    public long getSequenceNumber();
    public MarshalledObject getRegistrationObject();
}
```

Like the `EventRegistration` class, `RemoteEvent` provides `getID` and `getSource` methods, which uniquely identify the event as belonging to a particular event registration. When a listener receives an event notification, calling `getSource` and `getID` on the remote event will return the same values as calling `getSource` and `get ID` on the `EventRegistration` object returned by the corresponding `notify` registration. If the listener is registered to receive multiple event registrations, it can use these values to identify the event as belonging to a particular registration.

Likewise, the `RemoteEvent` provides a `getSequenceNumber` method that returns the unique sequence number of this event, which can be used to compare this event against the beginning sequence number in the event registration and also the event numbers of other events from the same notification.

Unlike the `EventRegistration` class, the remote event contains a `getRegistrationObject` method that returns a `MarshalledObject` (a wrapper around a serialized object); this object is the “handback” that was passed as the final parameter to the space’s `notify` method, and it gets handed back to a listener at notification time as part of the event object. We’ll have more to say about the handback object, and what it is used for, in Section 8.5.

8.4 Putting the Pieces Together

Now that we’ve seen the details of the various classes and interfaces that come into play in distributed event programming, we’ll see how to use them by building an advanced example. The example we’re going to build is a channel relay, based on channel distributed data structures. A thread will continually append messages to two channels. At the same time, a relayer object acts as a listener that’s notified whenever the tail of either channel changes—indicating that new items have been added—and is responsible for transferring all messages appearing in those channels to a third. The functionality of the application will serve to illustrate the important points of event notification and event handling.

We could, of course, build the same application in a non-event-driven style: We would need a separate relayer thread for each channel we wish to monitor, which would continually wait for new messages to arrive in the channel. The more channels, the more threads we need to create, which could quickly get out of hand. This is a clear example of a context in which the reactive programming style made possible by `notify` is more appropriate: We are able to have just one relayer object, which will be notified of updates to any number of channels (whether one or one hundred) and respond appropriately.

8.4.1 The Channel Relay Application

Here is the basic skeleton of our `ChannelRelay` application:

```
public class ChannelRelay implements Runnable
{
    private JavaSpace space;
    private Thread channelReader;

    public static void main(String[] args) {
        JavaSpace space = SpaceAccessor.getSpace();
        ChannelRelay relay = new ChannelRelay(space);
        relay.test();
    }

    public ChannelRelay(JavaSpace space) {
        this.space = space;
    }

    public void test() {
        createChannel("Channel A");
        createChannel("Channel B");
        createChannel("Channel C");

        channelReader = new Thread(this);
        channelReader.start();

        try {
            Relayer relay = new Relayer();
        } catch (RemoteException e) {
            throw new RuntimeException(
                "Can't create Relayer object.");
        }
    }
}
```

```
for (int i = 1; ; i++) {
    append("Channel A", "A Message " + i);
    sleep(1000);
    append("Channel B", "B Message " + i);
    sleep(1000);
}
}

//... createChannel, append, & getMessageNumber (from Pager
//... example, Ch. 5) go here (with minor changes to append)

//... run method of channelReader thread goes here

//... Relayer inner class goes here

//... head & tail utility methods & sleep methods go here
}
```

After obtaining a handle to a space, the `main` method constructs a `ChannelRelay` object and calls the object's `test` method. The `test` method calls `createChannel` (reused from the pager example of Chapter 5) three times to create Channel A, Channel B, and Channel C. At that point, the application spawns a `channelReader` thread that continually reads and prints the contents of Channel C (as we'll see in a moment); the reader operates in its own thread so that the channel relay application will be free to continue with its own business at the same time.

After starting the reader, the `test` method instantiates an object called `relay` from a `Relayer` inner class. The `relay` object will be a listener for events that correspond to new messages in Channels A and B, and will remove the messages and transfer them to Channel C as it finds them.

Next, the application enters a loop that carries out its main function: repeatedly appending messages to both Channels A and B (sleeping for one second in between appends). Our code reuses the `append` and `getMessageNumber` methods from the pager application (with very minor changes to `append`, as you can see for yourself in the complete source code).

That's all there is to the channel relay application. Its components, the channel reader, and especially the `relayer` object, deserve a more in-depth look.

8.4.2 The Channel Reader Thread

The channel reader thread executes the following `run` method that continually reads and prints the messages it finds in Channel C. This code should appear very familiar; it's a channel reader, like the ones we saw in Chapter 5:

```
public void run() {
    int position = 0;
    Message msg = null;
    Message template = new Message("Channel C", null, null);

    while(true) {
        template.position = new Integer(++position);
        try {
            msg = (Message)
                space.read(template, null, Long.MAX_VALUE);
        } catch (Exception e) {
            e.printStackTrace();
        }
        System.out.println("C " + position + ": " + msg.content);
    }
}
```

This method assigns zero to the variable `position`, and creates a `Message` template, specifying “Channel C” as the channel name.

Then the channel reader enters a `while` loop, where we first fill in the `position` field of the template with the index of the message we wish to read; the `position` starts off at one, and will be incremented with each iteration. The `content` field of the `Message` template is left as a wildcard. Next we call `read` to find a matching message in Channel C, and then print the message to the output screen.

8.4.3 The Relayer Listener

The `Relayer` object serves as a listener for new messages in Channels A and B, and transfers them into Channel C (where they'll be read and printed by the channel reader thread we just saw).

The `Relayer` inner class of `ChannelRelay` is defined as follows:

```
public class Relayer implements RemoteEventListener {
    private EventRegistration regA, regB;
    private long seqA, seqB;
```

```
public Relayer() throws RemoteException {
    UnicastRemoteObject.exportObject(this);

    try {
        Index template = new Index("tail", "Channel A");
        regA = space.notify(template, null, this, 60000, null);
        seqA = regA.getSequenceNumber();

        template = new Index("tail", "Channel B");
        regB = space.notify(template, null, this, 60000, null);
        seqB = regB.getSequenceNumber();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

public void notify(RemoteEvent event) {
    NotifyHandler nh = new NotifyHandler(event);
    new Thread(nh).start();
}

public class NotifyHandler implements Runnable {
    private RemoteEvent event;

    public NotifyHandler(RemoteEvent event) {
        this.event = event;
    }

    //... run and relayMessages methods go here
}
}
```

This class's constructor calls the `UnicastRemoteObject` class's static method `exportObject`. As we explained earlier for the `HelloWorldNotify` example, this call is needed to export the relayer as a remote object that can be notified by a space when an event occurs.

In the constructor, we next register the relayer object to be a listener for events occurring in the space. First we create an `Index` template that will match the tail index of Channel A. Then we call the space's `notify` method, passing it the `Index` template, a `null` transaction, the object to receive the notification (here the relayer object itself), a lease time of one minute, and `null` for the

`MarshalledObject`. In other words, the relayer object is asking to be notified every time the tail entry of Channel A is written into the space, within the next minute. We could have specified a lease time of `Lease.FOREVER` to receive notifications indefinitely, but here we've placed a time limit on the notification request to demonstrate notification expiration: After the one-minute lease is up, the notification agreement between the relayer and the space will expire.

You'll notice that we assign the `EventRegistration` object returned by the call to `notify` to the variable `regA`. Then we call `getSequenceNumber` on `regA` and assign the result to `seqA`; this long integer holds the starting sequence number issued by the space for all events generated from this `notify` registration. We'll see shortly when and how the registration and sequence information will be useful.

Next, we go through an almost identical notification registration, except this time we register the relayer object to be notified of updates to the tail entry of Channel B. The notification agreement is subject to a one-minute lease this time as well. And again, we save the `EventRegistration` object returned by the call to `notify`, as well as the initial sequence number for the events. If either of the two event registrations experiences problems, we will catch an exception.

Since the relayer object will act as an event listener, it implements the `RemoteEventListener` interface and therefore supplies the `notify` method. This method instantiates a `NotifyHandler` object to handle the notification, passing `RemoteEvent` to its constructor. The `NotifyHandler` implements the runnable interface so it can handle the processing of the remote event in a separate thread (recall that all lengthy processing in a `notify` method should be offloaded to another thread so the space does not have to wait). After the relayer instantiates the handler, it creates a thread for it (we will see the processing it does in the next section) and calls `start`. Finally, the `notify` returns, letting the space and the notify handler thread continue their business.

8.4.4 The Notify Handler Thread

The `NotifyHandler` object runs as a thread and executes the `run` method you see here:

```
public void run() {  
    Object source = event.getSource();  
    long id = event.getID();  
    long seqNum = event.getSequenceNumber();  
    String channel;  
  
    // determine if the event source was Channel A or Channel B  
    // and is more recent than the last event notification
```

```

    if (source.equals(regA.getSource()) &&
        id == regA.getID())
    {
        if (seqNum > seqA) {
            seqA = seqNum;
            channel = "Channel A";
        } else {
            return;
        }
    } else if (source.equals(regB.getSource()) &&
        id == regB.getID())
    {
        if (seqNum > seqB) {
            seqB = seqNum;
            channel = "Channel B";
        } else {
            return;
        }
    } else {
        System.out.println("Unknown Event.");
        return;
    }

    // then forward the messages to Channel C
    relayMessages(channel, "Channel C");
}

```

The job of the handler is to analyze the event notification and take appropriate action. Here, the handler extracts information about its remote event. The space that generated the event is assigned to `source`, the identifier for the event is assigned to `id`, and the number of the event (in the sequence of notifications generated by a particular event registration) is assigned to `seqNum`.

The handler uses this information to determine which of our two `notify` registrations triggered the remote event. If the event source and identifier match those of our `regA` event registration, then an update to Channel A's tail generated the notification. If, on the other hand, the event source and identifier match those of our `regB` event registration, then an update to Channel B's tail was the trigger. In any other case, the handler received an unknown event and simply returns (we include this case for completeness, but it should never happen).

Once the handler knows the reason for the event, it performs an additional check on sequence numbers. If the seqNum of the notification is less than or equal to the previous one the handler processed for the same channel (stored in seqA or seqB), then we know this event notification is an older one that arrived out of sequence, and the handler ignores it and returns. If the seqNum is greater than the last one processed, then the handler updates the sequence variable for the channel (seqA or seqB), and assigns the channel name to channel.

The next step is to call `relayMessages`, passing it the source channel name and the destination channel, which in this case is always Channel C. The job of `relayMessages` is to take any new messages that are in the source channel, remove them, and transfer them to Channel C. Here's the definition:

```
private void relayMessages(String source, String destination) {
    int end = readTail(source);
    Index head = takeHead(source);
    int start = head.position.intValue();
    if (end < start) {
        return;
    }

    Message template = new Message();
    template.channel = source;

    for (int i = start; i <= end; i++) {
        template.position = new Integer(i);
        try {
            Message msg = (Message)
                space.take(template, null, Long.MAX_VALUE);
            append(destination, msg.content);
            head.increment();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    writeHead(source, head);
}
```

The `relayMessages` method calls `readTail`, which determines the position number of the last message currently in the source channel, and `takeHead`, which removes the head entry of the source channel from the space. We then compare the positions of the head and tail; if the tail is less than the head, then the channel

is empty and we return. Otherwise, we iterate through the sequence numbers from the head to the tail, taking each message entry from the source channel and appending it to the destination channel. Each time we also increment the head by one. Once we've removed all the messages, we write the head index back into the space.

8.5 Summary

In this chapter, we've introduced the JavaSpace interface's `notify` method as a way to write space-based programs that have a more reactive style. We've also taken a close look at the classes and interfaces of the Jini Distributed Event model upon which the `notify` method is built.

In building our channel relay application, we were able to put these various classes and interfaces to use and gain experience writing an event-driven program. Every space-based application you might want to build using `notify` can certainly be rearchitected to use `read` and `take`, but as we saw in the channel relay example, there are definite cases where applications are cleaner to build in a reactive style.

There are a couple of advanced topics that are beyond the scope of this book but worth keeping in mind as you begin to build more complex distributed applications. One involves passing a `MarshalledObject` (a wrapper around a serialized object) as the `handback` parameter of `notify`; the space will pass this object back to the remote event listener as part of an event notification, and the listener can unpack the object, instantiate it, and call its methods. For instance, a client may wish to pass an entire graphical user interface in the marshalled object, and then run the interface at the appropriate time in response to an event. A second advanced topic is how `notify` can be used to allow outside listeners (non-space programs) to react to events in a space, thereby tying together space and non-space processes by using distributed events. For details on these topics, you'll need to refer to the *Jini™ Distributed Event Specification* and the *Java™ Remote Method Invocation Specification*.

We'll return to distributed events in the next chapter, as we review the semantics of `notify` under a transaction. We will also make use of them in Chapter 10 in a collaborative application.

8.6 Exercises

Exercise 8.1 Draw a picture of the various objects that are part of the distributed event lifecycle, including the space, the event listener, the event, and the event registration. If you have trouble, please review Section 8.3.

Exercise 8.2 Rewrite PingPong from Chapter 5 using distributed events.

Exercise 8.3 Write a relay that forwards messages from one channel to several others (but doesn't delete them from the original).

Exercise 8.4 Write a sieve that forwards messages from several channels to one.

Exercise 8.5 Write a program that tracks the number of all entries written into a space.

Transactions

In computing, the mean time to failure keeps getting shorter.
—Alan Perlis, **Epigrams in Programming**

IN this chapter we are going to explore a fundamental tool for dealing with partial failure: *transactions*. Transactions provide a means of enforcing consistency over a set of (one or more) space-based operations. By consistency we mean that transactions allow space-based operations to be bundled together in such a way that either all complete, or none of them do—in other words, a set of operations will never be “half” completed.

An example can help in understanding the value of transactions. Consider our web counter example from Chapter 3—each time a web page is visited, an applet takes a page count entry, increments its value, and writes it back into the space. Now consider how this applet might perform in the presence of partial failure: The applet removes the page counter, increments its value and then experiences some form of partial failure (the process’s machine crashes or network problems occur). This scenario would result in a counter entry being retrieved from the space but then lost, since the applet that took it from the space is no longer able to return it.

We can solve this problem by performing the `take` and `write` of the counter entry within a transaction. When these operations are executed within a transaction they become one atomic operation so that either both operations complete (the web counter is both removed and returned to the space), or neither operation completes (the web counter is effectively never removed from the space). This technique can be applied to more sophisticated protocols to ensure that distributed data structures always remain in a consistent state, even in the presence of partial failure.

9.1 The Distributed Transaction Model

The JavaSpaces technology uses the Jini™ Transaction model, which provides a generic transaction service for distributed computing. Here is a brief overview of how the model operates: All transactions are overseen by a *transaction manager*. When a distributed application needs operations to occur in a transactionally secure manner, a process asks the transaction manager to create a *transaction*. Once a transaction has been created, then one or more processes can perform operations *under the transaction*. A transaction can complete in two ways. If a transaction *commits* successfully, then all the operations performed under it occur. However, if problems arise (such as partial failure), then the transaction is *aborted* and none of the operations occur. These semantics are provided by a *two-phase commit protocol* that is performed by the transaction manager as it interacts with the transaction participants. For detailed information on this protocol, please refer to the *Jini™ Transaction Specification*.

Using transactions in space-based operations is fairly straightforward: A process first obtains a transaction from a manager and then passes that transaction to each space-based operation (which may occur over one or more spaces) to be performed under it. The process then commits the transaction. If the commit is successful, then all operations under the transaction are guaranteed to have completed successfully. On the other hand, if problems occur, the transaction is aborted and none of the operations occur, leaving the space unchanged.

A transaction can be aborted in two ways: A process can explicitly call the `abort` method on the transaction if it encounters unexpected problems, or, if a transaction is not explicitly committed or aborted, then it will eventually be aborted by the transaction manager. This happens because the transaction manager maintains each transaction as a leased resource, and when the lease expires the manager aborts the transaction.

Transactions have the following effect on the semantics of the space-based operations (as shown in Figure 9.1): When you `write` an entry under a non-null transaction, the entry is not seen or accessible outside of the transaction (that is, to operations that don't occur under the same transaction) until the transaction commits. When you `take` or `read` entries from the space under the transaction, they can come from entries written under the transaction, or entries in the space. If the transaction commits, then all the entries written under the transaction (and not later taken within the same transaction) become visible in the entire space. However, if the transaction aborts, the entries written under the transaction are removed and never become visible, and any entries taken under the transaction are returned to the space. In effect, after the abort the space reflects that the operations never occurred.

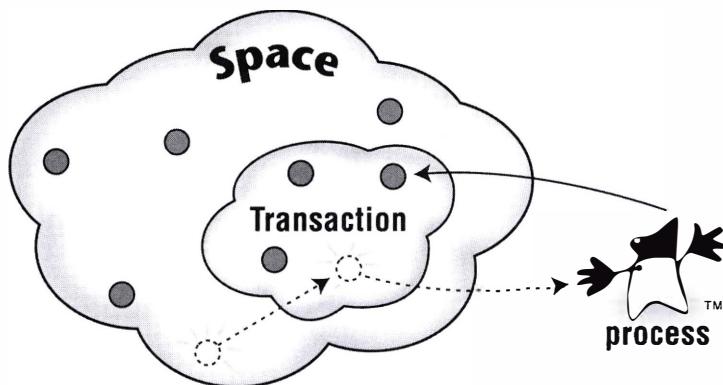


Figure 9.1 Space-based operations within a transaction.

Given this brief overview of transactions, let's now look at an example. Afterward, we will return to a more detailed explanation of the semantics of the space-based operations under transactions.

9.2 Creating a Transaction

To create a transaction, we first need access to a transaction manager. Transaction managers are remote services, like spaces, that are typically located through a registry or lookup service such as the RMI registry or the Jini Lookup service. Like our `SpaceAccessor` class that we've used to obtain access to spaces, we have also defined a `TransactionManagerAccessor` class that can be used to obtain access to a transaction manager. As with the `SpaceAccessor`, you can either use this class in your code or plug in your own means of accessing a transaction manager. The definition of `TransactionManagerAccessor` can be found in the complete source code.

Using `TransactionManagerAccessor` we can obtain a reference to a transaction manager like this:

```
TransactionManager mgr =  
    TransactionManagerAccessor.getManager();
```

Here we call the `getManager` static method of the `TransactionManagerAccessor` class, which returns a reference to a `TransactionManager`.

With a reference to a transaction manager in hand, we can now create a transaction. Here is how:

```

Transaction.Created trc = null;
try {
    trc = TransactionFactory.create(mgr, 3000);
} catch (Exception e) {
    System.err.println(
        "Could not create transaction " + e);
}

```

Here we first declare a variable `trc` of type `Transaction.Created` (an inner class of `Transaction`). Then we call the `TransactionFactory` class's static method `create`. The `create` method takes a transaction manager and a lease time (in milliseconds) and creates a transaction that is managed by the supplied manager for the given lease time. If the call to `create` is successful, then a `Transaction.Created` object is returned and assigned to the variable `trc`. If a transaction cannot be created, an exception is thrown.

The `Transaction.Created` object may look a bit strange, but it is simply an instantiation of a public inner class of the `Transaction` class that looks like this:

```

public static class Created implements Serializable {
    public final Transaction transaction;
    public final Lease lease;

    Created(Transaction transaction, Lease lease) {...}
}

```

This class acts as a wrapper for two values: the transaction and the granted lease. We use the `transaction` field to obtain a reference to the transaction object that can be passed to space-based operations; here is how we obtain this reference:

```
Transaction txn = trc.transaction;
```

Likewise, we can retrieve the transaction's lease by accessing the transaction's `lease` field. Leases on transactions work analogously to entry-based leases—if a transaction's lease expires (in the example above, after three seconds) then the transaction is automatically aborted, and none of the operations under the transaction will occur on the space.

9.3 Web Counter Revisited

With this groundwork behind us, it is now time for an example. Below we've taken the web counter applet and updated it to use transactions; you might first turn back to the previous version (see Section 3.2.3) to refamiliarize yourself with the original code before taking a look at this version. After we have stepped

through this example, and have a feel for how transactions are used, we will study the various APIs and semantics of transactions in detail.

Here is the code for the transaction-based web counter applet:

```
public class WebCounterClient extends Applet {
    private SharedVar counter;

    public void init() {
        JavaSpace space = SpaceAccessor.getSpace();
        String url = getDocumentBase().toString();

        TransactionManager mgr =
            TransactionManagerAccessor.getManager();

        Transaction.Created trc = null;
        try {
            trc = TransactionFactory.create(mgr, 3000);
        } catch (Exception e) {
            System.err.println(
                "Could not create transaction " + e);
            return;
        }

        Transaction txn = trc.transaction;

        SharedVar template = new SharedVar(url);
        try {
            try {
                // take counter under a transaction
                counter = (SharedVar)
                    space.take(template, txn, Long.MAX_VALUE);

                counter.increment();

                // write back counter under a transaction
                space.write(counter, txn, Lease.FOREVER);
            } catch (Exception e) {
                System.err.println("Not written into space " + e);
                txn.abort();
                return;
            }
        }
```

```

        txn.commit();
    } catch (Exception e) {
        System.err.println("Transaction failed");
        return;
    }
}

public void paint(java.awt.Graphics gc) {
    if (counter != null) {
        Integer count = counter.value;
        gc.drawString("Page visited " +
            count + " times before.", 20, 20);
    }
}
}

```

If you refer back to the original code, you'll see the changes we made to add transactions are fairly minimal. Stepping through the code, we first obtain a reference to a space and then obtain the URL of the web page for which we are displaying the counter.

We then obtain a transaction using the same steps we described in the previous section: We obtain a reference to a transaction manager, ask it to create a transaction and then obtain a reference `txn` to the transaction object.

Once we have a transaction in hand, we are ready to retrofit the space operations to work under it. Before doing so, we first create a template that will be used for taking the counter entry from the space. Then we call the `take` method: Instead of passing a `null` transaction parameter, as we've been doing, we specify the transaction `txn`. Then we increment the counter (a local operation) and `write` it back to the space, again supplying `txn` for the transaction parameter.

When we execute the `take` and `write` operations under the `txn` transaction, three things can happen. One possibility is that the operations complete without any exceptions, and we attempt to commit the transaction by calling the transaction's `commit` method:

```
txn.commit();
```

When this method is called, the transaction manager is asked to commit the transaction. If the `commit` is successful, then the operations invoked under the transaction occur in the space as one atomic operation.

The second possibility is that an exception may occur during the `write` or `take` operations, and in this case we attempt to abort the transaction in the `catch` clause. If the `abort` is successful, then no operation occurs in the space.

A third possibility is that an exception may be thrown in the process of committing or aborting the transaction. In these cases, the outer catch clause catches the exception and prints a message indicating the transaction failed. The transaction will expire when its lease time ends (in this case after three seconds), and no operations will occur. The transaction will also expire if this client unexpectedly dies or becomes disconnected from the network during the sequence of calls.

In a short amount of code we've seen the steps needed to use transactions with spaces:

- ◆ Obtain a reference to a transaction manager.
- ◆ Create a transaction by specifying a manager and lease time to the `TransactionFactory.create` method, which returns a `Transaction.Created` object.
- ◆ Obtain a reference to the transaction through the `Created` object.
- ◆ Pass the transaction to all operations you'd like performed under it.
- ◆ Commit the transaction when finished, or abort it if something goes wrong.

Now that we've had a taste of transactions, let's understand the details a little better.

9.4 The Space's Transactional Properties

Space-based transactions adhere to a set of properties known as the *ACID properties*. The word ACID is an acronym that stands for atomicity, consistency, isolation, and durability. Let's look at what each one means:

- ◆ **Atomicity** means that a transaction's changes to a space are atomic: either all of the state changes occur or none of them do. These changes include the effects of all `read`, `write`, `take`, and `notify` operations that occur within the transaction.
- ◆ **Consistency** means that a transaction's changes do not violate the correctness of the state within the space. This property requires that your program be correct. For instance, our web counter example ensures that an entry is both taken and returned; given that this algorithm is correct, the transaction ensures that the space will not be put into an inconsistent state.
- ◆ **Isolation** means that while many transactions may execute concurrently, it ap-

pears to each transaction that all other transactions complete either before or after it, but not both. In addition, any observer should be able to see the transactions executing as if they were completed one after the other in some sequential order. In essence, this property says that two transactions should not affect each other, which means that you can write transaction-based space programs without worrying about what is happening in other transactions.

- ◆ **Durability** means that when a transaction commits, its changes to a space will survive failures. In the Jini Transaction model, this requirement is loosened and depends on the guarantees of the underlying space implementation. In persistent spaces, the results of your transactions will survive failures of the space. However, if a space implementation does not provide persistence, then a transaction's changes may not survive a space's crash that occurs after it commits.

These properties provide an important set of semantics for processes that execute concurrently and may encounter failures. In light of the ACID properties, we are going to return to the basic space operations and describe their semantics more precisely.

9.5 Operational Semantics Under Transactions

In Chapter 2, we covered the operational semantics of each space operation (with the exception of `notify`, which we covered in Chapter 8). Taking another look at the basic space operations is appropriate at this point, because transactions introduce subtle changes to their operational semantics. If you plan to make use of transactions in order to build robust distributed programs, then it's important to understand the details of this section.

9.5.1 Writing Entries Under a Transaction

When you write an entry into a space under a transaction, the entry is only seen within the transaction until it commits. This means that the entry is invisible to any attempts to read, take or be notified of the entry outside of the transaction. If the entry is taken from within the transaction, it will never be seen outside of the transaction. If the transaction aborts, then the entry is discarded.

Once the transaction commits, the entry is available for reads, takes, and notifications outside of the transaction.

9.5.2 Reading Entries Under a Transaction

When you issue a read operation within a transaction, it may match an entry written under the transaction or any entry in the space (that is, outside of the transaction). According to the *JavaSpaces Specification*, space implementations are not required to prefer matching entries within the transaction over entries in the space, so you should not rely on that property in your algorithm design.

In the case where we match an entry in the space, it is added to the list of entries read by the transaction. Processes operating under other transactions are allowed to read this entry, but are not allowed to take the entry after it's been read. This constraint is imposed by the ACID properties—if a process in another transaction is allowed to take an entry we've already read, then we may end up with an inconsistent operational semantics for the entire space. What may happen is that the other transaction could commit before our transaction, leaving the space in a state where we've been able to read an entry that was technically taken by another process before our read effectively occurred (because our read doesn't really occur until we've committed our transaction). To prevent this inconsistency, once an entry is read under a transaction, it cannot be taken under another transaction until the first transaction completes.

9.5.3 read versus readIfExists

Earlier in the book you may have asked yourself, “Is a `readIfExists` just a `read` with its timeout parameter set to `JavaSpace.NO_WAIT`?” The answer is no, and the distinction between the two operations has to do with transactions.

A `read` will wait for an entry until its timeout period and then return `null` if no matching entry was found. If `read` is given a timeout of `JavaSpace.NO_WAIT` and no matching entry exists in the space, then it will return immediately. A `readIfExists` always returns immediately if no entry is found, except when operating under a transaction. In effect, `readIfExists` is able to “peak inside” transactions: If a matching entry exists within a transaction, then `readIfExists` will wait up until its timeout value for a transaction to complete before it returns. So, it’s only in the case where a matching entry exists under a transaction that the `readIfExists` method will wait before returning.

9.5.4 Taking Entries Under a Transaction

The `take` and `takeIfExists` methods operate analogously to `read` and `readIfExists` under a transaction and can match an entry inside or outside of their provided transaction. Once either method has removed an entry under a transaction, the entry is added to a list of entries that can’t be read or taken by other processes.

9.5.5 Notifications Under Transactions

When we register for a `notify` under a transaction, we receive notifications of entries that are written within the transaction, as well as entries that are written to the general space. When the transaction completes (whether it commits or aborts) or expires, all registrations for notification under the transaction are withdrawn. If the transaction commits, the entries remaining in the transaction may result in notifications in response to registrations in the general space. The entries also become eligible for `read` and `take` operations from the space.

9.6 Summary

At this point we have covered the basics and many of the details of space-based transactions. We've seen that transactions are a fundamental tool for providing failure semantics in distributed systems. Space-based transactions simplify the use of the more general Jini Transaction model and provide a means of grouping together space-based operations to ensure consistency. We also covered the mechanics of using a transaction manager to create and complete transactions.

At this point, rather than providing a more full-blown transaction-based example, we are going to move on to the next two chapters, both of which make use of transactions. There you will see the use of transactions in more fully developed examples.

9.7 Exercises

Exercise 9.1 Choose a few distributed data structures from Chapters 3-5 and analyze them to see where partial failure could corrupt the data structures. Reimplement them using transactions to make them safe.

Exercise 9.2 Reimplement the chat channel application from Chapter 5 and make it robust by using transactions.

Exercise 9.3 Rewrite the ChannelRelay from Chapter 8, so that transfers from two channels to a third channel are transactionally secure.

CHAPTER 10

A Collaborative Application

Collaboration, n.: A literary partnership based on the false assumption that the other fellow can spell.
—Anonymous

IN this chapter we are going to put together everything we've learned—from the basics of spaces programming, to useful distributed data structures and patterns, to the advanced topics of distributed events, leasing, and transactions—and build a collaborative application.

We are going to build an interactive messenger service, similar to many commercially available applications that allow you to keep track of a group of online friends and communicate with them. We'll start off by describing the messenger from the user's perspective, and then we'll build the application incrementally from the bottom up. We'll present each of the underlying distributed data structures, explaining their design and code, and then show how the pieces are woven together into a robust distributed application.

10.1 The Messenger

The messenger's user interface is shown in Figure 10.1. Three main components make up the interface: a login window, a main console, and one or more message windows. The login window (at the top right of Figure 10.1) appears when the application is first started and asks the user for a username and password. If the username and password are registered with the service, the user is logged in. If the password is incorrect, the user is prompted again. In the case where the user does not yet have an account, and a unique username was entered, then a new account is created automatically and the user is logged in.

Once the user has successfully logged in, the main console is displayed (shown at the left of Figure 10.1). Near the bottom of the console, an “Add” button

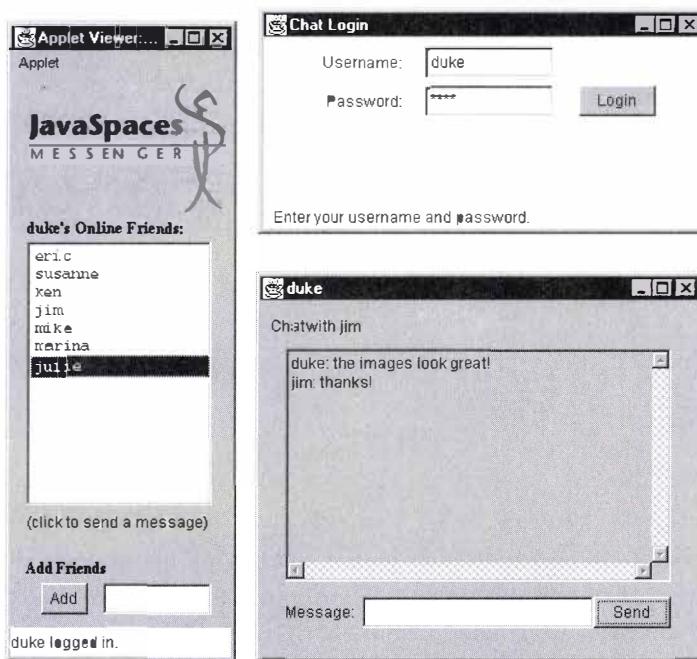


Figure 10.1 The Messenger console, login window, and message window.

and a corresponding text field allow the user to enter usernames and add them to a list of friends. Whenever friends in this list sign into the service, their usernames appear in the “Online Friends” text area at the top of the console. When they log off by closing the application (or are disconnected by some form of partial failure), their names disappear from the list of online friends.

At any time the user can send a private message to a friend by clicking on a name in the list of friends; this causes a message window to open (see Figure 10.1). By typing a message and clicking on the “Send” button, the message is sent to the remote friend’s messenger, where a message window will open on the screen. Likewise, if a message is received from a friend, a new window automatically appears (if one for that friend isn’t already open). Each conversation with a friend gets its own message window.

When the user is finished using the messenger service, the user can close the main console. Any remote users monitoring that user will see the name disappear from their list of online friends.

From this description of the messenger, you’ll notice that we have omitted several features that you would be likely to find in a full-fledged commercial version. For instance, we provide no way to conveniently manage your list of friends

(you can add as many friends as you'd like, but you can't view the friends list or remove anyone from it). We skimp on such user features, because our main focus is on the distributed aspects of the application. As simple as it is, this messenger service presents many challenges to the distributed application designer, especially in the face of partial failure. As you will see, our primary concern is not usability, but designing a correct and robust distributed application.

Now that we have an overview of the messenger, we'll spend the rest of the chapter examining its design and implementation.

10.2 Implementing a Messenger User

Each user of the messenger service is represented by a distributed data structure made up of the following:

- ◆ An account entry that holds the username and password
- ◆ A session entry that holds information about the user's current session and exists in the space only as long as the user is online
- ◆ A "friends list" entry that maintains the user's list of friends
- ◆ A channel that holds a user's messages

So basically a user is represented in the space by three entries and a channel (like the channels we developed in Chapter 5). These pieces are held together by a common, unique `username` field: For instance, the user "duke" will have the value "duke" in the `username` field of his account entry, friends list entry, and session entry, and in all the entries making up his channel.

Let's cover each piece of the distributed data structure in turn.

10.3 The Account

Our account entry holds two basic pieces of information about a user: a username and a password. Here is how the account entry is defined:

```
public class AccountEntry implements Entry {  
    public String username;  
    public String password;  
  
    public AccountEntry() {  
    }
```

```
public AccountEntry(String username) {
    this.username = username;
}

public AccountEntry(String username, String password) {
    this.username = username;
    this.password = password;
}
}
```

The `AccountEntry` is a simple entry, having fields for the username and password and a couple constructors. An account entry is used as follows: When a user logs into the service for the first time, the messenger creates a new account entry and writes it into the space. This acts to reserve a unique username for the user, and allows the user to be authenticated via a password on future logins.

Once an account entry is written to the space, it remains indefinitely—whether the user is online or not—and is never removed by the messenger service, even temporarily. This allows the messenger to test for the existence of an account entry in the space, which is used to determine if an account already exists for a given username.

10.3.1 The Account Object

Rather than manipulating account entries directly, the messenger typically relies on a helper class called `Account` that provides a wrapper around the space-based operations on account entries. This helper class provides two methods—one that creates new account entries and another that validates login attempts. Here is the code for the `Account` class:

```
public class Account {
    private Messenger messenger;
    private String username;
    private String password;
    private JavaSpace space;
    private TransactionManager mgr;

    public Account(Messenger messenger,
                  String username, String password)
    {
        this.messenger = messenger;
        this.username = username;
        this.password = password;
    }
}
```

```
    this.space = messenger.space();
    this.mgr = messenger.mgr();
}

public void create() {
    SharedVar template = new SharedVar("messenger-lock");
    SharedVar lock;

    AccountEntry account =
        new AccountEntry(username, password);
    FriendsList friendsList =
        new FriendsList(messenger, username);
    Channel channel = new Channel(messenger, username);

    try {
        Transaction.Created trc =
            TransactionFactory.create(mgr, 10000);
        Transaction txn = trc.transaction;
        lock = (SharedVar)
            space.take(template, txn, Long.MAX_VALUE);

        AccountEntry acctTemplate = new AccountEntry(username);

        if (space.readIfExists(acctTemplate, txn,
            JavaSpace.NO_WAIT) != null)
        {
            txn.abort();
            throw new UserExistsException();
        }

        space.write(account, txn, Lease.FOREVER);
        friendsList.create(txn);
        channel.create(txn);

        space.write(lock, txn, Lease.FOREVER);
        txn.commit();
    } catch (Exception e) {
        System.err.println("Can't create new account");
        e.printStackTrace();
    }
}
```

```

public void validate()
    throws UnknownUserException, InvalidPasswordException
{
    AccountEntry template = new AccountEntry(username);
    AccountEntry account = null;

    try {
        account = (AccountEntry)
            space.readIfExists(template,
                null, JavaSpace.NO_WAIT);
    } catch (Exception e) {
        System.err.println("Can't validate account");
        e.printStackTrace();
    }

    if (account == null) {
        throw new UnknownUserException();
    } else {
        if (!password.equals(account.password)) {
            throw new InvalidPasswordException();
        }
    }
}
}

```

The Account constructor takes a handle to our Messenger applet (which we will explain in a bit), as well as a username and password, and assigns them to local fields. It then obtains a handle to a space and a transaction manager by calling the messenger's space and mgr methods, respectively.

The create method creates an account for a new user. To properly create a new account it is crucial that it is done atomically—so that no two processes can create the same account at the same time such that we end up with two identical accounts in the space. As we mentioned, we use the existence of an AccountEntry to determine whether we need to create a new account, but this does not prevent a race condition in which two processes perform this test and then create the same account in an interleaved manner.

To ensure that accounts are created correctly and uniquely we make use of a shared variable that will serve as a lock (this lock must be created by a special messenger service installer that is supplied in the complete source code). A process can only create an account if they obtain the lock. Let's step through the create method and see how this works.

First, a shared variable template is created to retrieve the “messenger-lock” from the space. Then a new `AccountEntry` is instantiated with the supplied user-name and password. Since each user must also have an associated friends list and communication channel, `FriendsList` and `Channel` objects (which we’ll cover in more detail shortly) are instantiated as well. Next, we are going to create a transaction, because we want to either create the entire distributed data structure (the account, friends list and channel) successfully, or else leave the space untouched.

As we did in Chapter 9, we create a `Transaction.Created` object by calling the `TransactionFactory.create` method and supplying a manager and a lease time (here ten seconds). We then assign the returned transaction to the variable `txn`. Now we are ready to test to make sure we have the rights to create the account. First we retrieve the lock, waiting as long as necessary. Then we need to test to make sure that this account was not created by another process while we were waiting. We do this by creating an account entry template and calling `readIfExists`. If no entry exists, then we can create the account. Otherwise we abort the transaction and throw a `UserExistsException`.

Let’s think about the semantics before moving on: Since a process needs to have the lock (and there is only one) to create an account, and since an account entry is never removed from the space after it is created, if we own the lock and no matching account entry exists in the space, then we can be sure it is safe to create the account. One final concern is that the lock may be a bottleneck in the case where new accounts are quickly being created. This is easily fixed by “distributing” the lock. For instance, we could break the lock into 26 locks if we use the first letter of the user’s name as a key to a particular lock. If you need even finer-grained locks, a variety of hashing techniques can be used to map usernames to a set of locks.

Now we are finally ready to create the account. First, the account entry is written into the space. Then the `create` methods of the `FriendList` and `Channel` objects are invoked to write friends list and channel data structures into the space as well. These methods are passed the transaction so that they can perform their space operations under it. Finally, we return the lock and commit the transaction, which causes all the new user’s data structures to atomically appear in the space.

The `validate` method is used to verify that a username/password pair is valid. Validation is a two-step process. First we need to ensure that an account already exists for the user. To do this we create an account entry template, specifying the username but leaving the password field as a wildcard, and then call `readIfExists` to determine whether a matching account entry exists in the space. Because an account entry is written into the space at account creation time and never removed, if no matching account entry is found, it means the account doesn’t exist, and `validate` throws an `UnknownUserException` (see the com-

plete source code). If a matching entry is found, then an account already exists, and the supplied password is compared with the account's password. If the passwords aren't equal, then an `InvalidPasswordException` is thrown, otherwise the `validate` method returns normally.

10.4 User Sessions

In addition to the account entry (which stores persistent information for a user), we also need to store data about each user that is present only as long as the user is online. When a user logs in, the messenger creates and writes a `SessionEntry` object into the space, which serves two purposes. First, its existence in the space means that the user can be considered online. It also provides an object in which per-session information, such as the date and time the user signed on, can be stored. Here is the definition of the session entry:

```
public class SessionEntry implements Entry {  
    public String username;  
    public Long loginTime;  
  
    public SessionEntry() {  
    }  
  
    public SessionEntry(String username) {  
        this.username = username;  
    }  
  
    public SessionEntry(String username, long loginTime) {  
        this.username = username;  
        this.loginTime = new Long(loginTime);  
    }  
}
```

This is a simple entry, providing only a couple of convenience constructors and two fields: The `username` field associates the entry with an account, while the `loginTime` field records the user's login time.

The role of a session entry is to indicate—by its existence in the space—that the corresponding user is actively using the messenger service. When a user logs in, a session entry is created in the space, and when a user logs out, the entry is removed. However, in a distributed environment with the potential for partial failure, a user's messenger session could crash, or become disconnected from the space, before the user voluntarily logs out. In this case, unless we take precau-

tions, a user's session entry will stay behind in the space, falsely indicating that the user is still online. In addition, such session entries could eventually consume significant resources in a space, growing in an unbounded manner. To ensure that the lifetimes of session entries don't extend beyond their usefulness, we are going to lease session entries and place them under the control of a lease manager. Let's take a look at a class that's designed to take care of this.

10.4.1 The Session Object

Just as we defined a helper class that manipulates account entries, we also define a helper class that operates on session entries:

```
public class Session implements LeaseListener {  
    private JavaSpace space;  
    private String username;  
    private LeaseManager leaseManager;  
    private Lease lease;  
    private final long granularity = 1000 * 60 * 2; // two minutes  
  
    public Session(JavaSpace space, String username) {  
        this.space = space;  
        this.username = username;  
    }  
  
    static public void exists(JavaSpace space, String username)  
        throws UnknownUserSession  
{  
    SessionEntry template = new SessionEntry(username);  
    SessionEntry session = null;  
  
    try {  
        session = (SessionEntry)  
            space.readIfExists(template, null,  
                JavaSpace.NO_WAIT);  
    } catch (Exception e) {  
        System.err.println("Couldn't check session");  
        e.printStackTrace();  
    }  
    if (session == null) {  
        throw new UnknownUserSession();  
    }  
}
```

```

public void start() {
    long loginTime = System.currentTimeMillis();

    SessionEntry session =
        new SessionEntry(username, loginTime);
    try {
        lease = space.write(session, null, granularity * 2);
        leaseManager = new AutomatedLeaseManager(granularity);
        leaseManager.renewUntil(lease, Lease.FOREVER, this);
    } catch (Exception e) {
        System.err.println("Couldn't create session");
        e.printStackTrace();
    }
}

public void stop() {
    try {
        if (lease != null) {
            leaseManager.cancel(lease);
        }
    } catch (Exception e) {
        System.err.println("Couldn't cancel lease manager");
        e.printStackTrace();
    }
}

public void notify(LeaseRenewalEvent e) {
    System.out.println("got a lease renew event!" + e);
    // we ignore it
}
}

```

The `Session` class has a constructor that takes a space and a username as parameters and assigns them to local fields. Besides the constructor, the class provides four methods: `start`, `notify`, `stop`, and `exists`.

The `start` method is called to begin the user's session when he logs in. To do this, we first instantiate a `SessionEntry` object, passing it the username and the current time. We then `write` the entry into the space with an initial lease time of twice the value of `granularity` (a final long field in the `Session` object). We discussed the notion of `granularity` in Chapter 7: It is the time interval (in milliseconds) our lease manager regularly checks to see if any leases need to be renewed.

Here we write the session object into the space with an initial lease time of twice that value (in this case four minutes) to be reasonably sure the lease manager has a chance to renew the lease before it expires. Then we turn the management of the lease over to a lease manager.

To do this, we instantiate a `LeaseManager` object (the same lease manager we developed in Chapter 7), passing it `granularity`, and ask it to renew the lease forever. The `granularity` parameter tells the manager to check the leases for renewal every `granularity` milliseconds. As long as the application is running, the lease manager will actively renew the lease. If the user's application is closed or the machine crashes or the network fails, the renewals stop, the lease expires, and the session entry is removed from the space. Since the entry will disappear within a reasonable amount of time, friends won't mistakenly think the user is still logged in. Note that we also pass the lease manager a reference to `this` as the lease's listener—if anything goes awry in renewing the lease, the session object's `notify` method is called, which prints an error message.

The `stop` method is called when a user logs out of the messenger service. If a lease has been created for the session entry, then we ask the lease manager to cancel the lease, which results in the session entry being removed from the space.

Finally, `exists` is a static utility method used to determine if a user is online. The method first sets up a `SessionEntry` template, passing it the `username` and leaving the login time as a wildcard. Since session entries exist in the space only when the user is logged in, the method calls `readIfExists` to determine if a matching session entry resides in the space. If an entry is found, the method returns normally, otherwise the method throws an `UnknownUserSession` exception.

10.5 Friends List

We still have two components of the user distributed data structure to cover—the friends list and the channel. As we saw in Section 10.3, part of creating a new user account is setting up a `FriendsListEntry` object. This object maintains a list of the friends you'd like to keep track of and communicate with. To create a friends list, the `Account` code first instantiates a `FriendsList` object (which is a wrapper class that is used to deal with `FriendsListEntry` objects), and calls its `create` method.

Before looking at the `FriendsList` object, let's look at the `FriendsListEntry`:

```
public class FriendsListEntry implements Entry {  
    public String username;  
    public Vector list;
```

```
public FriendsListEntry() {  
}  
  
public FriendsListEntry(String username) {  
    this.username = username;  
}  
  
public FriendsListEntry(String username, Vector list) {  
    this.username = username;  
    this.list = list;  
}  
  
public void add(String username) {  
    list.addElement(username);  
}  
  
public Enumeration elements() {  
    return list.elements();  
}  
}
```

The `FriendsListEntry` has a `username` field, used to identify the user, and a `list` field, which is a `Vector` used to store friends' usernames.

Besides a handful of constructors, the `FriendsListEntry` provides two methods: `add` and `elements`. The `add` method takes a `username`, and adds that name to the list of friends by calling the `Vector` method `addElement`. The `elements` method returns an `Enumeration` of the usernames stored in the `list` field, which is convenient when we want to iterate over them, as we'll see shortly.

When an account is created for a user, a friends list entry with an empty list of friends is written into the space. Like an account entry, the friends list entry persists in the space as the user comes and goes from the messenger service and can be retrieved whenever the user logs back in—even if from a different location. Unlike an account entry, which is never removed from the space, a friends list entry is taken from the space whenever a user wants to update the list. We define a `FriendsList` object to operate on friends list entries; let's see how it works.

10.5.1 The `FriendsList` Object

`FriendsList` is a helper class, in the same spirit of other helper classes we've already seen in this chapter. Here is its definition:

```
public class FriendsList {  
    private JavaSpace space;  
    private TransactionManager mgr;  
    private String username;  
    private java.awt.List display;  
    private FriendsListMonitor friendsMonitor;  
  
    public FriendsList(Messenger messenger, String username) {  
        this.username = username;  
        this.space = messenger.space();  
        this.mgr = messenger.mgr();  
    }  
  
    public void create(Transaction txn)  
        throws RemoteException, TransactionException  
{  
    FriendsListEntry friendsList =  
        new FriendsListEntry(username, new Vector());  
    space.write(friendsList, txn, Lease.FOREVER);  
}  
  
public void add(String friendName) {  
    FriendsListEntry friendsTemplate =  
        new FriendsListEntry(username);  
    FriendsListEntry friendsList = null;  
    Transaction.Created trc = null;  
  
    try {  
        trc = TransactionFactory.create(mgr, 3000);  
        Transaction txn = trc.transaction;  
        friendsList = (FriendsListEntry)  
            space.take(friendsTemplate, txn, Long.MAX_VALUE);  
        friendsList.add(friendName);  
        space.write(friendsList, txn, Lease.FOREVER);  
        txn.commit();  
    } catch (Exception e) {  
        System.err.println("Couldn't add friend");  
        e.printStackTrace();  
    }  
}
```

```
public void monitor(java.awt.List display) {
    this.display = display;
    if (friendsMonitor == null) {
        friendsMonitor = new FriendsListMonitor();
        friendsMonitor.start();
    }
}

//... inner class to monitor friends list goes here
}
```

A `FriendsList` object has five private fields: `space`, `mgr`, `username`, `display`, and `friendsMonitor`. The `space` and `mgr` fields hold references to the space and transaction manager used by the messenger service implementation, respectively. The `username` identifies the user whose friends list we'll operate on, and `display` refers to the `java.awt.List` component of the user's messenger interface, in which the friends list is displayed (we will return to this topic later). Finally, the `friendsMonitor` field refers to an object instantiated from a `FriendsListMonitor` inner class—a thread that continually tracks and displays the contents of the user's friends list, as we'll see shortly.

The `FriendsList` constructor takes two parameters, a reference to our messenger applet and a `username`. We set the `space` and `mgr` fields to the values returned by calling the messenger's `space` and `mgr` methods, respectively. The `username` is assigned to the `username` field.

The `FriendsList` class's `create` method (called when an account is created for a new user) instantiates a new `FriendsListEntry`, passing the `username` and a zero-length vector representing an empty list of friends, and then writes the entry into the space. These operations occur under the transaction passed to `create`; if any transaction-related exceptions occur, they are passed back to the caller.

The `FriendsList` class also supplies an `add` method, which will be called whenever we want to add a friend to the user's friends list. The `add` method takes the friend's `username` as a parameter. The method removes the user's friends list entry from the space under a transaction, adds the friend's name to the list, and writes the entry back to the space, under the same transaction. If we didn't use a transaction and partial failure occurred during the update process, the space could be left in an inconsistent state. For instance, the friends list might be taken out of the space and never returned. Using a transaction makes the application more robust. Let's look at the details.

The `add` method first sets up a `FriendsListEntry` template, specifying the `username`. After declaring a variable `trc` of type `Transaction.Created`, we call the `TransactionFactory` class's static `create` method, passing it the transaction manager `mgr` and a three-second lease time. If successful, the call to `create`

returns a `Transaction.Created` object, which we assign to the variable `trc`; if a transaction cannot be created, an exception is thrown. Next we assign the variable `txn` a reference to the transaction object `trc.transaction`.

Once we have a transaction in hand, we use the `FriendsListEntry` template to take the user's friends list entry from the space, under the transaction `txn`, and we call the entry's `add` method to append the friend's name to the list. Then we write the updated entry back to the space, again under the transaction `txn`. Finally, we call `commit` on the transaction. The `take` and `write` are performed atomically: If partial failure occurs while they're being executed, the transaction is not committed, and the effect is as if the space operations never happened.

Finally, the `FriendsList` class provides a `monitor` method that is used to track whether the users in the friends list are online (we will see how this is used later). The `monitor` method instantiates an object from an inner class called `FriendsListMonitor`, which is used to start the actual monitoring process. Let's take a closer look.

10.5.2 The `FriendsListMonitor` Object

The `FriendsListMonitor` class is defined as follows:

```
private class FriendsListMonitor extends Thread {  
    private final long SLEEPTIME = 5000; // five seconds  
    private boolean done = false;  
  
    public void run() {  
        Entry template = null;  
        try {  
            Entry temp = new FriendsListEntry(username);  
            template = space.snapshot(temp);  
        } catch (RemoteException e) {  
            System.err.println("Couldn't snapshot");  
            e.printStackTrace();  
        }  
  
        while (!done) {  
            Vector activeFriends = new Vector();  
            FriendsListEntry friendsList = null;  
  
            try {  
                friendsList = (FriendsListEntry)  
                    space.read(template, null, Long.MAX_VALUE);  
            } catch (RemoteException e) {  
                System.err.println("Couldn't read");  
                e.printStackTrace();  
            }  
            if (friendsList != null) {  
                for (int i = 0; i < friendsList.size(); i++) {  
                    String name = (String) friendsList.get(i);  
                    if (space.isOnline(name)) {  
                        activeFriends.addElement(name);  
                    }  
                }  
            }  
            if (activeFriends.size() > 0) {  
                System.out.println("Active friends: " +  
                    activeFriends);  
            }  
            sleep(SLEEPTIME);  
        }  
    }  
}
```

```
        } catch (Exception e) {
            System.err.println("Couldn't read FriendsList");
            e.printStackTrace();
        }

        Enumeration enum = friendsList.elements();
        while (enum.hasMoreElements()) {
            String name = (String)enum.nextElement();
            try {
                Session.exists(space, name);
                activeFriends.addElement(name);
            } catch (UnknownUserSession e1) {
                ; // friend isn't online
            }
        }

        display(activeFriends);

        try {
            Thread.sleep(SLEEPTIME);
        } catch (InterruptedException e) {
            return;
        }
    }
}

private void display(Vector friends) {
    display.removeAll();
    for (int i = 0; i < friends.size(); i++) {
        display.add((String) friends.elementAt(i));
    }
}
```

The `FriendsListMonitor` extends `Thread`, which means when a `FriendsListMonitor` object is instantiated, it creates a separate thread of control that begins executing in the `run` method, which repeatedly loops, reading the user's friends list, determining which of the friends are online, and displaying their names. The thread will stop when the messenger application itself terminates. Let's look at the details.

In the `run` method, we first create a `FriendsListEntry` template, specifying the `username`, and assign it to the variable `temp`. Since the template will be used within a loop to repeatedly read the user's friends list, we call `snapshot` on the template and assign the result to `template`. Snapshotting the template is a wise optimization in this case, since it means the same template won't need to be re-serialized each time through the loop.

Next the `run` method enters a loop. Within the loop, we first set up a zero-length vector `activeFriends`, which we will use to store the names of friends who are currently online. Using the `template`, we read a copy of the user's friends list entry from the space. Then we iterate through an enumeration of the names in the list, calling `Session.exists` on each to determine if that friend is currently online. For each friend that is online, we add the name to the `activeFriends` vector. Once we've enumerated through all the names, we call the `display` method, which, as we will see, prints the names to the messenger console.

10.6 Communication Channel

Our last piece of the user distributed data structure is a communication channel, which is a consumer channel like those we discussed in Chapter 5. This channel is used to collect all messages sent to the user. When new messages arrive they are appended to the tail of the channel, and as they're displayed on the user's messenger console, they're deleted from the head of the channel.

The channel—like the user's account and friends list—persists in the space as the user logs in and out of the service. If the user happens to log out just as new messages are being appended to the channel, but before they are displayed to the console, the messages aren't lost: The next time the user logs back into the messenger, the unread messages are displayed. Let's take a look at the code that implements the communication channel.

10.6.1 Message Entries and Indices

Each message in the channel is represented by a `ChannelMessageEntry`, defined as follows:

```
public class ChannelMessageEntry implements Entry {  
    public String username;  
    public String from;  
    public Date time;  
    public String content;  
    public Integer position;
```

```
public ChannelMessageEntry() {  
}  
  
public ChannelMessageEntry(String username, int position) {  
    this.username = username;  
    this.position = new Integer(position);  
}  
  
public ChannelMessageEntry(String to, String from,  
    String content)  
{  
    this.username = to;  
    this.from = from;  
    this.content = content;  
    this.time = new Date();  
}  
}
```

Each channel message entry has five fields: the `username` of the recipient, a `from` field holding the `username` of the person who sent the message, the `time` and `content` of the message, and a `position` field giving the sequence number of the message in the channel. The `ChannelMessageEntry` class also supplies a few convenience constructors.

Recall that a consumer channel uses two index entries to represent the head and tail of the channel. Here we reuse the `Index` class we introduced in Section 5.6 to implement the head and the tail:

```
public class Index implements Entry {  
    public String type;          // head or tail  
    public String channel;  
    public Integer position;  
  
    public Index() {  
    }  
  
    public Index(String type, String channel) {  
        this.type = type;  
        this.channel = channel;  
    }  
  
    public Index(String type, String channel, Integer position) {  
        this.type = type;
```

```
        this.channel = channel;
        this.position = position;
    }

    public Integer getPosition() {
        return position;
    }

    public void increment() {
        position = new Integer(position.intValue() + 1);
    }
}
```

10.6.2 The Channel Object

To encapsulate operations on the channel distributed data structure—such as creating the channel, appending messages to it, and consuming messages from it—we define a `Channel` class:

```
public class Channel {
    private String username;
    private JavaSpace space;
    private TransactionManager mgr;
    private final long MESSAGELEASETIME =
        604800000; // one week

    public Channel(Messenger messenger, String username) {
        this.username = username;
        this.space = messenger.space();
        this.mgr = messenger.mgr();
    }

    public void create(Transaction txn)
        throws RemoteException, TransactionException
    {
        Index head =
            new Index("head", username, new Integer(1));
        Index tail =
            new Index("tail", username, new Integer(0));

        space.write(head, txn, Lease.FOREVER);
        space.write(tail, txn, Lease.FOREVER);
    }
}
```

```

public void append(ChannelMessageEntry message) {
    try {
        Transaction txn = createTransaction();

        Index tail = takeTail(message.username, txn);
        tail.increment();
        message.position = tail.getPosition();
        writeTail(tail, txn);
        space.write(message, txn, MESSAGELEASE_TIME);

        txn.commit();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

private Transaction createTransaction()
    throws RemoteException, LeaseDeniedException
{
    Transaction.Created trc =
        TransactionFactory.create(mgr, 3000);
    Transaction txn = trc.transaction;
    return txn;
}

//... tail and head helper methods go here

//... code for consuming messages goes here
}

```

A `Channel` object has three fields: the `space` the channel resides in, a transaction manager `mgr` that we use to create transactions involving operations on the channel, and the `username` of the user who “owns” the channel. The `Channel` constructor takes a handle to the messenger applet and the `username` as parameters. It obtains references to the `space` and the transaction manager by calling the applet’s `space` and `mgr` methods, respectively, and assigns the value of the `username` parameter to the `username` field.

The `create` method sets up a new channel in the `space`. As we saw in Chapter 5, this involves creating initial head and tail `Index` entries and writing them to the `space` under the supplied transaction. The `head` starts off with an index of 1, while the `tail` starts off with an index of 0 (recall that the tail being less than the head indicates an empty channel).

The append method adds a new message to the tail end of a channel. We've implemented append much as we did in Chapter 5, where we remove the tail entry of the channel from the space, increment the position of the tail by one, write the updated tail back to the space, and then write the new message to the channel at the new tail position. We write the message to the space with a lease time of one week so that at some point, if messages aren't read by the user, they are expired and removed. Note the head and tail entries of the channel are always written to the space with a lease time of `Lease.Forever`, so they are never expired by the space. The problem with the append code from Chapter 5, however, is that it does nothing to protect against partial failure. If failure occurs anywhere within the append sequence—say, the tail is removed from the space but can't be written back, or the message can't be written to the new tail position—the space may be left in an inconsistent state, and the application breaks. Now that we've learned how to create and use transactions, we've redesigned our append method to make it robust.

The revised append method first creates a transaction (by calling a `createTransaction` helper method we've defined). We take the tail from the space, write the tail back to the space, and write the message to the space—all under the transaction—and then commit the transaction. If partial failure occurs, the space will be left in the same consistent state it was in prior to the transaction.

You'll notice here and in the next section that we use several convenient helper methods for operating on the head and tail entries of the user's channel: `readTail`, `takeTail`, `writeTail`, `takeHead`, and `writeHead` (which you'll find defined in the complete source code). The `readTail` method reads the tail index of the channel and returns the tail's position. The `takeTail` method removes and returns the tail entry for the channel, while `writeTail` writes a tail entry for the channel into the space. The `takeHead` and `writeHead` methods are analogous, but work on head entries instead. Besides `readTail`, all the methods take a transaction parameter and perform their space operation under that transaction.

10.6.3 Retrieving Messages from the Channel

The `Channel` object provides two methods for consuming messages from a channel. One method, `getMessage`, removes a message with a particular sequence number from the channel. The other method, `getMessages`, removes all messages currently in the channel, up to and including the one at the tail.

Here is how the `getMessage` method is defined:

```
public ChannelMessageEntry getMessage(int num, Transaction txn) {  
    ChannelMessageEntry template =  
        new ChannelMessageEntry(username, num);
```

```

try {
    return (ChannelMessageEntry)
        space.take(template, txn, 500);
} catch (Exception e) {
    e.printStackTrace();
    return null;
}
}

```

The method takes two parameters: the position number of the message to retrieve, and the transaction under which the operations should take place. We first instantiate a message template, specifying the username (which uniquely identifies the channel) and the position of the message we want. Then we use the template to take the matching message entry from the space.

Note that the `take` is performed under the transaction that's been passed to the method and with a time-out of 500 milliseconds (if the `take` hasn't found a matching entry within half a second, it will return `null`). We put a time limit on the retrieval, because there is a chance that messages may have been expired and removed from the space if they have been in the space longer than a week (recall that we wrote messages to the space with a lease time of one week). In this case, we don't want to block on a message that will never appear in the space, so after half a second we skip it and continue on to the next message.

The `getMessages` method is used when we want to remove all the messages currently in the channel, from the head to the tail:

```

public Enumeration getMessages() {
    Transaction txn = null;

    try {
        txn = createTransaction();

        // read tail & remove head
        int end = readTail();
        Index head = takeHead(txn);
        int start = head.position.intValue();
        if (end < start) {
            return null;
        }

        // retrieve messages from head to tail
        Vector messages = new Vector();
        for (int i = start; i <= end; i++) {
            ChannelMessageEntry message = getMessage(i, txn);

```

```

        if (message != null) {
            messages.addElement(message);
        }
        head.increment();
    }

    writeHead(head, txn);

    txn.commit();
    return messages.elements();
} catch (Exception e) {
    if (txn != null) {
        try {
            txn.abort();
        } catch (Exception e2) {
            ; // failed to abort
        }
    }
    System.err.println("Failed reading messages");
    e.printStackTrace();
    return null;
}
}

```

The method first creates a transaction, under which it will perform all of its space operations. This ensures that the channel will be left in a consistent state even if partial failure occurs.

Next, `getMessages` calls `readTail`, to determine the sequence number of the last message currently in the channel, and `takeHead`, which removes the head index from the space. You may be wondering why we remove the head from the space but just read the tail. By removing the head, we're ensuring that this is the only process removing messages from this channel for the time being. We don't need to remove the tail, because there's no reason other processes can't append new messages to the channel even as we're removing and displaying older messages.

Next we compare the position numbers of the head and tail; if the tail is less than the head, then the channel is empty and we return `null`. Otherwise, we iterate through the sequence numbers from the head to the tail, calling `getMessage` to remove each message from the channel and adding it to a vector called `messages`. As we consume each message, we increment the head index. Once we've consumed them all, we write the head back into the space; its position will be one greater than the tail we recorded (which means the channel will be empty, unless

other processes have appended messages to it as we've been busy removing messages). Finally we commit the transaction and return the vector of messages (which will be displayed to the messenger console, as we'll see shortly).

Note that the whole sequence of operations to remove channel messages is performed within a `try/catch` clause. If any exceptions occur, we check to see if a transaction has already been created and, if so, we explicitly abort it.

10.7 The Messenger Applet

Finally, we have the various entries and supporting classes behind us and we can now turn our attention to the `Messenger` class that ties the application together. Let's start by looking at the skeleton of the `Messenger` class:

```
public class Messenger extends Applet
    implements ActionListener
{
    //... user interface variables go here

    private JavaSpace space;
    private TransactionManager mgr;
    private String username;
    private String password;

    public void init() {
        super.init();
        showStatus("Getting space");
        space = SpaceAccessor.getSpace();
        mgr = TransactionManagerAccessor.getManager();

        login = new Login(this);
    }

    //... method to set up messenger console interface goes here

    //... actionPerformed method (for console events) goes here

    //... inner class MouseHandler (for console events) goes here

    //... login callback method goes here

    //... inner class Listener goes here
```

```
public JavaSpace space() {
    return space;
}

public TransactionManager mgr() {
    return mgr;
}
}
```

The Messenger is an applet that primarily serves as the basis for the user interface. Our skeleton shows the applet's `init` method, which first gains access to a space and a transaction manager and assigns them to two class variables. These variables are available through two public methods (`space` and `mgr`), which are shown at the bottom of the skeleton and used, as we have already seen, by our helper objects to gain access to a space or a transaction manager.

The applet then instantiates a `Login` object, which is where the fun begins. The `Login` object is responsible for managing the user login. The `Login` object is also responsible for creating an account and its associated data structures (if the user is new), starting a user session, and letting the Messenger know the user has been authenticated by calling the Messenger's `loginCallback` method. Let's see how.

10.7.1 Logging in via the Login Object

When the Messenger instantiates a `Login` object, it passes itself as a parameter. The `Login` object brings up the login window shown in Figure 10.1. We will skip the full implementation details of the user interface, but it's important to examine the `actionPerformed` method that's called whenever the user presses the "Login" button or hits "Enter" in the password field:

```
public void actionPerformed(ActionEvent event) {
    String username = usernameTextField.getText();
    String password = passwordTextField.getText();

    if (username.equals("") || password.equals("")) {
        setStatus("Enter both username & password.");
        return;
    }

    Account userAccount =
        new Account(messenger, username, password);
```

```
try {
    userAccount.validate();
} catch (InvalidPasswordException e) {
    setStatus("Invalid password.");
    return;
} catch (UnknownUserException e1) {
    try {
        userAccount.create();
        messenger.showStatus("Created new account for " +
            username + ".");
    } catch (Exception e2) {
        messenger.showStatus(
            "Could not create account for " +
            username + ".");
        return;
    }
}

Session userSession = new Session(space, username);
try {
    userSession.start();
} catch (Exception e) {
    messenger.showStatus("Could not create session for " +
        username + ".");
    return;
}

dispose();
messenger.loginCallback(username, password);
}
```

This method first ensures that both a username and password have been provided. If not, an error message is printed, the method returns, and the user is free to try again. Otherwise, the method instantiates an Account object (passing it a handle to the messenger applet and the supplied username and password) and invokes its validate method. If validate throws an InvalidPasswordException, the method displays an error message and returns, and the user may try again. If validate throws an UnknownUserException, the account's create method is called to create a new user account—writing an account entry, friends list entry, and channel distributed data structure into the space.

After `create` completes, or if `validate` completes without exceptions, a new session is started for the user. As we saw in Section 10.4.1, this results in a `SessionEntry` being written into the space, indicating that the user is logged in.

Finally, the method destroys the login window by calling `dispose`, and notifies the messenger applet by calling its `loginCallback` method.

10.7.2 The `loginCallback` Method

The `loginCallback` method is defined in `Messenger` and acts to notify the messenger when the user has logged in. This callback also does some important work; namely, it sets up the messenger console shown in Figure 10.1, and adds listeners that respond to graphical events (such as clicking on a name to chat, or pressing the “Add” button to add a friend). The callback also takes care of setting up a friends list and ensuring that it’s continually monitored and displayed to the console. In addition, the callback starts a listener object that will be notified whenever new messages are appended to the user’s channel, so that they can be displayed. Finally, the callback creates a message dispatcher object that is used to display messages in the appropriate message windows.

This is an important method; here is its definition:

```
public void loginCallback(String username, String password) {  
    this.username = username;  
    this.password = password;  
  
    consoleSetup();  
  
    friendsList = new FriendsList(this, username);  
    friendsList.monitor(friendsDisplay);  
  
    try {  
        Listener listener = new Listener();  
        Index template = new Index("tail", username);  
        space.notify(template, null, listener,  
                     Lease.FOREVER, null);  
    } catch (RemoteException e) {  
        e.printStackTrace();  
    } catch (TransactionException e) {  
        e.printStackTrace();  
    }  
    showStatus(username + " logged in.");  
  
    dispatch = new MessageDispatcher(this);  
}
```

The callback is passed the authenticated user's username and password and these are assigned to the `username` and `password` fields of the `Messenger` object. The method then invokes `consoleSetup` to set up the user interface for the messenger console shown in Figure 10.1. Part of the console setup involves adding event listeners that will be called when the user presses the “Add” button, presses “Enter” in the text field, or clicks on a name in the online friends display area. We will come back to these actions shortly.

The messenger also needs to continually monitor the friends who are online and display them to the friends console. To do this, the callback constructs a `FriendsList` object and invokes its `monitor` method. As we discussed in Section 10.5.1, `monitor` starts a `FriendsListMonitor` thread that repeatedly reads a user's friends list entry and prints its contents to the console.

Next, `loginCallback` sets up a mechanism for retrieving and displaying the new messages in the user's channel. In this example, we use a reactive style of finding out about new messages by making use of `notify`. This approach contrasts with the channels we built in Chapter 5, in which a thread looped and repeatedly performed a `read` on the next message in the channel.

To use `notify`, the `loginCallback` method sets up a listener object that will be informed whenever the tail entry of the channel is updated in the space, which indicates that messages have been appended. This object, assigned to `listener`, is instantiated from an inner class of `Messenger`, called `Listener`. The `loginCallback` method then creates a `template` that will match the user's tail entry. With a template and listener in hand, the method calls `notify` on the space to register interest in being informed whenever the tail of the user's channel is updated. The first parameter to `notify` is the template of interest, the second indicates a `null` transaction, the third specifies that the `listener` object should be informed of matching entries, the fourth means the `notify` registration will be remembered indefinitely by the space, and the fifth indicates that there is no hand-back object. So as a result of this registration, whenever the tail entry is written to the space, the listener's `notify` method will be called; we'll get to the details of the `Listener` class in Section 10.7.6.

Finally, `loginCallback` instantiates a `MessageDispatcher` object and assigns it to the `dispatch` field. We'll see shortly that, when new messages are removed from the user's channel, the dispatcher comes into play to direct chat messages to the appropriate chat window on the user's screen.

10.7.3 Adding to the Friends List

Once the user is logged in, the messenger enables the user to manage a friends list and keep track of which friends are online. Let's see how. Recall that we said the `consoleSetup` method creates the messenger console interface and adds event

listeners to respond to events. In particular, when the “Add” button is clicked or “Enter” is pressed in the text field, the messenger’s `actionPerformed` method is called:

```
public void actionPerformed(ActionEvent event) {
    String friend = friendTextField.getText();
    Object object = event.getSource();

    if ((object == addButton) || (object == friendTextField)) {
        if (friend.equals("")) {
            showStatus("Enter a friend's username.");
        } else {
            showStatus("Adding friend " + friend + " to list.");
            friendTextField.setText("");
            friendsList.add(friend);
        }
    }
}
```

This method first makes sure that a name has been typed into the text field, and if so, calls the `add` method on the messenger’s `friendsList` to add the friend. Recall that this causes the user’s `FriendsListEntry` to be retrieved from the space and a new name to be added to the entry’s vector of names. The next time the `friendsList` monitor thread retrieves this entry, the new name will be included in the set of names that are checked for their online status.

10.7.4 Chatting with Friends

When the user clicks on the name in the friends list to initiate a chat session, the `mouseReleased` method of the messenger’s `MouseHandler` inner class is called:

```
class MouseHandler extends java.awt.event.MouseAdapter {
    public void mouseReleased(java.awt.event.MouseEvent event)
    {
        Object object = event.getSource();
        if (object == friendsDisplay) {
            try {
                dispatch.handle(username,
                    friendsDisplay.getSelectedItem(),
                    username, new Date(), "");
            } catch (Exception e) {
```

```
        System.err.println(
            "Couldn't display message window");
        e.printStackTrace();
    }
}
}
```

The `mouseReleased` method calls the `handle` method of the messenger dispatcher. It passes several parameters: the username of the person initiating contact, the name of the friend we clicked on, and the author, date, and content (which in this case is empty) of the message.

If there isn't already a window open on the screen for chatting with this friend, the dispatcher is responsible for setting up a new window. Here's what the `MessageDispatcher` code looks like:

```
public class MessageDispatcher {  
    private Messenger messenger;  
    private Hashtable frames = new Hashtable();  
  
    public MessageDispatcher(Messenger messenger) {  
        this.messenger = messenger;  
    }  
  
    public void handle(String username, String friend,  
                      String author, Date time, String content)  
    {  
        MessageFrame frame = (MessageFrame)frames.get(friend);  
        if (frame == null || !frame.isShowing()) {  
            frame = new MessageFrame(messenger, username, friend);  
            frames.put(friend, frame);  
        }  
        frame.addMessage(author, time, content);  
    }  
}
```

A message dispatcher is essentially a hashtable that maintains the chat windows currently displayed for chatting with various friends. These windows are `MessageFrame` objects; each message frame allows a user to direct messages to, and receive messages from, a particular friend.

When the `handle` method is called, it looks in the hashtable to see if a message frame for chatting with the given friend already exists. If it does, the

addMessage method is called on the frame, to display the author, time, and content of the message. If the frame doesn't yet exist, one is created and displayed (in the form of a chat window as shown in Figure 10.1) and also put into the hashtable.

10.7.5 Sending Messages

Once we have a chat window (a MessageFrame object) open to a friend, we can send messages. The bulk of the code in MessageFrame sets up the user interface. What we are interested in is what happens when the user types a message into a message frame and clicks on the “Send” button or presses the “Enter” key, which is handled by the actionPerformed method:

```
public void actionPerformed(ActionEvent event) {
    String content = messageTextField.getText();

    if (content.equals("")) {
        setStatus("Enter a message.");
        return;
    }

    messageTextField.setText("");

    // add the message here
    addMessage(username, new Date(), content);
    ChannelMessageEntry message =
        new ChannelMessageEntry(friend, username, content);
    Channel channel = new Channel(messenger, friend);
    channel.append(message);
}
```

This method first extracts the text from the message text field (making sure it's not empty) and calls its own addMessage method to display the message in the message display area of the local frame. The method then constructs a ChannelMessageEntry that contains the message's recipient, the local user's name, and the content of the message. Next the method instantiates a Channel object representing the friend's communication channel, and calls append to add the message to that channel.

This is how messages are sent. Now let's see how they are received as they come in on the channel.

10.7.6 Listening for Messages

In Section 10.7.2 we set up a listener object that is notified whenever the tail of the user's communication channel is updated in the space. Let's now take a look at the messenger's inner class `Listener`, which performs this work:

```
public class Listener implements RemoteEventListener, Runnable {  
    private int last = 0;  
  
    public Listener() throws RemoteException {  
        UnicastRemoteObject.exportObject(this);  
    }  
  
    public void notify(RemoteEvent ev) {  
        Thread temp = new Thread(this);  
        temp.start();  
    }  
  
    public void run() {  
        Channel channel = new Channel(Messenger.this, username);  
        Enumeration e = channel.getMessages();  
        while (e.hasMoreElements()) {  
            ChannelMessageEntry message =  
                (ChannelMessageEntry) e.nextElement();  
            dispatch.handle(username, message.from,  
                            message.from, message.time, message.content);  
        }  
    }  
}
```

You'll recall from Chapter 8 that the methods that handle `notify` events are expected to return as quickly as possible, in order to prevent the remote space from blocking. Here, the `notify` method instantiates and starts a new thread that executes the `run` method, thereby offloading the event processing and returning promptly.

The `run` method calls `getMessages` to enumerate all messages currently in the user's channel, and then iterates through them. For each message, the message dispatcher's `handle` method is called. Note that the second argument passed to `handle` is the sender of the message, which `handle` uses to locate a message frame corresponding to that sender. If a message frame exists for that sender, then the message is displayed to that frame, otherwise `handle` creates a new message frame and opens up a chat window for talking to that sender.

10.8 Summary

In this chapter, we've built an interactive messenger service. In doing so, we've had a chance to use most of the techniques and patterns we've been gathering throughout this book.

In the course of developing the messenger, we've made use of the entire Java-Spaces API (from `read` to `notify` to `snapshot`). Our application makes use of advanced techniques we've encountered in recent chapters as well. We've put distributed events to use by reading channel messages reactively (through the use of the `notify` method). We've used leasing on our session entries to ensure that they won't persist long after their corresponding users have logged out of the messenger service. Finally, we've made use of transactions, to make sure our application is robust in the distributed environment, even in the face of partial failure.

Of course, a commercial version of a messenger service would deliver additional features (some of which you'll add through the exercises that follow). But our focus wasn't on bells and whistles: we've concentrated on building a correct and robust application. You can use all the techniques we tied together in the messenger service as a springboard to building other robust distributed applications.

10.9 Exercises

Exercise 10.1 Design and implement a hashing scheme that maps usernames to a set of at least a couple hundred locks.

Exercise 10.2 The messenger doesn't provide any way to view or delete the friends in your friends list, regardless of whether they are logged in. Add code to support this functionality.

Exercise 10.3 Add functionality that invites a third party to an existing chat between two other users.

Exercise 10.4 Our messenger service does not provide "remove user account" functionality. Add code to support this feature. There are at least two approaches you can take. Once logged in, a user could be allowed to remove his own account (but you must figure out a way to keep the user from continuing the current session). Alternately, there could be two classes of users: administrative users, who have the ability to remove user accounts, and non-administrative, who do not have this power.

CHAPTER 11

A Parallel Application

In pioneer days, they used oxen for heavy pulling, and when one ox couldn't budge a log they didn't try to grow a larger ox. We shouldn't be trying for bigger computers, but for more systems of computers.

—Grace Hopper

THIS chapter builds on the compute server we developed in Chapter 6 and explores its use in two directions: First, we are going to extend our simple compute server to make use of a couple of the technologies we covered in the last few chapters, namely transactions and leases. Then, we are going to develop a parallel application on top of the server that will allow us to explore some of the more subtle issues that arise when developing space-based parallel applications. These issues range from keeping the workers busy with tasks (yet not overwhelming the space with too many tasks), to the computation/communication ratio, to cleaning up the space after we use it.

The application we are going to develop is a fun one: breaking password encryption. Let's get started.

11.1 The Compute Server

Before building our application we are first going to rework the simple compute server we developed in Chapter 6. At this point you might want to return to that chapter and quickly review the compute server code.

11.1.1 The Command Interface Revisited

Recall that all tasks computed by the compute server must implement the Command interface, which contains one method, `execute`. To submit a task to a compute server, we create a task entry that implements the command interface and

drop it into the server's bag of tasks. A worker in the compute server eventually removes this task and invokes its `execute` method. The `execute` method computes a task and then optionally returns an entry, which the worker will write back into the space.

Below we present the `Command` interface, with a couple of small changes:

```
public interface Command extends Entry {
    public Entry execute(JavaSpace space);
}
```

First, note that the `execute` method now takes a space as a parameter, which allows tasks to make use of the space within their code. For instance, a task may want to obtain additional data from the space, return intermediate results to the space, or communicate with other tasks.

We've also changed the return type of `execute` to make it more general. Rather than returning an entry type of `ResultEntry`, we return an `Entry` so that tasks no longer have to subclass the `ResultEntry` class.

11.1.2 The Task Entry

While we've gotten rid of `ResultEntry`, we still have a base `TaskEntry` class, which we first introduced in Chapter 6. Here is our updated `TaskEntry`:

```
public class TaskEntry implements Command {
    public Entry execute(JavaSpace space) {
        throw new RuntimeException(
            "TaskEntry.execute() not implemented");
    }

    public long resultLeaseTime() {
        return 1000 * 10 * 60; // 10 minutes
    }
}
```

The changes to this class are fairly minimal. The `execute` method has been updated to reflect the new space parameter specified in the `Command` interface, and also to return an `Entry` rather than a `ResultEntry`. The `execute` still throws a runtime exception, thus forcing subclasses to implement the method. Recall the reason we gave in Section 6.2.1 for defining `TaskEntry` in this manner (rather than defining an abstract class): It's so the compute server can create a `TaskEntry` template and remove any task from a space. If `TaskEntry` where declared `abstract`, then we couldn't use it to instantiate a template.

We've also added one new method, `resultLeaseTime`, that is used to supply a lease time for the result entries that are returned from the `execute` method (and then written back into the space by the worker). The implementor of the `TaskEntry` can override this method to specify a time other than the default for the results of the computation to remain in the space.

Now that we've gotten these minor differences out of the way, let's move on to more interesting changes, starting by reworking the generic worker.

11.1.3 The Generic Worker

To review, the basic operation of the worker is straightforward: In a loop we take a task entry from the space and invoke its `execute` method. If the method returns an entry then we write it back into the space (for the master to pick up). We then return to the top of the loop and begin again.

Our reworked version includes a few enhancements. Here is the code for the generic worker's `run` method:

```
public void run() {
    Entry taskTmp1 = null;
    try {
        taskTmp1 = space.snapshot(new TaskEntry());
    } catch (RemoteException e) {
        throw new RuntimeException("Can't obtain a snapshot");
    }

    while (true) {
        System.out.println("getting task");

        Transaction txn = getTransaction();
        if (txn == null) {
            throw new RuntimeException(
                "Can't obtain a transaction");
        }

        try {
            TaskEntry task = (TaskEntry)
                space.take(taskTmp1, txn, Long.MAX_VALUE);
            Entry result = task.execute(space);
            if (result != null) {
                space.write(result, txn, task.resultLeaseTime());
            }
            txn.commit();
        }
    }
}
```

```

        } catch (RemoteException e) {
            e.printStackTrace();
        } catch (TransactionException e) {
            e.printStackTrace();
        } catch (UnusableEntryException e) {
            e.printStackTrace();
        } catch (InterruptedException e) {
            try {
                txn.abort();
            } catch(Exception ex) {
                // RemoteException or BadTransactionException
                // lease expiration will take care of the
                // transaction
            }
            System.out.println("Task cancelled");
        }
    }
}

```

The first enhancement is the use of `snapshot`. Since we use the same task template over and over, we go ahead and create a snapshotted version of the template to avoid re-serialization on every `take` operation.

We have also improved the generic worker so that it takes into account the possibility of partial failure. Let's first step back and examine one of several scenarios that can occur with partial failure: consider the case where the generic worker removes a task, starts computing it, and then partial failure occurs. As a result, the task entry will be lost, which may have a detrimental effect on a parallel computation.

To make our compute server robust in the face of partial failure, we use transactions. To create a transaction we make use of the method `getTransaction`, which is shown below:

```

public Transaction getTransaction() {
    TransactionManager mgr =
        TransactionManagerAccessor.getManager();

    try {
        long leaseTime = 1000 * 10 * 60; // ten minutes
        Transaction.Created created =
            TransactionFactory.create(mgr, leaseTime);
        return created.transaction;
    }
}

```

```
        } catch(RemoteException e) {
            e.printStackTrace();
            return null;
        } catch(LeaseDeniedException e) {
            e.printStackTrace();
            return null;
        }
    }
}
```

Here we first obtain access to a transaction manager, which is done through our `TransactionManagerAccessor`. We then call `TransactionFactory.create`, passing it a transaction manager and a lease time of ten minutes. This lease time has the following effect on the task entry: If a task is not computed within ten minutes then the transaction is aborted by the transaction manager and the `TaskEntry` is returned to the space (note that our code does nothing to stop the `execute` method if it does not finish in that time period; we leave this as an exercise to the reader).

If an exception occurs creating the transaction, then the exception is printed and `null` is returned from the call to `getTransaction`; otherwise we return the transaction.

Returning to the generic worker's `run` method, when the call to `getTransaction` returns, if a transaction hasn't been created we throw a runtime exception. Otherwise, we call `take` (passing it our snapshotted template and transaction) and wait indefinitely until it returns a task entry. Once a task entry is returned we call its `execute` method and assign the return value to the local variable `result`. If `result` is non-null, then we write `result` back into the space under the transaction, with a lease time obtained from calling `resultLeaseTime`. By overriding the `resultLeaseTime` method, the designer of a `TaskEntry` can specify how long a result entry should live on a per-task basis. Finally, once the result is written back into the space, we commit the transaction.

If any exceptions occur along the way we display the exception and then continue at the top of the loop. If however, we receive an `InterruptedException`, then we explicitly abort the transaction, assuming the user has interrupted the computation.

11.1.4 A Generic Master

We are also going to make a few updates to our `GenericMaster` class. Recall that our previous version from Chapter 6 called the `generateTasks` method to generate a set of tasks and then called `collectResults` to collect any results. To create your own master process you subclass these two methods and supply the code to generate tasks and collect results.

In practice it is often the case that generating tasks and collecting results are two activities that can occur concurrently. Many parallel (as well as distributed) applications generate small sets of tasks in phases over time and collect the results from each phase as they become available. The results from one phase may, in fact, influence the set of tasks that are generated in the next. It may also be the case that a master needs to generate so many tasks that the space would be overwhelmed (or at least resource constrained) in a shared environment; in such a case the master needs to dole out tasks over time, as previous tasks are completed, and not all at once. That will be the case with our application, and we will talk about a technique called *watermarking* that allows us to keep workers busy, while minimizing the impact on the space.

Here is the new code for our updated GenericMaster, which is designed to allow the `generateTasks` and `collectResults` methods to run concurrently:

```
public abstract class GenericMaster extends Applet {  
    protected JavaSpace space;  
  
    public void init() {  
        space = SpaceAccessor.getSpace();  
  
        GenerateThread gThread = new GenerateThread();  
        gThread.start();  
        CollectThread cThread = new CollectThread();  
        cThread.start();  
    }  
  
    protected abstract void generateTasks();  
  
    protected abstract void collectResults();  
  
    //... writeTask and takeTask methods here  
}
```

To allow both methods to work concurrently, we've made a simple alteration to the `GenericMaster` class; rather than successively calling the two methods, it instead creates two new threads: one that calls `generateTasks` and the other that calls `collectResults`.

That concludes our changes to the compute server.

11.2 The Crypt Application

We are now going to develop a parallel application on top of our compute server. Our application is an exercise in applied cryptography, which is another way of saying we are going to write an application that breaks encrypted passwords.

11.2.1 Background

A common method of encrypting passwords, particularly in UNIX systems, is the use of a *one-way encryption algorithm*, which takes the user's password as input and returns the user's encrypted password. For instance, a password of "secret" may result in the output "BgU8DFSLhhz6Q." One-way functions that work well for encryption have the property that, given the encrypted version of a password, it is difficult to compute the inverse of the function to produce the original password (which makes it hard to guess passwords from their encrypted form).

The parallel application we are going to be developing will break passwords through the brute force technique of trying every possible combination of ASCII characters that make up a password. This technique works as follows: given a one-way function, let's call it `crypt`, and an encrypted password `e`, we can iterate through every possible ASCII password `p` and compute `crypt(p)`. We then compare the result of `crypt(p)` with `e`, and if the two are equal, then `p` is the password. If not, then we keep trying.

It isn't difficult to enumerate every possible ASCII password; it can be done by generating a sequence like this:

```
aaaaaaaa  
aaaaaaab  
aaaaaaac  
. . .  
zzzzzzzz
```

with the caveat that characters other than a-z can be used for passwords (in general, all 95 printable ASCII characters can be used).

There are, of course, other methods of trying to break passwords. The most common method is to use a dictionary of words that are then encrypted and checked against the stored encrypted password. This technique often works well in practice, because users often choose common words as passwords. However, it isn't foolproof, because users may choose passwords that aren't in dictionaries.

Exploring the space of every possible password—not just dictionary words—can be a very compute-intensive problem (if not intractable in some cases). Because of this we are going to implement a scaled-down application that breaks passwords of four characters or less. Even with four characters, the space of possible passwords would take a standalone Java application on the order of several hours to compute, which makes a nice size for our parallel experiments.

Our encrypted passwords are based on the UNIX password encryption scheme. The garden-variety UNIX algorithm normally takes a password of eight characters that is prepended by two characters of “salt”—an arbitrary sequence that is prepended to prohibit simple dictionary schemes of password breaking. So to encrypt “secret,” UNIX adds two random characters, say “aw” and then encrypts “awsecret” via a one-way function.

The UNIX `crypt` one-way function is provided in a class called `JCrypt`, written by John F. Dumas for the Java platform, and can be found in the source code for this book. The details of the class are not important, with the exception of the `crypt` method that we make use of in our code. The signature for this method is as follows:

```
public static String crypt(byte[] password);
```

The `crypt` method is a static method that takes a password (already prepended with salt) in the form of a byte array and returns an encrypted version of the password in the form of a `String`.

With this knowledge under our belt, let’s design and build the application. Our application is going to follow the standard replicated-worker pattern: Given an encrypted password, a master process enumerates all possible passwords and generates tasks for the workers to verify whether or not the possible passwords match the encrypted version. The workers pick up the tasks and do the verification by the technique mentioned above—they first encrypt the potential password and then compare it against the encrypted password. If the two match we’ve found the password.

11.2.2 The Generic Worker and Crypt Task

The task entry is the heart of the compute server computation—tasks are written into the space for the `GenericWorker` to compute. Because we make use of the command pattern, the `GenericWorker` simply calls each task’s `execute` method. Even if a worker has never seen a specific subclass of a task entry before, the space and its underlying RMI transport take care of the details of shipping the executable behavior to the worker.

Our workers look for tasks of type `TaskEntry`. Here we extend the task entry to create a new task: the `CryptTask`.

```
public class CryptTask extends TaskEntry {
    public Integer tries;
    public byte[] word;
    public String encrypted;

    public CryptTask() {
    }

    public CryptTask(int tries, byte[] word, String encrypted) {
        this.tries = new Integer(tries);
        this.word = word;
        this.encrypted = encrypted;
    }

    public Entry execute(JavaSpace space) {
        int num = tries.intValue();
        System.out.println("Trying " + getPrintableWord(word));
        for (int i = 0; i < num; i++) {
            if (encrypted.equals(JCrypt.crypt(word))) {
                CryptResult result = new CryptResult(word);
                return result;
            }
            nextWord(word);
        }
        CryptResult result = new CryptResult(null);
        return result;
    }
}
```

Let's step through the `CryptTask` code. A crypt task holds three pieces of information: `tries`, an integer that specifies the number of potential passwords each task should enumerate and try as a possible match against the encrypted version; `word`, a byte array that holds the word with which to begin the enumeration; and `encrypted`, a string that holds the encrypted password we are trying to break. So each task is given a starting point (`word`) and a number of enumerations (`tries`), to attempt matching words against the encrypted password (`encrypted`). All three values can be initialized in a crypt task by calling the supplied convenience constructor.

Next we have the crypt task's `execute` method: we first convert `tries` from its wrapper class into a primitive `integer` that is assigned to `num`. The rest of `execute` consists of one main loop that iterates for `num` times. Each time through the loop, we encrypt a word using the `JCrypt` class's `crypt` method and then compare it against the encrypted word. If the two are equal, then we've broken the password, and we create a `CryptResult` object and return it to the `Worker` process. The `CryptResult` object is simply an entry that holds the broken password in the form of a byte array.

If the two are not equal, then we call `nextWord`, which is shown below:

```
static void nextWord(byte[] word) {
    int pos = 5;
    for (;;) {
        word[pos]++;
        if ((word[pos] & 0x80) != 0) {
            word[pos--] = (byte) '!';
        } else {
            break;
        }
    }
}
```

The `nextWord` method is a static method (as we will see it is also used by the `CryptMaster`) that takes a word in the form of a byte array and alters the array so that it is set to the next word in the ASCII sequence. For instance, if we start at the word "aaaa" and prepend it with the salt sequence "aw" then the word "awaaaa" looks like this if we run it through `nextWord` a few times:

```
awaaaa
awaab
awaac
.
.
.
awaaa!
awaaba
awaabb
.
.
.
awa!!!
awaabaa
awabab
.
.
```

If we take a look at the code for `nextWord`, we see that the method first increments the right-most character. If this character goes beyond the end of the ASCII sequence (which is the hexadecimal number `0x80`) then the character is set to ! (which is the first printable character in the ASCII set), and the neighboring character to the left is incremented by one. The test for `0x80` is then repeated on that character; if equal, the character is also set to ! and the character to the left of it is incremented. And so on.

If `CryptTask` iterates through `num` possible passwords without finding a match, then it falls out of the loop and returns a result entry that has its `word` field set to `null` (the result entry consists of one field, `word`, that is used to hold the password when it is found).

To recap, the `CryptTask` has instructions to check a fixed number of potential passwords against the encrypted password. If a password is found, it is returned in a result entry, but if no password is found, then a result entry is still returned with its `word` field set to `null` (we will see why shortly).

11.2.3 The Crypt Master

Now that we've seen the worker code in detail, let's take a look at the `CryptMaster`, which drives the entire computation. Let's first examine the user interface of the `CryptMaster` (shown in Figure 11.1), which will give us a sense of its operation before we dive into the code details.

The `CryptMaster` interface works as follows: we enter salt and a password (in the first two text entries in the upper left corner of the interface), and then optionally tweak the parameters below the password text entry, such as the “tries

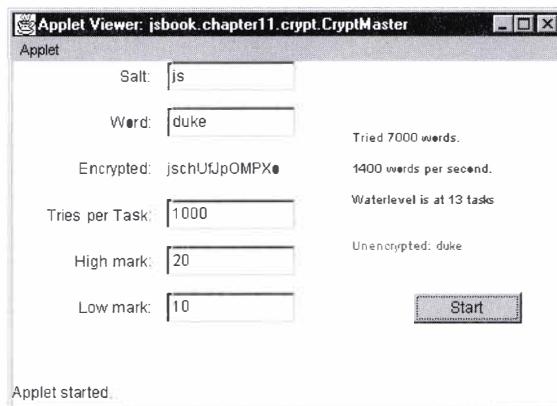


Figure 11.1 The CryptMaster Interface

per task” parameter, which controls how many words each task checks against the encrypted password. We then click on the “Start” button, which starts the application.

Once started, the CryptMaster encrypts our password and displays it just below the password text entry field. The compute server will now be asked to figure out what our original password was, given the encrypted version. To do this, the crypt master begins generating tasks and collecting results until the workers break the password. During the computation, the right side of the interface provides an information display that shows the total number of words that have been checked against the encrypted password, the average number of words being computed per minute by all workers, and the number of tasks in the space waiting to be computed (this is called the “water level,” which we will discuss shortly along with the “high mark” and “low mark” configuration information that appears in the interface).

Below is the skeleton of the CryptMaster code:

```
public class CryptMaster extends GenericMaster
    implements ActionListener
{
    // ... GUI declarations here

    private int lowmark;
    private int highmark;
    private int triesPerTask;
    private int waterlevel = 0;
    private boolean start = false;
    private long startTime;
    private String salt;
    private String unencrypted;

    public void init() {
        super.init();
        // ... GUI setup here
    }

    public void actionPerformed(ActionEvent event) {
        if (start) {
            return;
        }

        String msg = null;
        salt = saltTextField.getText();
```

```
if (salt.equals("")) {
    msg = "Supply a salt value";
}

unencrypted = wordTextField.getText();
if (unencrypted.equals("")) {
    msg = "Supply a word to encrypt";
}

if (unencrypted.length() != 4) {
    msg = "Supply a word of four characters";
}

// JCrypt expects 8 chars
unencrypted = "!!!!" + unencrypted;

try {
    triesPerTask = Integer.parseInt(
        triesTextField.getText());
} catch (NumberFormatException e) {
    msg = "Enter an integer value in \"Tries per Task\".";
}
try {
    highmark = Integer.parseInt(highTextField.getText());
} catch (NumberFormatException e) {
    msg = "Enter an integer value in \"High Mark\".";
}
try {
    lowmark = Integer.parseInt(lowTextField.getText());
} catch (NumberFormatException e) {
    msg = "Enter an integer value in \"Low Mark\".";
}

if (msg == null) {
    start = true;
} else {
    showStatus(msg);
}
}
```

```

    // ... generateTasks() goes here
    // ... collectResults() goes here
    // ... other helper methods go here
}

```

CryptMaster subclasses our GenericMaster class; this basic skeleton shows where fields are declared and initialized and where the user interface is set up.

The notable part of this code is the actionPerformed method, which is called when the “Start” button is clicked. This method first checks to see if the start field has already been set to true (indicating that this method has been called before), and if so simply returns. Otherwise we check the value of the fields in the user interface to make sure they contain appropriate values. If they do, we set the value of several fields, including the start field; otherwise we display an error message in the status area of the applet.

11.2.4 Generating Tasks

Now that our user interface code is out of the way, let’s look at how CryptMaster generates and collects tasks. Recall that GenericMaster (which CryptMaster subclasses) creates two threads, one that calls generateTasks and another that calls collectResults. Here is the code for generateTasks:

```

public void generateTasks() {
    while (!start) {
        try {
            Thread.sleep(250);
        } catch (InterruptedException e) {
            return;      // thread told to stop
        }
    }

    startTime = System.currentTimeMillis();

    String encrypted = JCrypt.crypt(salt, unencrypted);
    encryptedTextField.setText(encrypted);

    byte[] testWord = getFirstWord();

    for (;;) {
        if (testWord[1] != salt.charAt(1)) {
            return;
        }
    }
}

```

```

        CryptTask task =
            new CryptTask(triesPerTask, testWord, encrypted);
        System.out.println("Writing task");
        writeTask(task);
        for (int i = 0; i < triesPerTask; i++) {
            CryptTask.nextWord(testWord);
        }
    }
}

```

The `generateTasks` method begins by waiting for the `start` field to become `true`. As we saw in the last section, this field is set to `true` when the user clicks on the “Start” button.

Once `start` is set to `true`, the `generateTasks` method sets the `startTime` field to the current time and the `encrypted` field to the encrypted version of the password entered in the text entry field (prepended with salt); then the interface is updated to display the encrypted version. Next the method `getFirstWord` is called to obtain the first ASCII sequence (see the complete source code for the method definition), which gets assigned to the field `testWord`; this will be the starting point for enumerating all potential password matches.

Then we enter the main loop of `generateTasks`, which first tests to see if we are at the last word of the ASCII sequence. Recall from our explanation of the `CryptTask`’s `nextWord` method that the sequence is enumerated by “incrementing” through the positions of the word from right to left. Here we check the first character of the password’s salt. When this character is no longer the salt character (in other words, when it, too, has been incremented) then we know we’ve exhausted all password possibilities.

Next we create a `CryptTask` by calling its convenience constructor, supplying the number of tries per task, the encrypted version of the password, and the starting word to begin testing. We then write the crypt task to the space. Next we need to update `testWord`, so that the next task will begin at the word which is `triesPerTask` away from this task’s starting word. To do this we simply call the `nextWord` method `triesPerTask` times (a very fast operation compared to doing the password check of each possibility).

That is the basic version of the `generateTasks` method, which simply enumerates all possible values and partitions them into a number of tasks that need to be checked by workers. The task generation is problematic, however, in that it can generate a very large number of tasks before the password is discovered. In fact, there are over 81 million possible passwords of length four or less. If we choose a “tries per task” of 1,000 then the master will (in the worst case) generate over 80,000 tasks. This could have a serious impact on the resources of the space. Let’s look at one way to improve this code.

How many passwords are possible? For a four-character password we need to check just over 81 million potential passwords. This number comes from the following computation: a password has four character “slots” to be filled, each of which can hold one of 95 possible ASCII characters (we will assume that passwords of 3 characters or less are filled with the space character at the end). Because the choice of each slot is independent of the others we obtain all possibilities by computing $95 \times 95 \times 95 \times 95 = 95^4 = 81,450,625$.

How many eight-character passwords are possible?

11.2.5 Improving Space Usage with Watermarking

One observation we can make is that the space needs to hold only enough tasks to keep workers busy. A technique for keeping a space full enough, yet not *too* full, is called *watermarking*. The word watermarking comes from the two marks often found on shorelines marking high and low tide. Here we choose a high point and low point, meaning the upper and lower limits on the number of entries we’d like in the space. We fill the space with entries until it reaches the high mark, and then wait until the number of entries falls below the low mark, at which point we start filling it with tasks again.

As an illustration of how watermarking works, assume you have a set of ten workers at your disposal. You might want to set the high and low marks to, say, twelve and twenty respectively. That way you always have at least enough tasks for all workers but at most twenty entries in the space. Tuning watermarks is more of an art than a science. In our case we are going to assume that the high and low marks are set when the computation begins and never change. Often the number of workers varies over time, and the high and low values could be modified adaptively—but we won’t get that sophisticated here.

Let’s apply the watermarking principle to our `generateTasks` code. Here is a new version of the `generateTasks` main loop, updated to use watermarking:

```
for (;;) {
    waitForLowWaterMark(lowmark);

    while (waterlevel < highmark) {
        if (testWord[1] != salt.charAt(1)) {
            return;
        }
        CryptTask task =
            new CryptTask(triesPerTask, testWord, encrypted);
```

```

        System.out.println("Writing task");
        writeTask(task);
        changeWaterLevel(1);
        for (int i = 0; i < triesPerTask; i++) {
            CryptTask.nextWord(testWord);
        }
    }
}

```

And here are the definitions of a couple helper methods that are used:

```

synchronized void waitForLowWaterMark(int level) {
    while (waterlevel > level) {
        try {
            wait();
        } catch (InterruptedException e) {
            ; // continue
        }
    }
}

private synchronized void changeWaterLevel(int delta) {
    waterlevel += delta;
    waterMarkLabel.setText("Waterlevel is at " +
        waterlevel + " tasks");
    notifyAll();
}

```

Our first addition to the main loop is the call to `waitForLowWaterMark`, which causes the method to wait until the water level is equal to our low watermark (which is specified in the user interface). This is determined by the `waterlevel` field, which is initially set to zero in its declaration.

Next we've inserted an additional loop, which continues as long as the water level is less than the high watermark. For each task we write into the space we increase the water level by one, until it exceeds the high watermark. As we will see in the next section, as the results are collected this watermark is decremented. Only when the low watermark is reached do we begin adding more tasks.

11.2.6 Collecting Results

Now let's look at how results are collected. The code for `collectResults` is given below:

```

public void collectResults() {
    int count = 0;
    Entry template;

    try {
        template = space.snapshot(new CryptResult());
    } catch (RemoteException e) {
        throw new RuntimeException("Can't create a snapshot");
    }

    for (;;) {
        CryptResult result = (CryptResult)takeTask(template);
        if (result != null) {
            count++;
            triedLabel.setText("Tried " +
                (count * triesPerTask) + " words.");
            updatePerformance(count * triesPerTask);
            changeWaterLevel(-1);
            if (result.word != null) {
                String word =
                    CryptTask.getPrintableWord(result.word);
                answerLabel.setText("Unencrypted: " +
                    word.substring(6));
                addPoison();
                return;
            }
        }
    }
}

```

In the `collectResults` method we first declare a local variable `count`, which will be used to keep a count of the number of tasks that have been computed. We then create a snapshotted version of the `CryptResult` entry, which will be used as a template to take results from the space. Once again, the result template does not vary when it is used, so snapshotting the entry is better than reserializing the entry each time we call `take`.

Next we enter the main loop of the method; in this loop we first take a result entry from the space, waiting as long as necessary. Once we have retrieved a result entry, we increment the `count` variable and update the information display of the user interface by printing the total number of words that have been tried against the encrypted password (computed by `count * triesPerTask`). We then update

the performance area of the information display by calling `updatePerformance` with the total number of word tries as a parameter. The code for `updatePerformance` is given as follows:

```
public void updatePerformance(long wordsTried) {
    long now = System.currentTimeMillis();
    long wordRate = wordsTried / ((now - startTime) / 1000);
    perfLabel.setText(wordRate + " words per second.");
}
```

First we get the current time and assign it to `now`. Then we compute the average number words computed per second, which is `wordsTried` divided by the number of seconds since `CryptMaster` began running (which is `((now - startTime)/1000)`). This gives us an average number of words computed per second. We then set the appropriate label in the information display.

Now, returning to `collectResults`, after updating the display we decrement the water level, since we know a task has been removed and computed. We then check to see if the result entry we retrieved has the cracked password in it. If `result.word` is non-null, then we have the password and we print it in the information display.

At this point we are finished, but before we return from `collectResults`, we first call `addPoison`, which we describe in the next section.

11.2.7 A Little Poison

The `addPoison` method allows us to clean up the space, which is needed because, when a valid password is found, the space may be left with task entries that no longer need to be computed. There may also be result entries that need to be cleaned up as well. In the latter case, we can expect leases to take care of removing the results. However, the disadvantage of leaving the task entries around is that the workers will continue to compute them, which would be a waste of resources in a shared compute server environment.

One approach to cleaning up task entries is to have the master collect the remaining tasks until none remain. A better approach is to let the workers clean up the tasks themselves. This can be done using a variant of a barrier (see Chapter 4), which is referred to as a poison pill. Poison pills work like this: When a task's `execute` method is called by the worker, the task checks to see if a poison pill exists in the space. If one exists, then the task simply returns (and is therefore not computed), but if no pill exists, the task is computed.

To implement a poison pill we first create a poison entry:

```
public class PoisonPill implements Entry {
    public PoisonPill() {
    }
}
```

As you can see the `PoisonPill` is a simple entry, whose only purpose is to act as a barrier in the space.

To make use of the poison pill we also need to alter the `execute` method of our `CryptTask` entry:

```
public Entry execute(JavaSpace space) {
    PoisonPill template = new PoisonPill();
    try {
        if (space.readIfExists(template, null,
                               JavaSpace.NO_WAIT) != null)
        {
            return null;
        }
    } catch (Exception e) {
        ; // continue on
    }
    int num = tries.intValue();
    System.out.println("Trying " + getPrintableWord(word));
    for (int i = 0; i < num; i++) {
        if (encrypted.equals(JCrypt.crypt(word))) {
            CryptResult result = new CryptResult(word);
            return result;
        }
        nextWord(word);
    }
    CryptResult result = new CryptResult(null);
    return result;
}
```

Here we first look for a poison pill in the space using `readIfExists`. If one exists then we skip the computation and return `null`. Otherwise, we compute the entry as normal. When the poison pill exists in the space, it has a flushing effect on the remaining task entries: Whenever a worker takes a task and calls `execute`, a poison pill is found and the method returns immediately, so tasks are removed from the space instead of being computed.

11.2.8 Exploring Crypt

Finally, our application is complete. Now it is time to fire it up and experiment. As with the previous version of the compute server we first start up one or more generic workers and then start up the *CryptMaster* applet. At this point you should get a feel for how quickly the words are computed using one machine. If you are using a garden variety machine, then you should expect to see thousands of words processed per second.

With that benchmark behind you, begin to experiment with adding more machines. Pay attention to how the word rate increases as processors are added to the computation.

At this point you may want to tweak the “tries per task” parameter. If you collect data across a number of runs, you should see the effect of computation versus communication. As the tries-per-task parameter is raised, computation plays a larger role in the runtime behavior; as it is lowered, communication plays a larger role.

If you have access to a large number of machines, you will also want to study the *speedup* gained with each new processor. For a reasonable number of processors and a large enough “tries per task” you should expect to see perfect speedup. Perfect speedup means that if you use n processors you will compute a problem n times as fast as one processor. However as more and more processors are added, the effect may eventually level off, given contention for the space resource.

11.3 Summary

In this chapter we’ve explored not only the mechanics of creating a compute server, but also some of the subtle issues that arise in building parallel applications. As we’ve seen, creating a replicated-worker application isn’t as easy as just generating tasks, dropping them into a space, and letting workers compute them. Often we need to consider the granularity of the tasks as well as the resource constraints of the space in a shared environment.

The compute server application we developed in this chapter is simple, yet an interesting basis for further study and exploration. Much would need to be done to commercialize such a server, and there is much that you can learn from experimenting and extending the current server; the exercises provide a starting point for further study.

11.4 Exercises

Exercise 11.1 Improve the compute server such that it enforces a strict limit on the time a task can be computed. After that time, the task is aborted and returned to the space.

Exercise 11.2 How intractable is the brute force technique of breaking eight character UNIX passwords (how long would it take)?

Exercise 11.3 Experiment with the tries-per-task parameter and its effect on the running time of the crypt application. Make a graph of the words computed per second as you vary the tries-per-task. How does “tries per task” affect the computation/communication ratio?

Exercise 11.4 Design and implement an adaptive scheme for watermark values—as workers are added and removed from the compute server, alter the low and high watermarks to use the space more efficiently.

CHAPTER 12

Further Exploration

It would appear that we have reached the limits of what it is possible to achieve with computer technology, although one should be careful with such statements, as they tend to sound pretty silly in five years.

—John Von Neumann (circa 1949)

JAVASPACES technology is a new way of building distributed programs. In this book we have attempted to provide a set of principles and patterns that can be used as a foundation upon which our understanding and use of the technology can mature. Throughout the book, you've experimented with the spaces programming model to build applications ranging from simple "Hello World" programs to rather sophisticated collaborative and parallel applications.

As you use JavaSpaces technology to solve your own real-world programming problems, new principles and patterns will emerge. We'd very much like to learn from your practical experience; we encourage you to contact the authors at the following address: javaspaces@awl.com.

In the meantime, we leave you with a set of resources so that you can further explore space-based programming and its foundational Jini technology, which we touched upon only briefly. Here you will find a mixture of historical resources, background reading on distributed systems and transactions, and official Sun Microsystems specifications that are relevant to programming with spaces.

12.1 Online Resources

- ◆ Sun Microsystems web site for this book (where you can find complete source code to all the examples in the book): <http://java.sun.com/docs/books/jini/javaspaces>
- ◆ Sun Microsystems Java Distributed Computing web site: <http://java.sun.com/products/javaspaces/>

- ◆ Sun Microsystems Jini technology page: <http://www.sun.com/jini/>
- ◆ Sun Microsystems mailing list for discussion of the Javaspaces technology:
<http://archives.java.sun.com/archives/javaspaces-users.html>

12.2 Related Java and Jini Technologies

The following is the list of Sun Microsystems' official specifications that relate to space-based programming:

- ◆ *Jini™ Entry Specification* (appears in Appendix A)
- ◆ *Jini™ Entry Utilities Specification* (appears in Appendix B)
- ◆ *JavaSpaces™ Specification* (appears in Appendix C)
- ◆ *Java™ Remote Method Invocation Specification*: <http://java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.html>
- ◆ *Java™ Object Serialization Specification*: <http://java.sun.com/products/jdk/1.2/docs/guide/serialization/spec/serialTOC.doc.html>
- ◆ *Jini™ Distributed Event Specification*: <http://www.sun.com/jini/specs/>
- ◆ *Jini™ Distributed Leasing Specification*: <http://www.sun.com/jini/specs/>
- ◆ *Jini™ Transaction Specification*: <http://www.sun.com/jini/specs/>

12.3 Background Reading

12.3.1 Distributed Systems

For background reading on principles and theory of distributed systems, refer to the following:

- ◆ *Distributed Algorithms*, by Nancy A. Lynch (Morgan Kaufman, 1997).
- ◆ *Distributed Systems*, Sape Mullender, editor (Addison-Wesley, 1993).
- ◆ *Distributed Systems: Concepts and Design*, by George Coulouris, Jean Dollimore, Tim Kindberg (Addison-Wesley, 1994).

- *A Note on Distributed Computing*, Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. Sun Microsystems technical report SMLI TR-94-29, <http://www.sunlabs.com/technical-reports/1994/abstract-29.html>

For discussion of interprocess communication and synchronization issues relevant to Chapters 4 and 5, refer to:

- ◆ Chapter 2 of *Operating Systems: Design and Implementation*, by Andrew S. Tanenbaum (Prentice-Hall, 1987).

12.3.2 Transactions

For general information on the topic of transactions, refer to the following:

- ◆ *Transaction Processing: Concepts and Techniques*, by Jim Gray and Andreas Reuter (Morgan Kaufman, 1992).

For information on transactions in space-based systems in particular, refer to:

- ◆ “Adding Fault-tolerant Transaction Processing to LINDA,” by Scott R. Cannon, David Dunn, *Software—Practice and Experience*, Vol. 24(5) (May, 1994), pp. 449-446.
- ◆ “Persistent Linda: Linda + Transactions + Query Processing,” by Brian G. Anderson, Dennis Shasha, *Proceedings of the 13th Symposium on Fault-Tolerant Distributed Systems*, 1994.

12.4 Historical Resources

JavaSpaces technology is the descendant of almost two decades of academic research on space-based systems. The genesis of space-based systems is David Gelernter’s Linda coordination language, which was developed at Yale University in the early 1980s. Gelernter’s original paper remains the definitive work on the space- and time-uncoupled nature of space-based systems and offers many insights into the paradigm:

- ◆ “Generative Communication in Linda,” by David Gelernter, *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 1 (January, 1985), pp. 80-112.

More recent publications by Gelernter and Nicholas Carriero focus on common programming idioms in tuple-based systems. While these publications have a bias toward parallel computation, the techniques are applicable to distributed applications as well. For a short read, refer to:

- ◆ “How to Write Parallel Programs: A Guide to the Perplexed,” by Nicholas Carriero and David Gelernter, *ACM Computing Surveys* (Sept., 1989).

A more in-depth treatment can be found in:

- ◆ *How to Write Parallel Programs: A First Course*, by Nicholas Carriero and David Gelernter (MIT Press, 1990).

For a vision of how tuple-based systems can harness the computing power of networked machines and can be used to create continually updating software models of the real world, please see:

- ◆ *Mirror Worlds: Or the Day Software Puts the Universe in a Shoebox . . . How It Will Happen and What It Will Mean*, by David Gelernter (Oxford University Press, 1991).

Appendix A

The Jini™ Entry Specification

A.1 Entries and Templates

ENTRIES are designed to be used in distributed algorithms for which exact-match lookup semantics are useful. An entry is a typed set of objects, each of which may be tested for exact match with a template.

A.1.1 Operations

A service that uses entries will support methods that let you use entry objects. In this document we will use the term “operation” for such methods. There are three types of operations:

- ◆ *Store operations*—operations that store one or more entries, usually for future matches.
- ◆ *Match operations*—operations that search for entries that match one or more templates.
- ◆ *Fetch operations*—operations that return one or more entries.

It is possible for a single method to provide more than one of the operation types. For example, consider a method that returns an entry that matches a given

template. Such a method can be logically split into two operation types (match and fetch), so any statements made in this specification about either operation type would apply to the appropriate part of the method's behavior.

A.1.2 Entry

An entry is a typed group of object references represented by a class that implements the marker interface `net.jini.core.entry.Entry`. Two different entries have the same type if and only if they are of the same class.

```
package net.jini.core.entry;
public interface Entry extends java.io.Serializable { }
```

For the purpose of this specification, the term “field” when applied to an entry will mean fields that are public, non-static, non-transient, and non-final. Other fields of an entry are not affected by entry operations. In particular, when an entry object is created and filled in by a fetch operation, only the public non-static, non-transient, and non-final fields of the entry are set. Other fields are not affected, except as set by the class’s no-arg constructor.

Each `Entry` class must provide a public no-arg constructor. Entries may not have fields of primitive type (`int`, `boolean`, etc.), although the objects they refer to may have primitive fields and non-public fields. For any type of operation, an attempt to use a malformed entry type that has primitive fields or does not have a no-arg constructor throws `IllegalArgumentException`.

A.1.3 Serializing Entry Objects

`Entry` objects are typically not stored directly by an entry-using service (one that supports one or more entry operations). The client of the service will typically turn an `Entry` into an implementation-specific representation that includes a serialized form of the entry’s class and each of the entry’s fields. (This transformation is typically not explicit but is done by a client-side proxy object for the remote service.) It is these implementation-specific forms that are typically stored and retrieved from the service. These forms are not directly visible to the client, but their existence has important effects on the operational contract. The semantics of this section apply to all operation types, whether the above assumptions are true or not for a particular service.

Each entry has its fields serialized separately. In other words, if two fields of the entry refer to the same object (directly or indirectly), the serialized form that is compared for each field will have a separate copy of that object. This is true only of different fields of an entry; if an object graph of a particular field refers to the same object twice, the graph will be serialized and reconstituted with a single copy of that object.

A fetch operation returns an entry that has been created by using the entry type's no-arg constructor, and whose fields have been filled in from such a serialized form. Thus, if two fields, directly or indirectly, refer to the same underlying object, the fetched entry will have independent copies of the original underlying object.

This behavior, although not obvious, is both logically correct and practically advantageous. Logically, the fields can refer to object graphs, but the entry is not itself a graph of objects and so should not be reconstructed as one. An entry (relative to the service) is a set of separate fields, not a unit of its own. From a practical standpoint, viewing an entry as a single graph of objects requires a matching service to parse and understand the serialized form, because the ordering of objects in the written entry will be different from that in a template that can match it.

The serialized form for each field is a `java.rmi.MarshalledObject` object instance, which provides an `equals` method that conforms to the above matching semantics for a field. `MarshalledObject` also attaches a codebase to class descriptions in the serialized form, so classes written as part of an entry can be downloaded by a client when they are retrieved from the service. In a store operation, the class of the entry type itself is also written with a `MarshalledObject`, ensuring that it, too, may be downloaded from a codebase.

A.1.4 UnusableEntryException

A `net.jini.core.entry.UnusableEntryException` will be thrown if the serialized fields of an entry being fetched cannot be deserialized for any reason:

```
package net.jini.core.entry;

public class UnusableEntryException extends Exception {
    public Entry partialEntry;
    public String[] unusableFields;
    public Throwable[] nestedExceptions;
    public UnusableEntryException(Entry partial,
        String[] badFields, Throwable[] exceptions) {...}
    public UnusableEntryException(Throwable e) {...}
}
```

The `partialEntry` field will refer to an entry of the type that would have been fetched, with all the usable fields filled in. Fields whose deserialization caused an exception will be `null` and have their names listed in the `unusableFields` string array. For each element in `unusableFields` the corresponding element of `nestedExceptions` will refer to the exception that caused the field to fail deserialization.

If the retrieved entry is corrupt in such a way as to prevent even an attempt at field deserialization (such as being unable to load the exact class for the entry), `partialEntry` and `unusableFields` will both be `null`, and `nestedExceptions` will be a single element array with the offending exception.

The kinds of exceptions that can show up in `nestedExceptions` are:

- ◆ `ClassNotFoundException`: The class of an object that was serialized cannot be found.
- ◆ `InstantiationException`: An object could not be created for a given type.
- ◆ `IllegalAccessException`: The field in the entry was either inaccessible or `final`.
- ◆ `java.io.ObjectStreamException`: The field could not be deserialized because of object stream problems.
- ◆ `java.rmi.RemoteException`: When a `RemoteException` is the nested exception of an `UnusableEntryException`, it means that a remote reference in the entry's state is no longer valid (more below). Remote errors associated with a method that is a fetch operation (such as being unable to contact a remote server) are not reflected by `UnusableEntryException` but in some other way defined by the method (typically by the method throwing `RemoteException` itself).

Generally speaking, storing a remote reference to a non-persistent remote object in an entry is risky. Because entries are stored in serialized form, entries stored in an entry-based service will typically not participate in the garbage collection that keeps such references valid. However, if the reference is not persistent because the referenced server does not export persistent references, that garbage collection is the only way to ensure the ongoing validity of a remote reference. If a field contains a reference to a non-persistent remote object, either directly or indirectly, it is possible that the reference will no longer be valid when it is deserialized. In such a case the client code must decide whether to remove the entry from the entry-fetching service, to store the entry back into the service, or to leave the service as it is.

In the 1.2 Java™ Development Kit (JDK) software, activatable object references fit this need for persistent references. If you do not use a persistent type, you will have to handle the above problems with remote references. You may choose instead to have your entries store information sufficient to look up the current reference rather than putting actual references into the entry.

A.1.5 Templates and Matching

Match operations use entry objects of a given type, whose fields can either have *values* (references to objects) or *wildcards* (`null` references). When considering a template T as a potential match against an entry E , fields with values in T must be matched exactly by the value in the same field of E . Wildcards in T match any value in the same field of E .

The type of E must be that of T or be a subtype of the type of T , in which case all fields added by the subtype are considered to be wildcards. This enables a template to match entries of any of its subtypes. If the matching is coupled with a fetch operation, the fetched entry must have the type of E .

The values of two fields match if `MarshalledObject.equals` returns `true` for their `Marshall edObject` instances. This will happen if the bytes generated by their serialized form match, ignoring differences of serialization stream implementation (such as blocking factors for buffering). Class version differences that change the bytes generated by serialization will cause objects not to match. Neither entries nor their fields are matched using the `Object.equals` method or any other form of type-specific value matching.

You can store an entry that has a `null`-valued field, but you cannot match explicitly on a `null` value in that field, because `null` signals a wildcard field. If you have a field in an entry that may be variously `null` or not, you can set the field to `null` in your entry. If you need to write templates that distinguish between set and unset values for that field, you can (for example) add a `Boolean` field that indicates whether the field is set and use a `Boolean` value for that field in templates.

An entry that has no wildcards is a valid template.

A.1.6 Serialized Form

Class	serialVersionUID	Serialized Fields
<code>UnusableEntryException</code>	<code>-219908366668626172L</code>	<i>all public fields</i>

Appendix B

The Jini™ Entry Utilities Specification

B.1 Entry Utilities

ENTRIES are designed to be used in distributed algorithms for which exact-match lookup semantics are useful. An entry is a typed set of objects, each of which may be tested for exact match with a template. The details of entries and their semantics are discussed in the *Jini Entry Specification*.

When designing entries, certain tasks are commonly done in similar ways. This specification defines a utility class for such common tasks.

B.1.1 AbstractEntry

The class `net.jini.entry.AbstractEntry` is a specific implementation of `Entry` that provides useful implementations of `equals`, `hashCode`, and `toString`:

```
package net.jini.entry;

public abstract class AbstractEntry implements Entry {
    public boolean equals(Object o) {...}
    public int hashCode() {...}
    public String toString() {...}
    public static boolean equals(Entry e1, Entry e2) {...}
```

```
public static int hashCode(Entry entry) {...}
public static String toString(Entry entry) {...}
}
```

The static method `AbstractEntry.equals` returns `true` if and only if the two entries are of the same class and for each field F , the two objects' values for F are either both `null` or the invocation of `equals` on one object's value for F with the other object's value for F as its parameter returns `true`. The static method `hashCode` returns zero XOR the `hashCode` invoked on each non-`null` field of the entry. The static method `toString` returns a string that contains each field's name and value. The non-static methods `equals`, `hashCode`, and `toString` return a result equivalent to invoking the corresponding static method with `this` as the first argument.

B.1.2 Serialized Form

Class	serialVersionUID	Serialized Fields
<code>AbstractEntry</code>	5071868345060424804L	<i>none</i>

Appendix C

The JavaSpaces™ Specification

C.1 Introduction

DISTRIBUTED systems are hard to build. They require careful thinking about problems that do not occur in local computation. The primary problems are those of partial failure, greatly increased latency, and language compatibility. The Java™ programming language has a remote method invocation system called RMI that lets you approach general distributed computation in the Java programming language using techniques natural to the Java programming language and application environment. This is layered on the Java platform's object serialization mechanism to marshal parameters of remote methods into a form that can be shipped across the wire and unmarshalled in a remote server's Java™ virtual machine (JVM).

This specification describes the architecture of JavaSpaces™ technology, which is designed to help you solve two related problems: distributed persistence and the design of distributed algorithms. JavaSpaces services use RMI and the serialization feature of the Java programming language to accomplish these goals.

C.1.1 The JavaSpaces Application Model and Terms

A JavaSpaces service holds *entries*. An entry is a typed group of objects, expressed in a class for the Java platform that implements the interface `net.jini.core.entry.Entry`. Entries are described in detail in the *Jini™Entry Specification*.

An entry can be *written* into a JavaSpaces service, which creates a copy of that entry in the space¹ that can be used in future lookup operations.

You can look up entries in a JavaSpaces service using *templates*, which are entry objects that have some or all of its fields set to specified *values* that must be matched exactly. Remaining fields are left as *wildcards*—these fields are not used in the lookup.

There are two kinds of lookup operations: *read* and *take*. A *read* request to a space returns either an entry that matches the template on which the read is done, or an indication that no match was found. A *take* request operates like a read, but if a match is found, the matching entry is removed from the space.

You can request a JavaSpaces service to *notify* you when an entry that matches a specified template is written. This is done using the distributed event model contained in the package `net.jini.core.event` and described in the *Jini™ Distributed Event Specification*.

All operations that modify a JavaSpaces service are performed in a transactionally secure manner with respect to that space. That is, if a write operation returns successfully, that entry was written into the space (although an intervening take may remove it from the space before subsequent lookup of yours). And if a take operation returns an entry, that entry has been removed from the space, and no future operation will read or take the same entry. In other words, each entry in the space can be taken at most once. Note, however, that two or more entries in a space may have exactly the same value.

The architecture of JavaSpaces technology supports a simple transaction mechanism that allows multi-operation and/or multi-space updates to complete atomically. This is done using the two-phase commit model under the default transaction semantics, as defined in the package `net.jini.core.transaction` and described in the *Jini™ Transaction Specification*.

Entries written into a JavaSpaces service are governed by a lease, as defined in the package `net.jini.core.lease` and described in the *Jini™ Distributed Lease Specification*.

Distributed Persistence

Implementations of JavaSpaces technology provide a mechanism for storing a group of related objects and retrieving them based on a value-matching lookup for specified fields. This allows a JavaSpaces service to be used to store and retrieve objects on a remote system.

¹ The term “space” is used to refer to a JavaSpaces service implementation.

Distributed Algorithms as Flows of Objects

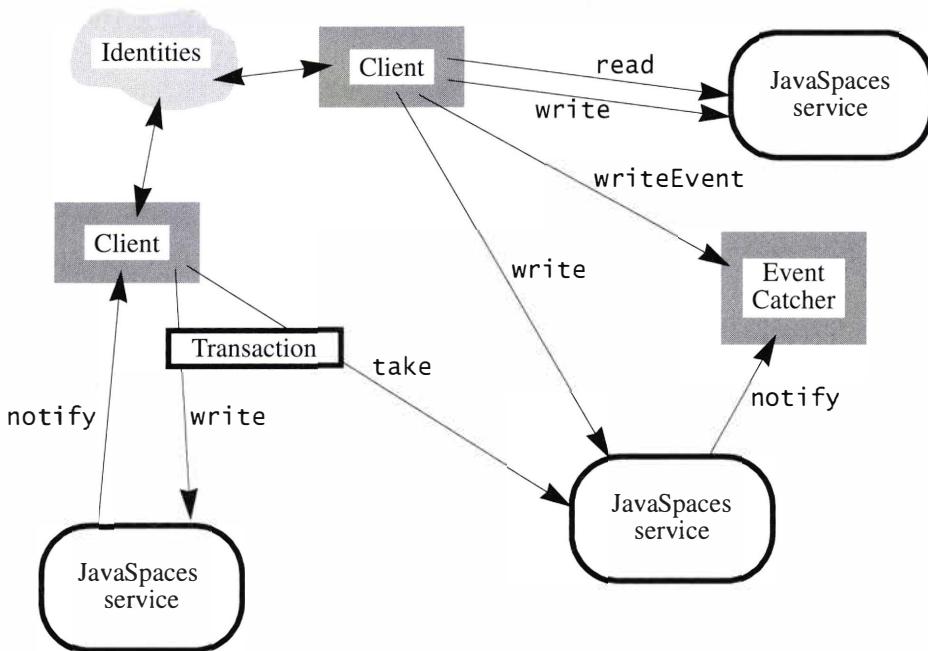
Many distributed algorithms can be modeled as a flow of objects between participants. This is different from the traditional way of approaching distributed computing, which is to create method-invocation-style protocols between participants. In this architecture’s “flow of objects” approach, protocols are based on the movement of objects into and out of implementations of JavaSpaces technology.

For example, a book-ordering system might look like this:

- ◆ A book buyer wants to buy 100 copies of a book. The buyer writes a request for bids into a particular public JavaSpaces service.
- ◆ The broker runs a server that takes those requests out of the space and writes them into a JavaSpaces service for each book seller who registered with the broker for that service.
- ◆ A server at each book seller removes the requests from its JavaSpaces service, presents the request to a human to prepare a bid, and writes the bid into the space specified in the book buyer’s request for bids.
- ◆ When the bidding period closes, the buyer takes all the bids from the space and presents them to a human to select the winning bid.

A method-invocation-style design would create particular remote interfaces for these interactions. With a “flow of objects” approach, only one interface is required: the `net.jini.space.JavaSpace` interface.

In general, the JavaSpaces application world looks like this:



Clients perform operations that map entries or templates onto JavaSpaces services. These can be singleton operations (as with the upper client), or contained in transactions (as with the lower client) so that all or none of the operations take place. A single client can interact with as many spaces as it needs to. Identities are accessed from the security subsystem and passed as parameters to method invocations. Notifications go to event catchers, which may be clients themselves or proxies for a client (such as a store-and-forward mailbox).

C.1.2 Benefits

JavaSpaces services are tools for building distributed protocols. They are designed to work with applications that can model themselves as flows of objects through one or more servers. If your application can be modeled this way, JavaSpaces technology will provide many benefits.

JavaSpaces services can provide a reliable distributed storage system for the objects. In the book-buying example, the designer of the system had to define the protocol for the participants and design the various kinds of entries that must be passed around. This effort is akin to designing the remote interfaces that an equivalent customized service would require. Both the JavaSpaces system solution and the customized solution would require someone to write the code that presented

requests and bids to humans in a GUI. And in both systems, someone would have to write code to handle the seller's registrations of interest with the broker.

The server for the model that uses the JavaSpaces API would be implemented at that point.

The customized system would need to implement the servers. These servers would have to handle concurrent access from multiple clients. Someone would need to design and implement a reliable storage strategy that guaranteed the entries written to the server would not be lost in an unrecoverable or undetectable way. If multiple bids needed to be made atomically, a distributed transaction system would have to be implemented.

All these concerns are solved in JavaSpaces services. They handle concurrent access. They store and retrieve entries atomically. And they provide an implementation of the distributed transaction mechanism.

This is the power of the JavaSpaces technology architecture—many common needs are addressed in a simple platform that can be easily understood and used in powerful ways.

JavaSpaces services also help with data that would traditionally be stored in a file system, such as user preferences, e-mail messages, and images. Actually, this is not a different use of a JavaSpaces service. Such uses of a file system can equally be viewed as passing objects that contain state from one external object (the image editor) to another (the window system that uses the image as a screen background). And JavaSpaces services enhance this functionality because they store objects, not just data, so the image can have abstract behavior, not just information that must be interpreted by some external application(s).

JavaSpaces services can provide distributed *object* persistence with objects in the Java programming language. Because code written in the Java programming language is downloadable, entries can store objects whose behavior will be transmitted from the writer to the readers, just as in an RMI using Java technology. An entry in a space may, when fetched, cause some active behavior in the reading client. This is the benefit of storing objects, not just data, in an accessible repository for distributed cooperative computing.

C.1.3 JavaSpaces Technology and Databases

A JavaSpaces service can store persistent data which is later searchable. But a JavaSpaces service is not a relational or object database. JavaSpaces services are designed to help solve problems in distributed computing, not to be used primarily as a data repository (although there are many data storage uses for JavaSpaces applications). Some important differences are:

- ◆ Relational databases understand the data they store and manipulate it directly via query languages. JavaSpaces services store entries that they understand

only by type and the serialized form of each field. There are no general queries in the JavaSpaces application design, only “exact match” or “don’t care” for a given field. You design your flow of objects so that this is sufficient and powerful.

- ◆ Object databases provide an object oriented image of stored data that can be modified and used, nearly as if it were transient memory. JavaSpaces systems do not provide a nearly transparent persistent/transient layer, and work only on copies of entries.

These differences exist because JavaSpaces services are designed for a different purpose than either relational or object databases. A JavaSpaces service can be used for simple persistent storage, such as storing a user’s preferences that can be looked up by the user’s ID or name. JavaSpaces service functionality is somewhere between that of a filesystem and a database, but it is neither.

C.1.4 JavaSpaces System Design and Linda² Systems

The JavaSpaces system design is strongly influenced by Linda systems, which support a similar model of entry-based shared concurrent processing. In Section C.4.1 you will find several references that describe Linda-style systems.

No knowledge of Linda systems is required to understand this specification. This section discusses the relationship of JavaSpaces systems with respect to Linda systems for the benefit of those already familiar with Linda programming. Other readers should feel free to skip ahead.

JavaSpaces systems are similar to Linda systems in that they store collections of information for future computation and are driven by value-based lookup. They differ in some important ways:

- ◆ Linda systems have not used rich typing. JavaSpaces systems take a deep concern with typing from the Java platform type-safe environment. In JavaSpaces systems, entries themselves, not just their fields, are typed—two different entries with the same field types but with different data types for the Java programming language are different entries. For example, an entry that had a string and two double values could be either a named point or a named vector. In JavaSpaces systems these two entry types would have specific different

² “Linda” is the name of a public domain technology originally propounded by Dr. David Gelernter of Yale University. “Linda” is also claimed as a trademark for certain goods by Scientific Computing Associates, Inc. This discussion refers to the public domain “Linda” technology.

classes for the Java platform, and templates for one type would never match the other, even if the values were compatible.

- ◆ Entries are typed as objects in the Java programming language, so they may have methods associated with them. This provides a way of associating behavior with entries.
- ◆ As another result of typed entries, JavaSpaces services allow matching of subtypes—a template match can return a type that is a subtype of the template type. This means that the read or take may return more states than anticipated. In combination with the previous point, this means that entry behavior can be polymorphic in the usual object-oriented style that the Java platform provides.
- ◆ The fields of entries are objects in the Java programming language. Any object data type for the Java programming language can be used as a template for matching entry lookups as long as it has certain properties. This means that computing systems constructed using the JavaSpaces API are object-oriented from top to bottom, and behavior-based (agent-like) applications can use JavaSpaces services for co-ordination.
- ◆ Most environments will have more than one JavaSpaces service. Most Linda tuple spaces have one tuple space for all cooperating threads. So transactions in the JavaSpaces system can span multiple spaces (and even non-JavaSpaces system transaction participants).
- ◆ Entries written into a JavaSpaces service are leased. This helps keep the space free of debris left behind due to system crashes and network failures.
- ◆ The JavaSpaces API does not provide an equivalent of “eval” because it would require the service to execute arbitrary computation on behalf of the client. Such a general compute service has its own large number of requirements (such as security and fairness).

On the nomenclature side, the JavaSpaces technology API uses a more accessible set of terms than the traditional Linda terms. The term mappings are “entry” for “tuple”, “value” for “actual”, “wildcard” for “formal”, “write” for “out”, and “take” for “in”. So the Linda sentence “When you ‘out’ a tuple make sure that actuals and formals in ‘in’ and ‘read’ can do appropriate matching” would be translated to “When you write an entry make sure that values and wildcards in ‘take’ and ‘read’ can do appropriate matching.”

C.1.5 Goals and Requirements

The goals for the design of JavaSpaces technology are:

- ◆ Provide a platform for designing distributed computing systems that simplifies the design and implementation of those systems.
- ◆ The client side should have few classes, both to keep the client-side model simple and to make downloading of the client classes quick.
- ◆ The client side should have a small footprint, because it will run on computers with limited local memory.
- ◆ A variety of implementations should be possible, including relational database storage and object-oriented database storage.
- ◆ It should be possible to create a replicated JavaSpaces service.

The requirements for JavaSpaces application clients are:

- ◆ It must be possible to write a client purely in the Java programming language.
- ◆ Clients must be oblivious to the implementation details of the service. The same entries and templates must work in the same ways no matter which implementation is used.

C.1.6 Dependencies

This document relies upon the following other specifications:

- ◆ *Java Remote Method Invocation Specification*
- ◆ *Java Object Serialization Specification*
- ◆ *Jini™ Entry Specification*
- ◆ *Jini™ Entry Utilities Specification*
- ◆ *Jini™ Distributed Event Specification*

- ◆ *Jini™ Distributed Leasing Specification*
- ◆ *Jini™ Transaction Specification*

C.2 Operations

There are four primary kinds of operations that you can invoke on a JavaSpaces service. Each operation has parameters that are entries, including some that are templates, which are a kind of entry. This chapter describes entries, templates, and the details of the operations, which are:

- ◆ **write**: Write the given entry into this JavaSpaces service.
- ◆ **read**: Read an entry from this JavaSpaces service that matches the given template.
- ◆ **take**: Read an entry from this JavaSpaces service that matches the given template, removing it from this space.
- ◆ **notify**: Notify a specified object when entries that match the given template are written into this JavaSpaces service.

As used in this document, the term “operation” refers to a single invocation of a method; for example, two different `take` operations may have different templates.

C.2.1 Entries

The types `Entry` and `UnusableEntryException` that are used in this specification are from the package `net.jini.core.entry` and are described in detail in the *Jini™ Entry Specification*. In the terminology of that specification `write` is a store operation; `read` and `take` are combination search and fetch operations; and `notify` sets up repeated search operations as entries are written to the space.

C.2.2 `net.jini.space.JavaSpace`

All operations are invoked on an object that implements the `JavaSpace` interface. For example, the following code fragment would write an entry of type `AttrEntry` into the JavaSpaces service referred to by the identifier `space`:

```
JavaSpace space = getSpace();
```

```

AttrEntry e = new AttrEntry();
e.name = "Duke";
e.value = new GIFImage("dukeWave.gif");
space.write(e, null, 60 * 60 * 1000); // one hour
// lease is ignored -- one hour will be enough

```

The JavaSpace interface is:

```

package net.jini.space;

import java.rmi.*;
import net.jini.core.event.*;
import net.jini.core.transaction.*;
import net.jini.core.lease.*;

public interface JavaSpace {
    Lease write(Entry e, Transaction txn, long lease)
        throws RemoteException, TransactionException;
    public final long NO_WAIT = 0; // don't wait at all
    Entry read(Entry tmp1, Transaction txn, long timeout)
        throws TransactionException, UnusableEntryException,
               RemoteException, InterruptedException;
    Entry readIfExists(Entry tmp1, Transaction txn, long timeout)
        throws TransactionException, UnusableEntryException,
               RemoteException, InterruptedException;
    Entry take(Entry tmp1, Transaction txn, long timeout)
        throws TransactionException, UnusableEntryException,
               RemoteException, InterruptedException;
    Entry takeIfExists(Entry tmp1, Transaction txn, long timeout)
        throws TransactionException, UnusableEntryException,
               RemoteException, InterruptedException;
    EventRegistration notify(Entry tmp1, Transaction txn,
                           RemoteEventListener listener, long lease,
                           MarshalledObject handback)
        throws RemoteException, TransactionException;
    Entry snapshot(Entry e) throws RemoteException;
}

```

The Transaction and TransactionException types in the above signatures are imported from net.jini.core.transaction. The Lease type is imported from net.jini.core.lease. The RemoteEventListener and EventRegistration types are imported from net.jini.core.event.

In all methods that have the parameter, `txn` may be `null`, which means that no `Transaction` object is managing the operation (see Section C.3).

The JavaSpace interface is not a remote interface. Each implementation of a JavaSpaces service exports proxy objects that implement the JavaSpace interface locally on the client, talking to the actual JavaSpaces service through an implementation-specific interface. An implementation of any JavaSpace method may communicate with a remote JavaSpaces service to accomplish its goal; hence, each method throws `RemoteException` to allow for possible failures. Unless noted otherwise in this specification, when you invoke JavaSpace methods you should expect `RemoteExceptions` on method calls in the same cases in which you would expect them for methods invoked directly on an RMI remote reference. For example, invoking `snapshot` might require talking to the remote JavaSpaces server, and so might get a `RemoteException` if the server crashes during the operation.

The details of each JavaSpace method are given in the sections that follow.

InternalSpaceException

The exception `InternalSpaceException` may be thrown by a JavaSpaces service that encounters an inconsistency in its own internal state or is unable to process a request because of internal limitations (such as storage space being exhausted). This exception is a subclass of `RuntimeException`. The exception has two constructors: one that takes a `String` description and another that takes a `String` and a nested exception; both constructors simply invoke the `RuntimeException` constructor that takes a `String` argument.

```
package net.jini.space;

public class InternalSpaceException extends RuntimeException {
    public final Throwable nestedException;
    public InternalSpaceException(String msg) {...}
    public InternalSpaceException(String msg, Throwable e) {...}
    public printStackTrace() {...}
    public printStackTrace(PrintStream out) {...}
    public printStackTrace(PrintWriter out) {...}
}
```

The `nestedException` field is the one passed to the second constructor, or `null` if the first constructor was used. The overridden `printStackTrace` methods print out the stack trace of the exception and, if `nestedException` is not `null`, print out that stack trace as well.

C.2.3 write

A `write` places a copy of an entry into the given JavaSpaces service. The `Entry` passed to the `write` is not affected by the operation. Each `write` operation places a new entry into the specified space, even if the same `Entry` object is used in more than one `write`.

Each `write` invocation returns a `Lease` object that is `lease` milliseconds long. If the requested time is longer than the space is willing to grant, you will get a lease with a reduced time. When the lease expires, the entry is removed from the space. You will get an `IllegalArgumentException` if the lease time requested is negative.

If a `write` returns without throwing an exception, that entry is committed to the space, possibly within a transaction (see Section C.3). If a `RemoteException` is thrown, the `write` may or may not have been successful. If any other exception is thrown, the entry was not written into the space.

Writing an entry into a space might generate notifications to registered objects (see Section C.2.7).

C.2.4 readIfExists and read

The two forms of the `read` request search the JavaSpaces service for an entry that matches the template provided as an `Entry`. If a match is found, a reference to a copy of the matching entry is returned. If no match is found, `null` is returned. Passing a `null` reference for the template will match any entry.

Any matching entry can be returned. Successive read requests with the same template in the same JavaSpaces service may or may not return equivalent objects, even if no intervening modifications have been made to the space. Each invocation of `read` may return a new object even if the same entry is matched in the JavaSpaces service.

A `readIfExists` request will return a matching entry, or `null` if there is currently no matching entry in the space. If the only possible matches for the template have conflicting locks from one or more other transactions, the `timeout` value specifies how long the client is willing to wait for interfering transactions to settle before returning a value. If at the end of that time no value can be returned that would not interfere with transactional state, `null` is returned. Note that, due to the remote nature of JavaSpaces services, `read` and `readIfExists` may throw a `RemoteException` if the network or server fails prior to the timeout expiration.

A `read` request acts like a `readIfExists` except that it will wait until a matching entry is found or until transactions settle, whichever is longer, up to the `timeout` period.

In both read methods, a timeout of NO_WAIT means to return immediately, with no waiting, which is equivalent to using a zero timeout.

C.2.5 `takeIfExists` and `take`

The `take` requests perform exactly like the corresponding `read` requests (see Section C.2.4), except that the matching entry is removed from the space. Two `take` operations will never return copies of the same entry, although if two equivalent entries were in the JavaSpaces service the two `take` operations could return equivalent entries.

If a `take` returns a non-null value, the entry has been removed from the space, possibly within a transaction (see Section C.3). This modifies the claims to once-only retrieval: A `take` is considered to be successful only if all enclosing transactions commit successfully. If a `RemoteException` is thrown, the `take` may or may not have been successful. If an `UnusableEntryException` is thrown, the `take` removed the unusable entry from the space; the contents of the exception are as described in the *Jini Entry Specification*. If any other exception is thrown, the `take` did not occur, and no entry was removed from the space.

With a `RemoteException`, an entry can be removed from a space and yet never returned to the client that performed the `take`, thus losing the entry in between. In circumstances in which this is unacceptable, the `take` can be wrapped inside a transaction that is committed by the client when it has the requested entry in hand.

C.2.6 `snapshot`

The process of serializing an entry for transmission to a JavaSpaces service will be identical if the same entry is used twice. This is most likely to be an issue with templates that are used repeated to search for entries with `read` or `take`. The client-side implementations of `read` and `take` cannot reasonably avoid this duplicated effort, since they have no efficient way of checking whether the same template is being used without intervening modification.

The `snapshot` method gives the JavaSpaces service implementor a way to reduce the impact of repeated use of the same entry. Invoking `snapshot` with an `Entry` will return another `Entry` object that contains a *snapshot* of the original entry. Using the returned `snapshot` entry is equivalent to using the unmodified original entry in all operations on the same JavaSpaces service. Modifications to the original entry will not affect the `snapshot`. You can `snapshot` a null template; `snapshot` may or may not return null given a null template.

The entry returned from `snapshot` will be guaranteed equivalent to the original unmodified object only when used with the space. Using the `snapshot` with any

other JavaSpaces service will generate an `IllegalArgumentException` unless the other space can use it because of knowledge about the JavaSpaces service that generated the snapshot. The snapshot will be a different object from the original, may or may not have the same hash code, and `equals` may or may not return `true` when invoked with the original object, even if the original object is unmodified.

A snapshot is guaranteed to work only within the virtual machine in which it was generated. If a snapshot is passed to another virtual machine (for example, in a parameter of an RMI call), using it—even with the same JavaSpaces service—may generate an `IllegalArgumentException`.

We expect that an implementation of JavaSpaces technology will return a specialized `Entry` object that represents a pre-serialized version of the object, either in the object itself or as an identifier for the entry that has been cached on the server. Although the client may cache the snapshot on the server, it must guarantee that the snapshot returned to the client code is always valid. The implementation may not throw any exception that indicates that the snapshot has become invalid because it has been evicted from a cache. An implementation that uses a server-side cache must therefore guarantee that the snapshot is valid as long as it is reachable (not garbage) in the client, such as by storing enough information in the client to be able to re-insert the snapshot into the server-side cache.

No other method returns a snapshot. Specifically, the return values of the `read` and `take` methods are not snapshots and are usable with any implementation of JavaSpaces technology.

C.2.7 `notify`

A `notify` request registers interest in future incoming entries to the JavaSpaces service that match the specified template. Matching is done as it is for `read`. The `notify` method is a particular registration method under the *Jini™ Distributed Event Specification*. When matching entries arrive, the specified `RemoteEventListener` will eventually be notified. When you invoke `notify` you provide an upper bound on the lease time, which is how long you want the registration to be remembered by the JavaSpaces service. The service decides the actual time for the lease. You will get an `IllegalArgumentException` if the lease time requested is not `Lease.ANY` and is negative. The lease time is expressed in the standard millisecond units, although actual lease times will usually be of much larger granularity. A lease time of `Lease.FOREVER` is a request for an indefinite lease; if the service chooses not to grant an indefinite lease, it will return a bounded (non-zero) lease.

Each `notify` returns a `net.jini.core.event.EventRegistration` object. When an object is written that matches the template supplied in the `notify` invocation, the listener's `notify` method is eventually invoked, with a

RemoteEvent object whose evID is the value returned by the EventRegistration object's getEventID method, fromWhom being the JavaSpaces service, seqNo being a monotonically increasing number, and whose getRegistrationObject being that passed as the handback parameter to notify. If you get a notification with a sequence number of 103 and the EventRegID object's current sequence number is 100, there will have been three matching entries written since you invoked notify. You may or may not have received notification of the previous entries due to network failures or the space compressing multiple matching entry events into a single call.

If the transaction parameter is null, the listener will be notified when matching entries are written either under a null transaction or when a transaction commits. If an entry is written under a transaction and then taken under that same transaction before the transaction is committed, listeners registered under a null transaction will not be notified of that entry.

If the transaction parameter is not null, the listener will be notified of matching entries written under that transaction in addition to the notifications it would receive under a null transaction. A notify made with a non-null transaction is implicitly dropped when the transaction completes.

The request specified by a successful notify is as persistent as the entries of the space. They will be remembered as long as an un-taken entry would be, until the lease expires, or until any governing transaction completes, whichever is shorter.

The service will make a “best effort” attempt to deliver notifications. The service will retry at most until the notification request’s lease expires. Notifications may be delivered in any order.

See the *Jini™Distributed Event Specification* for details on the event types.

C.2.8 Operation Ordering

Operations on a space are unordered. The only view of operation order can be a thread’s view of the order of the operations it performs. A view of inter-thread order can be imposed only by cooperating threads that use an application-specific protocol to prevent two or more operations being in progress at a single time on a single JavaSpaces service. Such means are outside the purview of this specification.

For example, given two threads *T* and *U*, if *T* performs a write operation and *U* performs a read with a template that would match the written entry, the read may not find the written entry even if the write returns before the read. Only if *T* and *U* cooperate to ensure that the write returns before the read commences would the read be ensured the opportunity to find the entry written by *T* (although it still might not do so because of an intervening take from a third entity).

C.2.9 Serialized Form

Class	serialVersionUID	Serialized Fields
InternalSpaceException	-4167507833172939849L	<i>all public fields</i>

C.3 Transactions

The JavaSpaces API uses the package `net.jini.core.transaction` to provide basic atomic transactions that group multiple operations across multiple JavaSpaces services into a bundle that acts as a single atomic operation. JavaSpaces services are actors in these transactions; the client can be an actor as well, as can any remote object that implements the appropriate interfaces.

Transactions wrap together multiple operations. Either all modifications within the transactions will be applied or none will, whether the transaction spans one or more operations and/or one or more JavaSpaces services.

The transaction semantics described here conform to the default transaction semantics defined in the *Jini™ Transaction Specification*.

C.3.1 Operations Under Transactions

Any `read`, `write`, or `take` operations that have a `null` transaction act as if they were in a committed transaction that contained exactly that operation. For example, a `take` with a `null` transaction parameter performs as if a transaction was created, the `take` performed under that transaction, and then the transaction was committed. Any `notify` operations with a `null` transaction apply to `write` operations that are committed to the entire space.

Transactions affect operations in the following ways:

- ◆ **write:** An entry that is written is not visible outside its transaction until the transaction successfully commits. If the entry is taken within the transaction, the entry will never be visible outside the transaction and will not be added to the space when the transaction commits. Specifically, the entry will not generate notifications to listeners that are not registered under the writing transaction. Entries written under a transaction that aborts are discarded.
- ◆ **read:** A `read` may match any entry written under that transaction or in the entire space. A JavaSpaces service is not required to prefer matching entries writ-

ten inside the transaction to those in the entire space. When read, an entry is added to the set of entries read by the provided transaction. Such an entry may be read in any other transaction to which the entry is visible, but cannot be taken in another transaction.

- ◆ **take:** A `take` matches like a `read` with the same template. When taken, an entry is added to the set of entries taken by the provided transaction. Such an entry may not be read or taken by any other transaction.
- ◆ **notify:** A `notify` performed under a `null` transaction applies to `write` operations that are committed to the entire space. A `notify` performed under a non-`null` transaction additionally provides notification of writes performed within that transaction. When a transaction completes, any registrations under that transaction are implicitly dropped. When a transaction commits, any entries that were written under the transaction (and not taken) will cause appropriate notifications for registrations that were made under a `null` transaction.

If a transaction aborts while an operation is in progress under that transaction, the operation will terminate with a `TransactionException`. Any statement made in this chapter about `read` or `take` apply equally to `readIfExists` or `takeIfExists`, respectively.

C.3.2 Transactions and ACID Properties

The ACID properties traditionally offered by database transactions are preserved in transactions on JavaSpaces systems. The ACID properties are:

- ◆ **Atomicity:** All the operations grouped under a transaction occur or none of them do.
- ◆ **Consistency:** The completion of a transaction must leave the system in a consistent state. Consistency includes issues known only to humans, such as that an employee should always have a manager. The enforcement of consistency is outside of the transaction—a transaction is a tool to allow consistency guarantees, and not itself a guarantor of consistency.
- ◆ **Isolation:** Ongoing transactions should not affect each other. Any observer should be able to see other transactions executing in some sequential order (although different observers may see different orders).
- ◆ **Durability:** The results of a transaction should be as persistent as the entity on which the transaction commits.

The timeout values in `read` and `take` allow a client to trade full isolation for liveness. For example, if a `read` request has only one matching entry and that entry is currently locked in a `take` from another transaction, `read` would block indefinitely if the client wanted to preserve isolation. Since completing the transaction could take an indefinite amount of time, a client may choose instead to put an upper bound on how long it is willing to wait for such isolation guarantees, and instead proceed to either abort its own transaction or ask the user whether to continue or whatever else is appropriate for the client.

Persistence is not a required property of JavaSpaces technology implementations. A transient implementation that does not preserve its contents between system crashes is a proper implementation of the JavaSpace interface's contract, and may be quite useful. If you choose to perform operations on such a space, your transactions will guarantee as much durability as the JavaSpaces service allows for all its data, which is all that any transaction system can guarantee.

C.4 Further Reading

C.4.1 Linda Systems

1. *How to Write Parallel Programs: A Guide to the Perplexed*, Nicholas Carriero, David Gelernter, *ACM Computing Surveys*, Sept., 1989.
2. *Generative Communication in Linda*, David Gelernter, *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 1, pp. 80–112 (January 1985).
3. *Persistent Linda: Linda + Transactions + Query Processing*, Brian G. Anderson, Dennis Shasha, *Proceedings of the 13th Symposium on Fault-Tolerant Distributed Systems*, 1994.
4. *Adding Fault-tolerant Transaction Processing to LINDA*, Scott R. Cannon, David Dunn, *Software—Practice and Experience*, Vol. 24(5), pp. 449–446 (May 1994).
5. *ActorSpaces: An Open Distributed Programming Paradigm*, Gul Agha, Christian J. Callsen, University of Illinois at Urbana-Champaign, UILU-ENG-92-1846.

C.4.2 The Java Platform

6. *The Java Programming Language, Second Edition*, Ken Arnold, James Gosling, Addison Wesley, 1998.
7. *The Java Language Specification*, James Gosling, Bill Joy, Guy Steele, Addison Wesley, 1996.
8. *The Java Virtual Machine Specification, Second Edition*, Tim Lindholm, Frank Yellin, Addison Wesley, 1999.
9. *The Java Class Libraries, Second Edition*, Patrick Chan, Rosanna Lee, Doug Kramer, Addison Wesley, 1998.

C.4.3 Distributed Computing

10. *Distributed Systems*, Sape Mullender, Addison Wesley, 1993.
11. *Distributed Systems: Concepts and Design*, George Coulouris, Jean Dollimore, Tim Kindberg, Addison Wesley, 1998.
12. *Distributed Algorithms*, Nancy A. Lynch, Morgan Kaufmann, 1997.

Index

A

`AbstractEntry`, 313–14
Accepted bids, 177, 180
`Account`
 class, 250–52
 constructor, 252
`Account` object, 250–54
`AccountEntry` class, 249–50
`ACID` (atomicity, consistency, isolation and durability), 243–44
 properties and transactions, 331–32
`actionPerformed` method, 185–86, 190, 271–72, 277
Adding
 elements to arrays, 71
 to friends list, 274–75
 leases, 203–4, 208–9
`addLease`, 208–9
`addPoison` method, 299–300
Algorithms
 distributed, 317
 one-way encryption, 287
Anonymous communication, 114
API
 `JavaSpaces`, 51
 distributed events, 222–26
entry, 22–23
lease, 199
lease map, 204
notification, 222–26
transaction 239–40
Applets
 Buyer, 182–86

`ChannelCreator`, 141, 143, 148
`ChannelReader`, 128–30
`ChannelWriter`, 122, 128–30, 136
`CreateChannels`, 191
`CryptMaster`, 301
`Messenger`, 270–78
`PageTaker`, 141
`PageWriter`, 136
`Seller`, 186–91
Application basics, 21–50
 building applications, 25–28
 `SpaceAccessor` class, 26–28
entries, 22–25, 31–38
 adding fields and methods, 23–25
 associative lookup, 31–32
 basics of, 32–33
 dealing with `null` valued fields in entries, 35–36
 instantiating entries, 23
interface, 22–23
looking at read and take, 36–38
primitive fields, 36
rules of matching, 33–35
 See also Serialization
example, 38–44
subclassing, 39–40
 writing entries into spaces, 28–31
Application, collaborative; *See also* Messenger, 247–79
Application, crypt, 287–301
Application, parallel, 281–302
 compute server, 281–87
 crypt application, 287–301
 See also replicated-worker

- Application patterns, 153–95
 - command pattern, 166–76
 - marketplace pattern, 176–92
 - miscellaneous patterns, 192–94
 - collaborative patterns, 193–94
 - specialist patterns, 192–93
 - replicated-worker patterns, 153–65
- Arrays revisited, distributed, 67–73
- Associative lookup, 7, 24
- Atomic modification, 57–58
- Atomicity, 243–44
- AutomatedLeaseManager, 207–08
- Automobile marketplace, 176–77
- B**
- Bags, 64
 - defined, 63
 - result, 65–66, 157–58
 - task, 65–66, 157–58
- Ball class, 109
- Barrier
 - defined, 95
 - synchronization, 95–97
 - poison pill, 299–300
- BattleSpaceShip, 39, 40, 42–43
- BidEntry, 186
 - class, 180–81
 - Bids, 177, 180
 - accepted, 177, 180
 - entries, 180–81
- Blackboard pattern, 192–93
- Bounded channels, 141–48
 - Channel creation revisited, 143–44
 - demonstrating, 148
 - status entry, 142–43
 - taking from, 146–48
 - writing to, 144–46
- Building blocks, 51–74
 - introduction to distributed data structures, 52–55
 - ordered structures, 66–73
 - shared variables, 56–63
 - unordered structures, 63–66
- Buyer
 - applet, 182–86
 - class, 182–85
- C**
- calculateMandelbrot, 165
- Cancelling leases, 209–10
- catchBall, 112
- Channel
 - Appending messages to, 119–21
 - basic, 116–27
 - channel message entries, 117–18
 - channel tail, 118
 - Creating channels, 118–19
 - Demonstrating channel writer and channel reader, 126–27
 - Implementing channel reader, 123–26
 - Implementing channel writer, 121–23
 - Bounded, 141–48
 - Communication, 263–70
 - Consumer, 130–41
 - Creating, 118–19
 - Creation revisited, 143–44
 - Defined, 116
 - Message entries, 117–18
 - Object, 265–67
 - Reader, 123–26
 - Relay application, 227–28
 - Retrieving messages from, 267–70
 - Tail, 118
 - Tracking start and end of, 133–34
 - Writer, 121–23
 - Channel class, 265–66
 - Channel reading, 128–30
 - And removing messages from, 138–41
 - Channel relay application, 226–34
 - Channel reader thread, 229
 - Channel relay application, 227–28
 - Notify handler thread, 231–34
 - Relayer listener, 229–31
 - Channel writer, 126–27
 - Implementing, 121–23
 - Channel writing, 128–30
 - ChannelCreator, 131
 - Applet, 141, 143, 148
 - ChannelMessageEntry, 263–64, 267–68
 - ChannelReader class, 124–25
 - ChannelRelay application, 227–30
 - channelTextField, 123, 126
 - ChannelWriter applets, 122, 128–30, 136
 - Chat application, building, 127–30
 - checkLeases method, 210–12
 - ChessPartner class, 64
 - Classes
 - Account, 250–52

- Classes (Continued)*
- AccountEntry, 249–50
 - AutomatedLeaseManager, 207
 - Ball, 109
 - BattleSpaceShip, 39
 - BidEntry, 180–81
 - Buyer, 182–85
 - Channel, 265–66
 - ChannelCreator, 135–36
 - ChannelMessageEntry, 263–64
 - ChannelReader, 124–25
 - ChessPartner, 64
 - Counter, 99–100
 - CryptMaster, 292–94
 - CryptTask, 289
 - Customer, 90
 - DistribArray, 69, 72–73
 - DistributedSemaphore, 77–78
 - Element, 67
 - EventRegistration, 224–26
 - FactorialResult, 172
 - FactorialTask, 171
 - FibFactMaster, 173–74
 - FibonacciTask, 172–73
 - FriendsList, 258–60
 - FriendslistEntry, 257–58
 - FriendslistMonitor, 261–62
 - GenericMaster, 169–70, 285–86
 - GenericWorker, 167–68
 - HelloWorld, 12
 - HelloWorldClient, 13
 - HelloWorldNotify, 220
 - Index, 134–35, 264
 - LeaseRenewalEvent, 214
 - LicenseInstall, 79–80
 - LicenseManager, 80
 - Listener, 221–22, 278
 - Master, 154, 159–61
 - Message, 11
 - MessageDispatcher, 276
 - Messenger, 270–71
 - MouseHandler, 275–76
 - PageTaker, 138–39, 146–48
 - PageWriter, 137, 145–46
 - Philosopher, 83–85
 - PingPong, 110–11
 - PoisonPill, 300
 - ReaderWriter, 101–3
 - Relayer, 229–30
 - RemoteEvent, 226
 - RenewThread, 210
 - Result, 159
 - RoundRobinParticipant, 93
 - RoundRobinSharedVar, 92
 - Scores, 53
 - Seller, 186–90
 - SemaphoreEntry, 77
 - Session, 255–56
 - SessionEntry, 254
 - SharedVar, 56
 - SpaceAccessor, 9, 26–28, 112
 - SpaceGame, 25
 - SpaceShip, 22–25, 39
 - Start, 68
 - Status, 142
 - Tail, 118
 - Task, 158
 - TaskEntry, 167, 282
 - TestEntry, 45
 - TransactionManagerAccessor, 239
 - UnusableEntryException, 309–10
 - WebCounterClient, 60, 241–42
 - Worker, 66, 95–96, 154, 164–65
 - Collaborative patterns, 193–94
 - collectResults, 162–64, 174, 297–99
 - Command interface revisited, 166, 281–82
 - Command patterns, 166–76
 - creating specialized master, 173–75
 - creating specialized tasks and results, 170–73
 - generic master, 169–70
 - generic worker, 167–69
 - implementing compute servers, 167
 - running compute server, 175–76
 - Communication, 107–51
 - anonymous, 114
 - basic channel, 116–27
 - basic message passing, 108–13
 - benefits of loose coupling, 115–16
 - beyond message passing, 116
 - bounded channels, 141–48
 - building chat applications with channels, 127–30
 - combining channel writing and channel reading, 128–30
 - graphical user interface (GUI), 127–28

- characteristics of space-based, 113–16
- loosely coupled, 114–15
- space-based, 107
- tightly coupled, 113–14
- Communication channel, 263–70
 - message entries and indices, 263–65
- Computation/communication ratio, 157
- Computation, Mandelbrot, 157, 159, 162
- Compute servers, 281–87
 - command interface revisited, 281–82
 - generating tasks, 294–96
 - generic master, 285–87
 - generic worker, 283–85
 - implementing, 167
 - running, 175–76
 - task entry, 282–83
- `com.sun.jini.use.registry`, 27
- Consistency, 243–44
- Constructor, no-arg, 47–48
- Consumer channel, 130–41
 - creating channels, 135–36
 - demonstrating pager, 141
 - implementing pager service, 131–32
- Index entry, 134–35
- PagerMessage entry, 133
- reading and removing messages from channel, 138–41
- sending messages to channel, 136–38
- tracking start and end of channel, 133–34
- Counter, 12, 99–101
- Counter class, 99–100
- Counter revisited, web, 240–43
- create method, 78
- `createChannel`, 119–20, 130, 136
- `createLeaseMap`, 202
- Creating
 - arrays, 69–70
 - channels, 118–19
 - lease map, 202
 - specialized master, 173–75
 - specialized tasks and results, 170–73
 - transactions, 239–40
 - web page counter, 59–62
- Crypt applications, 287–301
 - background, 287–88
 - collecting results, 297–99
- `CryptMaster`, 291–94
- exploring crypt, 301
- generic worker and `CryptTask`, 288–91
- improving space usage with watermarking, 296–97
- use of Poison Pill, 299–300
- `CryptMaster`, 291–94, 299
 - applet, 301
- `CryptResult`
 - entry, 298
- `CryptTask`, 295
 - entry, 300
 - and generic worker, 288–91
- Customer
 - class, 90

D

- Databases, JavaSpaces technology and, 319–320
- Deadlock defined, 82
- `decreaseScore`, 24
- Dining philosophers, space-based implementation of, 83–85
- `DiscoveryLocator`, 28
- `displayThread`, 185–86
- `displayNextMessage`, 146–48
- `displayNextMessage` method, 140
- `DistribArray` class, 69, 72–73
- Distributed algorithms as flows of objects, 317
- Distributed arrays
 - adding elements to, 71
 - creating, 69–70
 - reading elements of, 72
- revisited, 67–73
- Distributed computing
 - benefits of, 2–3
 - availability, 3
 - elegance, 3
 - fault tolerance, 3
 - performance, 2
 - resource sharing, 2
 - scalability, 2
 - challenges of, 3–4
 - latency, 4
 - partial failure, 4
 - synchronization, 4
- Distributed data structures, 5, 52
- Distributed data structures
 - building with entries, 53–55
 - and collections of objects, 53

- Distributed data structures (*Continued*)
 - creating simple, 55
 - distributed, 5
 - introduction to, 52–55
 - pieces needed for any, 55
 - and protocol, 53
 - representation of, 53
- Distributed environment, events in, 217–20
- Distributed events, 217–35
 - events in distributed environment, 217–20
 - hello world using notify, 220–22
 - notification API, 222–26
 - Distributed persistence, 314
- Distributed processes, synchronizing, 75
- Distributed protocols, 5–6
- Distributed transaction model, 238–39
- DistributedSemaphore**
 - class, 77–78
 - object, 81, 86
- Durability, 243–44
- E**
- Encryption algorithm, one-way, 287
- Entries; *See also listings under entry*
 - building distributed data structures with, 53–55
 - defined, 308
 - null valued fields in, 35–36
 - and operations, 9–11
 - reading, 245
 - reading and taking, 31–38
 - subclassing, 39–40
 - task, 162, 282–83
 - writing, 244
- entry, 23
- Entry
 - defined, 308
 - interface, 48
 - serialization, 45–47, 308–9
- Event; *See also Events*
 - listeners, 217–18
 - objects, 217–18
 - sources, 217
- EventRegistration
 - class, 224–26
- EventRegistration.notify, 223
- Events
 - distributed, 217–35
 - in distributed environment, 217–20
- ExceptionMap object, 212–14
- Exceptions, 30, 37–38
- IllegalArgumentException, 203, 328
- InternalSpaceException, 325
- InterruptedException, 36–38
- LeaseDeniedException, 201
- LeaseMapException, 213
 - RemoteException, 30, 36–38, 48, 197–198, 201, 207, 209, 223, 225, 310, 324, 326–327
 - UnknownEventException, 225
 - UnknownLeaseException, 201, 207, 209–10
 - TransactionException, 30, 36–38, 197, 223, 324
- UnusableEntryException, 36–38, 44, 201, 309–10, 324, 327
- F**
- FactorialResult class, 172
- FactorialTask, 172
- Failure, partial, 4
- FibFactMaster class, 173–74
- FibonacciTask class, 172–73
 - Fields, primitive, 36
- Friends, chatting with, 275–77
- Friends list, 257–63
- Friends list, adding to, 274–75
- FriendsList
 - class, 258–60
- FriendsListEntry
 - class, 257–58
 - template, 260–61
- FriendsListMonitor, 260
 - class, 261–62
- G**
- generateTasks, 161–64, 174, 287, 294–95
- Generic master, 169–70, 285–87
- Generic worker, 167–69, 283–85
 - and CryptTask, 288–91
- GenericMaster class, 169–70, 285–86, 294
- GenericWorker, 288
 - class, 167–68
- getAmmo, 40
- getDocumentBase, 61
- getExpiration method, 200
- getLicense method, 80–81
- getMessageNumber, 119, 121, 130, 138
- getMessages method, 268–69

`getScore`, 32–33, 40
`getSpace`, 9, 27, 61, 112, 123
`getTransaction` method, 284–85

H

Handback defined, 224
Hello world using notify, 220–22
`HelloWorld`
 application, 11
`HelloWorldClient`, 14–15
`HelloWorldNotify` class, 220
Historical resources, 305–6

I

`incrementServiceNumber`, 90–91
`Index`, 68
 class, 134–35, 264
 entries, 134–35, 138
Indices, message entries and, 263–65
Interfaces
 `ChannelCreator`, 136
 command, 166, 281–82
 `RemoteEventListener`, 225
 `Serializable`, 48
`InternalSpaceException`, 325
Isolation, 243–44

J

Java
 and Jini technologies, 304–5
 programming language, 1
`java.rmi.MarshalledObject` object, 309
JavaSpace `notify` method, 223–24
JavaSpaces
 API, 51
 specification, 315–333
 application model and terms, 315–17
 benefits, 318–19
 distributed persistence, 316
 entries, 323
 goals and requirements, 322
 introduction, 315–317
 Jini Entry Utilities Specification, 313–14
 `net.jini.space.JavaSpace`, 323–25
 `notify`, 328–29
 operation ordering, 329
 operations, 323–30
 operations under transactions, 330–31
 read, 323, 326–27

`readIfExists`, 326–27
references and further reading, 332–33
serialized form, 330
`snapshot`, 327–28
`take`, 323, 327
`takeIfExists`, 327
transactions, 330–32
transactions and ACID properties, 331–32
`write`, 323, 326

system design and Linda Systems, 320–21

technologies
 advantages of, 16
 chapter preview, 17
 in context, 8
 defined, 4–8
 key features, 6–7
 on-line supplement, 18
 overview, 9–14, 9–14

`JavaSpaces.NO_WAIT`, 37

Jini
 Distributed Event model, 219
Jini Entry Specification, 307–12
 entry, 308
 operations, 307–9
 serialized form, 312
 serializing entry objects, 308–9
 templates and matching, 311
`UnusableEntryException`, 309–10
Jini Entry Utilities Specification, 313–14
 `AbstractEntry`, 313–14
 serialized form, 314
 `jobId` field, 158, 162
 `JSGame`, 79
 JVMs (Java virtual machines), 27, 218–19, 223,
 315

L

Latency, 4
Lease manager interface, 206–7
Lease renewal, automated, 206–15
 adding leases, 208–9
 cancelling leases, 209–10
 checking leases, 211–13
 implementing constructors, 207–8
 lease manager interface, 206–7
 processing renewal failures, 213–14
 putting it all together, 214–15
 renewing leases, 210
Lease time arguments, 30

- leaseDuration, 203
- Lease.FOREVER, 30, 198
- LeaseManager object, 257
- LeaseMap
 - cancelling leases in, 205–6
 - renewing leases in, 204–5
- LeaseMapException, 205, 213
- LeaseRenewalEvent, 213
- Leases, 197–215
 - adding, 203–4, 208–9
 - cancelling, 201–2, 209–10
 - checking, 211–13
 - on entries, 197–98
 - expiration of, 200
 - maps, 202–6
 - objects, 199–202
 - removing, 203–4
 - renewing, 200–201, 210
- Library, license manager client, 80–81
- License manager
 - client library, 80–81
 - implementing, 79
 - installer, 79–80
- LicenseInstall class, 79–80
- LicenseManager, 81
- Linda Systems, JavaSpaces Design and, 320–21
- Listener class, 221–22, 278
- Listeners
 - event, 217–18
 - relayer, 229–31
- Lists, friends, 257–63, 274–75
- Login object, logging in via, 271–73
- loginCallback method, 273–74
- Long.MAX_VALUE, 37, 41
- Lookup, associative, 7, 24
- Loosely coupled communication, 114–15
- M**
- Manager, transaction, 238, 243
- Mandelbrot
 - computation, 157, 159, 162
 - tasks, 166
- Mandelbrot set, computing, 155–57
- Marketplace
 - application framework, 181–82
 - automobile, 176–77
 - bid entry, 180–81
 - buyer, 182–86
- interaction in, 177–80
- running, 191–92
- seller, 186–91
- MarshallledObject, 224, 311
- Master-Worker/Replicated-Worker Pattern, 65, 159–64
 - creating specialized, 173–75
 - generic, 169–70, 285–87
 - process, 65
 - and workers, 154
- Master class, 154, 159–61
- Matching
 - rules of, 33–35
 - and templates, 311
- Message
 - class, 11
 - entry, 14–15
- Message entries
 - channel, 117–18
 - and indices, 263–65
- Message passing; *See also* Communication
 - basic, 108–13
- MessageDispatcher class, 276
- Messages
 - appending to channels, 119–21
 - listening for, 278
 - reading and removing messages from channel, 138–41
 - retrieving from Channel, 267–70
 - sending, 277
 - sending to channel, 136–38
- messageTextField, 123
- Messenger, 247–49
 - account, 249–54
 - Account object, 250–54
 - applet, 270–78
 - adding to friends list, 274–75
 - chatting with friends, 275–77
 - listening for messages, 278
 - Login in via login object, 271–73
 - loginCallback method, 273–74
 - sending messages, 277
 - communication channel, 263–70
 - friends list, 257–63
 - implementing messenger user, 249
 - messenger, 247–49
 - Messenger applet, 270–78
 - Session object, 255–57
 - user, 249

user sessions, 254–57
Messenger class, 270–71
MouseHandler class, 275–76
mouseReleased, 139, 164

N

nestedExceptions, 310
net.jini.core.entry, 21
net.jini.core.transaction, 330
net.jini.javaspace, 21
net.jini.outrigger, 27
net.jini.space.JavaSpace, 323–25
No-arg constructor, 47–48
Notification API, 222–26
 EventRegistration class, 224–25
 JavaSpace notify method, 223–24
 RemoteEventListener interface, 225
Notifications under transactions, 246
notify, 223–24, 328–29
Notify
 handler thread, 231–34
 hello world using, 220–22
NotifyHandler object, 231–33
null
 transaction defined, 30
 valued fields in entries, 35–36

O

Objects
 account, 250–54
 channel, 265–67
 ChannelRelay, 228
 CryptResult, 290
 Customer, 91
 DistributedSemaphore, 81, 86
 event, 217–18
 EventRegistration, 231
 FriendsList, 258–61
 FriendsListEntry, 257
 FriendsListMonitor, 261–63
 Integer, 12
 java.rmi.MarshalledObject, 309
 LeaseManager, 257
 logging in via login, 271–73
 NotifyHandler, 231–33
 Philosopher, 86
 PingPong, 112
 RemoteEvent, 225–27

serializing entry, 308–9
Transaction.Created, 240
Online resources, 303–4
Operations
 and entries, 9–11
 under transactions, 244–246, 330–31
Ordered structures; *See also* Unordered structures, 66–73
distributed arrays revisited, 67–73

P

Pager
 demonstrating, 141
 service, 131–32
PagerMessage, 140–41
 entry, 133
PageTaker, 131
 applet, 141
PageWriter, 131, 144–45
 applet, 136
Parallel application, 281–302
Parallel computing, 153
Passwords, 296
Patterns; *See also* Distributed data structures
 application, 153–95
 blackboard, 192–93
 collaborative, 193–94
 command, 166–76
 master-worker, 153–65
 miscellaneous, 192–94
 replicated-worker, 153–65
 specialist, 192–93
 trellis, 193
 workflow, 193–94
Persistence, distributed, 316
Philosopher
 class, 83–86
Philosophers problem, dining, 82–87
Ping, 109–10
Ping-pong, playing, 109–13
PingPong
 class, 110–11
PoisonPill class, 300
Pong, 109–10
Primitive fields, 36
Process, master, 65
processExceptionMap method, 213
processFormFields, 186, 191

- Producer/consumer problem, 142
 Properties
 space's transactional, 243–44
 transactions and ACID, 331–32
 Protocols, distributed, 5–6
- Q**
 Queues, using to take turns, 88–92
- R**
 read, 36–37, 323, 326–327
 Read defined, 57
 Read versus ReadIfExists, 245
 readAndDisplayRequests, 191
 Reader, channel, 126–27
 Readers/writers
 problem, 97–105
 solution, 98
 space-based application, 101–5
 ReaderWriter, 104
 readIfExists, 36–37, 253, 300, 326–27
 Read versus, 245
 Reading
 background, 304–5
 distributed systems, 304–5
 transactions, 305
 channel, 128–30
 elements of arrays, 72
 entries under transactions, 245
 references and further, 332–33
 and taking entries, 31–38
 readOrWrite, 104
 readTail, 233, 269
 References and further reading, 303–306,
 332–33
 Relayer class, 229–30
 Relayer listener, 229–31
 relayMessages, 233
 RemoteEvent object, 225–27
 RemoteEventListener interface, 225
 RemoteException, 30–31
 removeMessage, 140
 Removing leases, 203–4
 renew method, 200–201
 Renewing leases, 210
 in LeaseMap, 204–5
 RenewThread class, 210
 Replicated-worker; *See also* Master-worker
 computing, 65
 patterns, 153–65
 computing Mandelbrot set, 155–57
 master, 159–64
 task and result entries, 157–59
 worker, 164–65
 Request bids, 178
 Requests for bids, 177, 180
 Resources, 303–6
 background reading, 304–5
 fairly sharing, 87–94
 round-robin synchronization, 92–94
 using queues to take turns, 88–92
 historical, 305–6
 online, 303–4
 related Java and Jini technologies, 304–5
 sharing, 2
 Result bags, 65–66, 157–58
 Result class, 159
 ResultEntry, 168–70, 174
 Results, creating specialized tasks and, 170–73
 retrieveShip, 40–41
 returnLicense method, 80
 Round-robin synchronization, 92–94
 RoundRobinParticipant class, 93
 RoundRobinSharedVar class, 92
- S**
 Scores entry, 54–55
 Seller applet, 186–91
 SemaphoreEntry, 78–79
 class, 77
 Semaphores, 76–81
 create method, 78
 defined, 76
 down method, 78
 entries, 77
 implementing, 77–79
 pools of available resources, 76–77
 SemaphoreEntry, 78
 up method, 79
 using multiple, 81–87
 dining philosophers problem, 82–87
 sendButton, 123
 Serializable interface, 48
 Serialization, 44–49
 and effects, 44–49
 entry, 45–48
 entry serialization, 45–47

- improving entry serialization using snapshot, 48–49
- matching and equality, 47
- no-arg constructor, 47–48
- `serialVersionUID`, 311, 314, 330
- `Session` class, 255–56
- `SessionEntry` class, 254
- Shared variables
 - atomic modification, 57–58
 - creating web page counter, 59–62
 - and decrement, 58–59
 - defined, 56
 - distributed data structures, 56–63
 - and increment, 58–59
 - miscellaneous operations, 57–59
 - `SharedVar`, 92, 95–96
- `snapshot`, 284, 327–28
- `Snapshot`, improving entry serialization using, 48–49
- Sources, event, 217
- Space-based communication, 107
 - characteristics of, 113–16
- `SpaceAccessor`, 11, 61
- `SpaceAccessor` class, 9, 26–28, 112
- `SpaceGame` class, 25
- Spaces, 5
 - associative, 7
 - exchange executable content, 7
 - persistent, 7
 - shared, 6–7
 - store entries, 9
 - transactional properties, 243–44
 - transactionally secure, 7
 - writing entries into, 28–31
- `SpaceShip`, 28, 31, 40, 42
 - class, 22–25, 39
 - template, 41
- Specialist patterns, 192–93
- Specifications
 - `JavaSpaces`, 315–333
 - `Jini Entry`, 307–12
 - `Jini Entry Utilities`, 313–14
- Starvation defined, 87
- `Status`
 - class, 142
 - template, 145
- `String`
 - names, 68
- parameter, 13
- Structures, ordered, 66–73
- Subclassing entries, 39–40
- Synchronization, 4, 75–105
 - advanced, 97–105
 - barrier, 95–97
 - fairly sharing resources, 87–94
 - implementing license manager, 79
 - license manager client library, 80–81
 - license manager installer, 79–80
 - round-robin, 92–94
 - semaphores, 76–81
 - using multiple semaphores, 81–87

T

- `Tail`
 - channel, 118
 - entries, 138
- `take`, 323, 327
- `takeAndDisplayAcceptedBids`, 191
- `takeANumber`, 89, 91
- `takeHead`, 269
- `Task`
 - class, 158
 - entry, 162
 - template, 165
- Task bags, 65–66, 157–58
- Task entry, 282–83
- `TaskEntry`, 170–72
- `TaskEntry` class, 167, 282
- Tasks
 - generating, 294–96
 - and results, 170–73
- Templates, 10
 - `FriendsListEntry`, 260–61
 - and matching, 7, 10, 31, 33–36, 311
 - `MyEntry`, 49
 - `SharedVar`, 90
 - `SpaceShip`, 41
 - `Status`, 145
 - `Task`, 165
- `TestEntry` class, 45–46
- Thread, notify handler, 231–34
- Thread variable, 185
- `throwBall`, 112
- Tightly coupled communication, 113–14
- `totalTasks`, 65–66
- `Transaction`, 284

- T**
- Transaction
 - argument, 30
 - manager, 238, 243
 - model
 - distributed, 238–39
 - Jini, 238
 - Transactional properties, space's, 243–44
 - `Transaction.Created` object, 240, 253
 - `TransactionException`, 30–31
 - `TransactionManagerAccessor` class, 239, 285
 - Transactions, 237–46
 - and ACID properties, 243–244, 331–32
 - creating, 239–40
 - distributed transaction model, 238–39
 - notifications under, 246
 - operational semantics under, 244–46
 - notifications under transactions, 246
 - Read versus ReadIfExists, 245
 - reading entries under transactions, 245
 - taking entries under transactions, 245
 - writing entries under transactions, 244
 - operations under, 330–31
 - web counter revisited, 240–43
 - Trellis pattern, 193
- U**
- `UnicastRemoteObject`, 230
 - Unordered structure, 63–66
 - bags, 64
 - and put and get, 63
 - result bags, 65–66
 - task bags, 65–66
 - `UnusableEntryException` class 44, 309–10
- User**
- `messenger`, 249
 - sessions, 254–57
- V**
- Variables**
- shared, *see* Shared variables
- W**
- `waitForLowWaterMark`, 296–97
 - `waitForTurn`, 89, 91
 - `waitForRead`, 104
 - watermarking, 286
 - defined, 296
 - improving space usage with, 296–97
 - web
 - counter revisited, 240–43
 - page counter, 59–62
 - `webCounter`, 60
 - `WebCounterClient` class, 60, 62, 241–42
 - wildcards, 311
 - `Worker` class, 66, 95–96, 154, 164–65
 - Workers, 65, 164–65
 - generic, 167–69, 283–85
 - and master, 154
 - Workflow patterns, 193–94
 - `write`, 9, 28–31, 244, 323–24
 - Writer, channel, 126–27
 - `writeShip`, 40
 - Writing
 - channel, 128–30
 - entries under transactions, 244

How to Register Your Book

Register this Book

Visit: <http://www.awl.com/cseng/register>

Enter this unique code: csng-bauw-hgxn-oqqp

Then you will receive:

- Access to the book's code appendix which contains the complete source code from within the book and will appear as a zip file that you can download and compile!
- Exclusive offers on other Addison-Wesley Jini Technology Series and Java Series books

Visit our Web site

<http://www.awl.com/cseng>

When you think you've read enough, there's always more content for you at Addison-Wesley's web site. Our web site contains a directory of complete product information including:

- Chapters
- Exclusive author interviews
- Links to authors' pages
- Tables of contents
- Source code

You can also discover what tradeshows and conferences Addison-Wesley will be attending, read what others are saying about our titles, and find out where and when you can meet our authors and have them sign your book.

We encourage you to patronize the many fine retailers who stock Addison-Wesley titles. Visit our online directory to find stores near you or visit our online store:
<http://store.awl.com/> or call 800-824-7799.

Contact Us via Email

cepubprof@awl.com

Ask general questions about our books.

Sign up for our electronic mailing lists.

Submit corrections for our web site.

bexpress@awl.com

Request an Addison-Wesley catalog.

Get answers to questions regarding your order or our products.

innovations@awl.com

Request a current Innovations Newsletter.

webmaster@awl.com

Send comments about our web site.

mikeh@awl.com

Submit a book proposal.

Send errata for an Addison-Wesley book.

cepubpublicity@awl.com

Request a review copy for a member of the media interested in reviewing new Addison-Wesley titles.

Addison Wesley Longman

Computer and Engineering Publishing Group

One Jacob Way, Reading, Massachusetts 01867 USA

TEL 781-944-3700 • FAX 781-942-3076

*See website for contest rules and duration



JavaSpaces™ Principles, Patterns, and Practice



"As part of the fruits of Sun's Jini™ project, we now have the JavaSpaces™ technology, a wonderfully simple platform for developing distributed applications that takes advantage of the power of the Java™ programming language. This important book and its many examples will help you learn about distributed and parallel programming. I highly recommend it to students, programmers, and the technically curious."

—Bill Joy, Chief Scientist and co-founder, Sun Microsystems, Inc.

JavaSpaces™ technology is a powerful Jini™ service from Sun Microsystems, Inc. that facilitates building distributed applications. The JavaSpaces model provides persistent object exchange "areas" in which remote Java™ processes can coordinate their actions and exchange data. JavaSpaces technology supplies a necessary, cross-platform framework for distributed computing with Jini technology.

This book introduces the JavaSpaces technology architecture and provides a comprehensive description of the model. Using an example-driven approach, this book shows you how to use JavaSpaces technology to develop distributed computing applications.

You will find information on such vital topics as:

- Distributed data structures
- Synchronization and communication patterns
- Application patterns (replicated worker, command, and marketplace)
- Leases and automated lease renewal
- Using distributed events with spaces
- Handling partial failure with distributed transactions

JavaSpaces™ Principles, Patterns, and Practice also includes two full-scale applications—one collaborative and the other parallel—that demonstrate how to put the JavaSpaces model to work.

Eric Freeman is co-founder and CTO of Mirror Worlds Technologies, a Java and Jini-based software company. Dr. Freeman previously worked at Yale University on space-based systems, and is a Fellow at Yale's Center for Internet Studies. **Susanne Hupfer** is Director of Product Development for Mirror Worlds Technologies and a Fellow of the Yale University Center for Internet Studies. Dr. Hupfer previously taught Java network programming as an Assistant Professor of Computer Science at Trinity College. **Ken Arnold** is the lead engineer of the JavaSpaces product at Sun. He is one of the original architects of the Jini™ platform and is co-author of *The Java™ Programming Language, Second Edition*.

The Jini Technology Series



From the creators of the Jini™ technology at Sun Microsystems comes the official Series for reference material and programming guides. Written by those who design, implement, and document the technology, these books show you how to use, deploy, and create applications using the Jini architecture. The Series is a vital resource of unique insights for anyone utilizing the power of the Java™ programming language and the simplicity of Jini technology.

...from the Source



Cover design by Simone R. Payment

Cover art by Sara Connell

Interior illustrations by James P. Dustin

Text printed on recycled paper

ADDISON-WESLEY

Addison-Wesley is an imprint
of Addison Wesley Longman, Inc.



53995

9 780201 309553

ISBN 0-201-30955-6

\$39.95 US
\$59.95 CANADA