Team LiB

TIMELY. PRACTICAL. RELIABLE.

More Java™ Pitfalls

50 New Time-Saving Solutions and Workarounds

Michael C. Daconta
Kevin T. Smith
Donald Avondolio
W. Clay Richardson

More Java™ Pitfalls

50 New Time-Saving Solutions and Workarounds

More Java™ Pitfalls

50 New Time-Saving Solutions and Workarounds

Michael C. Daconta
Kevin T. Smith
Donald Avondolio
W. Clay Richardson

Wiley Publishing, Inc.

Publisher: Joe Wikert
Executive Editor: Robert M. Elliott
Assistant Developmental Editor: Emilie Herman
Managing Editor: Micheline Frederick
New Media Editor: Angela Denny
Text Design & Composition: Wiley Composition Services

This book is printed on acid-free paper. ♾

Copyright © 2003 by Michael C. Daconta, Kevin T. Smith, Donald Avondolio, and W. Clay Richardson. All rights reserved.

Published by Wiley Publishing, Inc., Indianapolis, Indiana

Published simultaneously in Canada

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8700. Requests to the Publisher for permission should be addressed to the Legal Department, Wiley Publishing, Inc., 10475 Crosspoint Blvd., Indianapolis, IN 46256, (317) 572-3447, fax (317) 572-4447, E-mail: permcoordinator@wiley.com.

Limit of Liability/Disclaimer of Warranty: While the publisher and author have used their best efforts in preparing this book, they make no representations or warranties with respect to the accuracy or completeness of the contents of this book and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. No warranty may be created or extended by sales representatives or written sales materials. The advice and strategies contained herein may not be suitable for your situation. You should consult with a professional where appropriate. Neither the publisher nor author shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

Wiley, the Wiley Publishing logo and related trade dress are trademarks or registered trademarks of Wiley Publishing, Inc., in the United States and other countries, and may not be used without written permission. Java is a trademark or registered trademark of Sun Microsystems, Inc.. All other trademarks are the property of their respective owners. Wiley Publishing, Inc., is not associated with any product or vendor mentioned in this book.

For general information on our other products and services please contact our Customer Care Department within the United States at (800) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

Library of Congress Cataloging-in-Publication Data:

ISBN: 0-471-23751-5

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

This book is dedicated to the memory of Edsger W. Dijkstra who said,

"I mean, if 10 years from now, when you are doing something quick and dirty, you suddenly visualize that I am looking over your shoulders and say to yourself, 'Dijkstra would not have liked this', well that would be enough immortality for me."

We humbly disagree: 10 years of Dijkstra is just not long enough; may he happily haunt our consciousness for 10^{10} years. Such an increase is more befitting his stature.

Contents

Introduction		**xi**
Acknowledgments		**xvii**
Part One	**The Client Tier**	**1**
Item 1:	**When Runtime.exec() Won't**	**4**
Item 2:	**NIO Performance and Pitfalls**	**17**
	Canonical File Copy	20
	Little-Endian Byte Operations	21
	Non-Blocking Server IO	26
Item 3:	**I Prefer Not to Use Properties**	**34**
Item 4:	**When Information Hiding Hides Too Much**	**39**
Item 5:	**Avoiding Granularity Pitfalls In java.util.logging**	**44**
Item 6:	**When Implementations of Standard APIs Collide**	**53**
Item 7:	**My Assertions are Not Gratuitous!**	**59**
	How to Use Assertions	59
Item 8:	**The Wrong Way to Search a DOM**	**66**
Item 9:	**The Saving-a-DOM Dilemma**	**73**
Item 10:	**Mouse Button Portability**	**80**
Item 11:	**Apache Ant and Lifecycle Management**	**88**
Item 12:	**JUnit: Unit Testing Made Simple**	**100**

Item 13: The Failure to Execute 108
Deploying Java Applications 109
The Java Extension Mechanism 110
Sealed Packages 111
Security 112

Item 14: What Do You Collect? 112

Item 15: Avoiding Singleton Pitfalls 117
When Multiple Singletons in Your VM Happen 119
When Singletons are Used as Global Variables, or Become Non-Singletons 120

Item 16: When setSize() Won't Work 122

Item 17: When Posting to a URL Won't 126
Connecting via HTTP with the java.net Classes 126
An Alternative Open Source HTTP Client 137

Item 18: Effective String Tokenizing 140

Item 19: JLayered Pane Pitfalls 146

Item 20: When File.renameTo() Won't 151

Item 21: Use Iteration over Enumeration 157

Item 22: J2ME Performance and Pitfalls 162

Part Two The Web Tier 199

Item 23: Cache, It's Money 200

Item 24: JSP Design Errors 208
Request/Response Paradigm 208
Maintaining State 209
JSP the Old Way 210
JSP Development with Beans (Model 1 Architecture) 214
JSP Development in the Model 2 Architecture 220

Item 25: When Servlet HttpSessions Collide 220

Item 26: When Applets Go Bad 227

Item 27: Transactional LDAP—Don't Make that Commitment 235

Item 28: Problems with Filters 244

Item 29: Some Direction about JSP Reuse and Content Delivery 255

Item 30: Form Validation Using Regular Expressions 261

Item 31: **Instance Variables in Servlets** **269**

Item 32: **Design Flaws with Creating Database Connections within Servlets** **279**

Item 33: **Attempting to Use Both Output Mechanisms in Servlets** **291**

Item 34: **The Mysterious File Protocol** **297**

Item 35: **Reading Files from Servlets** **302**
Web Application Deployment Descriptors 308

Item 36: **Too Many Submits** **312**
Preventing Multiple Submits 314
Handling Multiple Submits 316

Part Three The Enterprise Tier **327**

Item 37: **J2EE Architecture Considerations** **329**

Item 38: **Design Strategies for Eliminating Network Bottleneck Pitfalls** **335**
A Scenario 336
General Design Considerations 336
EJB Design Considerations 340

Item 39: **I'll Take the Local** **341**

Item 40: **Image Obsession** **348**

Item 41: **The Problem with Multiple Concurrent Result Sets** **353**

Item 42: **Generating Primary Keys for EJB** **359**
A Simple Scenario 359
A "Client Control" Approach 360
The Singleton Approach 362
The Networked Singleton Approach 363
An Application Server-Specific Approach 363
Database Autogeneration Approaches 363
Other Approaches 364

Item 43: **The Stateful Stateless Session Bean** **365**
Message-Driven Beans 366
Entity Bean 366
Stateful Session Bean 368
Stateless Session Bean 368

Item 44: **The Unprepared PreparedStatement** **372**

Item 45: Take a Dip in the Resource Pool 378

Item 46: JDO and Data Persistence 385

Item 47: Where's the WSDL? Pitfalls of Using JAXR with UDDI 398
Where's the WSDL? 404

Item 48: Performance Pitfalls in JAX-RPC Application Clients 417
Example Web Service 418
A Simple Client That Uses Precompiled Stub Classes 420
A Client That Uses Dynamic Proxies for Access 421
Two Clients Using the Dynamic Invocation Interface (DII) 423
Performance Results 427
Conclusion 428

Item 49: Get Your Beans Off My Filesystem! 429

Item 50: When Transactions Go Awry, or Consistent State in Stateful Session EJBs 433
The Memento Pattern 438

Index 443

Introduction

"Sometimes we discover unpleasant truths. Whenever we do so, we are in difficulties: suppressing them is scientifically dishonest, so we must tell them, but telling them, however, will fire back on us."

Edsger W. Dijkstra, "How do we tell truths that might hurt?"

Good programming is difficult. It is especially arduous for new programmers given the pace of change and the ever-expanding size of the software engineering body of knowledge (www.swebok.org) that they must master. The authors of this book have found that experience and in-depth understanding are key factors in both programmer productivity and reliable software. The bottom line is that experienced programmers don't stumble around in the dark. They know the lay of the land, they recognize patterns, and they avoid the hazardous areas. This book presents our experience and guidance on 50 discrete topics to assist you in avoiding some of those hazards.

What Is a Pitfall?

The formal definition, given in the first *Java Pitfalls* (Wiley, 2000) book, is as follows:

> *"A pitfall is code that compiles fine but when executed produces unintended and sometimes disastrous results."*

This rather terse definition covers what we consider the "basic" pitfall. There are many variations on this theme. A broader definition could be any language feature, API, or system that causes a programmer to waste inordinate amounts of time struggling with the development tools instead of making progress on the resulting software.

The causes of pitfalls can be loosely divided into two groups: the fault of the platform designer or the fault of the inexperienced programmer. This is not to cast blame, but rather to determine the source of the pitfall in the construction of a pitfall taxonomy. For the same reason we create a formal definition of pitfalls, we present the pitfall taxonomy in Figure i.1 in order to attempt to better understand the things that trip us up.

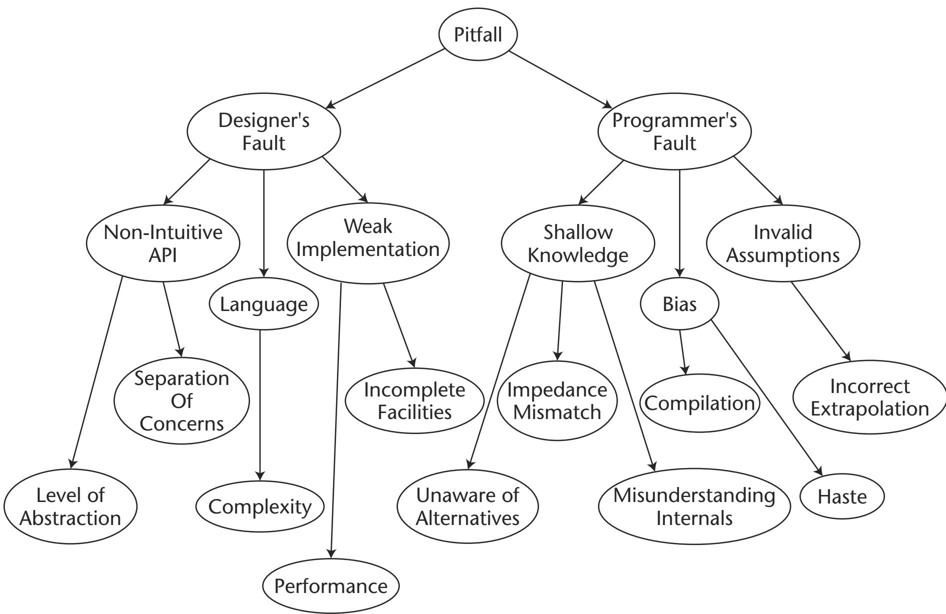


Figure i.1 A pitfall taxonomy.

The categories of pitfalls associated with the system designer are as follows:

Nonintuitive Application Programming Interfaces (APIs). The Java platform has thousands of classes and tens of thousands of methods. The sheer size of the platform has become a complexity pitfall. Some of these classes are well designed, like servlets, IO streams (excluding performance ramifications), and collections. Unfortunately, many APIs are nonintuitive for the following reasons:

- **Wrong level of abstraction.** Many APIs are layered over other software (like the operating system or native code) in order to simplify or aggregate functions. In layering, you must make a trade-off between simplicity and granularity of control. Thus, when setting the abstraction level, you must balance these appropriately for the particular context and target audience. Too high a level of abstraction (like URLConnection, Item 17) frustrates users with weak control mappings, while a too low level of abstraction reduces the average user's efficiency by over-cluttering the most common case.
- **Weak separation of concerns.** When an API tries to do too much, it often mixes metaphors and addresses its multiple concerns in a mediocre fashion. An example of this is the JAXR API that attempts to combine the diverse information models of UDDI and ebXML (Item 47).
- **Other deficiencies.** Too many method invocation sequences and dependencies will lead to incorrect ordering. Poor naming and weak parameters (object instead of a specific type) steer programmers toward dead ends.

Language Complexity. The Java language has many improvements over its predecessors yet also struggles with its own set of tradeoffs and idiosyncrasies. The cleanest language features are its strict object orientation, automatic memory management, and interfaces; while some overly complex areas are threading and synchronization, the tension between primitives and objects, and the effective use of exceptions.

Weak Implementation of Platform Areas. The most oft-cited example is poor performance. An example of this is the rewrite of the input/output facilities in the NIO package for performance reasons. Besides performance, there are thin APIs that ruin the Write Once, Run Anywhere (WORA) guarantee like the File.renameTo() (Item 20) method and the Runtime.exec() method (Item 1). There are also incomplete APIs where the programmer assumes complete functionality exists. These problems are fixable and often are resolved with each new release of the Java Development Kit (JDK).

The categories of pitfalls associated with the programmer are as follows:

Shallow Knowledge. Experience increases the depth of one's knowledge. It takes time to learn the underlying concepts, interactions, and nuances of a system. This is often manifest in choosing a weak implementation when a better alternative exists (like applets versus Web Start, Item 26), misunderstanding the internal workings of an API (like the consequence of instance variables in servlets, Item 31), and shock when implementations fail to meet your expectations of behavior (characterized as an impedance mismatch in Figure i-1). Such an impedance mismatch can occur with multiple concurrent result sets (Item 41).

Bias. Without many years of experience, a programmer can weigh previous experience too heavily to the point where it unfavorably biases him or her in a particular direction. Examples of this are to not take advantage of tools to automate the development process like Ant (Item 11) and JUnit (Item 12). Another example is to stick with legacy APIs over new ones for collections (Item 21) and regular expressions (Item 30). Lastly, one more effect of bias is to bring past habits into a new context like J2ME programming (Item 22).

Invalid Assumptions. A programmer can incorrectly base decisions on invalid assumptions—for example, assuming the most direct path to the solution is the best path. This often arises in designing larger systems with JSP (Item 24) and J2EE (Item 37).

Pitfalls can be extremely frustrating to programmers. We've experienced first hand this frustration. Our goal is to help you to avoid some situations we struggled through. So now that we understand pitfalls, let's see our method for exploring them.

Dissecting a Pitfall

There are three distinct parts of a pitfall:

The Symptom or Problem. The medium by which the pitfall manifests itself. We demonstrate this with a program entitled "BadXXX.java," where "XXX" refers to the type of pitfall in question.

The Root cause of the Problem. By far, this is the most important part of revealing the pitfall. Here, we go under the hood and explain the detailed internal workings, invalid assumptions, or API deficiencies that cause programmers to stumble into the trap. Usually this explanation is supported with a diagram.

The Solution or Workaround. The final part of the pitfall is to demonstrate a fix for the problem. This is done with a program entitled "GoodXXX.java" that is the reciprocal of the "BadXXX.java" program. The solution program will often be accompanied with a run of the results, or a table or graph, which proves the problem was indeed solved.

This method of dissecting a pitfall has proven an effective way to present these programming hazards.

How This Book Differs from *Java Pitfalls*

This book borrows all the good features from the first book and improves upon it in three ways:

Broader Coverage. The first book focused on the lang, util, io, and GUI packages, whereas this book covers the J2ME, J2SE, and J2EE platforms.

New Features. This book covers the majority of new features like regular expressions, NIO, assertions, JAXR, JAXM, JAX-RPC, and many more.

Better Coverage. The authors followed the "pitfall dissection" model more consistently and thoroughly, producing pitfalls with more detail and analysis.

In general, the authors strove to outdo the first book in every regard. We sincerely hope that we succeeded and encourage your feedback.

Organization of the Book

Like the first one, this book has 50 items. Unlike the first book, in this one they are divided into three parts corresponding to the three-tiered architecture:

Part One: The Client Tier. This part covers both J2ME and J2SE and explores pitfalls in developing both networked and standalone clients. Topics covered include preferences, application deployment, logging, IO performance, and many more. This part has 22 pitfalls.

Part Two: The Web Tier. This part examines pitfalls in components that run inside the Web container, like servlets and JavaServer Pages (JSPs). These applications generate dynamic Web pages or communicate with applets, JNLP, or standalone clients. This parts covers topics like JSP design, caching, servlet filters, database connections, form validation, and many others. This part includes 14 pitfalls.

Part Three: The Enterprise Tier. Here we look at components that are part of the J2EE platform or execute inside an Enterprise Java Beans (EJB) container, like session, entity, and message-driven beans. These components interact with other enterprise systems, legacy systems, the Web tier, or directly to clients. Because

Web services play a key role in the enterprise tier, pitfalls related to some of the Web services APIs (JAXR and JAX-RPC) are in this section. Some other topics in this part are J2EE design errors, session beans, Java Data Objects (JDO), security, transactions, and many more. This part includes 14 pitfalls.

How to Use the Book

This book can be used in three primary ways: as a reference manual on specific problems, as a tutorial on the topics exposed by the problems, or as a catalyst to your organization's technical mentoring program. Let's discuss each in detail:

As a Reference Manual. You can use the table of contents to examine a specific solution to a problem you are facing. The majority of readers use this book in this manner. Some readers reported using the first book as a corporate resource in the technical library.

As a Tutorial. You can read the book cover–to-cover to learn about the underlying cause of each pitfall. Another way to approach the book this way is to browse the contents or pages and then read the pitfalls that interest you. Lastly, another use is to treat each pitfall as a bite-sized tutorial to present at a "brown-bag lunch" internal training session or technical exchange meeting.

As Part of a Mentoring Program. You can use the book as a starting point for a technical mentoring program in your organization. This book is founded on the principle of peer mentoring. The dictionary definition of a mentor is a "wise and trusted counselor." This often miscasts a mentor as having great age or experience. I disagree with this definition because it leads to an extreme scarcity of good mentors. *At its essence, mentoring is one aspect in the search for truth*. Thus, the key quality for being a mentor is a deep understanding of at least one domain that you are willing to share with others. Anyone can participate in this process, and I encourage you to be involved in peer mentoring. Working together, I believe we can solve the software quality crisis.

What's on the Companion Web Site?

The companion Web site will contain four key elements:

Source Code. The source code for all listings in the book will be available in a compressed archive.

Errata. Any errors discovered by readers or the authors will be listed with the corresponding corrected text.

Examples. Sample chapters, the table of contents and index will be posted for people who have not yet purchased the book to get a feel for its style and content.

Contact Addresses. The email addresses of the authors will be available as well as answers to any frequently asked questions.

Comments Welcome

This book is written by programmers for programmers. All comments, suggestions, and questions from the entire computing community are greatly appreciated. It is feedback from our readers that both makes the writing worthwhile and improves the quality of our work. I'd like to thank all the readers who have taken time to contact us to report errors, provide constructive criticism, or express appreciation.

I can be reached via email at mike@daconta.net or via regular mail:

Michael C. Daconta
c/o Robert Elliott
Wiley Publishing, Inc.
111 River Street
Hoboken, NJ 07030

Best wishes,

Michael Daconta
Sierra Vista, Arizona

NOTE **About the code: In many of the code listings you will find a wrap character at the far right of some lines of code. We have used this character, ⮐, to indicate turnovers where the space available did not allow for all the characters to set on the same line. The line of code directly below a ⮐ is a direct unbroken continuation of the line above it, where the ⮐ appears.**

Acknowledgments

This book has been a difficult journey. It survived through three co-author changes, several delays, and a move to a new state. Along the way, the vision of the book never faded, and in some way it willed itself into existence. All the authors believe that uncovering pitfalls helps programmers produce better programs with less frustration. I would like to thank those people who helped me through this challenge: my family—Lynne, CJ, Greg, and Samantha; my editors at Wiley Publishing, Inc.—Bob Elliott and Emilie Herman; my co-authors—Kevin, Clay, and Donnie; Al Saganich for contributing two pitfalls; my supervisors, customer and coworkers on the Virtual Knowledge Base project—Ted Wiatrak, Danny Proko, Joe Broussard, Joe Rajkumar, Joe Vitale, Maurita Soltis, and Becky Smith; my editor at Javaworld—Jennifer Orr; my friends at Oberon—Jodi Johnson and Dave Young; and finally, I would like to thank our readers who share our goal of producing great programs. Thanks and best wishes!

Michael C. Daconta

First of all, I would like to thank my co-authors—Mike, Clay, and Don. Your hard work on this project, and our many brainstorming sessions together at Cracker Barrel, helped create a good book full of our Java experiences and lessons learned. Second, I would like to thank my other new friends at McDonald Bradley and our entire VKB team. What a team of incredible folks.

I would like to give special thanks to a few people who suggested pitfalls and ideas for this book—John Sublett from Tridium, Inc. in Richmond, Virginia, Kevin Moran from McDonald Bradley, and Jeff Walawender from Titan Systems. Lois G. Schermerhorn and Helen G. Smith once again served as readability editors for some of my material. Special thanks to Stan and Nicole Schermerhorn for allowing me to use their company's name, Lavender Fields Farm, in a fictional scenario in this book. Also, thanks to Al Alexander, who granted me permission to use DevDaily's DDConnectionBroker to demonstrate a solution to a common programming pitfall.

My experience on Java projects with many software engineers and architects over the years helped me in writing this book: Ralph Cook, Martin Batts, Jim Johns, John Vrankovich, Dave Musser, Ron Madagan, Phil Collins, Jeff Thomason, Ken Pratt, Adam Dean, Stuart Gaudet, Terry Bailey, JoAnn White, Joe Pecore, Dave Bishop, Kevin Walmsley, Ed Kennedy, George Burgin, Vaughn Bullard, Daniel Buckley, Stella Aquilina, Bill Flynn, Charlie Lambert, and Dave Cecil III. I would also like to thank Bill Lumbergh, and the TPS Report support team at Initech—Peter, Samir, and Michael.

I would like to express thanks to my dad, who jump-started my career in computer science by buying me a Commodore Vic-20 in 1981. Making the most of the 5 KB of memory on that box, I learned not to use memory-consuming spaces in my code—perhaps contributing to "readability" pitfalls when I started writing code in college. Thanks to my former teachers who helped me in my writing over the years—Audrey Guengerich-Baylor and Rebecca Wright-Reynolds.

Over the last year, I have been blessed with many new friends at New Hanover Presbyterian Church and neighbors in Ashcreek in Mechanicsville, Virginia. Special thanks to the guys in last year's Wednesday night Bible study—Rich Bralley, Billy Craig, Matt Davis, Dan Hulen, Chuck Patterson, Ben Roberson, Keith Scholten, Todd Tarkington, and Matt Van Wie. I would also like to thank folks who helped me take a break to focus on playing the trumpet this year—Ray Herbek, Jeff Sigmon, Rita Smith, and Kenny Stockman.

Finally, I would like to thank my wonderful wife Gwen. Without her love and support, this book would never have been possible!

Kevin T. Smith

All of my material for this book is drawn largely from an object-oriented class I teach and a lot of great developers I've worked with over the years. Specifically, I'd like to thank these people who inspired me with their probity and hard work: Peter Len, Joe Vitale, Scot Shrager, Mark "Mojo" Mitchell, Wendong Wang, Chiming Huang, Feng Peng, Henry Chang, Bin Li, Sanath Shetty, Henry, John and Andy Zhang, Swati Gupta, Chi Vuong, Prabakhar Ramakrishnan, and Yuanlin Shi.

Special thanks goes to my beloved wife Van and her support and assistance during the writing of this book and the three coauthors of this book who are really progressive thinkers and great guys to hang with.

Donald Avondolio

First, I would like to thank my wife Alicia for all of her patience and understanding while I wrote this book. You truly are the greatest and I love you more than you understand. To Jennifer, my baby doll, you are the greatest gift a father could ever receive. To Stephanie, I love you and I will never forget you. I would like to thank my parents, Bill and Kay, for being, well, my parents. Nothing I could write here could express the impact you have had on my life.

I would like to thank my fellow authors, Mike, Kevin, and Don, for being patient while I got up to speed. You guys are not only exceptional technical talents, but also exceptional people. To my team—Mark Mitchell (aka Mojo), Mauro Marcellino (Tre, who saw us here when we were riding ambulances together), Peter Len, Marshall Sayen, Scot Schrager, Julie Palermo/Hall/Bishop, and Joe Vitale, you guys are the

greatest and it was a privilege to serve as your lead. Mojo, it has been enjoyable to watch you progress from apprentice to master. Vic Fraenckel and Mike Shea, you aren't on my team, but you are certainly on *the* team, in spite of being part of the Borg. To Becky Smith, my fellow warrior, we have been through a lot of battles (some with each other), but it has been a pleasure working with you and your team.

To all the guys who have served with me on Gainesville District VFD Duty Crew A (and particularly its leader, Captain Bob Nowlen)—Patrick Vaughn, Marshall Sayen, Gary Sprifke, Mike Nelson, Matt Zalewski, Doug Tognetti, Frank Comer; we have seen some crazy things together, and I have been happy to be the one to drive you there. Chief Richard Bird, I would like to thank you for your leadership of our department and service to our community, which has been ongoing since before I was born. To the guys at the Dumfries-Triangle VFD, now you know where I went (writing this book): Brian Thomason, Tim Trax, Brian Kortuem, Brian Lichty, Nick Nanna, Scott Morrison, Brian Martin, Jack Hoffman, Craig Johnson, and Randy Baum—I wish I could name all of you. Volunteer firefighters truly are the salt of the earth, and I am happy to be among you.

To those who have served as mentors of mine through the years (in no particular order): Tom Bachmann, Seth Goldrich, Don Avondolio, Danny Proko, Burr Datz, Kevin McPhilamy, Shawn Bohner, John Viega, Pat Wolfe, Alex Blakemore (nonpolitical matters), Sam Redwine, and many others that I will kick myself for forgetting later. To Ted Wiatrak and Major Todd Delong, I would like to thank you guys for believing in us and giving us a shot to help some very important people. In closing, I would like to thank two of my Brother Rats, Matt Tyrrell and Jeff Bradford, for being, well, like brothers to me.

W. Clay Richardson

PART One

The Client Tier

"Now, if we regard a programming language primarily as a means of feeding problems into a machine, the quality of a programming language becomes dependent on the degree in which it promotes 'good use of the machine.'"

Edsger W. Dijkstra,
"On the Design of Machine Independent Programming Languages"

There have been a number of high-profile failures with using Java for major client-side applications. Corel developed an office suite in Java but scrapped the project after an early beta release. Netscape embarked on a pure Java version of a Web browser (referred to as "Javagator"), but the project was canceled before any code was released. Although these were the early days of client-side Java, in the 1997 to 1998 time frame, it was enough to cast a pall over client-side Java, and the momentum shifted to server-side Java. Yet, even under that shadow of server-side Java, the client tier continued to improve. Richer user interfaces, faster virtual machines, a fine-grained security model, and easier network deployment came to the platform piecemeal with each new release. So, borrowing a play from the Microsoft playbook, client-side Java has continued to address its shortcomings and improve with each release. Today we have high-profile and successful commercial applications in Java like ThinkFree Office, Borland's JBuilder, TIBCO's Turbo XML editor, and TogetherSoft's Together Control Center UML modeling tool. So, it is possible to develop rich client applications in Java. This part will assist you in that endeavor.

This part explores pitfalls in three general areas of client-side development: performance, nonintuitive application programming interfaces (APIs), and better alternatives. Here are some highlights of pitfalls in each area.

Performance has long been the bane of client-side Java. The first book, *Java Pitfalls: Time-Saving Solutions and Workarounds to Improve Programs*, had numerous pitfalls on performance, and many other books and Web sites have come out on Java performance tuning. This part has two pitfalls on performance:

NIO Performance and Pitfalls (Item 2). This pitfall examines the IO performance improvements of the New IO package (NIO). The pitfall examines file channels, ByteBuffers, and non-blocking server IO.

J2ME Performance and Pitfalls (Item 22). This pitfall ports a Swing application to the J2ME platform and uncovers both API pitfalls and over 20 optimizations for these small footprint devices.

Nonintuitive APIs cause hours of frustration, and the majority of pitfalls in this part are in this area. We carefully dissect the APIs, examine the internal workings of the software, and offer workarounds to the problem. The workarounds sometimes involve a proper sequence of operations, the use of a different class, or the abandonment of the standard API for an open-source alternative.

When `Runtime.exec()` Won't (Item 1). This pitfall is a canonical example of a mismatch between user expectations and the capability of an incomplete API.

Avoiding Granularity Pitfalls in `java.util.logging` (Item 5). The new `java.util.logging` API has some hidden complexities and relationships that affect the level of reporting granularity. You must understand the relationship between loggers and handlers to effectively use this API.

The Wrong Way to Search a DOM (Item 8). With JDK 1.4, the Java platform provided native support for XML with the javax.xml package. Unfortunately, the most intuitive representation of a Document Object Model (DOM) is not the correct representation, and this pitfall goes under the hood to see why.

The Saving-a-DOM Dilemma (Item 9). While JAXP introduced standard ways to create and manipulate XML DOM trees, it provides weak capabilities for persisting them—forcing developers to use implementation-specific methods. This pitfall discusses those challenges.

The Failure to Execute (Item 13). Java Archives or JAR files are the primary binary distribution for Java components. Unfortunately, there is great confusion about how these files work and how to make them executable. This pitfall explores those problems and provides an explanation of the best practices in using JAR files.

When Posting to a URL Won't (Item 17). The `URL` and `URLConnection` classes in the `java.net` API were designed at a very high level of abstraction that can be confusing. This pitfall demonstrates several incorrect ways to use the API, the reasons behind the deficiencies, and both a solution and open-source alternative to the problem.

The existence of better alternatives is an ever-growing problem as the platform ages and poor alternatives are kept to maintain backward compatibility. In addition to new APIs in the platform, again, open-source alternatives are proving themselves to be the best solution for many services.

I Prefer Not to Use Properties (Item 3). This pitfall demonstrates some weaknesses of the `Properties` class and how java.util.prefs package offers a better solution.

When Information Hiding Hides Too Much (Item 4). A frequent problem with abstracting things from developers is that it can hide important information

from developers. Exceptions are a classic example and this pitfall demonstrates how the new JDK 1.4 chained exception facility solves it.

When Implementations of Standard APIs Collide (Item 6). With XML becoming part of JDK 1.4, an immediate issue arose from the fact that the XML standards do not synchronize with JDK releases. This pitfalls address how JDK 1.4 supports upgrades to these endorsed standards.

My Assertions Are Not Gratuitous! (Item 7). There is often a lag between the introduction of a new feature and adoption of that feature by the majority of programmers. For key reliability enhancements like assertions, this adoption gap is a serious pitfall. This item walks you through this important facility.

Apache Ant and Lifecycle Management (Item 11). Though most pitfalls in this part occur at the language and API level, this pitfall takes a broader look at a better alternative for the software lifecycle. For team development, not using a build tool like Ant is a serious pitfall.

JUnit: Unit Testing Made Simple (Item 12). Much like the Ant build tool, JUnit is a critical tool in assuring the quality of code through unit tests. This pitfall demonstrates how to effectively use JUnit and why failing to use it is bad.

Use `Iteration` over `Enumeration` (Item 21). The `Collection` APIs have proven themselves both effective and popular. This pitfall uncovers a weakness in the `Enumeration` implementation, examines the internals to uncover the source of the problem, and reveals how `Iteration` solves the problem.

There are 22 pitfalls in this part of the book, covering a wide array of client-side traps to avoid. Using these workarounds and techniques will enable you to build robust client-side applications that make "good use of the machine." The remainder of the pitfalls in this section are:

Mouse Button Portability (Item 10). Java is an outstanding choice for cross development application development. Unfortunately, not all platforms are the same, especially when it comes to the most popular input device—the mouse. This pitfall shows the challenges involved in working with these different input devices.

What Do You Collect? (Item 14). The issue of over abstraction can be particularly acute when dealing with the `Collection` APIs. This pitfall shows examples of not knowing the type contained in a collection and discusses emerging strategies for solving this problem.

Avoid Singleton Pitfalls (Item 15). The Singleton pattern is a widely used pattern in Java development. Unfortunately, there are numerous mistakes that developers make in how they use a Singleton. This pitfall addresses these mistakes and suggests some remedies.

When `setSize()` Won't Work (Item 16). Frequently, developers, especially new developers, use methods without understanding the overall API associated with them. A perfect example is the use of `setSize()` which leads to unexpected results. This pitfall examines not only the mistaken use of `setSize()` but also the concepts of layout mangers.

Effective String Tokenizing (Item 18). The operation of the `StringTokenizer` class is frequently misunderstood. Interesting problems occur with multiple character delimiting strings. This pitfall examines those problems, and explains how they can be avoided.

`JLayered Pane` Pitfalls (Item 19). This pitfall examines issues with using Swing's `JLayered Pane`, particularly related to the use of layout mangers with it.

When `File.renameTo()` Won't (Item 20). The interaction with files in Java can produce unexpected results, particularly in working across filesystems or platforms. This pitfall examines those unexpected results and offers solutions for resolving them.

Item 1: When Runtime.exec() Won't[1]

The class `java.lang.Runtime` has a static method called `getRuntime()` to retrieve the current Java runtime environment. This is the only way to get a reference to the `Runtime` object. With that reference you can run external programs by invoking the `exec()` method of the `Runtime` class. One popular reason to do this is to launch a browser to display some kind of help page in HTML. There are four overloaded versions of the `exec()` command. Those method prototypes are:

- `public Process exec(String command);`
- `public Process exec(String [] cmdArray);`
- `public Process exec(String command, String [] envp);`
- `public Process exec(String [] cmdArray, String [] envp);`

The general idea behind all of the methods is that a command (and possible a set of arguments) are passed to an operating system-specific function call to create an operating system-specific process (a running program) with a reference to a `Process` class returned to the Java Virtual Machine (VM). The `Process` class is an abstract class because there will be a specific subclass of `Process` for each operating system. There are three possible input parameters to these methods: a single `String` that represents both the program to execute and any arguments to that program, an array of `Strings` that separate the program from its arguments, and an array of environment variables. The environment variables are passed in the form *name=value*. It is important to note that if you use the version of `exec()` with a single `String` for both the program and its arguments, the `String` is parsed using whitespace as the delimiter via the `StringTokenizer` class.

The prevalent first test of an API is to code its most obvious methods. For example, to exec a process that is external to the JVM, we use the `exec()` method. To see the value that the external process returns, we use the `exitValue()` method on the `Process` class. In our first example, we will attempt to execute the Java compiler (javac.exe). Listing 1.1 is a program to do that.

[1] This pitfall was first printed by *JavaWorld* (www.javaworld.com) in "When Runtime.exec() won't", December 2000 (http://www.javaworld.com/javaworld/jw-12-2000/jw-1229-traps.html?) and is reprinted here with permission. The pitfall has been updated from reader feedback.

```
01: package org.javapitfalls.item1;
02:
03: import java.util.*;
04: import java.io.*;
05:
06: public class BadExecJavac
07: {
08:     public static void main(String args[])
09:     {
10:         try
11:         {
12:             Runtime rt = Runtime.getRuntime();
13:             Process proc = rt.exec("javac");
14:             int exitVal = proc.exitValue();
15:             System.out.println("Process exitValue: " + exitVal);
16:         } catch (Throwable t)
17:           {
18:             t.printStackTrace();
19:           }
20:     }
21: }
```

Listing 1.1 BadExecJavac.java

A run of BadExecJavac produces the following:

```
E:\classes\org\javapitfalls\item1 >java
org.javapitfalls.item1.BadExecJavac
java.lang.IllegalThreadStateException: process has not exited
        at java.lang.Win32Process.exitValue(Native Method)
        at BadExecJavac.main(BadExecJavac.java:13)
```

The program failed to work because the `exitValue()` method will throw an `IllegalThreadStateException` if the external process has not yet completed. While this is stated in the documentation, it is strange in that it begs the question: why not just make this method wait until it can give me a valid answer? A more thorough look at the methods available in the `Process` class reveals a `waitFor()` method that does precisely that. In fact, the `waitFor()` method returns the exit value, which means that you would not use both methods in conjunction. You choose one or the other. The only possible reason for you to use the `exitValue()` method over the `waitFor()` method is that you do not want to have your program block waiting on an external process that may never complete. Personally, I would prefer a boolean parameter called `waitFor` be passed into the `exitValue()` method to determine whether or not the current thread should wait. I think a boolean would be better because the name `exitValue()` is a better name for this method and it is unnecessary to have two methods perform the same function under different conditions. Such simple "condition" discrimination is the domain of an input parameter.

So, the first pitfall relating to `Runtime.exec()` is beware the `IllegalThreadStateException` and either catch it or wait for the process to complete. Now, let's fix the problem in the above program and wait for the process to complete. In Listing 1.2, the program again attempts to exec the program javac.exe and then waits for the external process to complete.

```
01: package org.javapitfalls.item1;
02:
03: import java.util.*;
04: import java.io.*;
05:
06: public class BadExecJavac2
07: {
08:     public static void main(String args[])
09:     {
10:         try
11:         {
12:             Runtime rt = Runtime.getRuntime();
13:             Process proc = rt.exec("javac");
14:             int exitVal = proc.waitFor();
15:             System.out.println("Process exitValue: " + exitVal);
16:         } catch (Throwable t)
17:           {
18:             t.printStackTrace();
19:           }
20:     }
21: }
```

Listing 1.2 BadExecJavac2.java

Unfortunately, a run of BadExecJavac2 produces no output. The program hangs and never completes! Why is the javac process never completing? The javadoc documentation provides the answer. It says, "Because some native platforms only provide limited buffer size for standard input and output streams, failure to promptly write the input stream or read the output stream of the subprocess may cause the subprocess to block, and even deadlock." So, is this just a case of programmers not following "RTFM" (read the f-ing manual)? The answer is partially yes. In this case, reading the manual would get you halfway there. It tells you that you need to handle the streams to your external process but does not tell you how. Besides RTFM, there is another variable at play here that cannot be ignored when you examine the large number of programmer questions and errors over this API in the newsgroups. The `Runtime.exec()` and `Process` APIs seem extremely simple, but that simplicity is deceiving, because the simple (translate to obvious) use of the API is prone to error.

The lesson here for the API designer is to reserve simple APIs for simple operations. Operations prone to complexities and platform-specific dependencies should reflect the domain accurately. It is possible for an abstraction to be carried too far. An example of a more complete API to handle these operations is the JConfig library (available at http://www.tolstoy.com/samizdat/jconfig.html). So, now let's follow the documentation and handle the output of the javac process. When you run javac without any arguments, it produces a set of usage statements that describe how to run the program and the meaning of all the available program options. Knowing that this is going to the stderr stream, it is easy to write a program to exhaust that stream before waiting on the process to exit. Listing 1.3 does just that. While that approach will work, it is not a good general solution. That is why the program in Listing 1.3 is named MediocreExecJavac; it is only a mediocre solution. A better solution would empty both the standard error stream and the standard output stream. And the best solution would empty these streams simultaneously (this is demonstrated later).

```
01: package org.javapitfalls.item1;
02:
03: import java.util.*;
04: import java.io.*;
05:
06: public class MediocreExecJavac
07: {
08:     public static void main(String args[])
09:     {
10:         try
11:         {
12:             Runtime rt = Runtime.getRuntime();
13:             Process proc = rt.exec("javac");
14:             InputStream stderr = proc.getErrorStream();
15:             InputStreamReader isr = new InputStreamReader(stderr);
16:             BufferedReader br = new BufferedReader(isr);
17:             String line = null;
18:             System.out.println("<ERROR>");
19:             while ( (line = br.readLine()) != null)
20:                 System.out.println(line);
21:             System.out.println("</ERROR>");
22:             int exitVal = proc.waitFor();
23:             System.out.println("Process exitValue: " + exitVal);
24:         } catch (Throwable t)
25:           {
26:             t.printStackTrace();
27:           }
28:     }
29: }
```

Listing 1.3 MediocreExecJavac.java

A run of MediocreExecJavac produces the following:

```
E:\classes\org\javapitfalls\item1>java
org.javapitfalls.item1.MediocreExecJavac
<ERROR>
Usage: javac <options> <source files>
where <options> includes
  -g                     Generate all debugging info
  -g:none                Generate no debugging info
  -g:{lines,vars,source} Generate only some debugging info
  -O                     Optimize; may hinder debugging or enlarge class
files
  -nowarn                Generate no warnings
... some output removed for brevity ...
</ERROR>
Process exitValue: 2
```

So, MediocreExecJavac works and produces an exit value of 2. Normally, an exit value of 0 means success and nonzero means error. Unfortunately, the meaning of these exit values is operating system-specific. The Win32 error code for a value of 2 is the error for "file not found." This makes sense, since javac expects us to follow the program with the source code file to compile. So, the second pitfall to avoid with `Runtime.exec()` is ensuring you process the input and output streams if the program you are launching produces output or expects input.

Going back to windows, many new programmers stumble on `Runtime.exec()` when trying to use it for nonexecutable commands like `dir` and `copy`. So, we replace `"javac"` with `"dir"` as the argument to `exec()` like this:

```
Process proc = rt.exec("dir");
```

This line is replaced in the source file called BadExecWinDir, which when run produces the following:

```
E:\classes\org\javapitfalls\item1>java
org.javapitfalls.item1.BadExecWinDir
java.io.IOException: CreateProcess: dir error=2
        at java.lang.Win32Process.create(Native Method)
        at java.lang.Win32Process.<init>(Unknown Source)
        at java.lang.Runtime.execInternal(Native Method)
        at BadExecWinDir.main(BadExecWinDir.java:12)
```

As stated earlier, the error value of 2 means file not found—meaning that the executable named dir.exe could not be found. That is because the directory command is part of the window command interpreter and not a separate executable. To run the window command interpreter, you execute either command.com or cmd.exe depending on the windows operating system you are using. Listing 1.4 runs a copy of the Windows Command Interpreter and then executes the user-supplied command (like `dir`).

```
01: package org.javapitfalls.item1;
02:
03: import java.util.*;
04: import java.io.*;
05:
06: class StreamGobbler extends Thread
07: {
08:     InputStream is;
09:     String type;
10:
11:     StreamGobbler(InputStream is, String type)
12:     {
13:         this.is = is;
14:         this.type = type;
15:     }
16:
17:     public void run()
18:     {
19:         try
20:         {
21:             InputStreamReader isr = new InputStreamReader(is);
22:             BufferedReader br = new BufferedReader(isr);
23:             String line=null;
24:             while ( (line = br.readLine()) != null)
25:             {
26:                 System.out.println(type + ">" + line);
27:                 System.out.flush();
28:             }
29:         } catch (IOException ioe)
30:             {
31:                 ioe.printStackTrace();
32:             }
33:     }
34: }
35:
36: public class GoodWindowsExec
37: {
38:     public static void main(String args[])
39:     {
// ... command line check omitted for brevity ...
45:
46:         try
47:         {
48:             String osName = System.getProperty("os.name" );
49:             System.out.println("osName: " + osName);
50:             String[] cmd = new String[3];
```

Listing 1.4 GoodWindowsExec.java *(continued)*

```
51:
52:             if(osName.equals("Windows NT") ||
53:                 osName.equals("Windows 2000"))
54:             {
55:                 cmd[0] = "cmd.exe" ;
56:                 cmd[1] = "/C" ;
57:                 cmd[2] = args[0];
58:             }
59:             else if( osName.equals( "Windows 95" ) )
60:             {
61:                 cmd[0] = "command.com" ;
62:                 cmd[1] = "/C" ;
63:                 cmd[2] = args[0];
64:             }
65:
66:             Runtime rt = Runtime.getRuntime();
67:             System.out.println("Execing " + cmd[0] + " " + cmd[1]
68:                                + " " + cmd[2]);
69:             Process proc = rt.exec(cmd);
70:             // any error message?
71:             StreamGobbler errorGobbler = new
72:                 StreamGobbler(proc.getErrorStream(), "ERROR");
73:
74:             // any output?
75:             StreamGobbler outputGobbler = new
76:                 StreamGobbler(proc.getInputStream(), "OUTPUT");
77:
78:             // kick them off
79:             errorGobbler.start();
80:             outputGobbler.start();
81:
82:             // any error???
83:             int exitVal = proc.waitFor();
84:             System.out.println("ExitValue: " + exitVal);
85:
86:         } catch (Throwable t)
87:           {
88:             t.printStackTrace();
89:           }
90:     }
91: }
```

Listing 1.4 *(continued)*

Running GoodWindowsExec with the `dir` command produces:

```
E:\classes\org\javapitfalls\item1>java
org.javapitfalls.item1.GoodWindowsExec "dir *.java"
Execing cmd.exe /C dir *.java
OUTPUT> Volume in drive E has no label.
OUTPUT> Volume Serial Number is 5C5F-0CC9
OUTPUT>
OUTPUT> Directory of E:\classes\com\javaworld\jpitfalls\article2
OUTPUT>
OUTPUT>10/23/00  09:01p                   805 BadExecBrowser.java
OUTPUT>10/22/00  09:35a                   770 BadExecBrowser1.java
... (some output omitted for brevity)
OUTPUT>10/09/00  05:47p                23,543 TestStringReplace.java
OUTPUT>10/12/00  08:55p                   228 TopLevel.java
OUTPUT>              22 File(s)         46,661 bytes
OUTPUT>                         19,678,420,992 bytes free
ExitValue: 0
```

Running GoodWindowsExec with any associated document type will launch the application associated with that document type. For example, to launch Microsoft Word to display a Word document (a .doc extension), you type

```
>java org.javapitfalls.item1.GoodWindowsExec "yourdoc.doc"
```

Notice that GoodWindowsExec uses the `os.name` system property to determine which Windows operating system you are running in order to use the appropriate command interpreter. After execing the command interpreter, we handle the standard error and standard input streams with the `StreamGobbler` class. The `StreamGobbler` class empties any stream passed into it in a separate thread. The class uses a simple `String` type to denote which stream it is emptying when it prints the line just read to the console. So, the third pitfall to avoid is to know whether you are executing a standalone executable or an interpreted command. At the end of this section, I will demonstrate a simple command-line tool that will help you with that analysis.

It is important to note that the `Process` method used to get the output stream of the process is called `getInputStream()`. The thing to remember is that the perspective is from the Java program and not the external process. So, the output of the external program is the input to the Java program. And that logic carries over to the external programs input stream, which is an output stream to the Java program.

One final pitfall to cover with `Runtime.exec()` is not to assume that the `exec()` accepts any `String` that your command line (or shell) accepts. It is much more limited and not cross-platform. The primary cause of this pitfall is users attempting to use the `exec()` method that accepts a single `String` just like a command line. This confusion may be due to the parameter name for the `exec()` method being `command`. The

programmer incorrectly associates the parameter command with anything he or she can type on a command line instead of associating it with a single program and its arguments, for example, a user trying to execute a program and redirect its output in one call to `exec()`. Listing 1.5 attempts to do just that.

```
01: package org.javapitfalls.item1;
02:
03: import java.util.*;
04: import java.io.*;
05:
// StreamGobbler removed for brevity
32:
33: public class BadWinRedirect
34: {
35:     public static void main(String args[])
36:     {
37:         try
38:         {
39:             Runtime rt = Runtime.getRuntime();
40:             Process proc = rt.exec(
                        "java jecho 'Hello World' > test.txt");
// remaining code same as GoodWindowsExec.java
63: }
```

Listing 1.5 BadWinRedirect.java

Running BadWinRedirect produces:

```
E:\classes\org\javapitfalls\Item1>java
org.javapitfalls.Item1.BadWinRedirect
OUTPUT>'Hello World' > test.txt
ExitValue: 0
```

The program BadWinRedirect attempted to redirect the output of a simple Java version of an echo program into the file test.txt. Unfortunately, when we check to see if the file test.txt is created, we find that it does not exist. The jecho program simply takes its command-line arguments and writes them to the standard output stream. The source for jecho is available on the Web site. The user assumed you could redirect standard output into a file just like you do on a DOS command line. Unfortunately, that is not the way you redirect the output. The incorrect assumption here is that the `exec()` method acts like a shell interpreter, and it does not. The `exec()` method executes a single executable (a program or script). If you want to process the stream to either redirect it or pipe it into another program, you must do that programmatically using the java.io package. Listing 1.6 properly redirects the standard output stream of the jecho process into a file.

```
01: package org.javapitfalls.item1;
02:
03: import java.util.*;
04: import java.io.*;
05:
06: class StreamGobbler extends Thread
07: {
08:     InputStream is;
09:     String type;
10:     OutputStream os;
11:
12:     StreamGobbler(InputStream is, String type)
13:     {
14:         this(is, type, null);
15:     }
16:
17:     StreamGobbler(InputStream is, String type, OutputStream
redirect)
18:     {
19:         this.is = is;
20:         this.type = type;
21:         this.os = redirect;
22:     }
23:
24:     public void run()
25:     {
26:         try
27:         {
28:             PrintWriter pw = null;
29:             if (os != null)
30:                 pw = new PrintWriter(os);
31:
32:             InputStreamReader isr = new InputStreamReader(is);
33:             BufferedReader br = new BufferedReader(isr);
34:             String line=null;
35:             while ( (line = br.readLine()) != null)
36:             {
37:                 if (pw != null)
38:                 {
39:                     pw.println(line);
40:                     pw.flush();
41:                 }
42:                 System.out.println(type + ">" + line);
43:             }
44:             if (pw != null)
45:                 pw.flush();
46:         } catch (IOException ioe)
```

Listing 1.6 GoodWinRedirect.java *(continued)*

```
47:             {
48:             ioe.printStackTrace();
49:             }
50:     }
51: }
52:
53: public class GoodWinRedirect
54: {
55:     public static void main(String args[])
56:     {
// ... command argument check omitted for brevity ...
62:
63:         try
64:         {
65:             FileOutputStream fos = new FileOutputStream(args[0]);
66:             Runtime rt = Runtime.getRuntime();
67:             Process proc = rt.exec("java jecho 'Hello World'");
68:             // any error message?
69:             StreamGobbler errorGobbler = new
70:                 StreamGobbler(proc.getErrorStream(), "ERROR");
71:
72:             // any output?
73:             StreamGobbler outputGobbler = new
74:                 StreamGobbler(proc.getInputStream(), "OUTPUT", fos);
75:
76:             // kick them off
77:             errorGobbler.start();
78:             outputGobbler.start();
79:
80:             // any error???
81:             int exitVal = proc.waitFor();
82:             System.out.println("ExitValue: " + exitVal);
83:             fos.flush();
84:             fos.close();
85:         } catch (Throwable t)
86:           {
87:             t.printStackTrace();
88:           }
89:     }
90: }
```

Listing 1.6 *(continued)*

Running GoodWinRedirect produces

```
E:\classes\org\javapitfalls\item1>java
org.javapitfalls.item1.GoodWinRedirect test.txt
OUTPUT>'Hello World'
ExitValue: 0
```

After running GoodWinRedirect, we find that test.txt does exist. The solution to the pitfall was to simply handle the redirection by handling the standard output stream of the external process separate from the `Runtime.exec()` method. We create a separate `OutputStream`, read in the filename to redirect the output to, open the file, and write the output we receive from the standard output of the spawned process to the file. In Listing 1.7 this is accomplished by adding a new constructor to our `StreamGobbler` class. The new constructor takes three arguments: the input stream to gobble, the type `String`, which labels the stream we are gobbling, and lastly the output stream to redirect the input to. This new version of `StreamGobbler` does not break any of the previous code it was used in, since we have not changed the existing public API (just extended it). A reader of the *JavaWorld* article, George Neervoort, noted an important improvement to GoodWinRedirect is to ensure that the streams have completed reading input from the process. This is necessary because it is possible for the process to end before the threads have completed. Here is his solution to that problem (thanks George!):

```
int exitVal = proc.waitFor();
errorGobbler.join();
outputGobbler.join();
```

Since the argument to `Runtime.exec()` is operating system-dependent, the proper commands to use will vary from one operating system to another. So, before finalizing arguments to `Runtime.exec()`, it would be valuable to be able to quickly test the arguments before writing the code. Listing 1.7 is a simple command-line utility to allow you to do just that.

```
01: package org.javapitfalls.item1;
02:
03: import java.util.*;
04: import java.io.*;
05:
```

Listing 1.7 TestExec.java *(continued)*

```
// Class StreamGobbler omitted for brevity
32:
33: public class TestExec
34: {
35:     public static void main(String args[])
36:     {
// argument check omitted for brevity
42:
43:         try
44:         {
45:           String cmd = args[0];
46:           Runtime rt = Runtime.getRuntime();
47:           Process proc = rt.exec(cmd);
48:                                                    // remaining code
Identical to GoodWindowsExec.java
68:     }
69: }
70:
```

Listing 1.7 *(continued)*

Running TestExec to launch the Netscape browser and load the Java help documentation produces:

```
E:\classes\org\javapitfalls\item1>java org.javapitfalls.item1.TestExec ⮐
"e:\java\docs\index.html"
java.io.IOException: CreateProcess: e:\java\docs\index.html error=193
        at java.lang.Win32Process.create(Native Method)
        at java.lang.Win32Process.<init>(Unknown Source)
        at java.lang.Runtime.execInternal(Native Method)
        at TestExec.main(TestExec.java:45)
```

So, our first test failed with an error of 193. The Win32 Error for value 193 is "not a valid Win32 application." This tells us that there is no path to an associated application (like Netscape) and the process cannot run an HTML file without an associated application. So, we can try the test again, this time giving it a full path to Netscape (or we could have added Netscape to our `PATH` environment variable). A second run of TestExec produces:

```
E:\classes\com\javaworld\jpitfalls\article2>java TestExec ⮐
"e:\program files\netscape\program\netscape.exe e:\java\docs\index.html"
ExitValue: 0
```

This worked! The Netscape browser was launched, and it then loaded the Java help documentation.

One useful exercise would be to modify TestExec to redirect the standard input or standard output to a file. When you are executing the javac compiler on Windows 95 or Windows 98, this would solve the problem of error messages scrolling off the top of the limited command-line buffer. Here are the key lines of code to capture the standard output and standard error to a file:

```
FileOutputStream err = new FileOutputStream("stderror.txt");
FileOutputStream out = new FileOutputStream("stdout.txt");
Process proc = Runtime.getRuntime().exec(args);

// any error message?
StreamGobbler errorGobbler
  = new StreamGobbler(proc.getErrorStream(), "ERR", err);
// any output?
StreamGobbler outputGobbler
  = new StreamGobbler(proc.getInputStream(), "OUT", out);
```

One other improvement to TestExec would be a command-line switch to accept input from standard-in. This input would then be passed to the spawned external program using the `Process.getOutputStream()` method. This pitfall has explored several problems in using the `Runtime.exec()` method and supplied workarounds to assist you in executing external applications.

Item 2: NIO Performance and Pitfalls

The NIO packages represent a step forward for input/output (IO) performance and new features; however, these enhancements do not come without some cost in complexity and potential pitfalls. In this item, we examine some of the benefits and deficiencies in this new package. Before we examine specific features, let's get a high-level view of the package and its features. There are four major abstractions in the NIO packages:

Channels. A channel represents a connection to entities capable of performing IO operations like files and sockets. Thus, channels are characterized by the thing they are connected to like `FileChannel` or `SocketChannel`. For fast IO, channels work in conjunction with buffers to read from the source or write to the sink. There are also interfaces for different roles a channel can play like `GatheringByteChannel` or `ScatteringByteChannel`, which can write or read from multiple sequences of buffers. Lastly, you can get a channel from the `FileInputStream`, `FileOutputStream`, `ServerSocket`, `Socket`, and `RandomAccessFile` classes. Unfortunately, adding these new IO metaphors to the existing crowded field (streams, pipes, readers, writers) is extremely confusing to most programmers. Also, the metaphors overlap and their APIs are intertwined. Overall, the complexity and overlapping functionality of the numerous IO classes is its biggest pitfall.

Buffers. Buffers are generic data containers for random access manipulation of a single data type. These are integrated with the `Charsets` and `Channels` classes. Each buffer contains a mark, position, limit, and capacity value (and corresponding methods for manipulation). The position can be set to any point in the buffer. Buffers are set in a mode (reading or writing) and must be "flipped" to change the mode.

Charsets. A charset is a named mapping between bytes and 16-bit Unicode characters. The names of these character sets are listed in the IANA charset registry (http://www.iana.org/assignments/character-sets).

Selectors. A `Selector` is a class that multiplexes a `SelectableChannel`. A `SelectableChannel` has a single subclass called `AbstractSelectableChannel`. The `AbstractSelectableChannel` has four direct subclasses: `DatagramChannel`, `Pipe.SinkChannel`, `Pipe.SourceChannel`, `ServerSocketChannel`, and `SocketChannel`. In network communications, a multiplexer is a device that can send several signals over a single line. The Selector class manages a set of one or more "selectable" channels and provides a single point of access to input from any of them.

Table 2.1 lists all the key classes and interfaces in the NIO packages.

Table 2.1 Key NIO Classes and Interfaces

PACKAGE	CLASS/INTERFACE	DESCRIPTION
`java.nio`	`Buffer`	Random access data container for temporary data storage and manipulation.
`java.nio`	`ByteBuffer`, `CharBuffer`, `DoubleBuffer`, `FloatBuffer`, `IntBuffer`, `LongBuffer`, `ShortBuffer`	Buffer that stores the specific data type represented in its name.
`java.nio`	`MappedByteBuffer`	A `ByteBuffer` whose content is a memory-mapped region of a file. These are created via the `FileChannel.map()` method.
`java.nio`	`ByteOrder`	A typesafe enumeration of byte orders: `LITTLE_ENDIAN` or `BIG_ENDIAN`.
`java.nio.channels`	`Channel`	An interface that represents an open or closed connection to an IO device.
`java.nio.channels`	`Channels`	A class that contains utility methods for working with channels.

Table 2.1 *(Continued)*

PACKAGE	CLASS/INTERFACE	DESCRIPTION
`java.nio.channels`	`FileChannel`, `Pipe`, `DatagramChannel`, `SocketChannel`, `ServerSocketChannel`, `SelectableChannel`	Specific channel types to work with specific devices like files and sockets. A `DatagramChannel` is a channel for working with `DatagramSockets` (sockets that use User Datagram Protocol, or UDP). A pipe performs communication between threads.
`java.nio.channels`	`FileLock`	A class that represents a lock on a region of a file. An object of this type is created by calling either `lock()` or `tryLock()` on a `FileChannel`.
`java.nio.channels`	`Selector`, `SelectionKey`	As stated before, a `Selector` is a class that multiplexes one or more `SelectableChannels`. A `SelectionKey` is used to specify specific operations to listen for (or select) on a specific `SelectableChannel`.
`java.nio.charset`	`Charset`	An abstract class that provides a mapping between 16-bit Unicode characters and sequences of bytes. The standard character sets are US-ASCII (7-bit ASCII), ISO-8859-1 (ISO Latin), UTF-8, UTF-16BE (big endian order), UTF-16LE (little endian order), UTF-16 (uses a byte order mark).
`java.nio.charset`	`CharsetEncoder`, `CharsetDecoder`	A `CharsetEncoder` is an engine that encodes (or transforms) a set of 16-bit Unicode characters into a specific sequence of bytes specified by the specific character set. A `CharsetDecoder` performs the opposite operation.
`java.nio.charset`	`CoderResult`	A class that reports the state of an encoder or decoder. It reports any of five states: error, malformed, underflow, overflow, or unmappable.
`java.nio.charset`	`CodingActionError`	A class that is a typesafe enumeration that specifies how an encoder or decoder should handle errors. It can be set to `IGNORE`, `REPLACE`, or `REPORT`.

Unfortunately, we cannot demonstrate all of the functionality in the NIO package. Instead, we examine three examples: a canonical file copy, little endian byte operations, and non-blocking server IO.

Canonical File Copy

Our first example is a canonical file copy operation. The old way to implement the copy would be to loop while reading and writing from a fixed-size byte buffer until we had exhausted the bytes in the file. Listing 2.1 demonstrates that method.

```
01: package org.javapitfalls.item2;
02: 
03: import java.io.*;
04: import java.nio.*;
05: import java.nio.channels.*;
06: 
07: public class SlowFileCopy
08: {
09:      public static void main(String args[])
10:      {
// ...command line check omitted for brevity ...
16: 
17:           try
18:           {
19:                long start = System.currentTimeMillis();
20:                // open files
21:                FileInputStream fis =
22:                       new FileInputStream(args[0]);
23:                FileOutputStream fos =
24:                      new FileOutputStream(args[1]);
25: 
26:                int bufSize = 4096;
27:                byte [] buf = new byte[bufSize];
28: 
29:                int cnt = 0;
30:                while ((cnt = fis.read(buf)) >= 0)
31:                      fos.write(buf, 0, (int) cnt);
32: 
33:                fis.close();
34:                fos.close();
35:                long end = System.currentTimeMillis();
36:                System.out.println("Elapsed Time: " 
+ (end - start) + " milliseconds.");
37:           } catch (Throwable t)
38:             {
39:                t.printStackTrace();
40:             }
41:    }
42: }
```

Listing 2.1 SlowFileCopy.java

In line 26 of Listing 2.1 we set the buffer size to be 4 KB. Obviously, given the simple space/time trade-off, we can increase the speed of this for large files by increasing the buffer size. A run of Listing 2.1 on a 27-MB file produces:

```
E:\classes\org\javapitfalls\item2>java
org.javapitfalls.item2.SlowFileCopy j2sdk-1_4_1-beta-windows-i586.exe ⮐
j2sdk-copy1.exe


Elapsed Time: 13971 milliseconds.
```

Lines 30 and 31 of Listing 2.1 are the key workhorse loop of the program where bytes are transferred from the original file to the buffer and then are written from the buffer to a second file (the copy). The `FileChannel` class includes a method called `transferTo()` that takes advantage of low-level operating system calls specifically to speed up transfers between file channels. So, the loop in 26 to 31 can be replaced by the following code:

```
FileChannel fcin = fis.getChannel();
FileChannel fcout = fos.getChannel();
fcin.transferTo(0, fcin.size(), fcout);
```

The above snippet is from the program called FastFileCopy.java, which is identical to SlowFileCopy.java except for the lines above that replace the `while` loop. A run of FastFileCopy on the same large file produces:

```
E:\classes\org\javapitfalls\item2>java
org.javapitfalls.item2.FastFileCopy j2sdk-1_4_1-beta-windows-i586.exe ⮐
j2sdk-copy2.exe

Elapsed Time: 2343 milliseconds.
```

The performance of FastFileCopy is very fast for all large files, but slightly slower for smaller files.

Little-Endian Byte Operations

A nice feature of the `NIO Buffer` class is the ability to perform reads and writes of the numeric data types using either Big Endian or Little Endian byte order. For those not familiar with the difference, for multibyte data types like short (2 bytes), integer and float (4 bytes), and long and double (8 bytes), Little Endian stores the bytes starting from the least significant byte ("littlest" number) toward the most significant. Of course, Big Endian stores bytes in the opposite direction. Processors from Motorola and Sun use Big Endian order, while Intel uses Little Endian. Prior to JDK 1.4, you would have to perform the byte-swapping yourself. I created a class called `LittleEndian-OutputStream` to do just that for a BMP Image encoder. Listing 2.2 demonstrates the byte swapping. This is necessary because the only order available for `DataOutput-Stream` is Big Endian.

```
008: class LittleEndianOutputStream extends OutputStream
009: {
010:     OutputStream os;
011:
012:     public LittleEndianOutputStream(OutputStream os)
013:     {
014:         this.os = os;
015:     }
016:
017:     public void write(int b) throws IOException
018:     {
019:         os.write(b);
020:     }
021:
022:     public void writeShort(short s) throws IOException
023:     {
024:         int is = (int) s; // promote
025:         int maskB1 = 0xff;
026:         int maskB2 = 0xff00;
027:
028:         byte [] b = new byte[2];
029:         b[0] = (byte)  (s & maskB1);
030:         b[1] = (byte) ((s & maskB2) >>> 8);
031:
032:         os.write(b);
033:     }
034:
035:     public void writeInt(int i) throws IOException
036:     {
037:         byte [] b = new byte[4];
038:         int maskB1 = 0xff;
039:         int maskB2 = 0xff00;
040:         int maskB3 = 0xff0000;
041:         int maskB4 = 0xff000000;
042:
043:         b[3] = (byte) ((i & maskB4) >>> 24);
044:         b[2] = (byte) ((i & maskB3) >>> 16);
045:         b[1] = (byte) ((i & maskB2) >>> 8);
046:         b[0] = (byte)  (i & maskB1);
047:
048:         os.write(b);
049:     }
050: }
```

Listing 2.2 LittleEndianOutputStream.java

The `LittleEndianOutputStream` was used in combination with a `DataOutputStream` to write integers and shorts in a BMP encoder. Note how we manually swap the bytes when writing an integer to the underlying `OutputStream` in the `writeInt()` method. We swap the bytes by masking the byte in the original integer (Big Endian format), shifting it down to the lowest byte position and assigning it to its new byte position. LittleEndianOutputStream is now obsolete, as Listing 2.3 shows the encoder rewritten using NIO Buffers.

```
001: /** BmpWriter3.java */
002: package org.javapitfalls.item2;
003:
004: import java.awt.*;
005: import java.awt.image.*;
006: import java.io.*;
007: import java.nio.*;
008: import java.nio.channels.*;
009:
010: public class BmpWriter3
011: {
012:     // File Header - Actual contents (14 bytes):
013:     short fileType = 0x4d42;// always "BM"
014:     int fileSize;           // size of file in bytes
015:     short reserved1 = 0;    // always 0
016:     short reserved2 = 0;    // always 0
017:     int bitmapOffset = 54;  // starting byte position of image data
018:
019:     // BMP Image Header - Actual conents (40 bytes):
020:     int size = 40;          // size of this header in bytes
021:     int width;              // image width in pixels
022:     int height;   // image height in pixels (if < 0, "top-down")
023:     short planes = 1;       // no. of color planes: always 1
024:     short bitsPerPixel = 24;// number of bits per pixel: 1, 4, 8, ⮐
or 24 (no color map)
// Some data members omitted for brevity -- code available online.
037:
038:     public void storeImage(Image img, String sFilename) throws   ⮐
IOException
039:     {
// ... getting Image width and height omitted for brevity ...
056:
057:         width = imgWidth;
058:         height = imgHeight;
059:
```

Listing 2.3 BmpWriter3.java *(continued)*

```
060:         imgPixels = new int[imgWidth * imgHeight];
061:         // pixels are stored in rows
062:         try
063:         {
064:             PixelGrabber pg = new
PixelGrabber(img,0,0,imgWidth,imgHeight,imgPixels,
065:                                                 0,imgWidth);
066:             pg.grabPixels();
067:         } catch (Exception e)
068:           {
069:             throw new IOException("Exception. Reason: "
+ e.toString());
070:           }
071:
072:         // now open the file
073:         FileOutputStream fos = new FileOutputStream(sFilename);
074:         FileChannel fc = fos.getChannel();
075:
076:         // write the "header"
077:         boolean padded=false;
078:         // first calculate the scanline size
079:         iScanLineSize = 3 * width;
080:         if (iScanLineSize % 2 != 0)
081:         {
082:             iScanLineSize++;
083:             padded = true;
084:         }
085:
086:         // now, calculate the file size
087:         fileSize = 14 + 40 + (iScanLineSize * imgHeight);
088:         sizeOfBitmap = iScanLineSize * imgHeight;
089:
090:         // create a ByteBuffer
091:         ByteBuffer bbuf = ByteBuffer.allocate(fileSize);
092:         bbuf.order(ByteOrder.LITTLE_ENDIAN);
093:         bbuf.clear();
094:
095:         // now put out file header
096:         bbuf.putShort(fileType);
097:         bbuf.putInt(fileSize);
098:         bbuf.putShort(reserved1);
099:         bbuf.putShort(reserved2);
100:         bbuf.putInt(bitmapOffset);
101:
// ... some output to buffer code omitted for brevity ...
```

Listing 2.3 *(continued)*

```
113:
114:         // put the pixels
115:         for (int i= (imgHeight - 1); i >= 0; i--)
116:         {
117:             byte pad = 0;
118:             for (int j=0; j < imgWidth; j++)
119:             {
120:                 int pixel = imgPixels[(i * width) + j];
121:                 byte alpha = (byte) ((pixel >> 24) & 0xff);
122:                 byte red   = (byte) ((pixel >> 16) & 0xff);
123:                 byte green = (byte) ((pixel >>  8) & 0xff);
124:                 byte blue  = (byte) ((pixel      ) & 0xff);
125:
126:                 // put them bgr
127:                 bbuf.put(blue);
128:                 bbuf.put(green);
129:                 bbuf.put(red);
130:             }
131:             if (padded)
132:                 bbuf.put(pad);
133:         }
134:
135:         bbuf.flip();
136:         fc.write(bbuf);
137:         fos.close();
138:     }
139:
140:     public static void main(String args[])
141:     {
// method omitted for brevity - available on Web site.
171:
172: }
173:
```

Listing 2.3 *(continued)*

The key lines of Listing 2.3 are as follows:

- At lines 64 and 65, we grab the image pixels (as integers) using the `PixelGrabber` class.
- At lines 73 and 74, we first create the `FileOutputStream` for the BMP output file and then get the `FileChannel` using the `getChannel()` method. This demonstrates the integration between the existing IO packages (`FileOutputStream`) and the new IO packages (`FileChannel`).

- At line 91, we create a `ByteBuffer` via the `static allocate()` method. It is important to note that there is also an `allocateDirect()` method, which allows you to create `DirectBuffers`. A `DirectBuffer` is a buffer allocated by the operating system to reduce the number of copies between the virtual machine and the operating system. How `DirectBuffers` are implemented differs for each operating system. Additionally, a direct buffer may be more expensive to create because of the interaction with the operating system, so they should be used for long-standing buffers.
- There are three values associated with all buffers: position, limit, and capacity. The position is the current location to read from or write to. The limit is the amount of data in the array. The capacity is the size of the underlying array.
- At line 92, we use the `order()` method to set the endian order to `LITTLE_ENDIAN`.
- At lines 96 and 97, we use `putInt()` and `putShort()` to write integers and shorts, respectively, in Little Endian order, to the byte buffer.
- At line 135, we flip the buffer from reading to writing using the `flip()` method. The `flip()` method sets the limit to the current position and the position back to 0.
- At line 136, we write the byte buffer to the underlying file.

Now we can finish our discussion of NIO by examining how the package implements non-blocking IO.

Non-Blocking Server IO

Before NIO, Java servers would create a thread to handle each incoming client connection. You would see a code snippet like this:

```
while (true)
{
   Socket s = serverSocket.accept();
   Thread handler = new Thread(new SocketHandler(s));
   handler.start();
}
```

More experienced programmers would reuse threads via a thread pool instead of instantiating a new thread each time. So, the first key new feature of NIO is the ability to manage multiple connections by way of a multiplexer class called a Selector. *Multiplexing* is a networking term associated with protocols like the User Datagram Protocol (UDP) that multiplexes communication packets to multiple processes. Figure 2.1 demonstrates multiplexing.

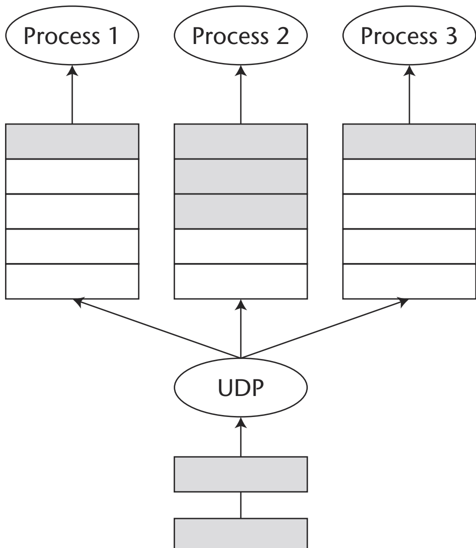


Figure 2.1 UDP multiplexing example.

So, after hearing about Selectable channels and deciding that they are a good thing, we set out to use them. We examine the JDK documentation and study the example programs like NBTimeServer.java. The "non-blocking time server" demonstrates non-blocking IO by creating a simple server to send the current time to connecting clients. For our demonstration program, we will write a server that is the start of a collaborative photo space. In that space we need to share images and annotations. Here we only scratch the surface of the application by creating a server to receive the images and annotations from clients. Listing 2.4 has the initial attempt at a non-blocking server:

```
001: /* ImageAnnotationServer1.java */
002: package org.javapitfalls.item2;
003:
004: import java.util.*;
005: import java.io.*;
006: import java.nio.*;
007: import java.net.*;
008: import java.nio.channels.*;
```

Listing 2.4 ImageAnnotationServer1.java *(continued)*

```
009:
010: public class ImageAnnotationServer1
011: {
012:      public static final int DEFAULT_IAS_PORT = 8999;
013:      boolean done;
014:
015:      public ImageAnnotationServer1() throws Exception
016:      {
017:           this(DEFAULT_IAS_PORT);
018:      }
019:
020:      public ImageAnnotationServer1(int port) throws Exception
021:      {
022:           acceptConnections(port);
023:      }
024:
025:      public void acceptConnections(int port) throws Exception
026:      {
027:           // get the ServerSocketChannel
028:           ServerSocketChannel ssc = ServerSocketChannel.open();
029:           System.out.println("Received a: " +
ssc.getClass().getName());
030:
031:           // get the ServerSocket on this channel
032:           ServerSocket ss = ssc.socket();
033:
034:           // bind to the port on the local host
035:           InetAddress address = InetAddress.getLocalHost();
036:           InetSocketAddress sockAddress = new
InetSocketAddress(address, port);
037:           ss.bind(sockAddress);
038:
039:           // set to non-blocking
040:           ssc.configureBlocking(false);
041:
042:           // create a Selector to multiplex channels on
043:           Selector theSelector = Selector.open();
044:
045:           // register this channel (for all events) with the
Selector
046:           // NOTE -- how do we know which events are OK????
047:           SelectionKey theKey = ssc.register(theSelector,
SelectionKey.OP_ACCEPT |
```

Listing 2.4 *(continued)*

```
048:                                               SelectionKey.OP_READ |
049:                                               SelectionKey.OP_CONNECT |
050:                                               SelectionKey.OP_WRITE);
051:
052:            while (theSelector.select() > 0)
053:            {
054:                 // get the ready keys
055:                 Set readyKeys = theSelector.selectedKeys();
056:                 Iterator i = readyKeys.iterator();
057:
058:                 // Walk through the ready keys collection and
process datarequests.
059:                 while (i.hasNext())
060:                 {
061:                      // get the key
062:                      SelectionKey sk = (SelectionKey)i.next();
063:
064:                      if (sk.isConnectable())
065:                      {
066:                           System.out.println("is Connectable.");
067:                      }
// ... other checks removed for brevity
083:                 }
084:            }
085:       }
086:
087:       public static void main(String [] args)
088:       {
// ... argument check removed for brevity
094:
095:            try
096:            {
097:                 int p = Integer.parseInt(args[0]);
098:                 ImageAnnotationServer1 ias1 = new
ImageAnnotationServer1(p);
099:            } catch (Throwable t)
100:              {
101:                 t.printStackTrace();
102:              }
103:       }
104: }
105:
```

Listing 2.4 *(continued)*

Let's walk through the logic in this program one step at a time. These follow the bold lines in Listing 2.4. The steps are as follows:

1. At line 28, the program opens the `ServerSocketChannel`. You do not directly instantiate a `ServerSocketChannel`. Instead, you get an instance of one from the `Service Provider Interface` classes for this channel type.
2. At line 32, the program gets the `ServerSocket` from the `ServerSocketChannel`. The connection between the original `java.net` classes (`Socket` and `ServerSocket`) and their respective channels is more intertwined than the relationship between the original IO streams and their respective channels. In this case, a `ServerSocketChannel` is not a bound connection to a port; instead, you must retrieve the `ServerSocket` and bind it to the network address and port. Without you blindly copying the example code, it is unintuitive when you must switch from channels to their IO counterparts.
3. At line 37, the program binds the local host and port to the server socket.
4. At line 40, we configure the `ServerSocketChannel` to be non-blocking. This means that a call to read or write on a socket from this channel will return immediately whether there is data available or not. For example, in blocking mode, the program would wait until there was data available to be read before returning. It is important to understand that you can use these selectable channels on both clients and servers. This is the chief benefit of non-blocking IO—your program never waits on the IO but instead is notified when it occurs. This concept of the program reacting to multiplexed data instead of waiting until it occurs is an implementation of the Reactor pattern [Schmidt 96]. Figure 2.2 is a UML diagram of the Reactor pattern.

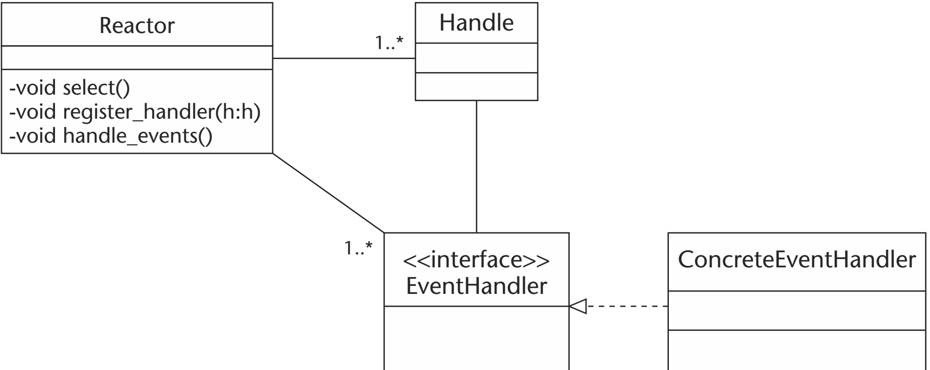


Figure 2.2 The Reactor pattern.

The Reactor pattern demultiplexes concurrent events to one or more event handlers. The key participants in the pattern are handles and a synchronous event demultiplexer. A handle represents the object of the event. In the NIO package implementation of this pattern, the handle is represented by the `SelectionKey` class. Thus, in relation to network servers, a handle represents a socket channel. The synchronous event demultiplexer blocks awaiting for events to occur on a set of handles. A common example of such a demultiplexer is the Unix `select()` system call. In the NIO package implementation of this pattern, the Selector class performs the synchronous event demultiplexing. Lastly, the Event Handler interface (and specific implementation) implements an object hook for a specific event handling. In the NIO implementation, the `SelectionKey` performs the object hook operation by allowing you to attach an object via the `attach()` method and retrieve the object via the `attachment()` method. Here is an example of attaching a `Callback` object to a key:

```
sk.attach(new Callback(sk.channel()));
```

1. At line 43, we create a `Selector` object (by calling the `Selector.open()` method) to multiplex all registered channels.
2. At line 47, we register the `SelectableChannel` with the `Selector` and specify which operations in the channel we want to be notified about. Here lies are first potential pitfall. If we do not register the correct operations for this channel object (which is provided by a Service Provider and not instantiated), an exception will be thrown. The operations that you can register for are `OP_ACCEPT`, `OP_CONNECT`, `OP_READ`, and `OP_WRITE`. In fact, in Listing 2.4, since we did not check which operations are valid on this channel with the `validOps()` method, an exception will be thrown.
3. At line 52, we call `select()` on the `Selector` to wait for operations on the registered channels.
4. At line 55, the `select()` method returns the number of `SelectionKeys` ready for processing. If the number is greater than 0, you can retrieve the set of selected `SelectionKeys`. A `SelectionKey` represents the registration and state of a `Channel` registered with a `Selector`. Once you have the `SelectionKey`, you can test its state and act accordingly.

Running Listing 2.4 produces:

```
E:\classes\org\javapitfalls\item2>java ⮐
org.javapitfalls.item2.ImageAnnotationServer1 5544
Received a: sun.nio.ch.ServerSocketChannelImpl

java.lang.IllegalArgumentException at
java.nio.channels.spi.AbstractSelectableChannel.register ⮐
(AbstractSelectableChannel.java:170)
```

The `IllegalArgumentException` is thrown because we attempted to register operations that were not valid on the `ServerSocketChannel`. The only operation we can register on that channel is `OP_ACCEPT`. Listing 2.5 registers the correct operation,

accepts the channel, and receives a file from the client. Listing 2.5 presents the changes to the `acceptConnections()` method.

```
025: public void acceptConnections(int port) throws Exception
026: {
// ... omitted code Identical to Listing 2.4
053:   SelectionKey theKey = ssc.register(theSelector,
SelectionKey.OP_ACCEPT);
054:
055:   int readyKeyCnt = 0;
056:   while ( (readyKeyCnt = theSelector.select()) > 0)
057:   {
058:      System.out.println("Have " + readyKeyCnt + " ready keys...");
059:
060:      // get the ready keys
061:      Set readyKeys = theSelector.selectedKeys();
062:      Iterator i = readyKeys.iterator();
063:
064:      // Walk through the ready keys collection and process the
requests.
065:      while (i.hasNext())
066:      {
067:         // get the key
068:         SelectionKey sk = (SelectionKey)i.next();
069:         i.remove();
070:
071:         if (sk.isAcceptable())
072:         {
073:            System.out.println("is Acceptable.");
074:            // accept it
075:
076:            // get the channel
077:            ServerSocketChannel channel = (ServerSocketChannel)
sk.channel();
078:
079:            // using method in NBTimeServer JDK example
080:            System.out.println("Accepting the connection.");
081:            Socket s = channel.accept().socket();
082:
083:            DataInputStream dis = new
DataInputStream(s.getInputStream());
084:            DataOutputStream dos = new
DataOutputStream(s.getOutputStream());
085:
086:            // Code to read file from the client ...
112:            }
113:       }
114:   }
115:}
```

Listing 2.5 Changes to `acceptConnections()` method

After working our way through the simple incorrect event registration pitfall, we can create a non-blocking server that properly accepts a socket connection. Here are the key changes highlighted in Listing 2.5:

- At line 53, we register the single operation `OP_ACCEPT` on the server socket channel.
- At line 56, we call `select()` to wait on any events on the registered channel.
- At line 69, we remove the `SelectionKey` from the set of `SelectionKey`s returned from the `select()` method. This is a potential pitfall, because if you do not remove the key, you will reprocess it. So, it is the programmer's responsibility to remove it from the set. This is especially dangerous if you have multiple threads processing the selection keys.
- At line 71, we test if the key `isAcceptable()`,which is the only operation we registered for. However, it is important to understand that once accepted, you get a channel for each incoming connection (each a separate key), which can in turn be registered with the `Selector` for other operations (reads and writes).
- At line 77, we get the registered channel (in this case the `ServerSocketChannel`) from the `SelectionKey` via the `channel()` method.
- At line 81, we call the `accept()` method to accept the socket connection and get a `SocketChannel` object. Given this object we can either process the channel (which is the approach of our simple server) or register it with the Selector like this:

```
SocketChannel sockChannel = channel.accept();
sockChannel.configureBlocking( false );
SelectionKey readKey =
             sockChannel.register( theSelector,
             SelectionKey.OP_READ|SelectionKey.OP_WRITE  );
```

A run of Listing 2.5 (ImageAnnotationServer2) accepts a single connection, receives the file, and then exits. The problem is in line 56 where the `while` loop (which follows Sun's NBTimeServer example) only continues if there are greater than 0 events returned from `select()`; however, the documentation clearly states that 0 events may be returned. Therefore to fix this pitfall, it is necessary to loop forever in the server and not assume `select()` will block until at least one event is ready, like this:

```
int readyKeyCnt = 0;
// loop forever (even if select() returns 0 ready keys)
while (true)
        {
readyKeyCnt = theSelector.select();
// ...
}
```

With the above change made, ImageAnnotationServer3.java is ready to continually accept files from clients. This pitfall has introduced you to some of the major features of the NIO package. The package has some clear benefits at the cost of some additional complexity. Readers should be ready for more changes to Java's IO packages. The most

glaring pitfall with this package is its separation from the IO package and the addition of brand-new metaphors. Having said that, most programmers will overlook that incongruity for the benefits of the new features. Overall, the performance improvements offered by NIO make up for the minor pitfalls mentioned here. All programmers should be encouraged to learn and use the NIO package.

Item 3: I Prefer Not to Use Properties

I have worked in a number of places where all development was done on an isolated network and a set of machines was used for office automation tasks like email, Web browsing, word processing, and time charging.

In this case, I really have two sets of properties that I need to handle. First, I have the set of properties that handle configuring the system in general. Examples of this would be the mail server that needs to be referenced, the network file server to point toward, and the timecard server. These are things that are clearly independent of any user and have more to do with the system than the individual user accessing the system.

Second, a multitude of user properties are required. It starts by being arranged by functional application, and then it is further organized by functional areas within the application.

Consider this properties file:

```
server=timecard.mcbrad.com
server=mail.mcbrad.com
server=ftp.mcbrad.com
```

This obviously wouldn't work, but it illustrates a simple problem in a properties file. You cannot give a common name to a particular property. Notice that naming a property "server" is remarkably appropriate in each of these cases. Furthermore, if you wanted to use a common piece of code to make a TCP/IP connection for all three apps listed, you couldn't do it without either writing custom code to parse out of three different files (a maintenance nightmare) or parsing the server subproperty.

This properties file shows a more typical example to avoid namespace collision:

```
timecard.server=timecard.mcbrad.com
mail.server=mail.mcbrad.com
ftp.server=ftp.mcbrad.com
```

Notice that these property names imply a sense of hierarchy. In fact, there is no hierarchy. There are only further qualified property names to make them more unique. However, our earlier example gave us the idea of being able to take the server subnode off of all of the primary nodes. There are no nodes since the properties file is not stored as a tree. This is where the Preferences API comes in handy. Listing 3.1 is an example of a preferences file.

```
01: <?xml version="1.0" encoding="UTF-8"?>
02: <!DOCTYPE preferences SYSTEM
'http://java.sun.com/dtd/preferences.dtd'>
03:
04: <preferences EXTERNAL_XML_VERSION="1.0">
05:
06:   <root type="user">
07:     <map />
08:     <node name="com">
09:       <map>
10:         <entry key="addr" value="8200 Greensboro Dr." />
11:         <entry key="pi" value="3.1416" />
12:         <entry key="number" value="23" />
13:       </map>
14:       <node name="mcbrad">
15:         <map />
16:         <node name="prefs">
17:           <map>
18:             <entry key="mail" value="mail" />
19:             <entry key="ftp" value="shortbus" />
20:             <entry key="timecard" value="spectator" />
21:           </map>
22:         </node>
23:       </node>
24:     </node>
25:
26:   </root>
27: </preferences>
28:
```

Listing 3.1 A preferences file

This preferences file shows the hierarchical organization of its XML format. It is very helpful when organizing multiple preferences under a particular user's settings.

Hang on, though. This just jumped from a discussion of system properties to user properties. Being able to do that in a single file is probably the best example of how we benefit from a hierarchical format. Now that we have a tree structure, not only can we separate nodes between different parts of the system, but we can also make a separation between the system and the user. Once that separation can be defined, we can make a distinction between users. This makes it easier to maintain a large number of users, all separated on the tree.

Using properties, you must store user properties within the context of the user's home directory, and then you almost always need to store those values in a file that is hard-coded into the system. This adds an additional problem with trying to ensure consistent access to these hard-coded locations. Listing 3.2 is an example of how a developer might use properties.

```
01: package org.pitfalls.prefs;
02:
03: import java.util.Properties;
04: import java.io.*;
05:
06: public class PropertiesTest {
07:
08:   private String mailServer;
09:   private String timecardServer;
10:   private String userName;
11:   private String ftpServer;
12:
13:
14:   public PropertiesTest() {
15:   }
16:
17:   [ GETTER AND SETTER METHODS FOR MEMBER VARIABLES... ]
18:
19:   public void storeProperties() {
20:
21:     Properties props = new Properties();
22:
23:     props.put("TIMECARD", getTimecardServer());
24:     props.put("MAIL", getMailServer());
25:     props.put("FTP", getFtpServer());
26:     props.put("USER", getTimecardServer());
27:
28:     try {
29:
30:       props.store(new FileOutputStream("myProps.properties"), 
"Properties");
31:
32:     } catch (IOException ex) {
33:
34:       ex.printStackTrace();
35:
36:     }
37:
38:   }
39:
```

Listing 3.2 Storing user properties

Here is the example of the properties file that is produced:

```
#Properties
#Sun Feb 24 23:16:09 EST 2002
TIMECARD=time.mcbrad.com
FTP=ftp.mcbrad.com
USER=time.mcbrad.com
MAIL=mail.mcbrad.com
```

Instead, Listing 3.3 shows the same example with preferences:

```
package org.pitfalls.prefs;

import java.util.prefs.Preferences;

public class PreferencesTest {

  private String mailServer;
  private String timecardServer;
  private String userName;
  private String ftpServer;

  public PreferencesTest() {

  }

  [ GETTER AND SETTER METHODS FOR MEMBER VARIABLES... ]

  public void storePreferences() {
    Preferences prefs = Preferences.userRoot();
    prefs.put("timecard", getTimecardServer());
    prefs.put("MAIL", getMailServer());
    prefs.put("FTP", getFtpServer());
    prefs.put("user", getTimecardServer());

  }

  public static void main(String[] args) {

    PreferencesTest myPFTest = new PreferencesTest();
    myPFTest.setFtpServer("ftp.mcbrad.com");
    myPFTest.setMailServer("mail.mcbrad.com");
    myPFTest.setTimecardServer("time.mcbrad.com");
    myPFTest.setUserName("Jennifer Richardson");

    myPFTest.storePreferences();

  }
```

Listing 3.3 Storing user preferences

Figure 3.1 shows the preferences stored, in this case, in the Windows Registry. Notice the slashes prior to each of the capital letters? This is due to the implementation on the Windows Registry, which does not support case-sensitive keys. The slashes signify a capital letter.

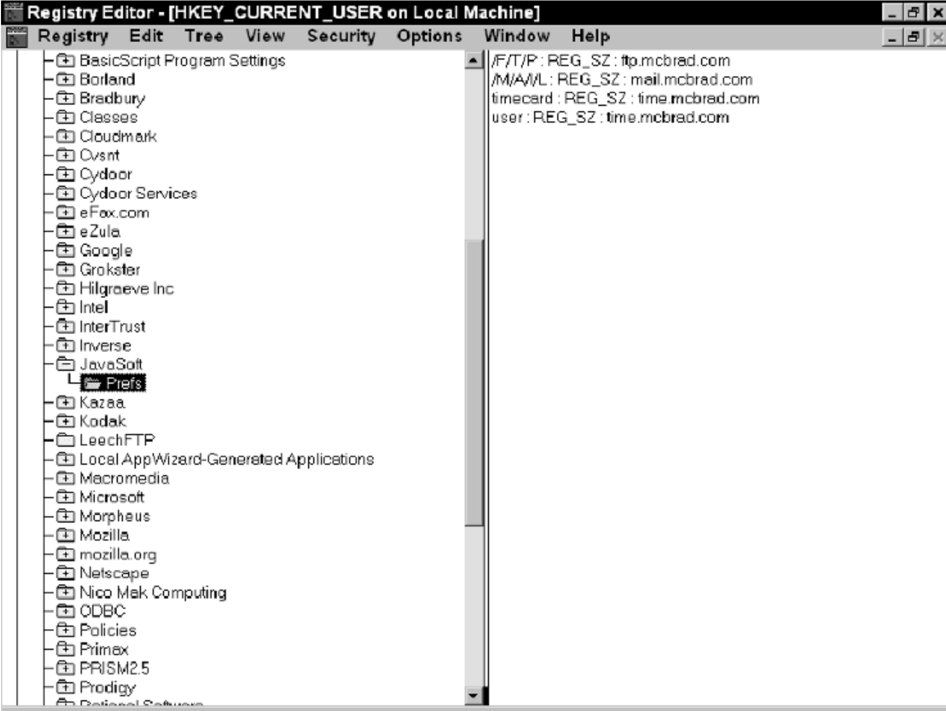


Figure 3.1 Preferences stored in the Windows Registry.

So is the hierarchical nature of preferences the reason to use them instead of properties? While that is certainly a great reason, it is not the only reason. Properties files have no standardized way of placing configuration information within the filesystem. This means that you need a configuration file to find your configuration file! Furthermore, you must have a filesystem available, so a lot of devices cannot use the `Properties` API.

What about using JNDI? JNDI is a hierarchical structure. JNDI is a solid choice for maintaining information about users, applications, and distributed objects. Two things run against JNDI, however:

- It doesn't give any indication of how the hierarchy is structured. Just because you can access the naming or directory service through JNDI doesn't give the information necessary to find the root of your specific context.
- It can seem like driving a tack with a sledgehammer. JNDI requires a directory server to be available. Often the directory server is maintained by a separate organization, which may not see value in maintaining the fact that a guy named Marshall likes to have his email messages display their text in hot pink. No matter what your application, there is likely to be something that should be maintained in a more simple fashion.

Why not have a solution that handles properties in a hierarchical fashion and is independent of the back end storage mechanism? This is what the `Preferences` API gives you.

Item 4: When Information Hiding Hides Too Much

When you are developing a framework that will be used by a large project, it is sometimes helpful to abstract the details of another API from the developers using your framework. For example, in an access control system, it may not be necessary to tell the users of your API that you are using database access control, directory server access control, or your own homegrown method of access control. Instead, you may simply hide the fact of what mechanism you are using and provide a public class (or interface) called `AccessControl`. When you write the implementation class, you will handle the mechanism of access control.

Many times, when API developers abstract the implementation of these classes, sometimes *too much* is abstracted where implementation-specific exceptions are involved. As an example, see Listing 4.1, which is an implementation of an access control mechanism with a Lightweight Directory Access Protocol (LDAP)-based directory server.

```
01: package org.javapitfals.item4;
02: import netscape.ldap.*;
03: public class AccessControl
04: {
05:     private String m_host       = null;
06:     private int    m_port       = 389;
07:     private int    m_ldapversion = 3;
08:     private LDAPConnection m_ld  = null;
09:
10:
// 1 and 2 argument constructors removed for brevity...
20:     public AccessControl(String hostname, int portnumber,
                               int ldapversion)
21:     {
22:         m_host = hostname;
23:         m_port = portnumber;
24:         m_ldapversion = ldapversion;
25:     }
26:     private void createConnection() throws LDAPException
27:     {
28:         m_ld = new LDAPConnection();
29:         m_ld.connect(m_host, m_port);
30:     }
31:     /**
32:      *  The purpose of this function is to authenticate to
33:      *  the Directory Server with credentials.
34:      *
35:      *  @param uname the user name
36:      *  @param pw the password
37:      *  @return successful authentication
38:      */
```

Listing 4.1 A bad example of abstracting details *(continued)*

```
39:     public boolean authenticate(String uname, String pw)
40:     {
41:         boolean result = false;
42:         String dn = "uid=" + uname + ",ou=People,dc=pitfalls.org";
43:         try
44:         {
45:             createConnection();
46:             m_ld.authenticate( m_ldapversion, dn, pw );
47:             result = true;
48:         }
49:         catch ( LDAPException e )
50:         {
51:             //here, result is originally set to false, so do nothing
52:         }
53:         return (result);
54:     }
55:  }
```

Listing 4.1 *(continued)*

In lines 39 through 54 of Listing 4.1, there exists a method called `authenticate()` that returns a boolean value, denoting a successful login to the access control mechanism. In line 42, the username is turned into a LDAP distinguished name, and in lines 45 and 46, the method creates a connection and attempts to authenticate the user. If an `LDAPException` is thrown, the method simply returns false.

This is a good example of how ignoring exceptions for the purpose of hiding detail can cause hours and hours of pain for developers using this code. Of course, this class compiles fine. If the infrastructure is in place for this example (network connectivity, a directory server, the correct username and password), the method will return a boolean true value, and everything will work correctly. However, if there is another problem, such as a directory server problem or network connectivity problems, it will return false. How does the implementer using this API handle the problem or know what the problem is? The original API used by this class throws an `LDAPException`, but the authenticate method in listing 4.1 simply ignores it and returns false.

What is the API developer of this class to do? First of all, a simple design change to use an interface that has the `authenticate()` method could be used along with a creational design pattern. This way, multiple implementation classes could be written (`LDAPAccessControl`, `DatabaseAccessControl`, etc.) that can realize this interface, depending on which mechanism we are using. The developer using the API would still not need to know the internals but would have to handle an exception thrown by that method, as shown in the code segment below.

```
public interface iAccessControl
{
  public boolean authenticate(String user, String passwd) throws
         AccessException;
}
```

The inclusion of a new exception brings up another possible pitfall, however. We have created a new `AccessException` class because we do not want the API user to have to handle exceptions such as LDAPException that are dependent on our hidden implementation. In the past, developers have handled this in a way shown in Listing 4.2.

```
01:   public boolean authenticate(String uname, String pw)
      throws AccessException
02:   {
03:      boolean result = false;
04:      String dn = "uid=" + uname + ",ou=People,dc=pitfalls.org";
05:         try
06:         {
07:             createConnection();
08:             m_ld.authenticate( m_ldapversion, dn, pw );
09:             result = true;
10:         }
11:         catch ( LDAPException e )
12:         {
13:             throw new AccessException(e.getMessage());
14:         }
15:         return (result);
16:     }
17:  }
```

Listing 4.2 Losing information with a new exception

On line 13 of Listing 4.2, we throw a new `AccessException` class to hide the LDAP-specific exception from the API user. Unfortunately, sometimes this complicates debugging because we lose a lot of information about the original cause of the exception, which was contained in our original `LDAPException`. For example, perhaps there was an actual bug in the LDAP API we are using. We lose a vast majority of debugging information by discarding the "causative exception," which was `LDAPException` in our example.

Prior to JDK 1.4, situations like these presented quite a few problems for debugging. Thankfully, JDK 1.4 released much-needed enhancements to allow "chained exceptions." Changes to the `java.lang.Throwable` class can be seen in Table 4.1, and the implementation of `Throwable.printStackTrace()` was also changed to show the entire "causal" chain for debugging purposes. As you can see by looking at Table 4.1, `Throwable` classes can now be associated as the "cause" for other `Throwable` classes.

Table 4.1 New Chained Exception Capabilities Added to Throwable in JDK 1.4

METHOD	DESCRIPTION
`public Throwable getCause()`	Returns the cause of this throwable or null if the cause is nonexistent or unknown. (The cause is the throwable that caused this throwable to get thrown.)
`public Throwable initCause(Throwable c)`	Initializes the cause of this throwable to the specified value. (The cause is the throwable that caused this throwable to get thrown.)
`public Throwable(Throwable cause)`	Constructs a new throwable with the specified cause.
`public Throwable(String message, Throwable cause)`	Constructs a new throwable with the specified detail message and cause.

Of course, `java.lang.Exception` and `java.lang.Error` are subclasses of `Throwable`, so now we can make minor adjustments to our code, passing in the cause of the exception to our new `AccessException` class. This is seen in Listing 4.3.

```
01:   public boolean authenticate(String uname, String pw)
      throws AccessException
02:   {
03:      boolean result = false;
04:      String dn = "uid=" + uname + ",ou=People,dc=pitfalls.org";
05:         try
06:         {
07:             createConnection();
08:             m_ld.authenticate( m_ldapversion, dn, pw );
09:             result = true;
10:         }
11:         catch ( LDAPException e )
12:         {
13:             throw new AccessException(e);
14:         }
15:         return (result);
16:     }
17:  }
```

Listing 4.3 Modifying `authenticate()`, passing causality

Listing 4.3 shows a simple way to handle our exception without losing information. Finally, Listing 4.4 shows the resulting class that replaces the listing in 4.1. As you can see in line 3 of Listing 4.4, we create a class that implements our `iAccessControl` interface, and we have modified our `authenticate()` method to throw an `AccessException`, passing the causal exception to the constructor in lines 39 to 55.

```
01: package org.javapitfals.item4;
02: import netscape.ldap.*;
03: public class LDAPAccessControl implements iAccessControl
04: {
05:     private String m_host       = null;
06:     private int    m_port       = 389;
07:     private int    m_ldapversion = 3;
08:     private LDAPConnection m_ld  = null;
09:
10:
11:     public LDAPAccessControl(String hostname)
12:     {
13:         this(hostname, 389, 3);
14:     }
15:
16:     public LDAPAccessControl(String hostname, int portnumber)
17:     {
18:         this(hostname, portnumber, 3);
19:     }
20:     public LDAPAccessControl(String hostname, int portnumber,
                               int ldapversion)
21:     {
22:         m_host = hostname;
23:         m_port = portnumber;
24:         m_ldapversion = ldapversion;
25:     }
26:     private void createConnection() throws LDAPException
27:     {
28:         m_ld = new LDAPConnection();
29:         m_ld.connect(m_host, m_port);
30:     }
31:     /**
32:      *  The purpose of this function is to authenticate to
33:      *  the Directory Server with credentials.
34:      *
35:      *  @param uname the user name
36:      *  @param pw the password
37:      *  @return successful authentication
```

Listing 4.4 The better implementation *(continued)*

```
38:      */
39:     public boolean authenticate(String uname, String pw)
40:     throws AccessException
41:     {
42:         boolean result = false;
43:         String dn = "uid=" + uname + ",ou=People,dc=pitfalls.org";
44:         try
45:         {
46:             createConnection();
47:             m_ld.authenticate( m_ldapversion, dn, pw );
48:             result = true;
49:         }
50:         catch ( LDAPException e )
51:         {
52:             throw new AccessException(e);
53:         }
54:         return (result);
55:     }
56:  }
```

Listing 4.4 *(continued)*

This pitfall showed the traps that can present themselves when you try to hide the implementation details when creating an API for use by other developers. The key points that you should keep in mind are as follows:

- Take advantage of interfaces when hiding your implementation details.
- Be wary of not handling exceptions properly. Instead of returning a value from a method (such as false or null), think about the cause of your returning these values. If there could be multiple causes, throw an exception.
- Instead of taking some information from one exception and placing it a new exception, take advantage of the new JDK 1.4 changes to Throwable and add the original exception to your new exception.

Item 5: Avoiding Granularity Pitfalls in java.util.logging

The release of J2SDK 1.4 brought us a new logging API—`java.util.logging`. For those of us who have been frustrated by our own implementations of logging over the years, this can be a powerful package that comes out of the box with the J2SE. An application can create a logger to log information, and several handlers (or objects that "log" the data) can send logging information to the outside world (over a network, to a file, to a console, etc.) Loggers and handlers can filter out information, and handlers can use Formatter objects to format the resulting output of the Handler object.

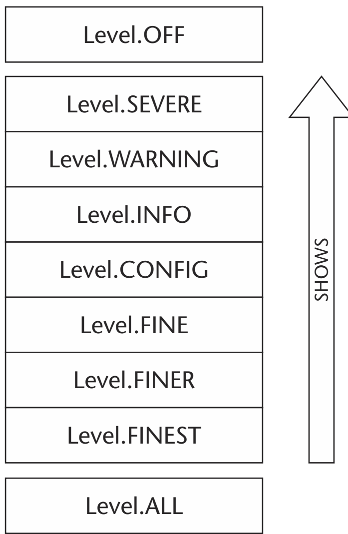


Figure 5.1 Levels of granularity.

At first glance, adding a logging solution into your application looks incredibly easy and quite intuitive. When you start changing the granularity of logging, however, there are some potential pitfalls that lie in your path. Figure 5.1 shows the levels of granularity from the class `java.util.logging.Level`, with the lowest level of granularity being `Level.FINEST`. The key point to know is that when a logger is set to show messages at a level of granularity, it will show messages labeled at that level and above. For example, a logger set to the `FINEST` level of granularity will show all messages labeled `Level.FINEST` and above in Figure 5.1. When a logger is set to show messages at `Level.INFO`, it will show messages labeled `Label.INFO`, `Label.WARNING`, and `Label.SEVERE`. Of course, as Figure 5.1 shows, the `Level` class also includes "ALL" for logging all messages and "OFF" for logging no messages.

A first attempt at using this log package, experimenting with levels of granularity, can be shown in Listing 5.1.

```
01: package org.javapitfalls.item5;
02:
03: import java.io.*;
04: import java.util.logging.*;
05:
06:  public class BadLoggerExample1
07:  {
08:    private Logger m_log = null;
09:
10:    public BadLoggerExample1(Level l)
```

Listing 5.1 BadLoggerExample 1 *(continued)*

```
11:     {
12:
13:
14:       //This creates the logger!
15:       m_log =
Logger.getLogger("org.pitfalls.BadLoggerExample1.logger");
16:
17:        m_log.setLevel(l);
18:    }
19:     /*
20:      * This tests the levels of granularity!
21:      */
22:     public void test()
23:     {
24:         System.out.println("The level for the log is: "
25:                           + m_log.getLevel());
26:         m_log.finest("This is a test for finest");
27:         m_log.finer("This is a test for finer");
28:         m_log.fine("This is a test for fine");
29:         m_log.info("This is a test for info");
30:         m_log.warning("This is a warning test");
31:         m_log.severe("This is a severe test");
32:     }
33:
34:     /*
35:      * A very simple example, where we will optionally
36:      * pass in the level of granularity to our application
37:      */
38:     public static void main(String[] args)
39:     {
40:         Level loglevel = Level.INFO;
41:
42:         if ( args.length !=0 )
43:         {
44:             if ( args[0].equals("ALL") )
45:             {
46:                 loglevel = Level.ALL;
47:             }
48:             else if ( args[0].equals("FINE") )
49:             {
50:                 loglevel = Level.FINE;
51:             }
52:             else if ( args[0].equals("FINEST") )
53:             {
54:                 loglevel = Level.FINEST;
55:             }
56:             else if ( args[0].equals("WARNING") )
57:             {
58:                 loglevel = Level.WARNING;
```

Listing 5.1 *(continued)*

```
59:             }
60:             else if ( args[0].equals("SEVERE") )
61:             {
62:                 loglevel = Level.SEVERE;
63:             }
64:
65:         }
66:         BadLoggerExample1 logex = new BadLoggerExample1(loglevel);
67:         logex.test();
68:     }
69:}
```

Listing 5.1 *(continued)*

In Listing 5.1, you can see that we create a simple logger and call a test function that tests the levels of granularity. In the `main()` method of this class, we pass it an argument pertaining to the level of granularity (`ALL`, `FINE`, `FINEST`, `WARNING`, `SEVERE`), or if there is no argument, the loglevel defaults to `INFO`. If you run this program without an argument, you will see the following printed to standard error, which is correct output:

```
The level for the log is: INFO
Feb 16, 2002 3:42:08 PM org.pitfalls.logging.BadLoggerExample1 test
INFO: This is a test for info
Feb 16, 2002 3:42:08 PM org.pitfalls.logging.BadLoggerExample1 test
WARNING: This is a warning test
Feb 16, 2002 3:42:08 PM org.pitfalls.logging.BadLoggerExample1 test
SEVERE: This is a severe test
```

Additionally, if you pass SEVERE as an argument, you will see the following correct output:

```
The level for the log is: SEVERE
Feb 16, 2002 3:42:09 PM org.pitfalls.logging.BadLoggerExample1 test
SEVERE: This is a severe test
```

However, if you run this program with the argument `FINE`, you will see the following:

```
The level for the log is: FINE
Feb 16, 2002 3:42:10 PM org.pitfalls.logging.BadLoggerExample1 test
INFO: This is a test for info
Feb 16, 2002 3:42:10 PM org.pitfalls.logging.BadLoggerExample1 test
WARNING: This is a warning test
Feb 16, 2002 3:42:10 PM org.pitfalls.logging.BadLoggerExample1 test
SEVERE: This is a severe test
```

What happened? Where are the "fine" messages? Is something wrong with the Logger? We set the level of granularity to `FINE`, but it still acts as if its level is INFO. We know that is wrong, because we printed out the level with the Logger's `getLevel()` method. Let us add a `FileHandler` to our example, so that it will write the log to a file, and see if we see the same output. Listing 5.2 shows the BadLoggerExample2, where we add a `FileHandler` to test this. On lines 20 and 21 of Listing 5.2, we create a new `FileHandler` to write to the log file log.xml, and we add that handler to our `Logger` object.

```
01: package org.javapitfalls.item5;
02:
03: import java.io.*;
04: import java.util.logging.*;
05:
06: public class BadLoggerExample2
07: {
08:    private Logger m_log = null;
09:
10:    public BadLoggerExample2(Level l)
11:    {
12:       FileHandler fh = null;
13:
14:      //This creates the logger!
15:      m_log = 
Logger.getLogger("org.pitfalls.BadLoggerExample2.logger");
16:
17:      //Try to create a FileHandler that writes it to file!
18:        try
19:        {
20:            fh = new FileHandler("log.xml");
21:            m_log.addHandler(fh);
22:        }
23:        catch ( IOException ioexc )
24:        {
25:            ioexc.printStackTrace();
26:        }
27:
28:        m_log.setLevel(l);
29:    }
30:    /*
31:     * This tests the levels of granularity!
32:     */
33:    public void test()
34:    {
35:        System.out.println("The level for the log is: "
```

Listing 5.2 BadLoggerExample2.java *(continued)*

```
36:                            + m_log.getLevel());
37:        m_log.finest("This is a test for finest");
38:        m_log.finer("This is a test for finer");
39:        m_log.fine("This is a test for fine");
40:        m_log.info("This is a test for info");
41:        m_log.warning("This is a warning test");
42:        m_log.severe("This is a severe test");
43:    }
44:
45:    /*
46:     * A very simple example, where we will optionally
47:     * pass in the level of granularity to our application
48:     */
49:    public static void main(String[] args)
50:    {
51:        Level loglevel = Level.INFO;
52:
53:        if ( args.length !=0 )
54:        {
55:            if ( args[0].equals("ALL") )
56:            {
57:                loglevel = Level.ALL;
58:            }
59:            else if ( args[0].equals("FINE") )
60:            {
61:                loglevel = Level.FINE;
62:            }
63:            else if ( args[0].equals("FINEST") )
64:            {
65:                loglevel = Level.FINEST;
66:            }
67:            else if ( args[0].equals("WARNING") )
68:            {
69:                loglevel = Level.WARNING;
70:            }
71:            else if ( args[0].equals("SEVERE") )
72:            {
73:                loglevel = Level.SEVERE;
74:            }
75:
76:        }
77:        BadLoggerExample2 logex = new BadLoggerExample2(loglevel);
78:        logex.test();
79:    }
80: }
```

Listing 5.2 *(continued)*

This time, we see the same output as before when we pass in the FINE argument, but a log file is generated, showing the XML output, shown in Listing 5.3! Now, standard error prints out the same seemingly incorrect thing as before, not showing the FINE test, and the FileHandler seems to write the correct output. What is going on? Why does the file output not match the output written to standard error?

```
<?xml version="1.0" encoding="windows-1252" standalone="no"?>
<!DOCTYPE log SYSTEM "logger.dtd">
<log>
<record>
  <date>2002-02-16T15:51:00</date>
  <millis>1013892660502</millis>
  <sequence>0</sequence>
  <logger>org.pitfalls.BadLoggerExample2.logger</logger>
  <level>FINE</level>
  <class>org.pitfalls.logging.BadLoggerExample2</class>
  <method>test</method>
  <thread>10</thread>
  <message>This is a test for fine</message>
</record>
<record>
  <date>2002-02-16T15:51:00</date>
  <millis>1013892660522</millis>
  <sequence>1</sequence>
  <level>INFO</level>
... <logger> <class>, <method> and <thread> elements same as above ...
  <message>This is a test for info</message>
</record>
<record>
  <date>2002-02-16T15:51:00</date>
  <millis>1013892660612</millis>
  <sequence>2</sequence>
  <level>WARNING</level>
  <message>This is a warning test</message>
</record>
<record>
  <date>2002-02-16T15:51:00</date>
  <millis>1013892660622</millis>
  <sequence>3</sequence>
  <level>SEVERE</level>
  <message>This is a severe test</message>
</record>
</log>
```

Listing 5.3 XML-formatted output from FileHandler

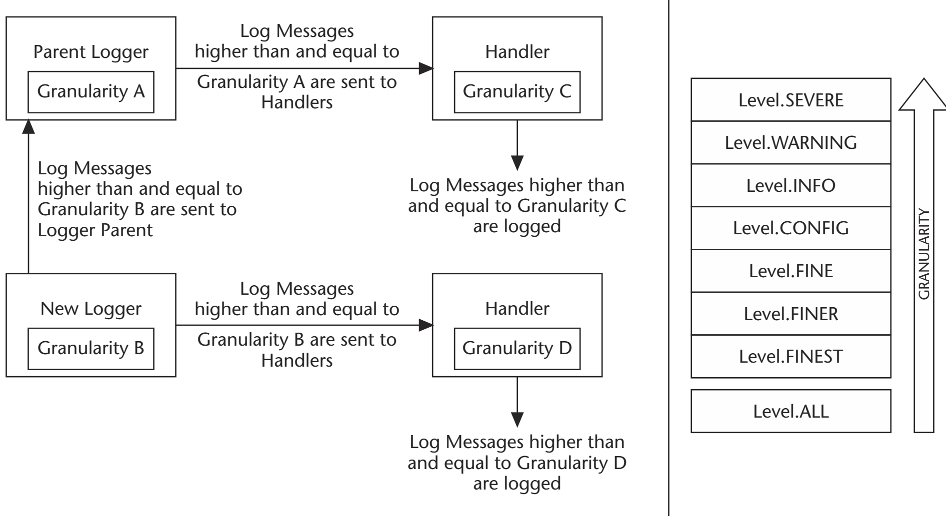


Figure 5.2 Logger/handler relationship diagram.

This behavior is quite strange. What happened? There are actually three things that we need to understand in order to understand this strange behavior:

Default Configurations of Loggers and Handlers. Loggers and handlers have default configurations, read from a preference file in your JRE's lib directory. A key thing to understand is that granularities may be set for each, using the `setLevel()` method.

Inheritance of Loggers. Another key thing to understand about the logging API is that each logger has a parent, and all data sent to a logger will go to its parent as well. The top parent is always the Root Logger, and you can create an inheritance tree descending from the Root Logger. In our initial example in Listing 5.1, we did not set a handler, but we still had output. Why? The reason is the Root Logger had a `ConsoleHandler`, whose default content level is `Level.INFO`. You can disable sending log messages to your parent's handler by calling the `setUseParentHandlers(false)` on the `Logger` class.

The Relationship between Handlers and Loggers. As we mentioned before, there are default levels of handlers. By default, all `ConsoleHandlers` log at the `Level.INFO` level, and all `FileHandlers` log at the `Level.ALL` level. The logger itself can set its level, and you can also set the level of the handlers. The key is that the level of the handler can never show a lower level of granularity than the logger itself. For example, if the logger's level is set to `Level.INFO`, the attached handlers will only see `Level.INFO` levels and above (from our diagram in Figure 5.1). In our example in Listing 5.2, we set our logger level to `Level.FINE`, and because the `FileHandler`'s level was the default level (`Level.ALL`), it only saw what the logger was able to pass through (`FINE` and below).

Confusing? We have presented this graphically in Figure 5.2 for your convenience. In our earlier example tests with BadLoggerExample1.java in Listing 5.1, everything seemed to work when we set the level to `Level.INFO` and `Level.SEVERE`, because those levels were higher than `Level.INFO`, the default level for the parent logger. However, when we set the level to `Level.FINE`, the parent's logger's handler was only passed messages higher than and equal to `Level.INFO`.

Luckily, it is very simple to set the levels for your handlers and loggers with the `setLevel()` method, and it is possible to not send messages to your logger's parent with the logger's `setUseParentsHandlers(false)` method. Listing 5.4 shows our changes to Listing 5.2, where we modify the constructor to set all of the handlers at the same level.

```
10:    public GoodLoggerExample(Level l)
11:    {
12:        FileHandler fh = null;
13:        ConsoleHandler ch = new ConsoleHandler();
14:
15:      //This creates the logger!
16:      m_log =
Logger.getLogger("org.pitfalls.GoodLoggerExample.logger");
17:      m_log.addHandler(ch);
18:      //Try to create a FileHandler that writes it to file!
19:      try
20:      {
21:          fh = new FileHandler("log.xml");
22:          m_log.addHandler(fh);
23:      }
24:      catch ( IOException ioexc )
25:      {
26:          ioexc.printStackTrace();
27:      }
28:
29:      /* This will set everything to the same level! */
30:      m_log.setLevel(l);
31:      m_log.setUseParentHandlers(false);
32:      fh.setLevel(l);
33:      ch.setLevel(l);
34:    }
```

Listing 5.4 Better constructor—GoodLoggerExample.java

In Listing 5.4, we want to create our own `ConsoleHandler` to log user-friendly messages to standard error, and we will continue to have our own `FileHandler` to write XML messages to file. On line 13, we instantiate a new `ConsoleHandler`, and on line 17, we add it to our logger. Finally, lines 29 to 33 fix the rest: we set the level of the logger (and every handler) to the same level, and we set our logger to not send messages to our parent's handler. The result of this program is the expected output.

Understanding the relationships of loggers and handlers and levels of granularity, shown in Figure 5.2, is very important. Our examples in this pitfall were created to give you an understanding of these relationships, and often, you will not want the same levels of granularity for every handler. Most of the time, logs to the console will be "user-friendly" warning messages, and log files may be debugging information for programmers and system administrators. The fact that you can create your own handlers, and set your logging levels at runtime, is very powerful. Studying this pitfall should lead to a better understanding of the `java.util.logging` package.

Item 6: When Implementations of Standard APIs Collide

Over the past three years, a plethora of XML processing software packages were released. First there were proprietary APIs, and then as the SAX and DOM standards evolved, vendors developed Java toolkits for developers to use. Developers began writing programs with these APIs, and as the standards evolved, you changed your code. As different toolkits used different levels of DOM compliancy, and as you integrated more software into your baseline, you had to go through your entire codebase, trying to determine which DOM API was needed for which application, and trying to determine if two modules using different levels of DOM compliancy could run together in the same application, since both used the `org.w3c.dom.*` classes, but some were based on different underlying implementations and different levels of DOM compliancy. If you started using one implementation of the DOM and later switched to another implementation of the DOM, sometimes your code needed to be tweaked. If you were one of the early adopters of processing XML in Java, you know our pain.

The release of JDK 1.4 brought a little complexity to the issue, because classes such as `org.w3c.dom.*` and `org.xml.sax.*` are now in the standard runtime environment, and thus read before your classpath which is chock full of your favorite XML JAR files. Because of this, if you use a later implementation of SAX or DOM than the classes in the JDK, or if you use a method that is implementation-specific, you may run into problems. A good example can be seen in Listing 6.1, ScheduleSwitcher.java. Here we simply parse an XML "schedule file," with an example schedule shown in Listing 6.2.

```
001: package org.javapitfalls.item6;
002: import org.w3c.dom.*;
003: import javax.xml.parsers.*;
004: :
005: /**
006:  * A simple class that demonstrates different functionality
007: * between DOM implementations
008: */
009: public class ScheduleSwitcher
010: {
011:   /* The DOM document loaded in memory */
```

Listing 6.1 ScheduleSwitcher.java *(continued)*

```
012:   Document m_doc = null;
013:   public ScheduleSwitcher(String filename)
014:   {
015:     /* Parse a file */
016:     try
017:     {
018:       DocumentBuilderFactory factory =
019:       DocumentBuilderFactory.newInstance();
020:       DocumentBuilder builder = factory.newDocumentBuilder();
021:       m_doc = builder.parse(filename);
022:     }
023:     catch ( Exception e )
024:     {
025:       System.err.println("Error processing " +
026:                          filename + "." +
027:                          "Stack trace follows:");
028:       e.printStackTrace();
029:     }
030:   }
031:
032:   /*
033:    * A very simple method that switches values of
034:    * days in an XML document and prints it out.
035:    *
036:    * @param a the value for one day
037:    * @param b the value for another day
038:    * @param keep a boolean value, designating if you
039:    *        want to keep this version of the DOM tree.
040:    */
041:   public void showSwitchedDays(String a , String b,
042:                                boolean keep)
043:   {
044:     Document newdoc = null;
045:
046:     if ( m_doc == null )
047:     {
048:       System.out.println("Error - no document.. ");
049:       return;
050:     }
051:
052:     /**
053:      * If the keep variable was set, do the manipulation
054:      * to the instance variable m_doc.. Otherwise, just
055:      * do the manipulation of the copy of the tree.
056:      *
057:      */
058:     if ( !keep )
059:       newdoc = (Document)m_doc.cloneNode(true);
060:     else
```

Listing 6.1 *(continued)*

```
061:        newdoc = m_doc;
062:
063:      /* Use the DOM API to switch days */
064:      NodeList nl = newdoc.getElementsByTagName("day");
065:      int len = nl.getLength();
066:      for ( int i = 0; i < len; i++ )
067:      {
068:
069:        Element e = (Element)nl.item(i);
070:
071:        if ( e.getAttribute("name").equals(a) )
072:        {
073:          e.setAttribute("name",b);
074:        } else if ( e.getAttribute("name").equals(b) )
075:        {
076:          e.setAttribute("name", a);
077:        }
078:      }
079:
080:      System.out.println(
081:        newdoc.getDocumentElement().toString()
082:        );
083:
084:    }
085:
086:    /* Print out the DOM Tree */
087:    public void showDays()
088:    {
089:      System.out.println(
090:        m_doc.getDocumentElement().toString()
091:        );
092:    }
093:
094:    public static void main(String[] args)
095:    {
096:      if ( args.length < 1 )
097:      {
098:        System.err.println("Usage: Argument must be the " +
099:                           "filename of an XML file");
100:        System.exit(-1);
101:      }
102:      String filename = args[0];
103:
104:      ScheduleSwitcher switcher = new ScheduleSwitcher(filename);
105:
106:      System.out.println("\nIf you switched " +
107:                         " the Wed & Thurs meetings ");
108:      System.out.println("this is what it would " +
109:                         "look like:\n*******");
```

Listing 6.1 *(continued)*

```
110:
111:     switcher.showSwitchedDays("wednesday", "thursday", false);
112:
113:     System.out.println("\nHere is the current " +
114:                        " schedule:\n********");
115:     switcher.showDays();
116:
117:   }
118: }
```

Listing 6.1 *(continued)*

In our simple example, we have written a method, `showSwitchedDays()`, on line 41 of Listing 6.1, that switches the name attribute on the `<day/>` tag of an XML file by manipulating the DOM and prints the resulting XML file to standard out. If the boolean value `keep` is true, then the DOM manipulation affects the DOM tree instance variable. If the Boolean value `keep` is false, then the method simply prints out a copy of the switched schedule but keeps the original DOM in memory. On lines 106-109 of Listing 6.1, we tell the class to print the meetings as if Wednesdays and Thursdays were switched, but to not permanently alter the schedule. Listing 6.3 shows the output of this program using the XML file in Listing 6.2, when we ran this with JDK 1.3.1, and implementations of the DOM in the crimson.jar file.

```
<?xml version="1.0"?>
<meetings>
 <day name="wednesday">
  <meeting desc="romans">
    <member name="ben"/><member name="billy"/><member name="chuck"/>
    <member name="dan"/><member name="keith"/><member name="kevin"/>
    <member name="matt d."/><member name="matt v."/><member
name="rich"/>
    <member name="todd"/>
  </meeting>
 </day>
 <day name="thursday">
  <meeting desc="all">
    <member name="avery"/><member name="catherine"/><member
name="dawn"/>
    <member name="doverly"/><member name="gwen"/><member name="heidi"/>
    <member name="holly"/><member name="jenny"/><member name="patrice"/>
    <member name="sandy b."/><member name="sandy c."/><member
name="sarah"/>
    <member name="shelly"/><member name="suzanne"/>
  </meeting>
 </day>
</meetings>
```

Listing 6.2 XML schedule file (schedule.xml)

```
C:\pitfalls\week5>c:\jdk1.3.1\bin\java -classpath .;crimson.jar;jaxp.jar
ScheduleSwitcher schedule.xml

If you switched the Wed & Thurs meetings,
this is what it would look like:
*******
<meetings>
 <day name="thursday">
  <meeting desc="romans">
   <member name="ben" /><member name="billy" /><member name="chuck" />
   <member name="dan" /><member name="keith" /><member name="kevin" />
   <member name="matt d." /><member name="matt v."                    ⮐
/><member name="rich" />
   <member name="todd" />
  </meeting>
 </day>
 <day name="wednesday">
  <meeting desc="all">
   <member name="avery"/><member name="catherine"                     ⮐
/><member name="dawn" />
   <member name="doverly"/><member name="gwen" /><member name="heidi" />
   <member name="holly"/><member name="jenny"/><member name="patrice"/>
   <member name="sandy b."/><member name="sandy c."/><member
name="sarah" />
   <member name="shelly" /><member name="suzanne" />
  </meeting>
 </day>
</meetings>

Here is the current schedule:
********
<meetings>
 <day name="wednesday">
  <meeting desc="romans">
    <member name="ben" /><member name="billy" /><member name="chuck" />
    <member name="dan" /><member name="keith" /><member name="kevin" />
    <member name="matt d." /><member name="matt v." /><member          ⮐
name="rich" />
    <member name="todd" />
  </meeting>
 </day>
 <day name="thursday">
  <meeting desc="all">
   <member name="avery" /><member name="catherine"                    ⮐
/><member name="dawn" />
   <member name="doverly" /><member name="gwen"                       ⮐
/><member name="heidi" />
   <member name="holly" /><member name="jenny"                        ⮐
/><member name="patrice" />
```

Listing 6.3 Output with JDK 1.3 and DOM implementation *(continued)*

```
   <member name="sandy b."/><member name="sandy c."/><member
name="sarah" />
   <member name="shelly" /><member name="suzanne" />
  </meeting>
 </day>
</meetings>
```

Listing 6.3 *(continued)*

Listing 6.3 has the expected output, and it ran correctly. As you can see, the "Wednesday" and "Thursday" meetings were switched, but when the final schedule was printed, the original DOM tree was printed out. However, when we upgraded our application to JDK 1.4, we ran into problems. The result of running this in JDK 1.4 is shown in Listing 6.4. As you can see, what ran fine in an earlier version of the JDK now throws an exception. When we call `cloneNode()` on `org.w3c.dom.Document` on line 27, we get a "HIERARCHY_REQUEST_ERR:" message! Looking into the documentation on the `cloneNode()` method, it is inherited from `org.w3c.dom.Node`, and the documentation says that cloning `Document`, `DocumentType`, `Entity`, and `Notation` nodes is implementation dependent.

```
C:\pitfalls\week5>java -classpath . ScheduleSwitcher schedule.xml

If you switched the Wed & Thurs meetings,
this is what it would look like:
*******
org.apache.crimson.tree.DomEx: HIERARCHY_REQUEST_ERR: This node isn't
allowed there. at
org.apache.crimson.tree.XmlDocument.changeNodeOwner(XmlDocument.java:115
6)
at
org.apache.crimson.tree.XmlDocument.changeNodeOwner(XmlDocument.java:117
7)
at org.apache.crimson.tree.XmlDocument.cloneNode(XmlDocument.java:1101)
          at ScheduleSwitcher.showSwitchedDays(ScheduleSwitcher.java:27)
                      at ScheduleSwitcher.main(ScheduleSwitcher.java:68)
Exception in thread "main"
```

Listing 6.4 Output with JDK 1.4, with built-in DOM

So what should we do in a situation like this? Since there are many implementations of DOM and SAX APIs, we are bound to run into these problems. Added to this dilemma is the fact that standards and new implementations evolve quickly, and we may want to use newer implementations of these standards before they are integrated

into the JDK. Luckily, with the release of JDK 1.4, there is the Endorsed Standards Override Mechanism (ESOM), which is used to replace implementations of these standard classes. Following is the complete list of the packages that can be overriden:

```
javax.rmi.CORBA
org.omg.CORBA,org.omg.CORBA.DynAnyPackage, org.omg.CORBA.ORBPackage,
org.omg.CORBA.portable,org.omg.CORBA.TypeCodePackage,org.omg.CORBA_2_3,
org.omg.CORBA_2_3.portable,org.omg.CosNaming,
org.omg.CosNaming.NamingContextExtPackage,
org.omg.CosNaming.NamingContextPackage, org.omg.Dynamic,
org.omg.DynamicAny,org.omg.DynamicAny.DynAnyFactoryPackage,
org.omg.DynamicAny.DynAnyPackage, org.omg.IOP ,
org.omg.IOP.CodecFactoryPackage, org.omg.IOP.CodecPackage,
org.omg.Messaging,
org.omg.PortableInterceptor,
org.omg.PortableInterceptor.ORBInitInfoPackage, org.omg.PortableServer,
org.omg.PortableServer.CurrentPackage,
org.omg.PortableServer.POAManagerPackage,
org.omg.PortableServer.POAPackage, org.omg.PortableServer.portable,
org.omg.PortableServer.ServantLocatorPackage, org.omg.SendingContext,
org.omg.stub.java.rmi, org.w3c.dom, org.xml.sax, org.xml.sax.ext,
org.xml.sax.helpers
```

As you can see, these include standard classes from the Object Management Group (OMG) as well as the W3C. To take advantage of the ESOM, follow these steps:

1. Create a directory called endorsed in the jre/lib directory.
2. Copy your JAR file implementations of your APIs into that directory.
3. Run your application.

In our simple example, we followed these steps, and the application worked perfectly. It is sometimes difficult to stay away from methods that are implementation-specific—as `Document.cloneNode()` was in this example. Remembering how to override the standards with the use of the ESOM will make your life a lot easier!

Item 7: My Assertions Are Not Gratuitous!

In *Java Pitfalls*, we finished the book by discussing the emerging JSR concerning a Java Assertion Facility. At the time, we suggested a mechanism that would allow an assertion-like facility. As of JDK 1.4, the language now includes an assertion facility.

Many developers do not understand why or how to use assertions, and this has caused them to go widely unused. Some developers use them improperly, causing them to be ineffective or counterproductive to the development effort.

How to Use Assertions

When writing code, we make assumptions all of the time. Assertions are meant to capture those assumptions in the code. By capturing those assumptions, you move closer to implementing a significant concept in quality software: design by contract.

Design by Contract holds there are three concepts embodied in any piece of code: preconditions, postconditions, and invariants. Preconditions involve the assumptions about the state prior to the execution of code. Postconditions involve the assumptions about the state after the execution of the code. Invariants capture the underlying assumptions upon which the execution occurs.

Consider the following method signature:

```
public String lookupPlayer (int number)
```

In this method, I provide a service to look up a player by his number. In explaining this signature, I have already specified a set of assumptions. The first assumption is players do not have numbers greater than 99 (they don't fit on a jersey well, and most teams don't have more than 100 players on their roster). Also, players do not have negative numbers either. Therefore, the preconditions are that we have a number provided that is less than or equal to zero and less than or equal to 99. An invariant is to assume that the list of players and numbers actually exists—that is, it has been initialized and is properly available (note that this action is subject to its own set of assumptions). The assumption on the postcondition is that there will be an actual `String` value returned from the method (and not a null).

How many times have you seen a problem with the code being debugged by a set of `System.out.println()` statements? This is because, essentially, the developer is going back in and creating visibility to check against assumptions that are made in writing the code. This can be relatively painless and easy for the developer who wrote the code—at the time he or she wrote the code. However, this becomes quite painful for others who may not understand what is going on inside the code.

Assertions can be thought of as automatic code checks. By writing assertions, you capture your assumptions, so that an error will be thrown at the incorrect assumption when the code is running. Even if a bug occurs that does not throw an `AssertionError`, they still enable the developer to eliminate many possible assumptions as causes of error.

There are really four conditions you should meet in order to determine that you do not need to use assertions:

- You wrote and maintain all of the code in question.
- You will always maintain every piece of that code.
- You have analyzed to perfection all of the different scenarios that your code may experience through some method like a formal code proof.
- You like and can afford to operate without a net.

With the exception of the last condition, if you have done all of these, you have already realized that you need some assertionlike capability. You probably have already littered your code with `System.out.println()` or `log()` statements trying to give yourself places to start like that demonstrated in Listing 7.1.

```
01: package org.javapitfalls.item7;
02:
03: public class AssertionExample {
04:
05: public AssertionExample() { }
06:
07: private void printLeague (String abbreviation) {
09:  if (abbreviation.equals("AL")) {
10:
11:          System.out.println("I am in the American League!");
12:
13:  } else if (abbreviation.equals("NL")) {
14:
15:          System.out.println("I am in the National League!");
16:
17:  } else {
19:          // I should never get here...
20:          assert false;
22:  }
23: }
24:
25: public static void main (String [] args) {
27:  String al = "AL";
28:  String nl = "NL";
29:  String il = "IL";
31:  AssertionExample myExample = new AssertionExample();
33:  myExample.printLeague(al);
34:  myExample.printLeague(il);
35:  myExample.printLeague(nl);
37: }
39: }
```

Listing 7.1 AssertionExample.java

So, we compile this code and the following happens:

```
C:\pitfallsBook\#7>javac AssertionExample.java
AssertionExample.java:20: warning: as of release 1.4, assert is a
keyword, and may not be used as an identifier
                assert false ;
                ^
AssertionExample.java:20: not a statement
                assert false ;
```

```
                ^
AssertionExample.java:20: ';' expected
                assert false ;
2 errors
1 warning
```

What happened? I thought assertions were supported in JDK 1.4! I am using JDK 1.4. Well, as it turns out, for backward compatibility purposes, you have to specify a flag in order to compile source with assertions in it.

So, you would want to compile it like this instead:

```
javac -source 1.4 AssertionExample.java
```

Now that I have successfully compiled my example, here is what I do to run it:

```
C:\pitfallsBook\#7>java -cp . org.javapitfalls.item7.AssertionExample
I am in the American League!
I am in the National League!
```

Notice that it didn't throw an `AssertionError` when I passed in the "IL" abbreviation. This is because I didn't enable assertions. The requirement to enable assertions is so that your code will not have any performance defect by having the assertions within the code when you choose not to use them. When assertions are not switched on, they are effectively the same as empty statements.

```
C:\pitfallsBook\#7>java -ea -cp . ⮐
org.javapitfalls.item7.AssertionExample
I am in the American League!
Exception in thread "main" java.lang.AssertionError
        at org.javapitfalls.item7.AssertionExample.printLeague(Unknown ⮐
Source)
        at org.javapitfalls.item7.AssertionExample.main(Unknown Source)
```

Now the assertion is thrown, but it is not very descriptive as to which assertion was the cause of the problem. In this simple example, this is not a problem; it is still very easy to tell which assertion caused the problem. However, with many assertions (as is good practice), we need to better distinguish between assertions. So if we change line 20 to read

```
assert false : "What baseball league are you playing in?";
```

this will give us a more descriptive response:

```
C:\pitfallsBook\#7>java -ea -cp . ⮐
org.javapitfalls.item7.AssertionExample
I am in the American League!
```

```
Exception in thread "main" java.lang.AssertionError: What baseball
league are you playing in?
        at org.javapitfalls.item7.AssertionExample.printLeague(Unknown
Source)
        at org.javapitfalls.item7.AssertionExample.main(Unknown Source)
```

An important thing to note is that `printLeague()` is declared private. While the example was meant to show assertions to mark unreachable conditions, it was declared private in order to avoid confusion about an important point: *You should never use assertions to do precondition checks for public methods*. When assertions are turned off, you will not receive the benefit of any precondition checking. Furthermore, the `AssertionError` is a relatively simple methodology for returning errors, and more sophisticated mechanisms like typed exceptions would serve the cause better.

In Listing 7.2, we show how to use assertions to check postconditions.

```
01: package org.javapitfalls.item7;
02:
03: public class AnotherAssertionExample {
05: private double taxRate;
06:
07: public AnotherAssertionExample(double tax) {
09:  taxRate = tax;
11: }
12:
13: private double returnMoney (double salary) {
15:  double originalSalary = salary;
17:  if (salary > 10000) salary = salary * (1 - taxRate);
19:  if (salary > 25000) salary = salary * (1 - taxRate);
21:  if (salary > 50000) salary = salary * (1 - taxRate);
23:  assert salary > 0 : "They can't take more than you have?";
25:  assert salary <= originalSalary : "You can't come out ahead!";
27:  return salary;
29: }
30:
31: public static void main (String [] args) {
33:  AnotherAssertionExample myExample = new
AnotherAssertionExample(.3);
34:  System.out.println("Tax Rate of 30%\n");
35:  System.out.println("Salary of 5000:"+myExample.returnMoney(5000));
36:  System.out.println("Salary of
24000:"+myExample.returnMoney(24000));
37:  System.out.println("Salary of
35000:"+myExample.returnMoney(35000));
38:  System.out.println("Salary of
75000:"+myExample.returnMoney(75000));
39: //      System.out.println("Salary of
75000:"+myExample.returnMoney(-75000));
40:
```

Listing 7.2 AnotherAssertionExample.java *(continued)*

```
41:   myExample = new AnotherAssertionExample(-.3);
42:   System.out.println("\n\nTax Rate of -30%\n");
43:   System.out.println("Salary of 5000:"+myExample.returnMoney(5000));
44:   System.out.println("Salary of
24000:"+myExample.returnMoney(24000));
45:   System.out.println("Salary of
35000:"+myExample.returnMoney(35000));
47: }
49: }
```

Listing 7.2 *(continued)*

This shows how to check postconditions with an assertion. Here is the output from this example:

```
C:\pitfallsBook\#7>java -ea -cp .
org.javapitfalls.item7.AnotherAssertionExample

Tax Rate of 30%
Salary of 5000:5000.0
Salary of 24000:16800.0
Salary of 35000:24500.0
Salary of 75000:36750.0

Tax Rate of -30%

Salary of 5000:5000.0
Exception in thread "main" java.lang.AssertionError: You can't come out
ahead!
        at
org.javapitfalls.item7.AnotherAssertionExample.returnMoney(Unknown Source)
        at org.javapitfalls.item7.AnotherAssertionExample.main(Unknown
Source)
```

You can see that there are two assertions about the postconditions. First, we assert that you cannot return with less than zero money, and then we assert that you cannot return with more money than you started. There are examples of how to break both assertions: a negative salary (commented out) and a negative tax rate. The first example is commented out, because execution stops after the assertion error, and we wanted to demonstrate that the second assertion would also work.

Now that we have covered the assertion basics, we will go over some other interesting things about assertions. First, you can enable and disable assertions as desired by specifying them in the switches to the JVM. Here are some more examples:

- Enable all assertions:

  ```
  java -ea org.javapitfalls.item7.MyClass
  ```

- Enable system assertions only:

  ```
  java -esa org.javapitfalls.item7.MyClass
  ```

- Enable all assertions in the `org.javapitfalls` package and its sub-packages:

  ```
  java -ea:org.javapitfalls org.javapitfalls.item7.MyClass
  ```

- Enable all assertions in the `org.javapitfalls` package and its sub-packages, but disable the ones in `AnotherAssertionExample`:

  ```
  java -ea:org.javapitfalls -da:
  org.javapitfalls.item7.AnotherAssertionExample
  org.javapitfalls.item7.MyClass
  ```

Also, there are situations where you want to require that assertions be enabled in your class. An example would be if you had some safety-sensitive class that should operate only if the assertions are true (in addition to normal control checking). Listing 7.3 shows our previous example with assertions always on.

```
01: package org.javapitfalls.item7;
02:
03: public class AnotherAssertionExample {
04:
05: static {
06:  boolean assertions = false;
07:  assert assertions = true;
08:
09:  if (assertions==false)
10:         throw new RuntimeException("You must enable assertions ⮐
to use this class.");
11: }
12:
13: private double taxRate;
14: // [...] remaining code Identical to listing 7.2
```

Listing 7.3 AnotherAssertionExample.java (modified)

So, if you run this example without assertions enabled, you receive the following message:

```
C:\pitfallsBook\#7>java -cp . ⮐
org.javapitfalls.item7.AnotherAssertionExample
Exception in thread "main" java.lang.ExceptionInInitializerError
Caused by: java.lang.RuntimeException: You must enable assertions to use
this class.
  at org.javapitfalls.item7.AnotherAssertionExample.<clinit>(Unknown
Source)
```

In closing, there are a few rules to follow in dealing with assertions:

- DO use assertions to test postconditions on methods.
- DO use assertions to test places where you believe control flow should not execute.
- DO NOT use assertions to test preconditions on public methods.
- DO use assertions to test preconditions on helper methods.
- DO NOT use assertions that affect the normal operation of the code.

Item 8: The Wrong Way to Search a DOM[2]

All well-formed XML files have a tree structure. For example, Listing 8.1 can be represented by a tree with two ADDRESS nodes:

```
01: <?xml version="1.0"?>
02: <!DOCTYPE ADDRESS_BOOK SYSTEM "abml.dtd">
03: <ADDRESS_BOOK>
04:  <ADDRESS>
05:         <NAME>Joe Jones </NAME>
06:         <STREET>4332 Sunny Hill Road </STREET>
07:         <CITY>Fairfax</CITY>
08:         <STATE>VA</STATE>
09:         <ZIP>21220</ZIP>
10:  </ADDRESS>
11:  <ADDRESS>
12:         <NAME>Sterling Software </NAME>
13:         <STREET> 7900 Sudley Road</STREET>
14:         <STREET> Suite 500</STREET>
15:         <CITY>Manassas</CITY>
16:         <STATE>VA </STATE>
17:         <ZIP>20109 </ZIP>
18:  </ADDRESS>
19: </ADDRESS_BOOK>
```

Listing 8.1 myaddresses.xml

The pitfall is assuming the DOM tree will look exactly like your mental picture of the XML document. Let's say we have a task to find the first NAME element of the first ADDRESS. By looking at Listing 8.1, you may say that the third node in the DOM (line 05) is the one we want. Listing 8.2 attempts to find the node in that way.

[2] This pitfall was first printed by *JavaWorld* (www.javaworld.com) in the article, "An API's looks can be deceiving", June 2001, (http://www.javaworld.com/javaworld/jw-06-2001/jw-0622-traps.html?) and is reprinted here with permission. The pitfall has been updated from reader feedback.

```
01: package org.javapitfalls.item8;
02:
03: import javax.xml.parsers.*;
04: import java.io.*;
05: import org.w3c.dom.*;
06:
07: public class BadDomLookup
08: {
09:     public static void main(String args[])
10:     {
11:         try
12:         {
13:             if (args.length < 1)
14:             {
15:                 System.out.println("USAGE: " +
16:              "org.javapitfalls.item8.BadDomLookup xmlfile");
17:                 System.exit(1);
18:             }
19:
20:             DocumentBuilderFactory dbf =
21:                      DocumentBuilderFactory.newInstance();
22:             DocumentBuilder db = dbf.newDocumentBuilder();
23:             Document doc = db.parse(new File(args[0]));
24:
25:             // get first Name of first Address
26:             NodeList nl = doc.getElementsByTagName("ADDRESS");
27:             int count = nl.getLength();
28:            System.out.println("# of \"ADDRESS\" elements: " + count);
29:
30:             if (count > 0)
31:             {
32:                 Node n = nl.item(0);
33:                 System.out.println("This node name is: " +
n.getNodeName());
34:                 // get the NAME node of this ADDRESS node
35:                 Node nameNode = n.getFirstChild();
36:                 System.out.println("This node name is: "
37:                                     + nameNode.getNodeName());
38:             }
39:         } catch (Throwable t)
40:           {
41:             t.printStackTrace();
42:           }
43:     }
44: }
45:
```

Listing 8.2 BadDomLookup.java

The simple program, BadDomLookup, uses the Java API for XML Processing (JAXP) to parse the DOM (this example was tested with both Xerces and Sun's default JAXP parser). After we get the W3C Document object, we retrieve a `NodeList` of `ADDRESS` elements (line 26) and then look to get the first `NAME` element by accessing the first child under `ADDRESS` (line 35).

Upon executing Listing 8.2, we get

```
e:\classes\org\javapitfalls\>java org.javapitfalls ... BadDomLookup
myaddresses.xml
# of "ADDRESS" elements: 2
This node name is: ADDRESS
This node name is: #text
```

The result clearly shows that the program fails to accomplish its task. Instead of an `ADDRESS` node, we get a text node. What happened? Unfortunately, the complexity of the DOM implementation is different from our simple conceptual model. The primary difference is that the DOM tree includes text nodes for what is called "ignorable whitespace," which is the whitespace (like a return) between tags. In our example, there is a text node between the `ADDRESS` and the first `NAME` element. The W3C XML specification states, "An XML processor must always pass all characters in a document that are not markup through to the application. A validating XML processor must also inform the application which of these characters constitute white space appearing in element content."[3] To visualize these whitespace nodes, Figure 8.1 displays all the DOM nodes in myaddresses.xml in a JTree.

There are three solutions to this problem, and our rewrite of the program demonstrates two of them. Listing 8.3, GoodDomLookup.java, fixes the problem demonstrated above in two ways.

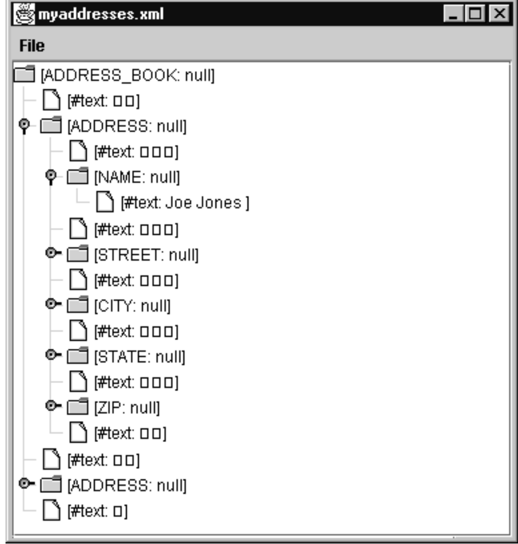


Figure 8.1 Display of all DOM nodes in myaddresses.xml.

[3] *Extensible Markup Language (XML) 1.0* (Second Edition). W3C recommendation; October 6, 2000; http://www.w3.org/TR/REC-xml.

```
001: package org.javapitfalls.item8;
002:
003: import javax.xml.parsers.*;
004: import java.io.*;
005: import org.w3c.dom.*;
006:
007: class DomUtil
008: {
009:     public static boolean isBlank(String buf)
010:     {
011:         if (buf == null)
012:             return false;
013:
014:         int len = buf.length();
015:         for (int i=0; i < len; i++)
016:         {
017:             char c = buf.charAt(i);
018:             if (!Character.isWhitespace(c))
019:                 return false;
020:         }
021:
022:         return true;
023:     }
024:
025:     public static void normalizeDocument(Node n)
026:     {
027:         if (!n.hasChildNodes())
028:             return;
029:
030:         NodeList nl = n.getChildNodes();
031:         for (int i = 0; i < nl.getLength(); i++)
032:         {
033:             Node cn = nl.item(i);
034:             if (cn.getNodeType() == Node.TEXT_NODE &&
035:                 isBlank(cn.getNodeValue()))
036:             {
037:                 n.removeChild(cn);
038:                 i--;
039:             }
040:             else
041:                 normalizeDocument(cn);
042:         }
043:     }
044:
045:     public static Element getFirstChildElement(Element elem)
046:     {
047:         if (!elem.hasChildNodes())
048:             return null;
```

Listing 8.3 GoodDomLookup.java *(continued)*

```
049:
050:         for (Node cn = elem.getFirstChild(); cn != null;
051:              cn = cn.getNextSibling())
052:         {
053:             if (cn.getNodeType() == Node.ELEMENT_NODE)
054:                 return (Element) cn;
055:         }
056:
057:         return null;
058:     }
059: }
060:
061: public class GoodDomLookup
062: {
063:     public static void main(String args[])
064:     {
065:         try
066:         {
// ... command line check omitted for brevity ...
073:
074:             DocumentBuilderFactory dbf =
075:                         DocumentBuilderFactory.newInstance();
076:             DocumentBuilder db = dbf.newDocumentBuilder();
077:             Document doc = db.parse(new File(args[0]));
078:
079:             // get first Name of first Address
080:             System.out.println("Method #1: Skip Ignorable White
space...");
081:             NodeList nl = doc.getElementsByTagName("ADDRESS");
082:             int count = nl.getLength();
083:             System.out.println("# of \"ADDRESS\" elements: " + count);
084:
085:             if (count > 0)
086:             {
087:                 Node n = nl.item(0);
088:                 System.out.println("This node name is: " +
n.getNodeName());
089:                 // get the NAME node of this ADDRESS node
090:                 Node nameNode =
DomUtil.getFirstChildElement((Element)n);
091:                 System.out.println("This node name is: " +
092:                                     nameNode.getNodeName());
093:             }
094:
095:             // get first Name of first Address
096:             System.out.println("Method #2: Normalize document...");
097:             DomUtil.normalizeDocument(doc.getDocumentElement());
098:             // Below is exact code in BadDomLookup
```

Listing 8.3 *(continued)*

```
099:            nl = doc.getElementsByTagName("ADDRESS");
100:            count = nl.getLength();
101:            System.out.println("# of \"ADDRESS\" elements: " +
count);
102:
103:            if (count > 0)
104:            {
105:                Node n = nl.item(0);
106:                System.out.println("This node name is: " +
107:                                    n.getNodeName());
108:                // get the NAME node of this ADDRESS node
109:                Node nameNode = n.getFirstChild();
110:                System.out.println("This node name is: " +
111:                                    nameNode.getNodeName());
112:            }
113:
114:        } catch (Throwable t)
115:          {
116:            t.printStackTrace();
117:          }
118:    }
119: }
120:
```

Listing 8.3 *(continued)*

The key class in GoodDomLookup is the DomUtil class that has three methods. Those three methods solve the DOM lookup problem in two ways. The first method is to retrieve the first child element (and not the first node) when performing a lookup. The implementation of the getFirstChildElement() method will skip any intermediate nodes that are not of type ELEMENT_NODE. The second approach to the problem is to eliminate all "blank" text nodes from the document. While both solutions will work, the second approach may remove some whitespace not considered ignorable.

A run of GoodDomLookup.java gives us the following:

```
e:\classes\org\javapitfalls >java org.javapitfalls.item8.GoodDomLookup
myaddresses.xml
Method #1: Skip Ignorable White space...
# of "ADDRESS" elements: 2
This node name is: ADDRESS
This node name is: NAME
Method #2: Normalize document...
# of "ADDRESS" elements: 2
This node name is: ADDRESS
This node name is: NAME
```

A better way to access nodes in a DOM tree is to use an XPath expression. XPath is a W3C standard for accessing nodes in a DOM tree. Standard API methods for evaluating XPath expressions are part of DOM Level 3. Currently, JAXP supports only DOM Level 2. To demonstrate how easy accessing nodes is via XPath, Listing 8.4 uses the DOM4J open source library (which includes XPath support) to perform the same task as GoodDomLookup.java.

```
01: package org.javapitfalls.item8;
02: 
03: import javax.xml.parsers.*;
04: import java.io.*;
05: import org.w3c.dom.*;
06: import org.dom4j.*;
07: import org.dom4j.io.*;
08: 
09: public class XpathLookup
10: {
11:     public static void main(String args[])
12:     {
13:         try
14:         {
15:             if (args.length < 1)
16:             {
17:                 System.out.println("USAGE: " +
18:              "org.javapitfalls.item8.BadDomLookup xmlfile");
19:                 System.exit(1);
20:             }
21: 
22:             DocumentBuilderFactory dbf =
23:                      DocumentBuilderFactory.newInstance();
24:             DocumentBuilder db = dbf.newDocumentBuilder();
25:             org.w3c.dom.Document doc = db.parse(new File(args[0]));
26: 
27:             DOMReader dr = new DOMReader();
28:             org.dom4j.Document xpDoc = dr.read(doc);
29:             org.dom4j.Node node = xpDoc.selectSingleNode(
30:                                   "/ADDRESS_BOOK/ADDRESS[1]/NAME");
31:             System.out.println("Node name : " + node.getName());
32:             System.out.println("Node value: " + node.getText());
33:         } catch (Exception e)
34:           {
35:             e.printStackTrace();
36:           }
37:     }
38: }
39: 
```

Listing 8.4 XpathLookup.java

A run of XpathLookup.java on myaddresses.xml produces the following output:

```
E:\classes\org\javapitfalls>javaorg.javapitfalls.item8.XpathLookup ⮐
myaddresses.xml

Node name : NAME
Node value: Joe Jones
```

The XpathLookup.java program uses the `selectSingleNode()` method in the DOM4J API with an XPath expression as its argument. The XPath recommendation can be viewed at http://www.w3.org/TR/xpath. It is important to understand that evaluation of XPath expressions will be part of the org.w3c.dom API when DOM Level 3 is implemented by JAXP. In conclusion, when searching a DOM, remember to handle whitespace nodes, or better, use XPath to search the DOM, since its robust expression syntax allows very fine-grained access to one or more nodes.

Item 9: The Saving-a-DOM Dilemma

One of the motivations for JAXP was to standardize the creation of a DOM. A DOM can be created via parsing an existing file or instantiating and inserting the nodes individually. JAXP abstracts the parsing operation via a set of interfaces so that different XML parsers can be easily used. Figure 9.1 shows a simple lifecycle of a DOM.

Figure 9.1 focuses on three states of a DOM: New, Modified, and Persisted. The New state can be reached either by instantiating a DOM object via the new keyword or by loading an XML file from disk. This "loading" operation action invokes a parser to parse the XML file. An edit, insert, or delete action moves the DOM to the modified

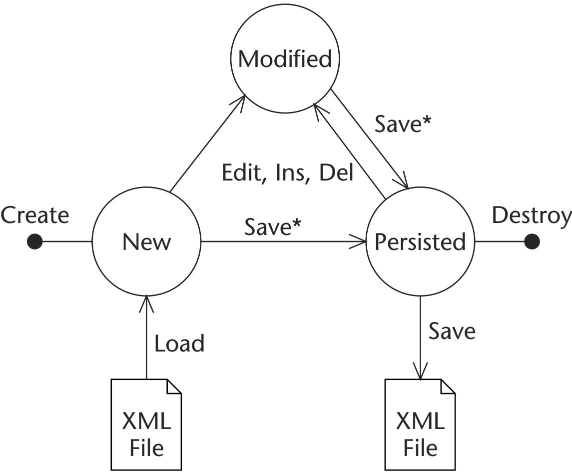


Figure 9.1 DOM lifecycle.

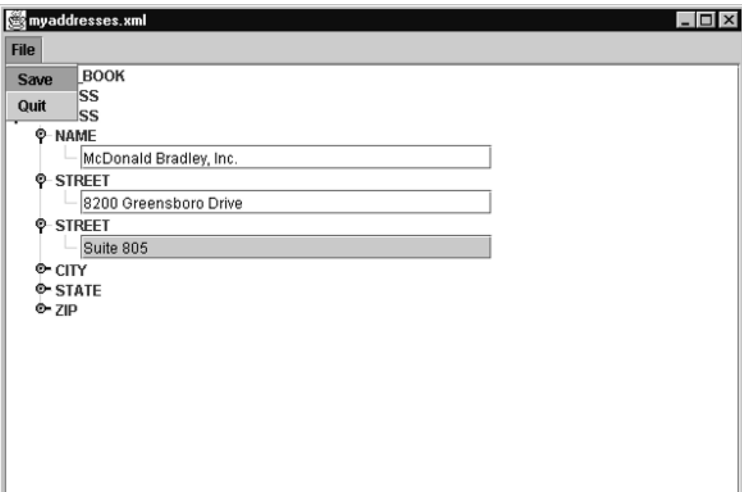


Figure 9.2 DomEditor saving a DOM.

state. The Save action transitions the DOM to the Persisted state. The asterisk by the save operation indicates that Save is implemented by saving the DOM to an XML file (the "save" operation without the asterisk). An enhancement to the DomViewer program demonstrated in Item 8 is to allow editing of the DOM nodes and then save the modified DOM out to a file (persist it). Figure 9.2 is a screen shot of the DomEditor program that implements that functionality.

When implementing the save operation in the DomEditor, we run into a dilemma on how to implement saving the DOM. Unfortunately, we have too many ways to save a DOM, and each one is different. The current situation is that saving a DOM depends on which DOM implementation you choose. The dilemma is picking an implementation that will not become obsolete with the next version of the parser. In this item, we will demonstrate three approaches to saving a DOM. Each listing will perform the same functionality: load a document from a file and save the DOM to the specified output file. The toughest part of this dilemma is the nonstandard and nonintuitive way prescribed by Sun Microsystems and implemented in JAXP to perform the save. The JAXP method for saving a DOM is to use a default XSLT transform that copies all the nodes of the source DOM (called a DOMSource) into an output stream (called a StreamResult). Listing 9.1 demonstrates saving an XML document via JAXP.

```
01: package org.javapitfalls.item9;
02:
03: import javax.xml.parsers.*;
```

Listing 9.1 JaxpSave.java

```
04: import javax.xml.transform.*;
05: import javax.xml.transform.dom.*;
06: import javax.xml.transform.stream.*;
07: import java.io.*;
08: import org.w3c.dom.*;
09:
10: class JaxpSave
11: {
12:     public static void main(String args[])
13:     {
14:         try
15:         {
// ... command-line check omitted for brevity ...
22:
23:             // load the document
24:             DocumentBuilderFactory dbf =
25:                         DocumentBuilderFactory.newInstance();
26:             DocumentBuilder db = dbf.newDocumentBuilder();
27:             Document doc = db.parse(new File(args[0]));
28:             String systemValue = doc.getDoctype().getSystemId();
29:
30:             // save to output file
31:             File f = new File(args[1]);
32:             FileWriter fw = new FileWriter(f);
33:
34:             /* write method USED To be in Sun's XmlDocument class.
35:                The XmlDocument class preceded JAXP. */
36:
37:             // Currently only way to do this is via a transform
38:             TransformerFactory tff = 
 TransformerFactory.newInstance();
39:             // Default transform is a copy
40:             Transformer tf = tff.newTransformer();
41:             tf.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM, 
systemValue);
42:             DOMSource ds = new DOMSource(doc.getDocumentElement());
43:             StreamResult sr = new StreamResult(fw);
44:             tf.transform(ds, sr);
45:             fw.close();
46:         } catch (Throwable t)
47:           {
48:             t.printStackTrace();
49:           }
50:     }
51: }
```

Listing 9.1 *(continued)*

While JAXP is the "official" API distributed with Sun's JDK, using a transform to save a DOM is not recommended. It is nonintuitive and not close to the proposed W3C DOM Level 3 standard discussed next. The method to perform the save via JAXP uses the XSLT classes in JAXP. Specifically, a TransformerFactory is created, which in turn is used to create a transformer. A transform requires a source and a result. There are various different types of sources and results. Line 40 of Listing 9.1 is the key to understanding the source, since it creates the default transformer. This line is key because normally an XSLT transform is performed via an XSLT script; however, since we are only interested in a node-for-node copy, we don't need a script, which is provided by the no-argument `newTransformer()` method. This way the "default" transformer provides the copy functionality. There is another `newTransformer(Source s)` that accepts an XSLT source script. To sum up, once we have a transformer, a source, and a result, the transform operation will perform the copy and thus serialize the DOM. One extra step is necessary if we want to preserve the Document Type declaration from the source document. Line 41 sets an output property on the transformer to add a Document Type declaration to the result document, which we set to be the same value as the source document.

Why has JAXP not defined the standard for saving a DOM? Simply because the DOM itself is not a Java standard but a W3C standard. JAXP currently implements DOM Level 2. The W3C is developing an extension to the DOM called DOM Level 3 to enhance and standardize the use of the DOM in multiple areas. Table 9.1 details the various parts of the DOM Level 3 specification.

Table 9.1 DOM Level 3 Specifications

DOM LEVEL 3 SPECIFICATIONS	DESCRIPTION
DOM Level 3 Core	Base set of interfaces describing the document object model. Enhanced in this third version.
DOM Level 3 XPath Specification	A set of interfaces and methods to access a DOM via XPATH expressions.
DOM Level 3 Abstract Schemas and Load and Save Specification	This specification defines two sub-specificatons: the Abstract Schemas specification and the Load and Save specification. The Abstract Schemas specification represents abstract schemas (DTDs and schemas) in memory. The Load and Save specification specifies the parsing and saving of DOMs.
DOM Level 3 Events Specification	This specification defines an event generation, propagation, and handling model for DOM events. It builds on the DOM Level 2 event model.
DOM Level 3 Views and Formatting	This specification defines interfaces to represent a calculated view (presentation) of a DOM. It builds on the DOM Level 2 View model.

Table 9.2 DOM Level 3 Load and Save Interfaces

W3C DOM LEVEL 3 LS INTERFACES	DESCRIPTION
`DOMImplementationLS`	A `DOMImplementation` interface that provides factory methods for creating the `DOMWriter`, `DOMBuilder`, and `DOMInputSource` objects.
`DOMBuilder`	A parser interface.
`DOMInputSource`	An interface that encapsulates information about the document to be loaded.
`DOMEntityResolver`	An interface to provide a method for applications to redirect references to external entities.
`DOMBuilderFilter`	An interface to allow element nodes to be modified or removed as they are encountered during parsing.
`DOMWriter`	An interface for serializing DOM Documents.
`DocumentLS`	An extended document interface with built-in load and save methods.
`ParseErrorEvent`	Event fired if there is an error in parsing.

The part of the DOM specification that solves our dilemma is the specification to Load and Save a DOM. Table 9.2 details the interfaces defined in the specification. It is important to note that JAXP will even change its method for bootstrapping parsers, since its current method is slightly different than the DOM Level 3 load interfaces (specifically, DOMBuilder versus DocumentBuilder).

The Xerces parser implements the DOM Level 3 specification. Listing 9.2 uses the Xerces parser to demonstrate both the loading and saving of a DOM via the DOM Level 3 standard.

```
01: package org.javapitfalls.item9;
02:
03: import org.apache.xerces.jaxp.*;
04: import org.apache.xerces.dom3.ls.*;
05: import org.apache.xerces.dom.DOMImplementationImpl;
06: import org.apache.xerces.dom3.ls.DOMImplementationLS;
```

Listing 9.2 XercesSave.java *(continued)*

```
07:
08: import javax.xml.parsers.*;
09: import java.io.*;
10: import org.w3c.dom.*;
11:
12: class XercesSave
13: {
14:     public static void main(String args[])
15:     {
16:         try
17:         {
// ... command line check omitted for brevity ...
24:
25:             // Xerces 2 implements DOM Level 3
26:             // get DOM implementation
27:             DOMImplementationLS domImpl =
28:              (DOMImplementationLS)
DOMImplementationImpl.getDOMImplementation();
29:
30:             // Create a DOM Level 3 - DOMBuilder
31:             DOMBuilder db =
32:                 domImpl.createDOMBuilder(
DOMImplementationLS.MODE_SYNCHRONOUS);
33:             DOMInputSource dis = domImpl.createDOMInputSource();
34:             dis.setByteStream(new FileInputStream(args[0]));
35:             Document doc = db.parse(dis);
36:
37:             // save to output file
38:             FileOutputStream fos = new FileOutputStream(args[1]);
39:
40:             // create a DOM Writer
41:             DOMWriter writer = domImpl.createDOMWriter();
42:             writer.writeNode(fos, doc);
43:
44:             fos.close();
45:         } catch (Throwable t)
46:           {
47:             t.printStackTrace();
48:           }
49:     }
50: }
51:
```

Listing 9.2 *(continued)*

XercesSave.java uses the "LS" (which stands for Load and Save) version of the DOMImplemenation to create the DOMBuilder (line 31) and DOMWriter (line 41)

objects. Once created, the DOMBuilder creates the DOM via the `parse()` method, and the DOMWriter saves the DOM via the `writeNode()` method. The `writeNode()` method can write either all or part of a DOM.

One final implementation worth mentioning is the load and save operations in the Java Document Object Model (JDOM). JDOM is a reimplementation of DOM for Java and is optimized for seamless integration into the Java platform. JDOM stresses ease of use for Java developers, whereas the W3C DOM is designed to be language-neutral (specified in CORBA IDL) and then provides bindings to specific languages. Of all the examples in this item, the JDOM implementation is the simplest; however, its functionality often lags behind the W3C DOM; its `Document` class is not a subclass of the `W3C Document` class and is thus incompatible with third-party software that expects a W3C DOM as an argument (although conversion is provided). Listing 9.3 demonstrates the load and save operations in JDOM.

```
01: package org.javapitfalls.item9;
02:
03: import org.jdom.*;
04: import org.jdom.input.*;
05: import org.jdom.output.*;
06: import java.io.*;
07:
08: class JdomSave
09: {
10:     public static void main(String args[])
11:     {
12:         try
13:         {
// ... command line check omitted for brevity ...
20:
21:             // load the document
22:             DOMBuilder db = new DOMBuilder();
23:             Document doc = db.build(new File(args[0]));
24:
25:             // save to output file
26:             FileOutputStream fos = new FileOutputStream(args[1]);
27:             XMLOutputter xout = new XMLOutputter();
28:             xout.output(doc, fos);
29:             fos.close();
30:         } catch (Throwable t)
31:           {
32:             t.printStackTrace();
33:           }
34:     }
35: }
36:
```

Listing 9.3 JdomSave.java

Like the W3C Load and Save specification, JDOM also uses a `DOMBuilder` class (but bootstraps it differently) and then builds an `org.jdom.Document` (line 23). It is important to note that a `JDOM Document` is NOT a `W3C Document`. To save the `JDOM Document`, an `XMLOutputter` class is instantiated that can `output()` (line 28) a document.

All three implementations of the program (JaxpSave, XercesSave, and JdomSave) produce nearly identical outputs, and thus it is not necessary to list them here. In conclusion, at this time of rapid evolution of the DOM, the safest bet is to align your code to the W3C standards and implementations that follow them. Thus, to save a DOM, the Xerces implementation is currently the best choice.

Item 10: Mouse Button Portability

Unfortunately for cross-platform computing, all computer mice are not created equal. There are one-button mice, two-button mice, three-button mice, and two-button-with-mouse-wheel mice. Like the AWT, to cover all of these options, Java initially took a least common denominator (LCD) approach that supported receiving a single mouse event and using modifiers and modifier keys to differentiate between the different types. A second problem Java programmers encounter regarding the mouse is that the Java platform evolves its mouse support with each major release, adding support for new features (like mouse wheels) and new convenience methods. Table 10.1 lists the interfaces and methods for capturing mouse events.

Table 10.1 Mouse Event Interfaces

INTERFACE	METHOD	DESCRIPTION
`MouseListener`	`mouseClicked`	Invoked when a mouse is clicked on a component
	`mousePressed`	Invoked when a mouse is pressed
	`mouseReleased`	Invoked when a mouse is released
	`mouseEntered`	Invoked when a mouse enters a component's bounds
	`mouseExited`	Invoked when a mouse exits a component's bounds
`MouseMotionListener`	`mouseDragged`	Invoked when a mouse button is pressed on a component and then dragged
	`mouseMoved`	Invoked when a mouse is moved around a component
`MouseWheelListener`	`mouseWheelMoved`	Invoked when the mouse wheel is rotated

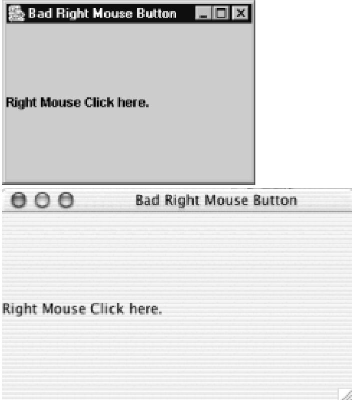


Figure 10.1 BadRightMouseButton on Windows (top) and Mac OS X (bottom).

A common problem is an application that needs to process clicks from a right mouse button across platforms. For example, say we had an application that captured right mouse clicks to both display a context-sensitive popup menu and change the mode of a tool from select, on the left mouse click, to select and execute for a right mouse click. Figure 10.1 displays the user interface of a simple application to capture these two activities of a right mouse click. Figure 10.1 shows the application run on both Windows NT and Mac OSX, since our Java application supports both operating systems.

Listing 10.1 displays the source code for BadRightMouseButton.java.

```
01: /* BadRightMouseButton.java */
02: import java.awt.event.*;
03: import java.awt.*;
04: import javax.swing.*;
05:
06: public class BadRightMouseButton extends JFrame implements
MouseListener
07: {
08:     public BadRightMouseButton()
09:     {
10:         super("Bad Right Mouse Button");
11:
12:         JLabel l = new JLabel("Right Mouse Click here.");
13:         getContentPane().add("Center", l);
14:
15:         addMouseListener(this);
16:
17:         setSize(400,200);
18:         setLocation(100,100);
19:         setVisible(true);
20:         addWindowListener(new WindowAdapter()
21:                             {
22:                              public void windowClosing(WindowEvent evt)
```

Listing 10.1 BadRightMouseButton.java *(continued)*

```
23:                              {
24:                                  System.exit(1);
25:                              }
26:                            });
27:
28:     }
29:
30:  public static void main(String [] args)
31:  {
32:          try
33:          {
34:              BadRightMouseButton win = new BadRightMouseButton();
35:
36:          } catch (Throwable t)
37:            {
38:                  t.printStackTrace();
39:            }
40:  }
41:
42:     public void mouseClicked(MouseEvent e)
43:     {
44:         int modifiers = e.getModifiers();
45:         if ((modifiers & InputEvent.BUTTON1_MASK) ==
InputEvent.BUTTON1_MASK)
46:             System.out.println("Button 1 clicked.");
47:
48:         if ((modifiers & InputEvent.BUTTON2_MASK) ==
InputEvent.BUTTON2_MASK)
49:             System.out.println("Button 2 clicked.");
50:
51:         if ((modifiers & InputEvent.BUTTON3_MASK) ==
InputEvent.BUTTON3_MASK) )
52:             System.out.println("Button 3 clicked.");
53:
54:         // modifier keys
55:         System.out.println("isControlDown? " + e.isControlDown());
56:         System.out.println("isMetaDown? " + e.isMetaDown());
57:         System.out.println("isAltDown? " + e.isAltDown());
58:         System.out.println("isShiftDown? " + e.isShiftDown());
59:         System.out.println("isAltGraphDown? " + e.isAltGraphDown());
60:
61:         /* 1.4 methods
62:         int buttonNumber = e.getButton();
63:         System.out.println("Button # is : " + buttonNumber);
64:
65:         int mods = e.getModifiersEx();
66:         System.out.println("Modifiers: " +
InputEvent.getModifiersExText(mods));
67:         */
```

Listing 10.1 *(continued)*

```
68:
69:         // is this a Popup Trigger?
70:         System.out.println("In mouseClicked(), isPopupTrigger? " + ⮐
e.isPopupTrigger());
71:     }
72:
73:     public void mousePressed(MouseEvent e)
74:     { }
75:     public void mouseReleased(MouseEvent e)
76:     { }
77:     public void mouseEntered(MouseEvent e)
78:     { }
79:     public void mouseExited(MouseEvent e)
80:     { }
81: }
```

Listing 10.1 *(continued)*

JDK 1.4 has added some new methods as demonstrated (but commented out) in lines 62 and 65. These are additional convenience methods that enable you to eliminate the need for ANDing the modifier integer with the constant flags (lines 45, 48, and 51).

Here is a run of BadRightMouseButton on Windows with a two-button mouse with a mouse wheel. When the program was executed, the right mouse button was clicked, followed by the left mouse button and then the mouse wheel. This produced the following `println` statements:

```
>>>java BadRightMouseButton  (on Windows NT with 2 button mouse with
mousewheel)
Button 3 clicked.
isControlDown? false
isMetaDown? true
isAltDown? false
isShiftDown? false
isAltGraphDown? false
In mouseClicked(), isPopupTrigger? false

Button 1 clicked.
isControlDown? false
isMetaDown? false
isAltDown? false
isShiftDown? false
isAltGraphDown? false
In mouseClicked(), isPopupTrigger? false

Button 2 clicked.
isControlDown? false
isMetaDown? false
isAltDown? true
```

```
isShiftDown? false
isAltGraphDown? false
In mouseClicked(), isPopupTrigger? false
```

Here is a run of BadRightMouseButton.java on Mac OSX with a single-button mouse. When the program was executed, the single mouse button was clicked (which maps to button 1), then the Ctrl key was held and the mouse button clicked, then the special "apple" key was held and the mouse button clicked.

```
>>>java BadRightMouseButton  (on MacOSX with a single mouse button)
Button 1 clicked.
isControlDown? false
isMetaDown? false
isAltDown? false
isShiftDown? false
isAltGraphDown? false
In mouseClicked(), isPopupTrigger? false

Button 2 clicked.
isControlDown? true
isMetaDown? false
isAltDown? true
isShiftDown? false
isAltGraphDown? false
In mouseClicked(), isPopupTrigger? false

Button 3 clicked.
isControlDown? false
isMetaDown? true
isAltDown? false
isShiftDown? false
isAltGraphDown? false
In mouseClicked(), isPopupTrigger? false
```

There are two problems you should notice when examining the results from both operating systems besides the fact that they are not consistent. First, there is no clear indication of a right mouse button click even though we know that the Windows mouse clearly has a right and left mouse button. Instead of "sides" of the mouse, we have the ability in the `InputEvent` class (which is the parent class of `MouseEvent`) to check which button was clicked: button 1, 2, or 3. Unfortunately, there is no way to correlate a button with a side for a cross-platform application. The second problem is that the call to `isPopupTrigger()` has always returned false when we know that a right-button mouse click is the trigger on Windows for a popup and the Ctrl-mouse click combination is the trigger on the Mac. Listing 10.2, GoodRightMouseButton.java, solves both of these problems.

```
01: /* GoodRightMouseButton.java */
02: import java.awt.event.*;
03: import java.awt.*;
04: import javax.swing.*;
05:
06: public class GoodRightMouseButton extends JFrame implements
MouseListener
07: {
08:     public GoodRightMouseButton()
09:     {
            // ... constructor identical to Listing #10.1
28:     }
29:
30:   public static void main(String [] args)
31:   {
32:         try
33:         {
34:             GoodRightMouseButton win = new GoodRightMouseButton();
35:
36:         } catch (Throwable t)
37:           {
38:                 t.printStackTrace();
39:           }
40:   }
41:
42:     public void mouseClicked(MouseEvent e)
43:     {
// ... getModifiers() code Identical to listing 10.1 ...
68:
69:         // is this a Popup Trigger?
70:         System.out.println("In mouseClicked(), isPopupTrigger? " +
e.isPopupTrigger());
71:
72:         // Use SwingUtilities to disambiguate
73:         boolean lb = SwingUtilities.isLeftMouseButton(e);
74:         boolean mb = SwingUtilities.isMiddleMouseButton(e);
75:         boolean rb = SwingUtilities.isRightMouseButton(e);
76:
77:         System.out.println("Left button? " + lb);
78:         System.out.println("Middle button? " + mb);
79:         System.out.println("Right button? " + rb);
80:     }
81:
82:     public void mousePressed(MouseEvent e)
```

Listing 10.2 GoodRightMouseButton.java *(continued)*

```
83:     {
84:         // is this a Popup Trigger?
85:         System.out.println("In mousePressed(), isPopupTrigger? " + ⮐
e.isPopupTrigger());
86:     }
87:     public void mouseReleased(MouseEvent e)
88:     {
89:         // is this a Popup Trigger?
90:         System.out.println("In mouseReleased(), isPopupTrigger? " + ⮐
e.isPopupTrigger());
91:     }
92:
93:     public void mouseEntered(MouseEvent e)
94:     { }
95:     public void mouseExited(MouseEvent e)
96:     { }
97: }
98:
```

Listing 10.2 *(continued)*

Here is a run of GoodRightMouseButton on Windows. When executing the program, the right mouse button was clicked, followed by the left mouse button and then the mouse wheel. This produced the following `println` statements:

```
>>>java GoodRightMouseButton  (on Windows NT with 2 button mouse with
mousewheel)
In mousePressed(), isPopupTrigger? false
In mouseReleased(), isPopupTrigger? true
Button 3 clicked.
isControlDown? false
isMetaDown? true
isAltDown? false
isShiftDown? false
isAltGraphDown? false
In mouseClicked(), isPopupTrigger? false
Left button? false
Middle button? false
Right button? true

In mousePressed(), isPopupTrigger? false
In mouseReleased(), isPopupTrigger? false
Button 1 clicked.
isControlDown? false
isMetaDown? false
isAltDown? false
isShiftDown? false
isAltGraphDown? false
```

```
In mouseClicked(), isPopupTrigger? false
Left button? true
Middle button? false
Right button? false

In mousePressed(), isPopupTrigger? false
In mouseReleased(), isPopupTrigger? false
Button 2 clicked.
isControlDown? false
isMetaDown? false
isAltDown? true
isShiftDown? false
isAltGraphDown? false
In mouseClicked(), isPopupTrigger? false
Left button? false
Middle button? true
Right button? false
```

Here is a run of GoodRightMouseButton.java on Mac OSX. When the program was executed, the single mouse button was clicked, then the Ctrl key was held and the mouse button clicked, then the special "apple" key was held and the mouse button clicked.

```
>>>java GoodRightMouseButton  (on MacOSX with a single mouse button)
In mousePressed(), isPopupTrigger? false
In mouseReleased(), isPopupTrigger? false
Button 1 clicked.
isControlDown? false
isMetaDown? false
isAltDown? false
isShiftDown? false
isAltGraphDown? false
In mouseClicked(), isPopupTrigger? false
Left button? true
Middle button? false
Right button? false

In mousePressed(), isPopupTrigger? true
In mouseReleased(), isPopupTrigger? false
Button 2 clicked.
isControlDown? true
isMetaDown? false
isAltDown? true
isShiftDown? false
isAltGraphDown? false
In mouseClicked(), isPopupTrigger? false
Left button? false
Middle button? true
Right button? false

In mousePressed(), isPopupTrigger? false
In mouseReleased(), isPopupTrigger? false
```

```
Button 3 clicked.
isControlDown? false
isMetaDown? true
isAltDown? false
isShiftDown? false
isAltGraphDown? false
In mouseClicked(), isPopupTrigger? false
Left button? false
Middle button? false
Right button? true
```

The new results show that we can determine which mouse button (left, right, or middle) was clicked and whether that click is the popup trigger event. The solution has two parts: first, the `SwingUtilities` class contains a set of methods that allow you to test a mouse event to determine which side of the mouse was clicked. It is slightly nonintuitive to have these separated from the other test methods in `InputEvent` or `MouseEvent`; however, that could change in a future release. Second, you should notice how you have to test for the `popupTrigger` in the `mousePressed()` and `mouseReleased()` to accurately determine the trigger event. It is interesting to note that the Ctrl-mouse combination on the Macintosh is considered the middle mouse button and not the right mouse button. It would be better if the popup trigger was consistent on Windows and the Mac (both being considered a "right click").

In conclusion, using the `SwingUtilities` class and understanding when to call `isPopupTrigger()` allows us to better process mouse events in cross-platform applications.

Item 11: Apache Ant and Lifecycle Management

The software lifecycle process describes the life of a software product from its conception to its implementation and deployment. An important aspect of this process is the need to impose consistency and structure on all lifecycle activities that can guide actions through development to deployment. This practice is often compared to a cookbook, where knowledge is captured and then transferred to others so that they can emulate similar actions. Unfortunately, most projects incorporate inconsistent practices where developers create and deploy on disparate platforms and apply their individual techniques for building and testing code, which becomes problematic during integration when disparities between scripts make them difficult to understand and implement.

An important solution to this problem is a utility available through the Apache Software Foundation called Ant ("Another Neat Tool"). Ant's main purpose is to facilitate application builds and deployments. This is achieved by combining Java programming language applications and XML build files, which can be run on multiple platforms and offer open-architecture flexibility. By maintaining applications and program builds with Ant, consistency levels can be achieved by disparate groups of developers, and the best practices can be propagated throughout a project.

With Ant, targets are generated in project files for compilation, testing, and deployment tasks. The aim of the Ant build file which follows is to highlight some useful

target generation tasks that can facilitate software lifecycle activities so that manual development processes can be automated, which will free up developers' time for greater creativity in their own applications rather than being tied down with mundane build and deployment activities.

Most Ant scripts start with the initialization of application properties demonstrated in lines 11 to 23. These same properties could also be established from the command line and delivered to the Ant build script in the same manner as Java programs with the `-D` parameter during the Ant build invocation. One important thing to consider about properties that are set in an Ant script is that these are immutable constants that cannot be changed once declared. Line 31 shows the `depends` tag that signals that the target `"compile"` depends on the `"init"` target being run prior to its execution. Most scripts will use the `depends` tag to execute sequential build processes.

```
001: <?xml version="1.0"?>
002: <project name="lifecycle management">
003:
004:         <!--
005:         *******************************************************
006:         ** Initialize global properties.
007:         ****************************************************-->
008: <target name="init" description="initialize lifecycle properties.">
009:
010: <tstamp/>
011: <property name="testdir" value="." />
012: <property name="rootdir" value="."/>
013: <property name="builddir" value="build"/>
014: <property name="driver" value="org.gjt.mm.mysql.Driver" />
015: <property name="url" value="jdbc:mysql://localhost/States" />
016: <property name="userid" value="" />
017: <property name="password" value="" />
018: <property name="destdir" value="bugrat" />
019: <property name="zip.bugrat" value="bugrat.zip" />
020: <property name="destdir.bugrat" value="${destdir}/test" />
021: <property name="catalina.home" value="c:\apache\tomcat403"/>
022:     <property name="cvs.repository" ⮐
value=":pserver:<username>@<hostname>:c:\cvsrep"/>
023:     <property name="cvs.package" value="tests"/>
024:
025: </target>
026:
027: <!--
028:         *******************************************************
029:         ** Java compilation
030:         ****************************************************-->
031: <target name="compile" depends="init">
032:         <javac srcdir="." destdir="." classpath="junit.jar" />
033: </target>
034:
```

Listing 11.1 lifecycle_build.xml *(continued)*

Lines 39 to 57 illustrate how Ant can be used to run JUnit tests on your Java applications. JUnit is an open-source testing framework that allows developers to build test suites so that unit testing can be performed on Java components. Personally, we like to use them to create unit tests on JavaBean applications to ensure that developers on our team do not corrupt these files during development activities. If a developer is experiencing a problem with some code, we run some tests that were crafted during development to verify that no bugs have been introduced into an application.

```
035: <!--
036:        *******************************************************
037:        ** JUnit tests
038:        ****************************************************-->
039: <target name="test" depends="compile">
040:
041:        <echo message="Running JUnit tests." />
042:        <junit printsummary="true">
043:        <!-- <formatter type="plain" usefile="false" /> -->
044:        <formatter type="xml" />
045:                <test name="AllTests2" />
046:                <classpath>
047:                        <pathelement location="." />
048:                </classpath>
049:        </junit>
050:        <junitreport todir=".">
051:                <fileset dir=".">
052:                        <include name="TEST-*.xml" />
053:                </fileset>
054:                <report format="frames" todir="." />
055:        </junitreport>
056:
057: </target>
058:
```

Listing 11.1 *(continued)*

Obviously, the setting of appropriate classpath properties is paramount when running Ant and compiling individual Java code. This is accomplished by setting the paths and their components on lines 63 to 69.

```
059:        <!--
060:        *******************************************************
061:        ** Classpath properties.
062:        ****************************************************-->
063:        <path id="classpath.path">
064:                <pathelement location="${builddir}"/>
065:                <fileset dir="cache/lib">
066:                <include name="*.jar"/>
067:                </fileset>
068:                <pathelement location="cache/lib/servlet.jar"/>
069:        </path>
070:
```

Listing 11.1 *(continued)*

Additionally, directory creation scripts on lines 75 to 104 are needed to move source code to proper deployment areas so that they are accessible to other Ant targets.

```
071:         <!--
072:         *******************************************************
073:         ** Create the output directory structure.
074:         ***************************************************-->.
075:         <target name="prepare">
076:
077:         <mkdir dir="${builddir}"/>
078:         <mkdir dir="${builddir}/tests"/>
079:         <mkdir dir="${builddir}/WEB-INF/lib"/>
080:         <mkdir dir="${builddir}/WEB-INF/classes/cache"/>
081:
082:         <copy todir="${builddir}/WEB-INF/classes/cache">
083:                <fileset dir="cache/beans"/>
084:         </copy>
085:         <copy todir="${builddir}">
086:                <fileset dir="cache/src"/>
087:         </copy>
<!-- some copy elements deleted for brevity ... -->
104: </target>
105:
```

Listing 11.1 *(continued)*

All Ant scripts should provide Help targets to assist end users in their build operations as verified on lines 110 to 115. Cleanup targets should be created to remove unwanted files and directories, and to ensure that legacy code that is no longer pertinent does not get added to production builds, which is demonstrated on lines 121 to 125.

```
106:         <!--
107:         ********************************************************
108:         ** Help
109:         ****************************************************-->
110:         <target name="help" description="Lifecycle Help">.
111:
112:         <echo message="Lifecycle Help"/>
113:         <echo message="Type 'ant -projecthelp' for more
assistance..."/>
114:
115: </target>
116:
117:         <!--
118:         ********************************************************
119:         ** Remove the (build/release) directories
120:         ****************************************************-->
121:         <target name="clean">.
122:
123:         <delete dir="${builddir}"/>
124:
125: </target>
126:
```

Listing 11.1 *(continued)*

On many development efforts, code repositories and versioning systems are deployed to share code among programmers and to save modifications for redistribution. Often, an open-source application called Concurrent Versioning System (CVS) is used to perform these tracking and coordination activities. With CVS, check-in and checkout procedures allow users to access source code repositories to ensure that proper builds are deployed and archived. Source code control is an absolute necessity during multiple developer projects because it prevents inconsistencies in program updates and automates coordination among developers. A very simple glimpse of a CVS update operation embedded in Ant is shown on lines 131 to 139.

The combination of these three open-source applications, CVS/JUnit/Bugrat, can be an effective configuration management toolset that developers can integrate with Ant to facilitate their development activities. Configuration management systems are important in that they minimize risk and promote traceability through source control, test coordination, bug tracking, and resolution. Normal CVS operations and JUnit test scripts can be embedded in Ant scripts. The BugRat tracking tool can be built or deployed by using an Ant script to place the zipped Web Archive BugRat file in your J2EE Web container. Lines 146 to 179 show how to initialize, deploy, and clean up a BugRat installation.

```
127: <!--
128:   *****************************************************
129:   ** CVS
130:   ***************************************************-->
131:   <target name="cvs" description="Check out CVS files...">
132:
133:   <echo message="Check out CVS files..."/>
134:   <cvs cvsRoot=":pserver:anoncvs@cvs.apache.org:/home/cvspublic"
135:           package="jakarta-ant"
136:           dest="c:\Java_Pitfalls\Antidote\jakarta-ant-antidote" />
137:          <cvs command="update -A -d"/>
138:
139: </target>
140:
141: <!--
142:   *****************************************************
143:   ** Bugrat - Bug Tracking Tool
144:   ***************************************************-->
145:
146: <target name="initBugrat">
147:
148:   <!-- Create the time stamp -->
149:   <tstamp/>
150:   <!-- Create the build directory structure used by compile -->
151:   <available property="haveBugrat" type="dir"                   ⮐
file="${destdir.bugrat}"/>
152:
153: </target>
154:
```

Listing 11.1 *(continued)*

```
155: <target name="prepareBugrat" depends="initBugrat">
156:
157:     <mkdir dir="${destdir}"/>
158:
159: </target>
160:
161: <target name="installBugrat" depends="prepareBugrat" 
unless="haveBugrat">
162: <unzip src="${zip.bugrat}" dest="${destdir}"/>
163: </target>
164:
165: <target name="deployBugrat" depends="installBugrat" 
description="Install Bugrat">
166:
167:     <pathconvert targetos="windows" property="bugrat_home">
168:             <path location="${destdir.bugrat}"/>
169:     </pathconvert>
170:
171: </target>
172:
173: <target name="cleanBugrat">
174:
175:     <!-- Delete the ${build} and ${dist} directory trees -->
176:     <delete dir="${destdir}" />
177:     <!-- For the sake of brevity, I've omitted the SQL scripts that 
need to be run to build the Bugrat repository. These files include: 
defconfig.sql, defproperties.sql, examplecats.sql, and mysqlschema.sql -->
178:
179: </target>
```

Listing 11.1 *(continued)*

As in life, with software, increasing complexity leads to the natural occurrence of more problems. During software development, source code needs to be consistently reviewed to determine where code can be refactored, or rewritten, to improve efficiencies. Refactoring increases quality through design improvements and the reduction of defects. Two open-source Java utilities, JDepend and JavaNCSS, can be used within an Ant script to provide metrics so that program behaviors can be observed and improvements can be tested.

The JDepend application reads Java class and source file directories to generate metric measurements that can be used to determine software quality. Designs are more extensible when they are independent of implementation details, which allows them to adapt to new modifications without breaking the entire system. JDepend isolates program couplings to determine where dependencies lie and where migrations can occur among lower levels of software hierarchies to higher levels so that redundancies can be decreased. JavaNCSS provides noncommented source code measurements so that large, cumbersome programs can be discovered and possibly be rewritten to improve readability or performance.

Certainly, measurements from both JDepend and JavaNCSS should be used only to gauge software quality and not be deemed absolute predictors as to what needs to be performed in order to make code more efficient.

```
181: <!--
182:        ******************************************************
183:        ** JDepend
184:        ***************************************************-->
185: <target name="jdepend">
200:        <jdepend outputfile="docs/jdepend-report.txt">
201:        <sourcespath>
202:            <pathelement location="./ant/src" />
203:        </sourcespath>
204:        <classpath location="." />
206:        </jdepend>
208: </target>
209:
210: <!--
211:        ******************************************************
212:        ** JavaNCSS
213:        ***************************************************-->
214: <target name="javancss">
223: <taskdef name="javancss" classname="javancss.JavancssAntTask" ⮐
classpath="${CLASSPATH}"/>
231: <javancss srcdir="./ant/src"
232:            generateReport="true"
233:                      outputfile="javancss_metrics.xml"
234:            format="xml"/>
236: </target>
```

Listing 11.1 *(continued)*

Developers can include a splash screen target to signal that something is actually occurring during an Ant command-line build operation by adding lines 242 to 248.

```
238: <!--
239:        ******************************************************
240:        ** Splash screen
241:        ***************************************************-->
242: <target name="splash" description="Display splash screen...">
243:
244:        <echo message="Display splash screen..."/>
245:        <splash imageurl="./ant/images/ant_logo_large.gif"
246:           showduration="5000" />
248: </target>
```

Listing 11.1 *(continued)*

Checkstyle is a Java development tool that helps programmers write Java code that adheres to a recognized coding standard. It automates the process of visually checking through Java code to ensure that previously established coding rules are incorporated into individual source code components.

Some of the useful standards that Checkstyle looks for are unused or duplicate `import` statements, that Javadoc tags for a method match the actual code, the incorporation of specified headers, that `@author` tags exist for class and interface Javadoc

comments, that periods (.) are not surrounded by whitespace, that brackets ({}) are used for if/while/for/do constructs, that lines do not contain tabs, and that files are not longer than a specified number of lines. To implement Checkstyle, a user would need to use lines 254 to 263.

```
250: <!--
251:         ****************************************************
252:         ** Checkstyle
253:         **************************************************-->
254: <target name="checkStyle" description="Coding standard met?...">
256:         <taskdef name="checkstyle"
257:
classname="com.puppycrawl.tools.checkstyle.CheckStyleTask"/>
258:         <echo message="Coding standard met?..."/>
259:         <checkstyle allowTabs="yes">
260:                 <fileset dir="./ant/src" includes="**/*.java"/>
261:                 </checkstyle>
263: </target>
```

Listing 11.1 *(continued)*

Document preparation is an important but often overlooked activity during software implementation. Users often need to understand what APIs are being used to propagate data across systems. Javadoc provides hyperlinked documents for Web browser viewing, which allows users to share copies and facilitates distribution. Javadocs are easily updateable, which helps maintain consistency.

```
266: <!--
267:         ****************************************************
268:         ** Javadoc
269:         **************************************************-->
270:         <target name="javadoc" description="Generate Javadoc
artifacts">
272:         <echo message="Generating Javadoc artifacts..."/>
273:         <javadoc packagenames="*"
274:                     sourcepath="./ant/src"
275:                       sourcefiles="./ant/src/**"
276:                     excludepackagenames="com.dummy.test.doc-files.*"
277:                     defaultexcludes="yes"
278:                     destdir="docs/api"
279:                     author="true"
280:                     version="true"
281:                     use="true"
282:                     windowtitle="Test API">
283:              <doctitle><![CDATA[<h1>Test</h1>]]></doctitle>
284:              <bottom><![CDATA[<i>Copyright &#169; 2002 Java
Pitfalls II All Rights Reserved.</i>]]></bottom>
285:         </javadoc>
286:
287: </target>
```

Listing 11.1 *(continued)*

In the Servlet/JavaServer Page model, Web ARchive (WAR) files are portable components that can be deployed across a wide range of J2EE Web containers. The code below shows how Java source is compiled and packaged for deployment.

```
288:
289:        <!--
290:        ********************************************************
291:        ** Build WAR file
292:        *****************************************************-->
293:        <target name="distribute" depends="prepare">
294:
295:        <echo message="Compiling source..."/>
296:
297:        <javac srcdir="cache/beans" destdir="${builddir}/WEB-
INF/classes/">
298:                <classpath><path
refid="classpath.path"/></classpath>
299:        </javac>
300:
301:        <echo message="Creating WAR file [cache.war]..."/>
302:        <war warfile="${builddir}/cache.war"
webxml="cache/deployment/web.xml">
303:                <fileset dir="${builddir}">
304:
        <patternset id="_source">
305:                                <include name="*.jsp"/>
306:                        </patternset>
307:                        <patternset id="_stylesheet">
308:                                <include name="*.css"/>
309:                </patternset>
310:                </fileset>
311:                <webinf dir="${builddir}/WEB-INF">
312:                        <patternset id="_tld">
313:                                <include name="*.tld"/>
314:                        </patternset>
315:                </webinf>
316:                <classes dir="${builddir}/WEB-INF/classes" >
317:                        <patternset id="_classes">
318:                                <include name="**"/>
319:                        </patternset>
320:                </classes>
321:                <lib dir="${builddir}/WEB-INF/lib"  />
322:        </war>
324:  </target>
```

Listing 11.1 *(continued)*

Database creation, population, and destruction can also be accomplished with Ant scripts. Prior to discovering this capability, our development team was experiencing great difficulties in performing these operations on both Intel and Unix platforms. Different scripts needed to be maintained in order to run the SQL commands, which proved quite cumbersome. By employing Ant, we were able to use the same script on both platforms, which allowed us to facilitate operations.

```
326:         <!--
327:         *******************************************************
328:         ** SQL - table creation, population and deletion
329:         ****************************************************-->
330:
331: <target name="createMySQLCacheTables">
333: <sql driver="${driver}" url="${url}" userid="${userid}"
password="${password}">
334:         <classpath>
335:         <fileset dir=".">
336:                 <include name="mm.mysql-2.0.4-bin.jar" />
337:         </fileset>
338:         </classpath>
339:
340:         <!--
341:         NOTE: Could logon to MySQL thru URL mysql
(default database) and create the States dB instance
342:         or do this manually through this step: create database
States;
343:         -->
345:         CREATE TABLE GeneralInfo (
346:                 State VARCHAR(40) NOT NULL,
347:                 Flower VARCHAR(50),
348:                 Bird VARCHAR(50),
349:                         Capital VARCHAR(40),
350:                 PRIMARY KEY(State)
351:         );
352:
353:                 CREATE TABLE Topics (
354:                 State VARCHAR(40) NOT NULL,
355:                 AutomobileDealers VARCHAR(40),
356:                 BikeTrails VARCHAR(50),
357:                 Gyms VARCHAR(50),
358:                         Hospitals VARCHAR(50),
359:                         Laundromats VARCHAR(50),
360:                         Parks VARCHAR(50),
361:                         Physicians VARCHAR(50),
362:                         PetStores VARCHAR(50),
363:                         Restaurants VARCHAR(50),
364:                         RestAreas VARCHAR(50),
365:                         Supermarkets VARCHAR(50),
366:                  PRIMARY KEY(State)
367:         );
369: </sql>
371: </target>
372:
373: <target name="dropMySQLCacheTables">
375: <sql driver="${driver}" url="${url}" userid="${userid}"
password="${password}">
376:         <classpath>
377:         <fileset dir=".">
378:                 <include name="mm.mysql-2.0.4-bin.jar" />
379:         </fileset>
```

Listing 11.1 *(continued)*

```
380:        </classpath>
382:                DROP TABLE GeneralInfo;
383:                DROP TABLE Topics;
385: </sql>
386: </target>
387:
388: <target name="populateMySQLCacheTables">
390: <sql driver="${driver}" url="${url}" userid="${userid}"
password="${password}">
391:        <classpath>
392:                <fileset dir=".">
393:                        <include name="mm.mysql-2.0.4-bin.jar" />
394:                </fileset>
395:        </classpath>
396:
397:        INSERT INTO GeneralInfo VALUES ('Alabama', 'Camellia',
'Yellowhammer', 'Montgomery');
399:        INSERT INTO Topics VALUES ('Alabama', 'KIA', 'Bama Path',
'Mr. Muscles', 'St. Lukes', 'Mr. Clean', 'Tuscaloosa', 'Dr. Nick', 'Mr.
Pickles', 'Joes Pizzaria', 'Selma', 'Mr. Goodshoes');
401: </sql>
402: </target>
```

Listing 11.1 *(continued)*

The key to having efficient operations during development is predicated on the automation of tedious operations, especially testing. An open-source offering called CruiseControl can be embedded in Ant scripts to allow developers to check out source code from CVS so that release builds can be made and modules can be tested. This is an important component of all software operations. Batch scripts can be used to create nightly builds with CruiseControl so that daily software modifications can be tested for defects overnight and addressed by developers before they are forgotten and remain undetected on a system.

```
404: <!--
405:        ********************************************************
406:        ** Cruise control
407:        *****************************************************-->
408:    <target name="all" depends="clean, compile, javadoc,
distribute, junit" description="prepare application for CruiseControl"/>
410: <target name="cruise" description="Start the automated build
process with CruiseControl">
412:        <copy todir="${catalina.home}/webapps/"
file="Cruisecontrol/cruisecontrol/buildservlet.war" />
414:        <java classname="net.sourceforge.cruisecontrol.MasterBuild"
fork="yes" >
415:            <arg line="-properties cruisecontrol.properties -
lastbuild 20020901010101 -label test 1"/>
416:            <classpath>
```

Listing 11.1 *(continued)*

```
417:                 <pathelement
location="Cruisecontrol\cruisecontrol\cruisecontrol.jar"/>
418:                 <pathelement path="${java.class.path}"/>
419:                 <pathelement path="${compile.classpath}"/>
420:                 <pathelement location="."/>
421:             </classpath>
422:         </java>
424:     </target>
425:
426:     <target name="checkout" description="Update package from CVS">
427:         <cvs cvsroot="${cvs.repository}" package="${cvs.package}"
dest="." passfile="etc\.passwd" />
428:     </target>
429:
430:     <target name="modificationcheck" depends="prepare"
description="Check modifications since last build">
431:
432:         <taskdef name="modificationset"
classname="net.sourceforge.cruisecontrol.ModificationSet"/>
433:         <echo message="Checking for modifications..."/>
434:         <modificationset lastbuild="${lastGoodBuildTime}"
quietperiod="30" dateformat="yyyy-MMM-dd HH:mm:ss">
435:             <cvselement cvsroot="${cvs.repository}"
localworkingcopy="." />
436:         </modificationset>
437:     </target>
438:
439:     <target name="masterbuild"
depends="modificationcheck,checkout,all"  description="Cruise Control
master build"/>
440:     <target name="cleanbuild" depends="clean,masterbuild"
description="Cruise Control clean build"/>
442: </project>
```

Listing 11.1 *(continued)*

Let's face it, the Internet has made it difficult for all software companies to keep up with platform and tool evolutions. Many integrated development environments (IDEs), both proprietary and open-source, have implemented Ant in their applications, but disparities in Ant versions and proprietary tag extensions in these applications have made these tools less desirable for application builds and deployments. These inconsistencies make Ant a much more powerful tool when run from the command line and not from an IDE application that could introduce incompatibility problems.

NOTE The lifecycle build script shown above highlights Ant's ability to ease the construction and deployment of Java projects by automating source code packaging, integration, script execution, and production system deployment. Hopefully, users can use these Ant build techniques to make their lifecycle activities more efficient.

Item 12: JUnit: Unit Testing Made Simple

My wife is a good cook, a really good cook, but she drives me crazy sometimes with experiments on existing recipes that I like and expect to taste a certain way. Sometimes she does this because she's run out of some ingredient or she just feels like trying something new. Me, I don't like to try new things and prefer consistency in what I eat. My wife has made several suggestions on what I should do about this problem, the cleanest version being that I cook for myself. Rather than taking a chance that I might actually learn how to cook, I've graciously learned to accept her unpredictable ways.

Software testing exhibits these same qualities and inconsistencies. Many times developers don't write proper tests because they feel that they don't have the time to do so, and when they do, they often introduce their own programming biases in their manual tests, making them irrelevant. Software testers often don't understand what they are testing because developers don't communicate very well what exactly needs to be tested. Growing code repositories and deadline pressures complicate this matter, as well as the attrition of developers on many projects.

To address this problem, development teams need to enforce some order into their systems to prevent the chaos that habitually occurs, and that can be accomplished by implementing the JUnit framework to test logic boundaries and to ensure that the overall logic of your software components is correct.

JUnit is an open-source unit-testing framework written in Java that allows users to create individual test cases that are aggregated in test suites for deployment. Lines 16 and 17 illustrate this. The test case for my unit test is called OneTestCase.class, and my test suite is "test". If a test class does not define a suite method, then the TestRunner application will extract a suite and fill it with all methods that start with "test" using the Java reflection mechanism.

```
01:
02: import junit.framework.*;
03: import junit.runner.BaseTestRunner;
04:
05: /**
06:  * TestSuite that runs all the sample tests
07:  *
08:  */
09: public class AllTests {
10:
11:  public static void main(String[] args) {
12:          junit.textui.TestRunner.run(suite());
13:  }
14:
15:  public static Test suite() {
16:          testsuite suite= new testsuite("Framework Tests");
17:          suite.addTestSuite(OneTestCase.class);
18:          return suite;
19:  }
20: }
```

Listing 12.1 AllTests.java

Since we develop Web applications on many of our projects, our unit tests focus primarily on JavaBean components to ensure that our program logic is sound and consistent. In the code below, an employee form validates user inputs from a Web page. Lines 59 to 71 demonstrate how to add JUnit test code to our JavaBean applications for unit testing.

```
01:
02: import java.util.*;
03: import junit.framework.*;
04: import junit.runner.BaseTestRunner;
05:
06: public class employeeFormBean {
07:  private String firstName;
08:  private String lastName;
09:  private String phone;
10:  private Hashtable errors;
11:
12:  public boolean validate() {
13:         boolean allOk=true;
14:
15:         if (firstName.equals("")) {
16:            errors.put("firstName","Please enter your first name");
17:            firstName="";
18:            allOk=false;
19:         }
20:         if (lastName.equals("")) {
21:            errors.put("lastName","Please enter your last name");
22:            lastName="";
23:            allOk=false;
24:         }
25:         return allOk;
26:  }
27:
28:  public String getErrorMsg(String s) {
29:         String errorMsg =(String)errors.get(s.trim());
30:      return (errorMsg == null) ? "":errorMsg;
31:  }
32:
33:  public employeeFormBean() {
34:         firstName="";
35:         lastName="";
36:         // errors
37:         errors = new Hashtable();
38:  }
39:
40:  // GET methods ------------------------------------------------
41:  public String getFirstName() {
42:         return firstName;
43:  }
```

Listing 12.2 employeeFormBean.java *(continued)*

```
44:
45:  public String getLastName() {
46:          return lastName;
47:  }
48:
49:  // SET methods ----------------------------------------------------
50:  public void setFirstName(String fname) {
51:          firstName =fname;
52:  }
53:
54:  public void setLastName(String lname) {
55:          lastName =lname;
56:  }
57:
58:  /* main */
59:  public static void main(String[] args) {
60:          junit.textui.TestRunner.run(suite());
61:  }
62:
63:  /**
64:   * TestSuite that runs all the sample tests
65:   *
66:   */
67:  public static Test suite() {
68:          TestSuite suite= new TestSuite("Employee Form Unit Tests");
69:          suite.addTestSuite(employeeFormBeanTestCase.class);
70:          return suite;
71:  }
73: }
```

Listing 12.2 *(continued)*

Listing 12.3 illustrates how a test case should be written to test the JavaBean that was created. Notice the assert statements that check the getter/setter methods in the employeeFormBean component. These asserts ensure that expected behavior is maintained in our code and that bugs that might be introduced to this application are captured and resolved easily. One thing to be aware of when you do create unit tests with JUnit is that if you insert `System.out.println()` statements in your test cases, it does not print the text output to the console when you run your test from the command line.

```
01: import junit.framework.TestCase;
02:
03: public class employeeFormBeanTestCase extends TestCase {
04:  public employeeFormBeanTestCase(String name) {
```

Listing 12.3 employeeFormBeanTestCase.java

```
05:         super(name);
06:  }
07:  public void noTestCase() {
08:  }
09:  public void testCase1() {
10:         employeeFormBean eForm = new employeeFormBean();
11:         assertTrue(eForm != null);
12:  }
13:  public void testCase2() {
14:         employeeFormBean eForm = new employeeFormBean();
15:         assertTrue(eForm != null);
16:
17:         eForm.setFirstName("Steven");
18:         assertTrue("Steven" == eForm.getFirstName());
19:  }
20:  public void testCase3() {
21:         employeeFormBean eForm = new employeeFormBean();
22:         assertTrue(eForm != null);
23:
24:         eForm.setLastName("Fitzgerald");
25:         assertTrue("Fitzgerald" == eForm.getLastName());
26:  }
27:  public void testCase4() {
28:         employeeFormBean eForm = new employeeFormBean();
29:         assertTrue(eForm != null);
30:
31:         eForm.setFirstName("John");
32:         eForm.setLastName("Walsh");
33:         assertTrue(eForm.validate());
34:  }
35:  public void testCase5() {
36:         employeeFormBean eForm = new employeeFormBean();
37:         assertTrue(eForm != null);
38:
39:         String s = eForm.getErrorMsg("firstName");
40:         assertTrue(!s.equals("Please enter your first name"));
41:  }
42:  public void testCase6() {
43:         employeeFormBean eForm = new employeeFormBean();
44:         assertTrue(eForm != null);
45:
46:         String s = eForm.getErrorMsg("lastName");
47:         assertTrue(!s.equals("Please enter your last name"));
48:  }
49:  public void testCase(int arg) {
50:  }
51: }
```

Listing 12.3 *(continued)*

Additionally, unit tests can be created to ensure that data in a database remains consistent and has not been corrupted during testing and integration operations. This is accomplished by creating unit tests that validate JDBC connections and user queries on database data.

```
01:
02: import java.sql.*;
03: import java.util.*;
04: import junit.framework.*;
05: import junit.runner.BaseTestRunner;
06:
07: public class dbQueryBean {
08:
09:   private static final String DRIVER_NAME="org.gjt.mm.mysql.Driver";
10:   private static final String DB_URL="jdbc:mysql://localhost/States";
11:   private static final String USERNAME="";
12:   private static final String PASSWORD="";
13:   private static final String QUERY="Select * from Topics";
14:
15:   Connection conn = null;
16:   Statement stmt = null;
17:   ResultSet rslt = null;
18:
19:   public dbQueryBean() {
20:
21:           try {
22:                   // get driver
23:                   Class.forName(DRIVER_NAME);
24:                   // connect to the MySQL db
25:                   Connection conn =
DriverManager.getConnection(DB_URL, USERNAME, PASSWORD);
26:                   Statement stmt = conn.createStatement();
27:                   ResultSet rslt = stmt.executeQuery(QUERY);
28:           }
29:           catch(Exception e) {
30:           }
31:   }
32:
33:   public void closeDb()
34:   {
35:           try {
36:                   // get driver
37:                   this.conn.close();
38:           }
39:           catch(Exception e) {
40:           }
41:   }
42:
```

Listing 12.4 dbQueryBean.java

```
43:  public ResultSet getResultSet() {
44:          return this.rslt;
45:  }
46:
47:  public Connection getConnection() {
48:          return this.conn;
49:  }
50:
51:  public Statement getStatement() {
52:          return this.stmt;
53:  }
54:
55:  /* main */
56:  public static void main(String[] args) {
57:          junit.textui.TestRunner.run(suite());
58:  }
59:
60:  /**
61:   * TestSuite that runs all the sample tests
62:   *
63:   */
64:  public static Test suite() {
65:          TestSuite suite= new TestSuite("DB Query Unit Tests");
66:          suite.addTestSuite(dbQueryBeanTestCase.class);
67:          return suite;
68:  }
70: }
```

Listing 12.4 *(continued)*

The dbQueryBeanTestCase in Listing 12.5 demonstrates how to assess database connections and result sets that are returned from database queries. In most cases, a simple instantiation of your bean followed by an assert after the invocation of your bean's methods is the way to unit test your code. In this example, static database information is tested; on many enterprise systems the data is dynamic and tests like this would not be proper.

```
01: import java.sql.*;
02: import junit.framework.TestCase;
03:
04: public class dbQueryBeanTestCase extends TestCase {
05:  public dbQueryBeanTestCase(String name) {
06:          super(name);
07:  }
```

Listing 12.5 dbQueryBeanTestCase.java *(continued)*

```
08:  public void noTestCase() {
09:  }
10:  public void testCase1() {
11:          employeeFormBean eForm = new employeeFormBean();
12:          assertTrue(eForm != null);
13:  }
14:  public void testCase2() {
15:
16:          try {
17:
18:                  dbQueryBean db = new dbQueryBean();
19:                  assertTrue(db != null);
20:
21:                  // Get the resultset meta-data
22:                  ResultSet rslt = db.getResultSet();
23:                  ResultSetMetaData rmeta = rslt.getMetaData();
24:
25:                  // Use meta-data to determine column #'s in each row
26:                  int numColumns = rmeta.getColumnCount();
27:                  String[] s = new String[numColumns];
28:
29:                  for (int i=1; i < numColumns; i++) {
30:                          s[i] = rmeta.getColumnName(i);
31:                  }
32:
33:                  // check to see if db columns are correct
34:                  assertTrue(s[1].equals("State"));
35:                  assertTrue(s[2].equals("AutomobileDealers"));
36:                  assertTrue(s[3].equals("BikeTrails"));
37:                  assertTrue(s[4].equals("Gyms"));
38:                  assertTrue(s[5].equals("Hospitals"));
39:                  assertTrue(s[6].equals("Laundromats"));
40:                  assertTrue(s[7].equals("Parks"));
41:                  assertTrue(s[8].equals("Physicians"));
42:                  assertTrue(s[9].equals("PetStores"));
43:                  assertTrue(s[10].equals("Restaurants"));
44:                  assertTrue(s[11].equals("RestAreas"));
45:          }
46:          catch(Exception e) {}
47:  }
48:  public void testCase3() {
49:
50:          try {
51:
52:                  dbQueryBean db = new dbQueryBean();
53:                  assertTrue(db != null);
54:
55:                  // Get the resultset meta-data
```

Listing 12.5 *(continued)*

```
56:                    ResultSet rslt = db.getResultSet();
57:                    ResultSetMetaData rmeta = rslt.getMetaData();
58:
59:                    // Use meta-data to determine column #'s in each row
60:                    int numColumns = rmeta.getColumnCount();
61:                    String[] s = new String[numColumns];
62:
63:                    for (int i=1; i < numColumns; i++) {
64:                           s[i] = rmeta.getColumnName(i);
65:                    }
66:
67:                    while (rslt.next()) {
68:                     for (int i=1; i < numColumns; ++i) {
69:                      if (rslt.getString(i).trim().equals("Alabama")) {
70:                         assertEquals(rslt.getString(i).trim(),
"Alabama");
71:                         assertEquals(rslt.getString(i+1).trim(),
"KIA");
72:                         assertEquals(rslt.getString(i+2).trim(),
"Bama Path");
73:                         assertEquals(rslt.getString(i+3).trim(),
"Mr. Muscles");
74:                         assertEquals(rslt.getString(i+4).trim(),
"St. Lukes");
75:                         assertEquals(rslt.getString(i+5).trim(),
"Mr. Clean");
76:                         assertEquals(rslt.getString(i+6).trim(),
"Tuscaloosa");
77:                         assertEquals(rslt.getString(i+7).trim(),
 "Dr. Nick");
78:                         assertEquals(rslt.getString(i+8).trim(),
"Mr. Pickles");
79:                         assertEquals(rslt.getString(i+9).trim(),
"Joes Pizzaria");
80:                         assertEquals(rslt.getString(i+10).trim(),
"Selma");
81:                         assertEquals(rslt.getString(i+11).trim(),
"Mr. Goodshoes");
82:                                          break;
83:                                   }
84:                           }
85:                    }
87:            }
88:            catch(Exception e) {}
89:     }
90:     public void testCase(int arg) {
91:     }
92: }
```

Listing 12.5 *(continued)*

JUnit allows developers and testers to assess module interfaces to ensure that information flows properly in their applications. Local data structures can be examined to verify that data stored temporarily maintains its integrity, boundaries can be checked for logic constraints, and error handling tests can be developed to ensure that potential errors are captured. All developers should implement the JUnit framework to test their software components and to make certain that development teams don't accept inconsistencies in their programs.

Item 13: The Failure to Execute

Executable JAR files, CLASSPATHs, and JAR conflicts challenge developers as they deploy their Java applications. Frequently, developers run into problems because of inadequate understanding of the Java extension mechanism. This pitfall costs a developer time and effort, and it can lead to substantial configuration control issues. I have seen numerous developers have problems with executable JAR files. They build desktop applications and deploy them via executable JAR files. However, for some reason, sometimes the JAR file will not execute.

When executed using the conventional `java classname` command, the code runs fine. The developer adds all of the classes to a JAR file and sets the main class attribute. Executing the `java -jar jarname` command, the following error returns:

```
Exception in thread "main" java.lang.NoClassDefFoundError:
com/borland/jbcl/layout/XYLayout
        at execution.application.ExecFrame.<init>(ExecFrame.java:23)
        at execution.application.ExecApp.<init>(ExecApp.java:11)
        at execution.application.ExecApp.main(ExecApp.java:40)
```

This is one example of this familiar error for these developers. It is unable to find a particular class, in this case, `com.borland.jbcl.layout.XYLayout`. In this case, the developer did not select the proper JAR packaging for the JBuilder IDE. This is not unique to JBuilder, though, nor IDEs. This is part of a bigger issue in regard to tracking the classpath of applications.

Another classic example of this issue, prior to JDK 1.4, was the use of XML libraries like Xerces. This instance is not a problem in JDK 1.4, because XML is now part of the JVM, but developers cannot wait for every additional JAR to be bundled into the JDK.

So since these developers still haven't learned the underlying problem, they try something else. They try to ensure that the JAR was made correctly, the CLASSPATH is correct, and the JAR is uncompressed and executed in the conventional manner. Everything works right. Recompress it and it no longer works.

```
Manifest-Version: 1.0
Main-Class: Example
Created-By: 1.3.1
```

Why doesn't this work? First, it is important to recognize that the command-line ("java Example") invocation of the JVM uses the same CLASSPATH as the compiler. So, obviously, if it compiled then, it will run.

However, double-clicking on a JAR file, or executing `java -jar` (except in the JDK HOME\bin directory), attempts to use the JRE, which has its own `CLASSPATH`. So, the additional necessary JARs can be placed in the JRE HOME\lib\ext directory. Note that it is not the lib directory itself, unlike the JDK.

However, unless you want to install the classes into that directory, you should add them into your JAR file and reference them in the JAR's manifest file.

```
Manifest-Version: 1.0
Main-Class: Example
Class-Path: jaxp.jar xalan.jar xerces.jar
Created-By: 1.3.1
```

Deploying Java Applications

A tremendous number of problems in installing, configuring, and running Java applications has to do with the misunderstanding of how to specify how the JVM should find classes to load. The first method used looks like the one found in Listing 13.1.

```
01: APPDATA=C:\Documents and Settings\crichardson\Application Data
02: CLASSPATH=D:\soap-2_2;C:\Program Files\Apache Tomcat
4.0\webapps\soap\WEB-INF\classes;D:\soap-2_2\lib\mail.jar;D:\soap-
2_2\lib\activation.jar;D:\soap-2_2\lib\mailapi.jar
03: CommonProgramFiles=C:\Program Files\Common Files
04: COMPUTERNAME=CLAYSVAIO
05: ComSpec=C:\WINNT\system32\cmd.exe
06: HOMEDRIVE=C:
07: HOMEPATH=\
08: OS=Windows_NT
09: [...]
```

Listing 13.1 Environment variables

This shows the classic and traditional way of handling the Java `CLASSPATH`. Set an environment variable, or rather, add to the already existing environment variable. A number of other environment variables were left in this example to highlight how broad the spectrum of things stored in the environment is and the number of applications that must make use of it.

Furthermore, we have one `CLASSPATH` that all applications use. This provides a major issue for deploying applications, because there are possible conflicts that can occur among executable JAR files. This is just the Java version of the "DLL hell" phenomenon—not only for the presence or absence of JAR files, but also the difference in versions of JAR files.

Another mechanism that is used for assembling a `CLASSPATH` is the `-cp` switch in executing the Java Virtual Machine execution.

```
C:\j2sdk1.4.0\jre\bin\javaw -cp
"D:\pkoDev\beans\classes;C:\dom4j\lib\dom4j.jar;D:\java_xml_pack-winter-
01-dev\jaxp-1.2-ea1\xalan.jar;D:\java_xml_pack-winter-01-dev\jaxp-1.2-
ea1\xerces.jar;D:\java_xml_pack-winter-01-dev\jaxm-1.0.1-
ea1\lib\activation.jar;D:\java_xml_pack-winter-01-dev\jaxm-1.0.1-
ea1\lib\dom4j.jar;D:\java_xml_pack-winter-01-dev\jaxm-1.0.1-
ea1\lib\jaxm.jar;D:\java_xml_pack-winter-01-dev\jaxm-1.0.1-
ea1\lib\log4j.jar;D:\java_xml_pack-winter-01-dev\jaxm-1.0.1-
ea1\lib\mail.jar;D:\jaxpack\java_xml_pack-fall01\jaxm-
1.0\jaxm\client.jar"  org.javapitfalls.jar.Example
```

While this option allows for a more explicit and direct way to control the CLASSPATH for your application, it still is neither very user-friendly nor very configuration-friendly. In effect, this is what happens with a lot of applications; everyone packages their own JAR files and references them using the CLASSPATH JVM option in order to control what is available in their applications.

However, this can lead to a great deal of redundancy. For example, I searched for "xerces*.jar" on my development machine. This would capture all of the versions of the popular Apache XML parsing API Xerces. Granted, a development box is not exactly the most representative example of a deployment machine, but I got 73 instances of Xerces. Basically, every Java application running on my machine that uses XML, some of which are not development tools or APIs, has a copy of Xerces to process XML. Those 73 instances take up 108 MB of disk space, to do essentially the same thing.

What if there is a bug fix out for Xerces that needs to be integrated into my applications, or even bigger, a performance improvement. I have to copy it to 73 different places in my application. Furthermore, with the introduction of XML into JDK 1.4, there is a potential for class conflicts. This is because the Java platform has evolved to the point where it provides new features, and XML processing is part of it.

So, why not be able to extend the platform? Earlier, we discussed an example of what people do wrong in creating an executable JAR file. Those mistakes are due to poor understanding of the Java extension mechanism.

The Java Extension Mechanism

The Java Extension Mechanism is how developers can extend the Java platform without having to worry about the issues concerning the CLASSPATH. The most obvious way to create an extension is to copy a JAR file into the JRE HOME/lib/ext directory. Furthermore, Java Plug-in and Java Web Start provide facilities for installing extensions on demand and automatically.

Figure 13.1 shows the architecture of the Java Extension Mechanism. There are sets of classes that make up the core system classes, and developers can provide the ability to add their own classes into the Java Runtime Environment. These are called "optional packages" (formerly known as "standard extensions").

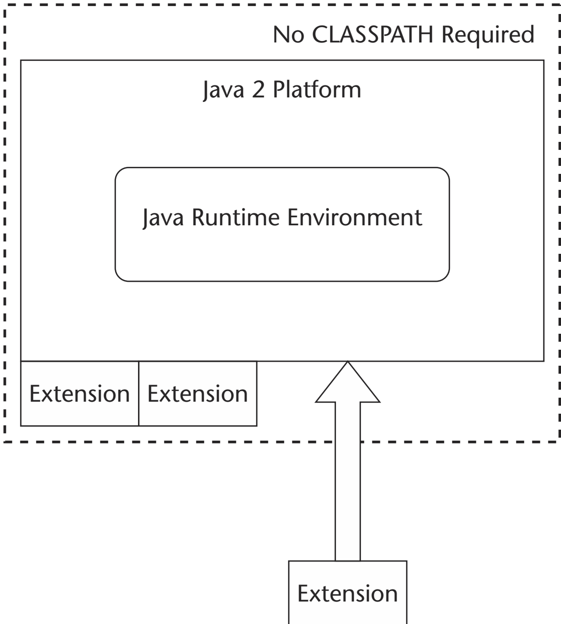


Figure 13.1 Java Extension Mechanism.

We should note that in our previous example, there were references to three optional packages: `jaxp.jar`, `xerces.jar`, and `xalan.jar`. When the executable JAR file shown in that example is executed, the Java Runtime Environment knows to look in those JAR files for classes needed to execute the application. It should be noted that it checks those only if they are not already available in the system class loader; therefore, redundant references to JAR files are not a problem. Class loading from the JAR files is lazy, or as needed.

Sealed Packages

Developers can choose to seal their optional packages. The purpose of sealing a package is to ensure consistency among versions of a particular JAR file. When a package is sealed, it means that every class defined in this JAR must originate from this JAR. Therefore, a sealed Xerces package would assume precedence over other `xerces.jar` files. This can cut both ways, and it requires that developers are cognizant of which of the optional packages may be controlling the loading of classes.

A perfect example of this is when it becomes necessary to change the order in which JAR files are placed in the `CLASSPATH`. If a sealed package is loaded first, and then a

newer version is loaded later in the sequence, the newer version will not be used; in fact, ClassNotFoundExceptions and ClassCastExceptions are common symptoms. Tomcat 3.x versions are known to have this problem in regard to XML. It should be noted that this is addressed in another pitfall regarding the endorsed standards override mechanism.

Listing 13.2 gives an example of the Java Versioning Specification combined with the optional package sealing mechanism.

```
01: Name: org/javapitfalls/
02: Sealed: true
03: Extension-List: ThisClass ThatClass
04: ThisClass-Extension-Name: org.javapitfalls.ThisClass
05: ThisClass-Specification-Version: 1.2
06: ThisClass-Implementation-Version: 1.2
07: ThisClass-Implementation-Vendor-Id: org.javapitfalls
08: ThisClass-Implementation-URL: http://javapitfalls.org/ThisClass.jar
09: ThatClass-Extension-Name: org.javapitfalls.ThatClass
10: ThatClass-Specification-Version: 1.2
11: ThatClass-Implementation-Version: 1.2
12: ThatClass-Implementation-Vendor-Id: org.javapitfalls
13: ThatClass-Implementation-URL: http://javapitfalls.org/ThatClass.jar
14:
```

Listing 13.2 Versioning and sealing an optional package

This shows that any files in the org.javapitfalls package will be sealed—that is, they must all come from this JAR file. The rest shows the appropriate information to allow ClassLoaders to understand the versioning of extensions. The Plug-in is an example of an application that will use this to determine the need to download this version of the package.

Security

These installed optional packages are restricted to the sandbox unless they are from a trusted source. This requires a signing of the JAR file and appropriate permissions being handled in the Java security policy for the application that uses the extension.

Executable JAR files, CLASSPATHs, JAR conflicts—these are all things that challenge developers as they deploy their Java applications. Frequently, developers run into problems because of inadequate understanding of the Java extension mechanism. This pitfall costs a developer time and effort and can lead to substantial configuration control issues.

Item 14: What Do You Collect?

We are building a system that provides a "virtual file cabinet" to browse articles available all over the World Wide Web. The Web site provides a hierarchical format that organizes

knowledge for visibility. The topics can be nested inside each other to go as deeply as the content organizer desires. Each topic maps over to one or more categories. Furthermore, a category can appear in multiple topics. This scenario might seem a bit confused, but it is based on a real system. Essentially, the topics refer to how the Web site wants to organize content, and the categories refer to how content syndicates tag their documents.

As the folders are opened, subfolders are displayed below it on the tree, and the appropriate documents for the categories are displayed in the right pane. Users can browse down the hierarchy to view the data that they want.

This whole logic is based on an XML file that maintains the nesting of topics and mappings to categories. Listing 14.1 shows an excerpt of the XML document that the Web site uses.

```
01: <?xml version = "1.0" encoding = "UTF-8"?>
02: <navigation>
03:  <taxonomy text = "Donnie's Subscription Service" value = "default">
04:          <topic value = "1" text = "Computer Technology News">
05:                  <topic value = "3" text = "Software Technology">
06:                     <topic value = "4" text = "J2SE">
07:                                 <category>2611</category>
08:                     </topic>
09:                     <topic value = "5" text = "J2EE">
10:                                 <category>2612</category>
11:                     </topic>
12:                     <topic value = "6" text = "J2ME">
13:                                 <category>2613</category>
14:                     </topic>
<!-- ... some topics omitted for brevity ... -->
32:                  </topic>
33:                  <topic value = "13" text = "Network Technology">
34:                         <category>2612</category>
35:                         <category>2700</category>
46:                  </topic>
47:                  <topic value = "22" text = "Hardware">
48:                     <topic value = "23" text = "Server Technology">
49:                                 <category>1511</category>
50:                     </topic>
51:                     <topic value = "24" text = "Desktop Technology">
52:                                 <category>1512</category>
53:                     </topic>
54:                     <topic value = "25" text = "Wireless Technology">
55:                                 <category>1513</category>
56:                     </topic>
57:          <...>
58:  </taxonomy>
59: </navigation>
```

Listing 14.1 Navigation.xml

Listing 14.2 demonstrates the code that is used to process the Navigation.xml file for supporting the Web site. It has been stripped down to show the necessary methods and the one of interest (`listCategories`).

```
001: package org.javapitfalls;
002:
003: import java.io.*;
004: import java.net.*;
005: import java.util.*;
006: import org.dom4j.Document;
// ... some dom4j Imports removed for brevity ... Code available on Web
site
014:
015: public class BadNavigationUtils {
017:   private Document document;
018:
019:   public BadNavigationUtils () {
020:         try {
021:         setFile(getClass().getResource("/Navigation.xml"));
022:         setDates();
023:         } catch (Exception e) { e.printStackTrace();  }
024:   }
025:
026:   public void setFile ( String path ) throws DocumentException {
027:         SAXReader reader = new SAXReader();
028:         document = reader.read(path);
029:   }
030:
031:   public void setFile ( URL path ) throws DocumentException {
032:         SAXReader reader = new SAXReader();
033:         document = reader.read(path);
034:   }
035:
036:
037:   public void writeFile (String path) {
039:     try {
040:
041:        // write to a file
042:         XMLWriter writer = new XMLWriter(
043:             new FileWriter( path ),
OutputFormat.createPrettyPrint() );
044:         writer.write( document );
045:         writer.close();
046:
047:     } catch (IOException ioe )  {
048:
049:         ioe.printStackTrace();
051:     }
```

Listing 14.2 BadNavigationUtils.java

```
052:   }
053:
054:
055:   public List listCategories (String[] topics) {
057:         HashSet mySet = new HashSet();
060:         for (int i=0; i < topics.length - 1; i++) {
062:         List list = document.selectNodes( "//topic[@value='" + 
topics[i] + "']/category" );
064:         mySet.addAll(list);
066:         }
067:
068:         ArrayList theList = new ArrayList();
070:         theList.addAll(mySet);
073:         return theList;
074:
075:   }
076:
077:   public static void main(String[] args) {
079:       try {
080:
081:    String [] topics = new String[3];
083:            topics[0] = "4";
084:    topics[1] = "5";
085:          topics[2] = "13";
086:
087:         BadNavigationUtils topicCategory = new BadNavigationUtils 
();
089:         topicCategory.setFile("Navigation.xml");
091:         List categories = topicCategory.listCategories(topics);
092:         for ( int i = 0; i < categories.size(); i++ ) {
093:                 Element myElement = (Element) categories.get(i);
094:                 System.out.println(myElement.getText());
096:       }
099:    } catch (Exception e) { e.printStackTrace();}
101:  }
102: }
103:
```

Listing 14.2 BadNavigationUtils.java *(continued)*

BadNavigationUtils.java uses DOM4J to parse an XML file into a DOM tree, and then uses an XPath expression to pull back a List of category nodes (in the `listCategories` method). This is the result of executing this code:

```
01: 2611
02: 2612
03: 2612
```

```
04: 2700
05: 2710
06: 2711
07: 2712
08: 2713
09: 2714
10: 2715
11: 2720
12: 2730
13: 2740
14: 2750
```

The idea was to filter out the duplicates in the list, but it is clear that this didn't work. The category "2612" is shown twice. Why didn't this work? After all, `HashSet` is supposed to not allow duplicates in the collection.

As it turns out, this method returns a list of nodes to the user. Those nodes represent the location of that particular element on the tree. Therefore, there is a distinction between the 2612 in the `topic` with `id` of 5 and the 2612 in the `topic` with `id` of 13. When we print out the text of each of the nodes, we find that there are duplicate values.

So, if we want to make sure we have a true `Set`, we need to pay more careful attention to the type of object being stored in the collection. To handle this problem, we modify the code to actually store the text value of the nodes. Listing 14.3 shows how we do that.

```
01:   public List listCategories (String[] topics) {
03:         TreeSet mySet = new TreeSet();
06:         for (int i=0; i < topics.length - 1; i++) {
08:         List list = document.selectNodes( "//topic[@value='" + 
topics[i] + "']/descendant-or-self::*/category" );
09:
10:             for ( Iterator it = list.listIterator(); it.hasNext(); )
{
11:                mySet.add( ((Element) it.next()).getText() );
12:             }
14:         }
15:
16:         ArrayList theList = new ArrayList();
18:         theList.addAll(mySet);
20:         return theList;
22:   }
```

Listing 14.3 GoodNavigationUtils.java (listCategories)

Notice in this example we are calling the `getText()` method on the `Element` returned in the `Iterator`. The returned `String` is then added to the `HashSet`.

An interesting development has evolved in the Java Community Process program (JCP). JSR 14, "Add Generic Types to the Java Programming Language," offers a

mechanism for providing parameterized classes (JSR stands for "Java Specification Request"). In this case, we have a `List` being returned to us, which we were unable to determine what type was in the list without closely consulting the API documentation. Of course, since all code is well documented in excruciating detail, we always know the types that are returned to us. In reality, the only documentation on which you can ever truly count (and frequently the only documentation that is read) is the method signature. Of course, if you were up on XML libraries, you would understand that a method called `selectNodes` is going to return `Node` objects.

This gets to the essence of the problem: The Collection classes are a very popular feature in the Java platform. Their power, efficiency, and flexibility are highly valued. However, as this example has shown, it is very difficult to know precisely how collections are handled. This requires that numerous casts and type checks are needed in order to handle something generically.

That is what JSR 14 hopes to accomplish. Now, you would be able to pass a type to the `List` class. For example, you could declare a vector like this:

```
Vector<String> x = new Vector<String>();
```

Generics are a feature to be added to the Java 2 Standard Edition 1.5, code-named "Tiger." Until then, developers will have to be aware of what types are being held in their collections.

Item 15: Avoiding Singleton Pitfalls

As programmers, we love design patterns! They allow us to use repeatable solutions to problems, and they offer us an easy way to communicate in software engineering projects. Used wisely, these patterns offer us the capability to quickly build elegant, flexible designs. Using them without fully understanding their purpose and impact could lead to a brittle software architecture. More importantly, creating an incorrect implementation of existing patterns could lead to disastrous results.

The Singleton design pattern, presented by Gamma, Helm, Johnson, and Vlissides in *Design Patterns: Elements of Reusable Software*, is a creational pattern that ensures that a class has one instance and provides a global point of access to it.[4] In the Java programming language, a correct implementation of a Singleton ensures that there will be one instance of a class per JVM. There are many Singleton classes throughout the Java API itself, with a few examples being *java.util.Calendar* and *java.util.DateFormat*. The Singleton is useful in cases where you want a single point of access to a resource. In Item 32, we created a Singleton class that was a single point of access to a database. Listings 15.1 and 15.2 show two common ways of implementing a Singleton class. The constructors in both classes are private, so that only the static `getInstance()` method can instantiate the class. In Listing 15.1, the Singleton's constructor is called when the class is first loaded, on line 4. When a class calls the `getInstance()` method shown in lines 16 to 18, the static `m_instance` variable is returned.

[4] Gamma, Helm, Johnson, Vlissides, *Design Patterns: Elements of Reusable Software*, Addison-Wesley, Reading, Massachusetts, 1984.

```
01: public class ClassLoadSingleton
02: {
03:    //called at class-load time
04:    private static ClassLoadSingleton
05:                     m_instance = new ClassLoadSingleton();
06:
07:    private ClassLoadSingleton()
08:    {
09:       //implementation details go here
10:    }
11:
12:    /**
13:     * point of entry to this class
14:     */
15:    public static ClassLoadSingleton getInstance()
16:    {
17:        return m_instance;
18:    }
20: }
```

Listing 15.1 Singleton instantiated at class loading time

Listing 15.2 shows a different approach. Instead of calling the constructor at class load time, the class uses *lazy instantiation*—that is, the constructor is not called until `getInstance()` is called the first time. Lines 14 to 20 of Listing 15.2 show the logic of returning the `FirstCallSingleton` class from the `getInstance()` method.

```
01: public class FirstCallSingleton
02: {
03:
04:    private static FirstCallSingleton m_instance = null;
05:
06:    private FirstCallSingleton()
07:    {
08:       //implementation details go here
09:    }
10:
11:    /**
12:     * point of entry to this class
13:     */
14:    public static synchronized FirstCallSingleton getInstance()
15:    {
16:        if (m_instance == null)
17:             m_instance = new FirstCallSingleton();
```

Listing 15.2 Singleton instantiated at first call

```
18:
19:         return m_instance;
20:     }
21: }
```

Listing 15.2 *(continued)*

Now that we've provided a review of the Singleton design pattern, let's discuss what could go wrong. The following sections discuss bad practices that we have commonly seen and provide suggestions on resolving the resulting dilemmas that occur.

When Multiple Singletons in Your VM Happen

Right now, you're thinking, "What? Isn't this contrary to the definition of a Singleton?" Yes, you're absolutely correct. However, if a Singleton class is not written correctly in a one-VM multithreaded system, problems can arise.

Consider a Singleton designed to use lazy instantiation, as implemented in Listing 15.2. If the `getInstance()` method is not synchronized, the following could happen: Two threads could call `getInstance()` simultaneously, and two different versions of `FirstCall` could be returned. Figure 15.1 shows a pictorial representation of how this could occur in one virtual machine. At time t=30, two threads in one virtual machine call `FirstCall.getInstance()`, and two different objects are returned. This could be extremely dangerous if, for example, the Singleton is supposed to guarantee a single point of access to an external resource. If the Singleton class is writing data to a file, for example, several problems could arise, ranging from corrupted data to unintended mutual exclusion problems.

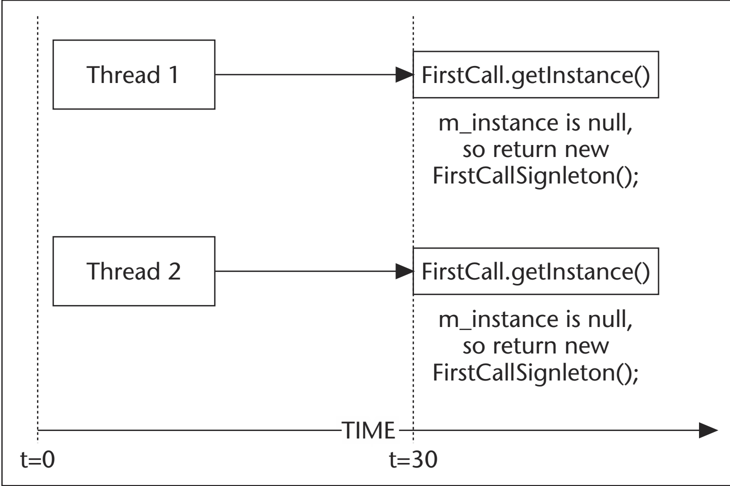


Figure 15.1 The synchronization problem.

Like most concurrency issues, this is a difficult problem to debug. *The key point here is to make certain that your* `getInstance()` *method is synchronized when you are using the lazy instantiation Singleton strategy shown in Listing 15.2.*

When Singletons are Used as Global Variables, or Become Non-Singletons

Sometimes the Singleton class is abused. One example that we've unfortunately seen quite a bit is when a Singleton class is used as a "global variable" to eliminate parameter passing. Using a Singleton in this manner is poor practice, and it may affect the flexibility of your project's software architecture. One example that we have seen is where a code author placed his main application's GUI object in a Singleton with synchronized `setGUI()` and `getGUI()` methods, to avoid passing the user interface component to other objects in the system. The strategy worked well, until the software customer requested multiple applications per VM. Because some of the classes got a handle to the application's user interface with the `getGUI()` method in the Singleton, this had to be rewritten.

We shudder to even show this next example, but an extreme case is shown in Listing 15.3.

```
01: public class GlobalVarSingleton
02: {
03:
04:    private static GlobalVarSingleton
05:                m_instance = new GlobalVarSingleton();
06:
07:    //The use of this class is NOT recommended!
08:    public int x = 0;
09:    public int y = 1;
10:    public int z = 2;
11:
12:    private void GlobalVarSingleton()
13:    {
14:    }
15:
16:    /**
17:     * point of entry to this class
18:     */
19:    public static synchronized GlobalVarSingleton getInstance()
20:    {
21:        return m_instance;
22:    }
23:
24:    public static void main(String[] args)
25:    {
26:        //Bad usage example:
27:
```

Listing 15.3 Singleton as global variable

```
28:         GlobalVarSingleton globals =
29:               GlobalVarSingleton.getInstance();
30:         globals.x = 333;
31:         globals.y = 40803;
32:         globals.z = 21;
33:
34:        /**
35:            At this point, the offender calls other classes
36:            who call GlobalVarSingleton.getInstance() to get
37:            (and change) values of x, y, and z.
38:          **/
39:
40:    }
41: }
```

Listing 15.3 *(continued)*

Listing 15.3 shows the poor choice of using a Singleton to hold variables. In lines 8, 9, and 10, there are public instance variables x, y, and z. The `main()` method of Listing 15.3 shows an example usage of this, where a class in the VM gets the Singleton, alters the variables, and calls another class, which gets the values of those variables and changes them. It goes without saying that there will be synchronization issues in this example, and it also goes without saying that this is poor design. Of course, many Singletons are not as blatant as this; some are designed correctly, but somewhere along the road, an unenlightened programmer could add things to it to accomplish ends like this. Please be aware that this is poor practice, and keep an eye on your existing Singleton classes.

Over time, a code base evolves. Software engineers modify classes, and every once in a while, we have seen times where Singletons, accidentally or intentionally, stop being Singletons. One of the most common events that we have seen is when a novice programmer changes the private constructor to a public one, leading to havoc throughout the baseline. Watch your baseline!

In conclusion, when you are deciding whether to create a Singleton class, first ask yourself the following questions:

- Does there need to be one global entry point to this class?
- Should there be only one instance to this class in the VM?

If your answer is yes, then use a Singleton. If no, don't use this design pattern.

If you do use the Singleton design pattern, be sure to implement your Singleton classes correctly—use a private constructor, and synchronize methods that need to be synchronized, looking at the code skeletons in Listing 15.1 and 15.2. Finally, again, watch your baseline! Poor practices, such as using Singletons as global variables, as well as the evolution of your Singletons into non-Singletons, can cause problems that will keep you up too late at night.

Item 16: When setSize() Won't Work[5]

Most developers stumble upon pitfalls sequentially, based on their experience level with Java. The `setSize()` pitfall usually presents itself shortly after Java developers begin serious GUI development, specifically when they try to set the size of their first newly created custom components. `BadSetSize`, as follows, creates a simple custom button that we want to size to 100 by 100 pixels. Here is the code to create our custom button:

```
class CustomButton extends Button
{
    public CustomButton(String title)
    {
        super(title);
        setSize(100,100);
    }
}
```

In the constructor, developers often mistakenly assume that they can use `setSize()(width, height)` in the same way they do when sizing a frame. The problem arises when the developer hasn't yet gained the knowledge of the Abstract Windowing Toolkit's (AWT) inner workings to understand that this code will only work under certain situations. He or she has no idea that `setSize()` will fail to correctly size the component. For example, when we place our custom button in the frame with other components using a simple grid layout, we get the results in Figure 16.1. Our button is 66 by 23, not 100 by 100! What happened to our call to `setSize()`? The method was executed, of course. However, it did not give the final word on the size of our component.

Listing 16.1 shows the source code for BadSetSize.java.

Figure 16.1 Run of BadSetSize.class.

[5] This pitfall was first published by *JavaWorld* (www.javaworld.com) in the article "Steer clear of Java Pitfalls", September 2000 (http://www.javaworld.com/javaworld/jw-09-2000/jw-0922-javatraps.html?) and is reprinted here with permission. The pitfall has been updated from reader feedback.

```
01: package org.javapitfalls.item16;
02:
03: import java.awt.*;
04: import java.awt.event.*;
05:
06: class CustomButton extends Button
07: {
08:     public CustomButton(String title)
09:     {
10:         super(title);
11:         setSize(100,100);
12:     }
13: }
14:
15: public class BadSetSize extends Frame
16: {
17:     TextArea status;
18:
19:     public BadSetSize()
20:     {
21:         super("Bad Set Size");
22:
23:         setLayout(new GridLayout(2,0,2,2));
24:         Panel p = new Panel();
25:         CustomButton button = new CustomButton("Press Me");
26:         p.add(button);
27:         add(p);
28:         status = new TextArea(3, 50);
29:         status.append("Button size before display: " + 
button.getSize() + "\n");
30:         add(status);
31:         addWindowListener(new WindowAdapter()
32:                         {
33:                         public void windowClosing(WindowEvent we)
34:                         { System.exit(1); }
35:                         });
36:         setLocation(100,100);
37:         pack();
38:         setVisible(true);
39:         status.append("Button size after display: " + 
button.getSize());
40:     }
41:
42:     public static void main(String args [])
43:     {
44:         new BadSetSize();
45:     }
46: }
```

Listing 16.1 BadSetSize.java

Let's examine the correct approach to sizing a component. The key to understanding why our code failed is to recognize that after we create the component, the layout manager—called GridLayout—reshapes the component in accordance with its own rules. This presents us with several solutions. We could eliminate the layout manager by calling setLayout(null), but as the layout manager provides numerous benefits to our code, this is a poor remedy. If the user resizes the window, we still want to be able to automatically resize our user interface, which is the layout manager's chief benefit. Another alternative would be to call setSize() after the layout manager has completed its work. This only provides us with a quick fix: By calling repaint(), the size would change, yet again when the browser is resized. That leaves us with only one real option: Work with the layout manager only to resize the component. Below we rewrite our custom component:

```
class CustomButton2 extends Button
{
    public CustomButton2(String title)
    {
        super(title);
        // setSize(100,100); - unnecessary
    }

    public Dimension getMinimumSize()
    { return new Dimension(100,100); }

    public Dimension getPreferredSize()
    { return getMinimumSize(); }
}
```

Our custom component overrides the getMinimumSize() and getPreferredSize() methods of the Component class to set the component size. The layout manager invokes these methods to determine how to size an individual component. Some layout managers will disregard these hints if their pattern calls for that. For example, if this button was placed in the center of a BorderLayout, the button would not be 100 by 100, but instead would stretch to fit the available center space. GridLayout will abide by these sizes and anchor the component in the center. The GoodSetSize class below uses the CustomButton2 class.

```
01: package org.javapitfalls.item16;
02:
03: import java.awt.*;
04: import java.awt.event.*;
05:
06: class CustomButton2 extends Button
07: {
```

Listing 16.2 GoodSetSize.java

```
08:     public CustomButton2(String title)
09:     {
10:         super(title);
11:         System.out.println("Size of button is : " + this.getSize());
12:     }
13:
14:     public Dimension getMinimumSize()
15:     { return new Dimension(100,100); }
16:
17:     public Dimension getPreferredSize()
18:     { return getMinimumSize(); }
19: }
20:
21: public class GoodSetSize extends Frame
22: {
23:     TextArea status;
24:
25:     public GoodSetSize()
26:     {
27:         super("Good Set Size");
28:
29:         setLayout(new GridLayout(2,0));
30:         Panel p = new Panel();
31:         CustomButton2 button = new CustomButton2("Press Me");
32:         p.add(button);
33:         add(p);
34:         status = new TextArea(3,50);
35:         status.append("Button size before display: " +
button.getSize() + "\n");
36:         add(status);
37:         addWindowListener(new WindowAdapter()
38:                          {
39:                          public void windowClosing(WindowEvent we)
40:                          { System.exit(1); }
41:                          });
42:         setLocation(100,100);
43:         pack();
44:         setVisible(true);
45:         status.append("Button size after display: " +
button.getSize());
46:     }
47:
48:     public static void main(String args [])
49:     {
50:         new GoodSetSize();
51:     }
52: }
```

Listing 16.2 *(continued)*

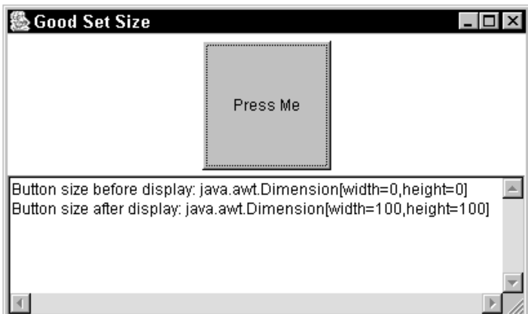


Figure 16.2 Run of GoodSetSize.class.

Running GoodSetSize.java results in Figure 16.2.

It is interesting to note that our solution to setting the size of a component involved not using the `setSize()` method. This pitfall is caused by the design complexity of a cross-platform user interface and a developer's unfamiliarity with the chain of events necessary to display and resize an interface. Unfortunately, the supplied documentation of `setSize()` fails to suggest these prerequisites.

This solution also highlights the importance of properly naming methods and parameters. Should you use `setSize()` when you only need to set some internal values that may or may not be used by your display mechanisms? A better choice would be `setInternalValues()`, which at least clearly warns a developer of the limited guarantee this method offers.

Item 17: When Posting to a URL Won't[6]

Now that the Simple Object Access Protocol (SOAP) and other variants of XML Remote Procedure Calls (RPC) are becoming popular, posting to a Uniform Resource Locator (URL) will be a more common and more important operation. While implementing a standalone SOAP server, I stumbled across multiple pitfalls associated with posting to a URL; starting with the nonintuitive design of the URL-related classes and ending with specific usability pitfalls in the `URLConnection` class.

Connecting via HTTP with the java.net Classes

To perform a Hypertext Transfer Protocol (HTTP) post operation on a URL, you would hope to find a simple `HttpClient` class to do the work, but after scanning the java.net package, you would come up empty. There are several open-source HTTP clients available, and we examine one of them after examining the built-in classes. As an aside, it is interesting to note that there is an `HttpClient` in the sun.net.www.http package that is shipped with the JDK (and used by `HttpURLConnection`) but not part of the

[6] This pitfall was first published by *JavaWorld* (www.javaworld.com) in the article, "Dodge the traps hiding in the URLConnection Class", March 2001 (http://www.javaworld.com/javaworld/jw-03-2001/jw-0323-traps.html?)and is reprinted here with permission. The pitfall has been updated from reader feedback.

public API. Instead, the java.net URL classes were designed to be extremely generic and take advantage of dynamic class loading of both protocols and content handlers. Before we jump into the specific problems with posting, let's examine the overall structure of the classes we will be using (either directly or indirectly). Figure 17.1 is a UML diagram (created with ArgoUML downloadable from www.argouml.org) of the URL-related classes in the java.net package and their relationships to each other. For brevity, the diagram only shows key methods and does not show any data members.

The main class this pitfall centers around is the `URLConnection` class; however, you cannot instantiate that class directly (it is abstract) but only get a reference to a specific subclass of URLConnection via the URL class. If you think that Figure 17.1 is complex, I would agree. The general sequence of events works like this: A static URL commonly specifies the location of some content and the protocol needed to access it. The first time the URL class is used, a URLStreamHandlerFactory Singleton is created. This factory will generate the appropriate `URLStreamHandler` that understands the access protocol specified in the URL. The `URLStreamHandler` will instantiate the appropriate `URLConnection` class that will then open a connection to the URL and instantiate the appropriate `ContentHandler` to handle the content at the URL. So, now that we know the general model, what is the problem? The chief problem is that these classes lack a clear conceptual model by trying to be overly generic. Donald Norman's book *The Design of Everyday Things* states that one of the primary principles of good design is a good conceptual model that allows us to "predict the effects of our actions."[7] Here are some problems with the conceptual model of these classes:

- The URL class is conceptually overloaded. A URL is merely an abstraction for an address or an endpoint. In fact, it would be better to have URL subclasses to differentiate static resources from dynamic services. What is missing conceptually is a `URLClient` class that uses the URL as the endpoint to read from or write to.
- The URL class is biased toward retrieving data from a URL. There are three methods you can use to retrieve content from a URL and only one way to write data to a URL. This disparity would be better served with a URL subclass for static resources that only has a read operation. The URL subclass for dynamic services would have both read and write methods. That would provide a clean conceptual model for use.
- The naming of the protocol handlers "stream" handlers is confusing because their primary purpose is to generate (or build) a connection. A better model to follow would be the one used in the Java API for XML Parsing (JAXP) where a `DocumentBuilderFactory` produces a `DocumentBuilder` that produces a `Document`. Applying that model to the URL classes would yield a `URLConnectorFactory` that produces a `URLConnector` that produces a `URLConnection`.

Now that we have the general picture, we are ready to tackle the `URLConnection` class and attempt to post to a URL. Our goal is to create a simple Java program that posts some text to a Common Gateway Interface (CGI) program. To test our programs, I created a simple CGI program in C that echoes (in an HTML wrapper) whatever is passed in to it. Listing 17.1 is the source code for that CGI program called echocgi.c.

[7] Norman, Donald A., *The Design of Everyday Things*, Doubleday, 1988, page 13.

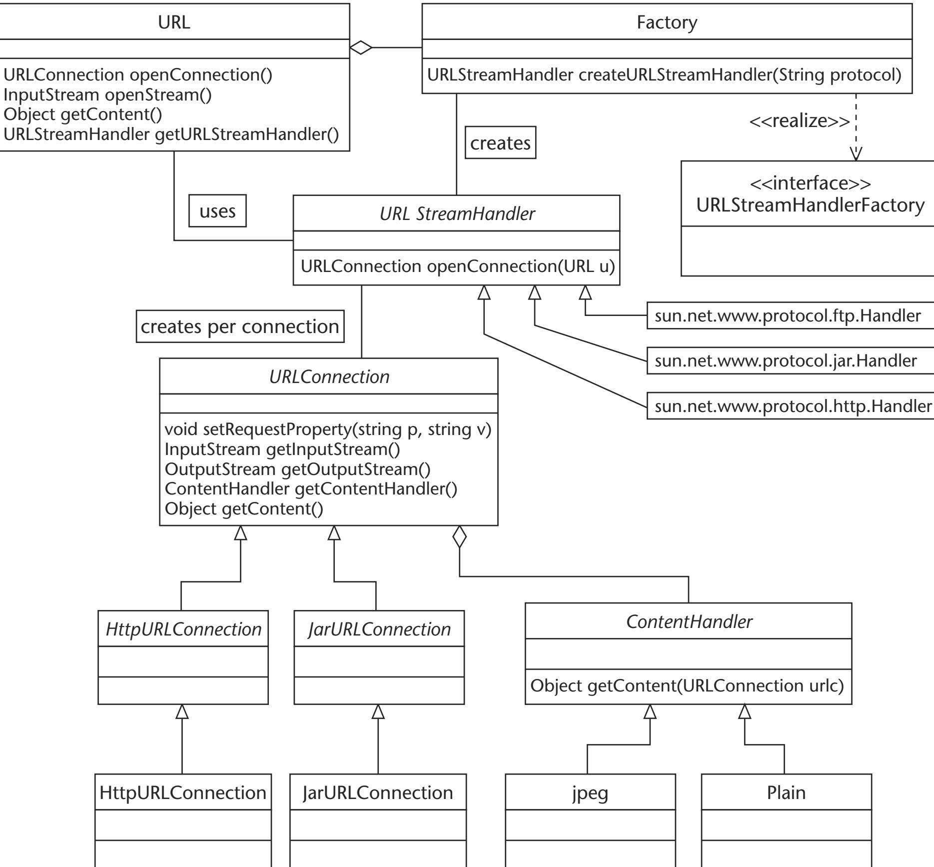


Figure 17.1 URL Classes in the java.net package.

```
01: #include <stdio.h>
02: #include <stdlib.h>
03: #include <string.h>
04:
05: void main(int argc, char **argv)
06: {
07:  char *request_method = NULL;
08:  char *content_length = NULL;
09:  char *content_type = NULL;
10:  int length=0;
11:  char *content = NULL;
12:  int read = 0;
13:
```

Listing 17.1 echocgi.c

```
14:  /* get the key environment variables. */
15:  request_method = getenv("REQUEST_METHOD");
16:  if (!request_method)
17:  {
18:          printf("Not being run as a CGI program.\n");
19:          exit(1);
20:  }
21:
22:  // set outgoing content type
23:  printf("Content-type: text/html\n\n");
24:
25:  if (strcmp(request_method, "POST") == 0)
26:  {
27:          content_length = getenv("CONTENT_LENGTH");
28:          content_type = getenv("CONTENT_TYPE");
29:
30:          length = atoi(content_length);
31:          if (length > 0)
32:          {
33:                  content = (char *) malloc(length + 1);
34:                  read = fread(content, 1, length, stdin);
35:                  content[length] = '\0'; /* NUL terminate */
36:          }
37:
38:          printf("<HEAD>\n");
39:          printf("<TITLE> Echo CGI program </TITLE>\n");
40:          printf("</HEAD>\n");
41:          printf("<BODY BGCOLOR='#ebebeb'>");
42:          printf("<CENTER>\n");
43:          printf("<H2> Echo </H2>\n");
44:          printf("</CENTER>\n");
45:          if (length > 0)
46:          {
47:                  printf("Length of content: %d\n", length);
48:                  printf("Content: %s\n", content);
49:          }
50:          else
51:                  printf("No content! ERROR!\n");
52:          printf("</BODY>\n");
53:          printf("</HTML>\n");
54:  }
55:  else
56:  {
57:          // print out HTML error
58:          printf("<HTML> <HEAD> <TITLE> Configuration Error ⮐
</TITLE></HEAD>\n");
59:          printf("<BODY> Unable to run the Echo CGI Program. <BR>\n");
60:          printf("Reason: This program only tests a POST method. ⮐
<BR>\n");
```

Listing 17.1 *(continued)*

```
61:         printf("Report this to your System Administrator. </BR>\n");
62:         printf("</BODY> </HTML>\n");
63:         exit(1);
64:   }
66: }
```

Listing 17.1 *(continued)*

Testing the CGI program requires two things: a Web server and a browser or program to post information to the program. For the Web server, I downloaded and installed the Apache Web server from www.apache.org. Figure 17.2 displays the simple HTML form used to post information (two fields) to the CGI program. When the "Submit your vote" button is clicked in the HTML form, the two values are posted to the CGI program (on the localhost) and the response page is generated as is shown in Figure 17.3.

Now that we have a simple CGI program to echo data posted to it, we are ready to write our Java program to post data. To send data to a URL, we would expect it to be as easy as writing data to a socket. Fortunately, by examining the `URLConnection` class we see that it has `getOutputStream()` and `getInputStream()` methods, just like the `Socket` class. Armed with that information and an understanding of the HTTP protocol, we write the program in Listing 17.2, BadURLPost.java.

Figure 17.2 HTML Form to test echocgi.exe.

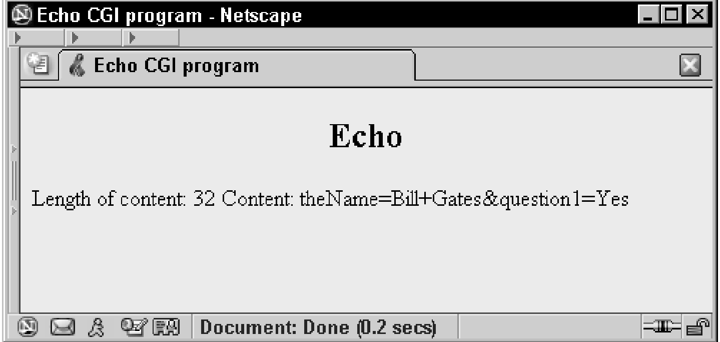


Figure 17.3 HTML response from echocgi.exe.

```
01: /** BadURLPost.java */
02: package org.javapitfalls.item17;
03:
04: import java.net.*;
05: import java.io.*;
06:
07: public class BadURLPost
08: {
09:     public static void main(String args[])
10:     {
11:         // get an HTTP connection to POST to
12:         if (args.length < 1)
13:         {
14:             System.out.println("USAGE: java
GOV.dia.mditds.util.BadURLPost url");
15:             System.exit(1);
16:         }
17:
18:         try
19:         {
20:             // get the url as a string
21:             String surl = args[0];
22:             URL url = new URL(surl);
23:
24:             URLConnection con = url.openConnection();
25:             System.out.println("Received a : " +
con.getClass().getName());
26:
```

Listing 17.2 BadURLPost.java *(continued)*

```
27:             con.setDoInput(true);
28:             con.setDoOutput(true);
29:             con.setUseCaches(false);
30:
31:             String msg = "Hi HTTP SERVER! Just a quick hello!";
32:             con.setRequestProperty("CONTENT_LENGTH", "5"); // Not
checked
33:             con.setRequestProperty("Stupid", "Nonsense");
34:
35:             System.out.println("Getting an input stream...");
36:             InputStream is = con.getInputStream();
37:
38:             System.out.println("Getting an output stream...");
39:             OutputStream os = con.getOutputStream();
40:
41:             /*
42:             con.setRequestProperty("CONTENT_LENGTH", "" +
msg.length());
43:             Illegal access error - can't reset method.
44:             */
45:
46:             OutputStreamWriter osw = new OutputStreamWriter(os);
47:             osw.write(msg);
48:             /** REMEMBER THIS osw.flush(); **/
49:             osw.flush();
50:             osw.close();
51:
52:             System.out.println("After flushing output stream. ");
53:
54:             // any response?
55:             InputStreamReader isr = new InputStreamReader(is);
56:             BufferedReader br = new BufferedReader(isr);
57:             String line = null;
58:
59:             while ( (line = br.readLine()) != null)
60:             {
61:                 System.out.println("line: " + line);
62:             }
63:         } catch (Throwable t)
64:           {
65:             t.printStackTrace();
66:           }
67:     }
68: }
```

Listing 17.2 *(continued)*

A run of Listing 17.2 produces the following:

```
E:\classes\org\javapitfalls\Item17>java
org.javapitfalls.item17.BadURLPost http://localhost/cgi-bin/echocgi.exe ↵
Received a : sun.net.www.protocol.http.HttpURLConnection
Getting an input stream...
Getting an output stream...
java.net.ProtocolException: Cannot write output after reading input.
        at
sun.net.www.protocol.http.HttpURLConnection.getOutputStream(HttpURLCo
nnection.java:507)
        at
com.javaworld.jpitfalls.article3.BadURLPost.main(BadURLPost.java:39)
```

When trying to get the output stream of the `HttpURLConnection` class, the program informed me that I cannot write output after reading input. The strange thing about this error message is that we have not tried to read any data yet. Of course, that assumes the `getInputStream()` method behaves in the same manner as in other IO classes. Specifically, there are three problems with the above code:

- The `setRequestProperty()` method parameters are not checked. This is demonstrated by setting a property called "`stupid`" with a value of "`nonsense`." Since these properties actually go into the HTTP request and they are not validated by the method (as they should be), you must be extra careful to ensure the parameter names and values are correct.
- The `getOutputStream()` method causes the program to throw a `ProtocolException` with the error message "Can't write output after reading input." By examining the JDK source code, we find that this is due to the `getInputStream()` method having the side effect of sending the request (whose default request method is "`GET`") to the Web server. As an aside, this is similar to a side effect in the `ObjectInputStream` and `ObjectOutputStream` constructors that are detailed in my first pitfalls book. So, the pitfall is the assumption that the `getInputStream()` and `getOutputStream()` methods behave just like they do for a `Socket` connection. Since the underlying mechanism for communicating to the Web server actually is a socket, this is not an unreasonable assumption. A better implementation of `HttpURLConnection` would be to postpone the side effects until the initial read or write to the respective input or output stream. This could be done by creating an `HttpInputStream` and `HttpOutputStream`. That would keep the socket metaphor intact. One could argue that HTTP is a request/response stateless protocol and the socket metaphor does not fit. The answer to that is that the API should fit the conceptual model. If the current model is identical to a socket connection, it should behave as such. If it does not, you have stretched the bounds of abstraction too far.

- Although it is commented out, it is also illegal to attempt to set a request property after getting an input or output stream. The documentation for `URLConnection` does state the sequence to set up a connection, although it does not state this is a mandatory sequence.

If we did not have the luxury of examining the source code (which definitely should not be a requirement to use an API), we would be reduced to trial and error (the absolute worst way to program). Neither the documentation nor the API of the `HttpURLConnection` class afford us any understanding of how the protocol is implemented, so we feebly attempt to reverse the order of calls to `getInputStream()` and `getOutputStream()`. Listing 17.3, BadURLPost1.java, is an abbreviated version of that program.

```
01: package org.javapitfalls.item17;
02:
03: import java.net.*;
04: import java.io.*;
05:
06: public class BadURLPost1
07: {
08:     public static void main(String args[])
09:     {
// removed for brevity
35:             System.out.println("Getting an output stream...");
36:             OutputStream os = con.getOutputStream();
37:
38:             System.out.println("Getting an input stream...");
39:             InputStream is = con.getInputStream();
// removed for brevity
67:     }
68: }
```

Listing 17.3 BadURLPost1.java

A run of Listing 17.3 produces the following:

```
E:\classes\org\javapitfalls\Item17>java org.javapitfalls. ⮐
item17.BadURLPost1 http://localhost/cgi-bin/echocgi.exe
Received a : sun.net.www.protocol.http.HttpURLConnection
Getting an output stream...
Getting an input stream...
After flushing output stream.
line: <HEAD>
line: <TITLE> Echo CGI program </TITLE>
line: </HEAD>
line: <BODY BGCOLOR='#ebebeb'><CENTER>
```

```
line: <H2> Echo </H2>
line: </CENTER>
line: No content! ERROR!
line: </BODY>
line: </HTML>
```

Although the program compiles and runs, the CGI program reports that no data was sent! Why? Again we were bitten by the side effects of `getInputStream()`, which caused the POST request to be sent before anything was put in the post's output buffer, thus sending an empty post request.

Now, after having failed twice, we understand that the `getInputStream()` is the key method that actually writes the requests to the server. Therefore, we must perform the operations serially (open output, write, open input, read) as we do in Listing 17.4, GoodURLPost.java.

```
01: package org.javapitfalls.item17;
02:
03: import java.net.*;
04: import java.io.*;
05:
06: public class GoodURLPost
07: {
08:     public static void main(String args[])
09:     {
10:         // get an HTTP connection to POST to
11:         if (args.length < 1)
12:         {
13:             System.out.println("USAGE: java
GOV.dia.mditds.util.GoodURLPost url");
14:             System.exit(1);
15:         }
16:
17:         try
18:         {
19:             // get the url as a string
20:             String surl = args[0];
21:             URL url = new URL(surl);
22:
23:             URLConnection con = url.openConnection();
24:             System.out.println("Received a : " +
con.getClass().getName());
25:
26:             con.setDoInput(true);
27:             con.setDoOutput(true);
28:             con.setUseCaches(false);
29:
30:             String msg = "Hi HTTP SERVER! Just a quick hello!";
```

Listing 17.4 GoodURLPost.java *(continued)*

```
31:             con.setRequestProperty("CONTENT_LENGTH", "" + ⮐
msg.length()); // Not checked
32:             System.out.println("Msg Length: " + msg.length());
33:
34:             System.out.println("Getting an output stream...");
35:             OutputStream os = con.getOutputStream();
36:
37:             OutputStreamWriter osw = new OutputStreamWriter(os);
38:             osw.write(msg);
39:             /** REMEMBER THIS osw.flush(); **/
40:             osw.flush();
41:             osw.close();
42:
43:             System.out.println("After flushing output stream. ");
44:
45:             System.out.println("Getting an input stream...");
46:             InputStream is = con.getInputStream();
47:
48:             // any response?
49:             InputStreamReader isr = new InputStreamReader(is);
50:             BufferedReader br = new BufferedReader(isr);
51:             String line = null;
52:
53:             while ( (line = br.readLine()) != null)
54:             {
55:                 System.out.println("line: " + line);
56:             }
57:         } catch (Throwable t)
58:           {
59:             t.printStackTrace();
60:           }
61:     }
62: }
```

Listing 17.4 *(continued)*

A run of Listing 17.4 produces the following:

```
E:\classes\
org\javapitfalls\Item17>javaorg.javapitfalls.item17.GoodURLPost
http://localhost/cgi-bin/echocgi.exe
Received a : sun.net.www.protocol.http.HttpURLConnection ⮐
Msg Length: 35
Getting an output stream...
After flushing output stream.
Getting an input stream...
line: <HEAD>
line: <TITLE> Echo CGI program </TITLE>
```

```
line: </HEAD>
line: <BODY BGCOLOR='#ebebeb'><CENTER>
line: <H2> Echo </H2>
line: </CENTER>
line: Length of content: 35
line: Content: Hi HTTP SERVER! Just a quick hello!
line: </BODY>
line: </HTML>
```

Finally, success! We now can post data to a CGI program running on a Web server. To summarize, to avoid the HTTP post pitfall, do not assume the methods behave as they do for a socket. Instead, the `getInputStream()` method has the side effect of writing the requests to the Web server. Therefore, the proper sequence must be observed.

One final note on this class is to understand the complexity of writing characters to the Web server. In the above programs, I use the default encoding when writing the `String` to the underlying socket. You could explicitly write bytes instead of characters by first retrieving the bytes via `getBytes()` of the `String` class. Additionally, you could explicitly set the encoding of the characters using the `OutputStreamWriter` class.

An Alternative Open Source HTTP Client

A more intuitive open source package called `HTTPClient` can be downloaded from http://www.innovation.ch/java/HTTPClient. We will use two classes in this package, `HTTPConnection` and `HTTPResponse`, to accomplish the same functionality in GoodURLPost.java. Listing 17.5 demonstrates posting raw data using this package.

```
01: package org.javapitfalls.item17;
02:
03: import HTTPClient.*;
04:
05: import java.net.*;
06: import java.io.*;
07:
08: public class HTTPClientPost
09: {
10:     public static void main(String args[])
11:     {
12:         // get an HTTP connection to POST to
13:         if (args.length < 2)
14:         {
15:             System.out.println("USAGE: java
org.javapitfalls.net.mcd.i1.HTTPClientPost host cgi-program");
16:             System.exit(1);
17:         }
```

Listing 17.5 HTTPClientPost.java *(continued)*

```
18:
19:         try
20:         {
21:             // get the url as a string
22:             String sHost = args[0];
23:             String sfile = args[1];
24:
25:             HTTPConnection con = new HTTPConnection(sHost);
26:
27:             String msg = "Hi HTTP SERVER! Just a quick hello!";
28:
29:             HTTPResponse resp = con.Post(sfile, msg);
30:             InputStream is = resp.getInputStream();
31:
32:             // any response?
33:             InputStreamReader isr = new InputStreamReader(is);
34:             BufferedReader br = new BufferedReader(isr);
35:             String line = null;
36:
37:             while ( (line = br.readLine()) != null)
38:                  System.out.println("line: " + line);
39:         } catch (Throwable t)
40:           {
41:             t.printStackTrace();
42:           }
43:     }
44: }
```

Listing 17.5 *(continued)*

A run of the HttpClientPost program produces:

```
E:\classes\org\javapitfalls>java org.javapitfalls.Item17.HTTPClientPost
localhost /cgi-bin/echocgi.exe
line: <HEAD>
line: <TITLE> Echo CGI program </TITLE>
line: </HEAD>
line: <BODY BGCOLOR='#ebebeb'><CENTER>
line: <H2> Echo </H2>
line: </CENTER>
line: Length of content: 35
line: Content: Hi HTTP SERVER! Just a quick hello!
line: </BODY>
line: </HTML>
```

As you can see, the results are the same as with GoodURLPost. Instead of raw data, you may want to send form input. Listing 17.6 is an example that sends the same form input as demonstrated in Figure 17.1.

```
01: package org.javapitfalls.item17;
02:
03: import HTTPClient.*;
04:
05: import java.net.*;
06: import java.io.*;
07:
08: public class HTTPClientPost2
09: {
10:     public static void main(String args[])
11:     {
12:         // get an HTTP connection to POST to
13:         if (args.length < 2)
14:         {
15:             System.out.println("USAGE: java
org.javapitfalls.net.mcd.i1.HTTPClientPost2 host cgi-program");
16:             System.exit(1);
17:         }
18:
19:         try
20:         {
21:             // get the url as a string
22:             String sHost = args[0];
23:             String sfile = args[1];
24:
25:             HTTPConnection con = new HTTPConnection(sHost);
26:
27:             NVPair form_data[] = new NVPair[2];
28:             form_data[0] = new NVPair("theName", "Bill Gates");
29:             form_data[1] = new NVPair("question1", "No");
30:
31:             HTTPResponse resp = con.Post(sfile, form_data);
32:             InputStream is = resp.getInputStream();
33:
34:             // any response?
35:             InputStreamReader isr = new InputStreamReader(is);
36:             BufferedReader br = new BufferedReader(isr);
37:             String line = null;
38:
39:             while ( (line = br.readLine()) != null)
40:                 System.out.println("line: " + line);
41:         } catch (Throwable t)
42:           {
43:             t.printStackTrace();
44:           }
45:     }
46: }
```

Listing 17.6 HTTPClientPost2.java

A run of the program HTTPClientPost2 produces the following:

```
E:\classes\org\javapitfalls\net\mcd\i1>java org.javapitfalls.net
.mcd.i1.HTTPClientPost2 localhost /cgi-bin/echocgi.exe
line: <HEAD>
line: <TITLE> Echo CGI program </TITLE>
line: </HEAD>
line: <BODY BGCOLOR='#ebebeb'><CENTER>
line: <H2> Echo </H2>
line: </CENTER>
line: Length of content: 31
line: Content: theName=Bill+Gates&question1=No
line: </BODY>
line: </HTML>
```

The results of HTTPClientPost2 are identical to the results in Figure 17.3. In conclusion, while you can use `URLConnection` to post data to Web servers, you will find it more intuitive to use an open-source alternative.

Item 18: Effective String Tokenizing[8]

This pitfall revealed itself when a junior developer needed to parse a text file that used a three-character delimiter (###) between tokens. His first attempt used the `StringTokenizer` class to parse the input text. He sought my advice after he discovered what he considered to be strange behavior. The run of the program below demonstrates code similar to his:

```
>java org.javapitfalls.util.mcd.i1.BadStringTokenizer
input: 123###4#5###678###hello###wo#rld###9
delim: ###
If '###' treated as a group delimiter expecting 6 tokens...
tok[0]: 123
tok[1]: 4
tok[2]: 5
tok[3]: 678
tok[4]: hello
tok[5]: wo
tok[6]: rld
tok[7]: 9
# of tokens: 8
```

As is demonstrated in the above listing, the developer expected six tokens, but if a single "#" character was present in any token, he received more. The junior developer wanted the delimiter to be the group of three pound characters, not a single pound

[8] This pitfall was first published by *JavaWorld* (www.javaworld.com) in the article, "Steer clear of Java Pitfalls", September 2000 (http://www.javaworld.com/javaworld/jw-09-2000/jw-0922-javatraps-p2.html) and is reprinted here with permission. The pitfall has been updated from reader feedback.

character. BadStringTokenizer.java in Listing 18.1 is the incorrect way to parse with a delimiter of "###".

```
01: package org.javapitfalls.item18;
02:
03: import java.util.*;
04:
05: public class BadStringTokenizer
06: {
07:     public static String [] tokenize(String input, String delimiter)
08:     {
09:         Vector v = new Vector();
10:         StringTokenizer t = new StringTokenizer(input, delimiter);
11:         String cmd[] = null;
12:
13:         while (t.hasMoreTokens())
14:             v.addElement(t.nextToken());
15:
16:         int cnt = v.size();
17:         if (cnt > 0)
18:         {
19:             cmd = new String[cnt];
20:             v.copyInto(cmd);
21:         }
22:
23:         return cmd;
24:     }
25:
26:     public static void main(String args[])
27:     {
28:         try
29:         {
30:             String delim = "###";
31:             String input = "123###4#5###678###hello###wo#rld###9";
32:             System.out.println("input: " + input);
33:             System.out.println("delim: " + delim);
34:             System.out.println("If '###' treated as a group
delimiter expecting 6 tokens...");
35:             String [] toks = tokenize(input, delim);
36:             for (int i=0; i < toks.length; i++)
37:                 System.out.println("tok[" + i + "]: " + toks[i]);
38:             System.out.println("# of tokens: " + toks.length);
39:         } catch (Throwable t)
40:           {
41:             t.printStackTrace();
42:           }
43:     }
44: }
```

Listing 18.1 BadStringTokenizer.java

The tokenize() method is simply a wrapper for the StringTokenizer class. The StringTokenizer constructor takes two String arguments: one for the input and one for the delimiter. The junior developer incorrectly inferred that the delimiter parameter would be treated as a group of characters instead of a set of single characters. Is that such a poor assumption? I don't think so. With thousands of classes in the Java APIs, the burden of design simplicity rests on the designer's shoulders and not on the application developer's. It is not unreasonable to assume that a String would be treated as a single group. After all, that is its most common use: a String represents a related grouping of characters.

A correct StringTokenizer constructor would require the developer to provide an array of characters, which would better signify that the delimiters for the current implementation of StringTokenizer are only single characters—though you can specify more than one. This incompletion is an example of API laziness. The API designer was more concerned with rapidly developing the API implementation than the intuitiveness of the implementation. We have all been guilty of this, but it is something we should be vigilant against.

To fix the problem, we create two new static tokenize() methods: one that takes an array of characters as delimiters, the other that accepts a Boolean flag to signify whether the String delimiter should be regarded as a single group. The code for those two methods (and one additional utility method) is in the class GoodStringTokenizer:

```
01: package org.javapitfalls.item18;
02:
03: import java.util.*;
04:
05: public class GoodStringTokenizer
06: {
07:     // String tokenizer with current behavior
08:     public static String [] tokenize(String input, char []
delimiters)
09:     {
10:         return tokenize(input, new String(delimiters), false);
11:     }
12:
13:     public static String [] tokenize(String input, String
delimiters, boolean delimiterAsGroup)
14:     {
15:         String [] result = null;
16:         List l = toksToCollection(input, delimiters,
delimiterAsGroup);
17:         if (l.size() > 0)
18:         {
19:             result = new String[l.size()];
20:             l.toArray(result);
21:         }
```

Listing 18.2 GoodStringTokenizer.java

```
22:         return result;
23:     }
24:
25:     public static List toksToCollection(String input, String
delimiters, boolean delimiterAsGroup)
26:     {
27:         ArrayList l = new ArrayList();
28:
29:         String cmd[] = null;
30:
31:         if (!delimiterAsGroup)
32:         {
33:             StringTokenizer t = new StringTokenizer(input, delimiters);
34:             while (t.hasMoreTokens())
35:                 l.add(t.nextToken());
36:         }
37:         else
38:         {
39:             int start = 0;
40:             int end = input.length();
41:
42:             while (start < end)
43:             {
44:                     int delimIdx = input.indexOf(delimiters,start);
45:                     if (delimIdx < 0)
46:                     {
47:                             String tok = input.substring(start);
48:                             l.add(tok);
49:                             start = end;
50:                     }
51:                     else
52:                     {
53:                             String tok = input.substring(start,
delimIdx);
54:                             l.add(tok);
55:                             start = delimIdx + delimiters.length();
56:                     }
57:             }
58:         }
59:
60:         return l;
61:     }
62:
63:     public static void main(String args[])
64:     {
65:         try
66:         {
```

Listing 18.2 *(continued)*

```
67:             String delim = "###";
68:             String input = "123###4#5###678###hello###wo#rld###9";
69:             // expecting     1     2     3      4       5      6 ⮐
tokens
70:             System.out.println("input: " + input);
71:             System.out.println("delim: " + delim);
72:             System.out.println("If '###' treated as a group ⮐
delimiter expecting 6 tokens...");
73:             String [] toks = tokenize(input, delim, true);
74:             for (int i=0; i < toks.length; i++)
75:                 System.out.println("tok[" + i + "]: " + toks[i]);
76:             System.out.println("# of tokens: " + toks.length);
77:         } catch (Throwable t)
78:           {
79:             t.printStackTrace();
80:           }
81:     }
82: }
83:
```

Listing 18.2 *(continued)*

Following is run of `GoodStringTokenizer` that demonstrates the new static method, `tokenize()`, that treats the token `String` “###” as a single delimiter:

```
>java org.javapitfalls.util.mcd.i1.GoodStringTokenizer
input: 123###4#5###678###hello###wo#rld###9
delim: ###
If '###' treated as a group delimiter expecting 6 tokens...
tok[0]: 123
tok[1]: 4#5
tok[2]: 678
tok[3]: hello
tok[4]: wo#rld
tok[5]: 9
# of tokens: 6
```

Beyond solving the “delimiter as a group” problem, GoodStringTokenizer adds a utility method to convert the set of tokens into a java `Collection`. This is important, as `StringTokenizer` is a pre-`Collection` class that has no built-in support for collections. By returning a collection, we can take advantage of the utility methods, specifically, those for sorting and searching, in the `Collections` class. The class below, `TokenCollectionTester.java`, demonstrates the benefits of a `Collection` of tokens.

```
01: package org.javapitfalls.item18;
02:
03: import java.util.*;
04:
05: public class TokenCollectionTester
06: {
07:     public static void main(String args[])
08:     {
09:         try
10:         {
11:             String input = "zuchinni, apple, beans, hotdog, 
hamburger," +
12:                            "wine, coke, drink, rice, fries, chicken";
13:             String delim = ", ";
14:             List l = GoodStringTokenizer.toksToCollection(input,
15:                                           delim, false);
16:             String top = (String) Collections.max(l);
17:             System.out.println("Top token is: " + top);
18:             Collections.sort(l);
19:             System.out.println("Sorted list: ");
20:             Iterator i = l.iterator();
21:             while (i.hasNext())
22:                 System.out.println(i.next());
23:
24:         } catch (Throwable t)
25:           {
26:             t.printStackTrace();
27:           }
28:     }
29: }
```

Listing 18.3 TokenCollectionTester.java

Running TokenCollectionTester produces the following output:

```
>java org.javapitfalls.util.mcd.i1.TokenCollectionTester
Top token is: zuchinni
Sorted list:
apple
beans
chicken
coke
drink
fries
hamburger
```

```
wine
hotdog
rice
zuchinni
```

In this item, we have carefully examined the workings of the `StringTokenizer` class, highlighted some shortcomings, and created some utility methods to improve the class.

Item 19: JLayered Pane Pitfalls[9]

While working on the jXUL project (an open-source effort to integrate XUL, or Extensible User-Interface Language, with Java) for the book *Essential XUL Programming*, I ported a Pacman arcade game clone called Pagman to a Java-based `XulRunner` platform. `XulRunner` is a Java class that executes XUL applications; it's similar to the JDK's `AppletRunner`. Figure 19.1 provides a screen shot of Pagman port's current version, which successfully allows the ghost sprites to move on a `JLayeredPane`'s top layer. The sprites move over the background images, which exist in a layer beneath. (Many thanks to my coauthor Kevin Smith, who worked through these pitfalls with me to bring Pagman to fruition.)

Instead of examining this pitfall in the `XulRunner` code, which is rather large, we will examine a simpler example that demonstrates the problem. Those interested in the Pagman code can download it from the jXUL Web site (http://www.sourceforge .net/jxul).

Our simple `BadLayeredPane` example in Listing 19.1 attempts to create a frame that has a colored panel in a background layer and a button in a foreground layer with a `JLayeredPane`:

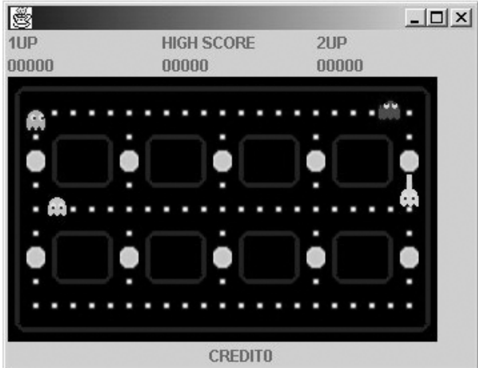


Figure 19.1 Pagman using a JlayeredPane.
Graphics © Dan Addix, Brian King, and David Boswell.

[9] This pitfall was first published by *JavaWorld* (www.javaworld.com) in the article, "Practice makes perfect" November 2001 (http://www.javaworld.com/javaworld/jw-11-2001/jw-1116-traps .html?) and is reprinted here with permission. The pitfall has been updated from reader feedback.

```
01: package org.javapitfalls.item19;
02:
03: import java.awt.*;
04: import javax.swing.*;
05: import java.awt.event.*;
06:
07: public class BadLayeredPane extends JFrame
08: {
09:     public BadLayeredPane()
10:     {
11:         // Error 1: using the Root layered pane
12:         JLayeredPane lp = getLayeredPane();
13:
14:         // set the size of this pane
15:         lp.setPreferredSize(new Dimension(100,100));
16:
17:         // add a Colored Panel
18:         JPanel jpnl = new JPanel();
19:         jpnl.setSize(100,100);
20:         jpnl.setOpaque(true);
21:         jpnl.setBackground(Color.red);
22:
23:         // Error 2: these MUST be of type Integer.
24:         lp.add(jpnl, 2);
25:
26:         // put a Button on top
27:         Button b = new Button("Hi!");
28:         // Error 3: adding button wrong
29:         lp.add(b, 1);
30:     }
31:
32:     public static void main(String [] args)
33:     {
34:         JFrame frame = new BadLayeredPane();
35:
36:         frame.addWindowListener(
37:         new WindowAdapter()
38:         {
39:             public void windowClosing(WindowEvent e)
40:             {
41:                 System.exit(0);
42:             }
43:         });
44:
45:         frame.pack();
46:         frame.setVisible(true);
47:     }
48: }
49:
```

Listing 19.1 BadLayeredPane.java

Figure 19.2 Run of BadLayeredPane.

When Listing 19.1 runs, it produces the screen in Figure 19.2.

Not only is our `JLayeredPane` not working properly, it has no size! We must first work through the size problem before we can approach the heart of our pitfall. Listing 19.1 features three errors (called out in the comments); I'll tackle the first two now and address the third later. First, the `JLayeredPane` that is part of the `JFrame`'s `JRootPane` causes our size problem. When you examine the source code for `JRootPane`, you see that the `JRootPane`'s `RootLayout` does not use the `JLayeredPane` to calculate its size; `JLayeredPane` only calculates the size of the content pane and the menu bar. Second, when adding components to our `JLayeredPane`, we use integers instead of `Integer` objects.

With this knowledge, let's examine our second attempt at displaying our two simple layers. Listing 19.2 fixes two of our problems.

```
01: package org.javapitfalls.item19;
02:
03: import java.awt.*;
04: import javax.swing.*;
05: import java.awt.event.*;
06:
07: public class BadLayeredPane2 extends JFrame
08: {
09:     public BadLayeredPane2()
10:     {
11:         // Fix 1: Create a JLayeredPane
12:         JLayeredPane lp = new JLayeredPane();
13:
14:         // set the size of this pane
15:         lp.setPreferredSize(new Dimension(100,100));
16:
17:         // add a Colored Panel
18:         JPanel jpnl = new JPanel();
19:         jpnl.setSize(100,100);
20:         jpnl.setOpaque(true);
21:         jpnl.setBackground(Color.red);
22:
23:         // Fix 2: using Integer objects
24:         lp.add(jpnl, new Integer(2));
```

Listing 19.2 BadLayeredPane2.java

```
25:
26:         // put a Button on top
27:         Button b = new Button("Hi!");
28:         lp.add(b, new Integer(1));
29:
30:         // Part of Fix 1
31:         getContentPane().add(lp);
32:     }
33:
// main method() Identical to BadLayeredPane.java
50: }
```

Listing 19.2 *(continued)*

We'll first study the fixes applied and then the results. There are two fixes in Listing 19.2 (called out in the comments):

- First, we create a new `JLayeredPane`, which we add to the `ContentPane`. The `RootLayout` manager uses the `ContentPane` to calculate the frame's size, so now the `JFrame` is packed properly.
- Second, we correctly add components to the `JLayeredPane` using an `Integer` object to specify the layer.

Figure 19.3 shows the result of these fixes.

Figure 19.3 clearly demonstrates that we have not yet accomplished our goal. Though the colored panel displays, the button fails to appear on the layer above the panel. Why? Because we assume we add components to a `JLayeredPane` the same way we add components to `Frame`s and `Panel`s. This assumption is our third error and the `JLayeredPane` pitfall. Contrary to `Frame` and `Panel`, the `JLayeredPane` lacks a default `LayoutManager`; thus, the components have no sizes or positions provided for them by default. Instead, a component's size and position must be explicitly set before adding them to the `JLayeredPane`, which Fix 1 achieves in Listing 19.3.

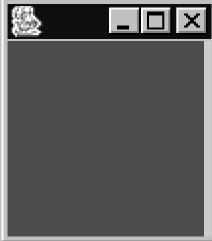


Figure 19.3 Run of BadLayeredPane2.

```
01: package org.javapitfalls.item19;
02:
03: import java.awt.*;
04: import javax.swing.*;
05: import java.awt.event.*;
06:
07: public class GoodLayeredPane extends JFrame
08: {
09:     public GoodLayeredPane()
10:     {
11:         JLayeredPane lp = new JLayeredPane();
12:
13:         // set the size of this pane
14:         lp.setPreferredSize(new Dimension(100,100));
15:
16:         // add a Colored Panel
17:         JPanel jpnl = new JPanel();
18:         jpnl.setSize(100,100);
19:         jpnl.setOpaque(true);
20:         jpnl.setBackground(Color.red);
21:
22:         lp.add(jpnl, new Integer(1));
23:
24:         // put a Button on top
25:         Button b = new Button("Hi!");
26:         // Fix 1: set the size and position
27:         b.setBounds(10,10, 80, 40);
28:         lp.add(b, new Integer(2));
29:
30:         getContentPane().add(lp);
31:     }
32:
// main() method Identical to BadLayeredPane.java
49: }
```

Listing 19.3 GoodLayeredPane.java

When run, Listing 19.3 produces the correct result, shown in Figure 19.4.

Figure 19.4 Run of GoodLayeredPane.

In summary, the key pitfall in our JLayeredPane example is wrongly assuming that the JLayeredPane has a default LayoutManager like JFrame and JPanel. Experience tells us to eliminate that assumption and position and size the components for each layer. Once we do so, the JLayeredPane works fine.

Item 20: When File.renameTo() Won't[10]

The File class and specifically the File.renameTo() method suffers from pitfalls in both design and implementation. Many pitfalls stem from confusion regarding the expected behavior of classes in the Java libraries. Unfortunately, the input/output (IO) classes in Java have been prone to significant revision as the Java platform has evolved. An early overhaul added readers and writers to the input and output streams to distinguish between character-based IO and byte-based IO. With JDK 1.4, another overhaul has taken place, adding a lower layer of high-performance access via Channel classes. Figure 20.1 displays the major file-related classes in the java.io and java.nio packages.

Unfortunately, just the fact that there are five classes and two interfaces all pertaining to different facets of a file increases the complexity of using these classes properly. That is especially true if the distinction between classes is small or if the role of the class is ill defined. Let's examine each class and its purpose:

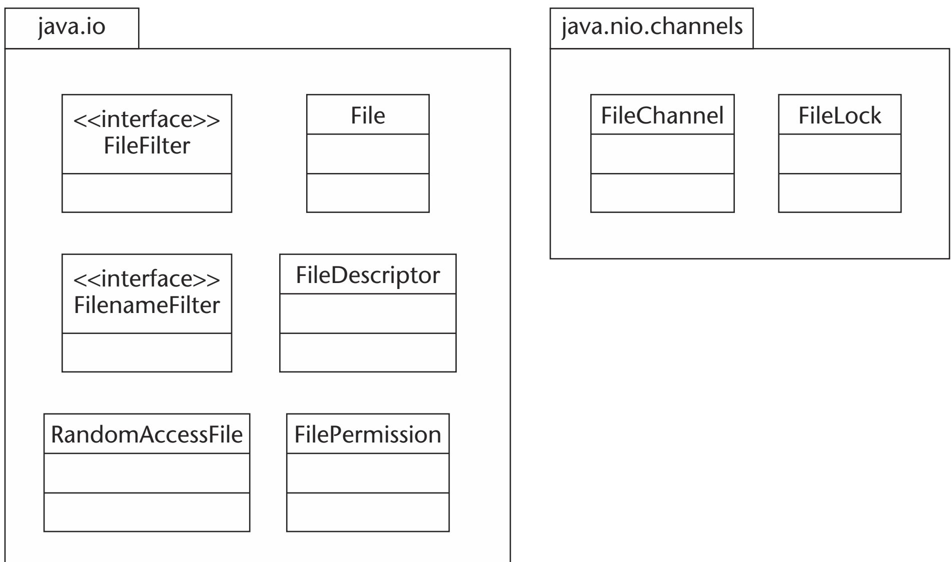


Figure 20.1 File-related classes in the Java class libraries.

[10] This pitfall was first published by *JavaWorld* (www.javaworld.com) in the article, "Practice makes perfect" November 2001 (http://www.javaworld.com/javaworld/jw-11-2001/jw-1116-traps-p2.html) and is reprinted here with permission. The pitfall has been updated from reader feedback.

File. Present since JDK 1.0, a class that represents a file or directory pathname. This class contains a wide array of methods to test characteristics of a file, delete a file, create directories, and, of course, rename a file. Unfortunately, this class suffers from a vague scope in that it incorporates behaviors of a directory entry, like `isFile()`, `isDirectory()`, and `lastModified()`, and behaviors of a physical file like `createNewFile()`, `delete()`, and `renameTo()`. We will discuss this more later.

FilenameFilter. Present since JDK 1.0, an interface to test the occurrence of a list of `File` objects via the `File.list()` method and `FileDialog.setFilename-Filter()` method. The confusion over scope stated above is evident in the contradiction between this interface and the next one (`FileFilter`) in terms of their names. This interface has a single method called `accept()` that receives a `File` object representing a directory and a `String` representing the name of the file to filter on.

FileFilter. Added in JDK 1.2, an interface to filter in the same manner as `File-nameFilter` except that the `accept()` method receives only a single `File` object. Unfortunately, this interface is a prime example of a superfluous convenience interface that does more harm than good because of the new name. It would have been far better to follow the precedent of the `awt` package where `LayoutManager2` extends `LayoutManager` to add methods. The difference between the two design strategies is that the `LayoutManager` interfaces are clearly semantically congruent, whereas `FilenameFilter` and `FileFilter` are not.

FileDescriptor. A class to provide an opaque handle to the operating system-specific File data structure. As its name implies, this class is a very thin abstraction over an operating system structure. The class only has two methods. As a general design rule, it would be preferable to combine our abstractions of a physical file into a single class. Unfortunately, the requirements of backward compatibility cause future developers to suffer with multiple abstractions. Since the New IO package (NIO), split IO operations at the package level (which also spoils the platform's cohesiveness), there is an opportunity to start from scratch with new classes in the NIO package.

FilePermission. A class to represent access to a file directory. This was part of the 1.2 fine-grained security mechanisms—again, a nice candidate for conceptual consolidation.

RandomAccessFile. Present since JDK 1.0, a class that represents the characteristics and behaviors (taken from the C standard library) of a binary file. This allows low-level reading and writing of bytes to a file from any random file position. This class stands on its own and does not fit in to the IO stream metaphor and does not interact with those classes. It is interesting to note that the word "File" in this class name actually refers to a physical file and not just its name. Unfortunately, this package lacks such consistency.

FileChannel. Added to JDK 1.4, a class to provide a high-performance pathway for reading, writing, mapping, and manipulating a file. You get a `FileChannel` from a `FileInputStream`, `FileOutputStream`, or `RandomAccessFile` class. A key benefit of this class is the ability to map a file to memory. Item 2

examined the NIO performance improvements. Lastly, a region of the file (or the whole file) may be locked to prevent access or modification by other programs via methods that return a `FileLock` object.

FileLock. Added to JDK 1.4, a class to represent a lock on a region of a file. A lock can be either exclusive or shared. These objects are safe for use by multiple threads but only apply to a single virtual machine.

Now let's narrow our focus to the `File` class. Listing 20.1 demonstrates some `File` class behaviors and pitfalls.

```
01: package org.javapitfalls.item20;
02: 
03: import java.io.*;
04: 
05: public class BadFileRename
06: {
07:     public static void main(String args[])
08:     {
09:         try
10:         {
11:             // check if test file in current dir
12:             File f = new File("dummy.txt");
13:             String name = f.getName();
14:             if (f.exists())
15:                 System.out.println(f.getName() + " exists.");
16:             else
17:                 System.out.println(f.getName() + 
" does not exist.");
18: 
19:             // Attempt to rename to an existing file
20:             File f2 = new File("dummy.bin");
21:             // Issue 1: boolean status return instead of Exceptions
22:             if (f.renameTo(f2))
23:                 System.out.println(
24:                     "Rename to existing File Successful.");
25:             else
26:                 System.out.println(
27:                     "Rename to existing File Failed.");
28: 
29:             // Attempt to rename with a different extension
30:             int dotIdx = name.indexOf('.');
31:             if (dotIdx >= 0)
32:                 name = name.substring(0, dotIdx);
33:             name = name + ".tst";
34:             String path = f.getAbsolutePath();
35:             int lastSep = path.lastIndexOf(File.separator);
36:             if (lastSep > 0)
```

Listing 20.1 BadFileRename.java *(continued)*

```
37:                  path = path.substring(0,lastSep);
38:              System.out.println("path: " + path);
39:              File f3 = new File(path + File.separator + name);
40:              System.out.println("new name: " + f3.getPath());
41:              if (f.renameTo(f3))
42:                  System.out.println(
43:                      "Rename to new extension Successful.");
44:              else
45:                  System.out.println(
46:                      "Rename to new extension failed.");
47:
48:              // delete the file
49:              // Issue 2: Is the File class a file?
50:              if (f.delete())
51:                  System.out.println("Delete Successful.");
52:              else
53:                  System.out.println("Delete Failed.");
54:
55:              // assumes program not run from c drive
56:              // Issue 3: Behavior across operating systems?
57:              File f4 = new File("c:\\" + f3.getName());
58:              if (f3.renameTo(f4))
59:                  System.out.println(
"Rename to new Drive Successful.");
60:              else
61:                  System.out.println("Rename to new Drive failed.");
62:          } catch (Throwable t)
63:           {
64:              t.printStackTrace();
65:           }
66:      }
67: }
68:
```

Listing 20.1 *(continued)*

When this code is run from a drive other than C, and with the file dummy.txt in the current directory, it produces the following output:

```
E:\classes\org\javapitfalls\Item20>java
org.javapitfalls.item20.BadFileRename
dummy.txt exists.
Rename to existing File Failed.
path: E:\classes\org\javapitfalls\Item20
new name: E:\classes\org\javapitfalls\Item20\dummy.tst
Rename to new extension Successful.
Delete Failed.
Rename to new Drive Successful.
```

Listing 20.1 raises three specific issues, which are called out in the code comments. At least one is accurately characterized as a pitfall, and the others fall under poor design:

- First, returning a Boolean error result does not provide enough information about the failure's cause. That proves inconsistent with exception use in other classes and should be considered poor design. For example, the failure above could have been caused by either attempting to `renameTo()` a file that already exists or attempting to `renameTo()` an invalid filename. Currently, we have no way of knowing.
- The second issue is the pitfall: attempting to use the initial `File` object after a successful rename. What struck me as odd in this API is the use of a `File` object in the `renameTo()` method. At first glance, you assume you only want to change the filename. So why not just pass in a `String`? In that intuition lies the source of the pitfall. The pitfall is the assumption that a `File` object represents a physical file and not a file's name. In the least, that should be considered poor class naming. For example, if the object merely represents a filename, then it should be called `Filename` instead of `File`. Thus, poor naming directly causes this pitfall, which we stumble over when trying to use the initial `File` object in a `delete()` operation after a successful rename.
- The third issue is `File.renameTo()`'s different behavior between operating systems. The `renameTo()` works on Windows even across filesystems (as shown here) and fails on Solaris (reported in Sun's Bug Parade and not shown here). The debate revolves around the meaning of "Write Once, Run Anywhere" (WORA). Sun programmers verifying reported bugs contend that WORA simply means a consistent API. That is a cop-out. A consistent API does not deliver WORA; there are numerous examples in existing APIs where Sun went beyond a consistent API to deliver consistent behavior. The best-known example of this is Sun's movement beyond the Abstract Windowing Toolkit's consistent API to Swing's consistent behavior. If you claim to have a platform above the operating system, then a thin veneer of an API over existing OS functionality will not suffice. A WORA platform requires consistent behavior; otherwise, "run anywhere" means "maybe run anywhere." To avoid this pitfall, you check the "os.name" System property and code `renameTo()` differently for each platform.

Out of these three issues, we can currently only fix the proper way to delete a file after a successful rename, as Listing 20.2 demonstrates. Because the other two issues result from Java's design, only the Java Community Process (JCP) can initiate these fixes.

```
01: package org.javapitfalls.item20;
02: 
03: import java.io.*;
04: 
05: public class GoodFileRename
06: {
```

Listing 20.2 GoodFileRename.java *(continued)*

```
07:     public static void main(String args[])
08:     {
09:         try
10:         {
11:             // check if test file in current dir
12:             File f = new File("dummy2.txt");
13:             String name = f.getName();
14:             if (f.exists())
15:                 System.out.println(f.getName() + " exists.");
16:             else
17:                 System.out.println(f.getName() +
" does not exist.");
18:
19:             // Attempt to rename with a different extension
20:             int dotIdx = name.indexOf('.');
21:             if (dotIdx >= 0)
22:                 name = name.substring(0, dotIdx);
23:             name = name + ".tst";
24:             String path = f.getAbsolutePath();
25:             int lastSep = path.lastIndexOf(File.separator);
26:             if (lastSep > 0)
27:                 path = path.substring(0,lastSep);
28:             System.out.println("path: " + path);
29:             File f3 = new File(path + File.separator + name);
30:             System.out.println("new name: " + f3.getPath());
31:             if (f.renameTo(f3))
32:                 System.out.println(
33:                     "Rename to new extension Successful.");
34:             else
35:                 System.out.println(
36:                     "Rename to new extension failed.");
37:
38:             // delete the file
39:             // Fix 1: delete via the "Filename" not File
40:             if (f3.delete())
41:                 System.out.println("Delete Successful.");
42:             else
43:                 System.out.println("Delete Failed.");
44:         } catch (Throwable t)
45:           {
46:             t.printStackTrace();
47:           }
48:     }
49: }
50:
```

Listing 20.2 *(continued)*

A run of Listing 20.2 produces the following output:

```
E:\classes\org\javapitfalls\Item20> java org.javapitfalls.item20
.GoodFileRename
dummy2.txt exists.
path: E:\classes\org\javapitfalls\Item20
new name: E:\classes\org\javapitfalls\Item20\dummy2.tst
Rename to new extension Successful.
Delete Successful.
```

Thus, don't use the `File` class as if it represents a file instead of the filename. With that in mind, once the file is renamed, operations such as `delete()` only work on the new filename.

Item 21: Use Iteration over Enumeration[11]

`Enumeration` is the original interface, available since JDK 1.0, to iterate over (step through) all the elements in a collection. In terms of semantics, it would have been better to call the interface "Enumerator," as it expresses the role a class is "putting on" by implementing the interface, instead of "Enumeration," which specifies an occurrence of the activity. This is in line with all the more recent interfaces in the `java.util` package like `Observer`, `Comparator`, and `Iterator`. Table 21.1 compares the `Enumeration` interface to the `Iterator` interface.

Table 21.1 Enumeration versus Iterator

ENUMERATION METHODS	DESCRIPTION	ITERATOR METHODS	DESCRIPTION
`boolean hasMoreElements();`	Checks if this enumeration has more elements.	`boolean hasNext()`	Checks if this iterator has more elements.
`Object nextElement();`	Returns the next element in the enumeration if there is at least 1 more.	`Object next()`	Returns the next element in the iterator if there is at least one more.
		`void remove()`	Removes from the underlying collection the last element returned. (optional)

[11] This pitfall was first published by *JavaWorld* (www.javaworld.com) in the article, "Practice makes perfect" November 2001, (http://www.javaworld.com/javaworld/jw-11-2001/jw-1116-traps-p2.html) and is reprinted here with permission. The pitfall has been updated from reader feedback.

The idiom for using both an Enumeration and Iterator is the same:

```
while (i.hasNext())
{
     Object o = i.next();
     // do something with o
}
```

As is evident in Table 21.1, both Iterator and Enumeration are functionally identical except for two differences:

- Iterators allow you to safely remove an element from the underlying collection in a well-defined way.
- The iterator method names have been simplified.

There is a removal of elements pitfall in the Enumeration implementation class of Vector where its behavior differs from Iteration. Listing 21.1 demonstrates the behavior in question.

```
01: package org.javapitfalls.item21;
02:
03: import java.util.*;
04:
05: public class BadVisitor
06: {
07:     public static void main(String args[])
08:     {
09:         Vector v = new Vector();
10:         v.add("one"); v.add("two"); v.add("three"); v.add("four");
11:
12:         Enumeration enum = v.elements();
13:         while (enum.hasMoreElements())
14:         {
15:             String s = (String) enum.nextElement();
16:             if (s.equals("two"))
17:                 v.remove("two");
18:             else
19:             {
20:                 // Visit
21:                 System.out.println(s);
22:             }
23:         }
24:
25:         // see what's left
```

Listing 21.1 BadVisitor.java

```
26:         System.out.println("What's really there...");
27:         enum = v.elements();
28:         while (enum.hasMoreElements())
29:         {
30:             String s = (String) enum.nextElement();
31:             System.out.println(s);
32:         }
33:     }
34: }
```

Listing 21.1 *(continued)*

When run, Listing 21.1 produces the following:

```
E:\classes>java org.javapitfalls.item21.BadVisitor
one
four
What's really there...
one
three
four
```

You would expect to have visited elements `"one"`, `"three"`, and `"four"`, but instead only visited elements `"one"` and `"four"`. The problem is that we are assuming that the `Enumeration` implementation and the `Vector` class work in sync, which is not the case. What has happened is the index integer (called `count`) is not modified when the `Vector.remove()` method is called. This is demonstrated in Figure 21.1. Listing 21.2 demonstrates how an iterator handles this situation.

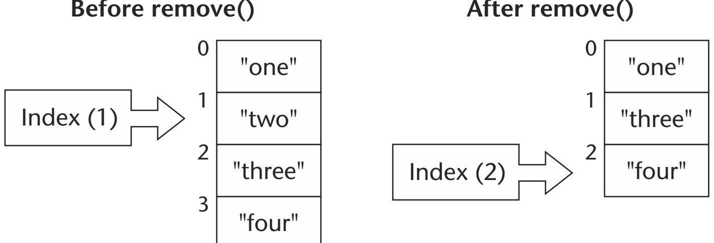


Figure 21.1 Under the hood of a `Vector`.

```
01: package org.javapitfalls.item21;
02:
03: import java.util.*;
04:
05: public class BadVisitor2
06: {
07:     public static void main(String args[])
08:     {
09:         Vector v = new Vector();
10:         v.add("one"); v.add("two"); v.add("three"); v.add("four");
11:
12:         Iterator iter = v.iterator();
13:         while (iter.hasNext())
14:         {
15:             String s = (String) iter.next();
16:             if (s.equals("two"))
17:                 v.remove("two");
18:             else
19:             {
20:                 // Visit
21:                 System.out.println(s);
22:             }
23:         }
24:
25:         // see what's left
26:         System.out.println("What's really there...");
27:         iter = v.iterator();
28:         while (iter.hasNext())
29:         {
30:             String s = (String) iter.next();
31:             System.out.println(s);
32:         }
33:     }
34: }
```

Listing 21.2 BadVisitor2.java

When run, Listing 21.2 produces the following:

```
E:\classes>java org.javapitfalls.item21.BadVisitor2
one
Exception in thread "main" java.util.ConcurrentModificationException
    at
java.util.AbstractList$Itr.checkForComodification(AbstractList.java:445)
    at java.util.AbstractList$Itr.next(AbstractList.java:418)
    at
com.javaworld.jpitfalls.article5.BadVisitor2.main(BadVisitor2.java:15)
```

As the output shows, the class implementing the `Iterator` interface specifically checks for this type of concurrent modification (modification outside of the iteration implemenation class while we are iterating) and throws an exception. It would be nice if the `Enumeration` implementation class was upgraded with this same behavior. Now let's examine the correct way to do this. Listing 21.3 demonstrates both visiting and modifying with an `Iterator`.

```
01: package org.javapitfalls.item21;
02:
03: import java.util.*;
04:
05: public class GoodVisitor
06: {
07:     public static void main(String args[])
08:     {
09:         Vector v = new Vector();
10:         v.add("one"); v.add("two"); v.add("three"); v.add("four");
11:
12:         Iterator iter = v.iterator();
13:         while (iter.hasNext())
14:         {
15:             String s = (String) iter.next();
16:             if (s.equals("two"))
17:                 iter.remove();
18:             else
19:             {
20:                 // Visit
21:                 System.out.println(s);
22:             }
23:         }
24:
25:         // see what's left
26:         System.out.println("What's really there...");
27:         iter = v.iterator();
28:         while (iter.hasNext())
29:         {
30:             String s = (String) iter.next();
31:             System.out.println(s);
32:         }
33:     }
34: }
```

Listing 21.3 GoodVisitor.java

When Listing 21.3 is run, it produces the following:

```
E:\classes>java org.javapitfalls.item21.GoodVisitor
one
```

```
three
four
What's really there...
one
three
four
```

Notice that the `remove` method is performed via the `Iterator` class and not using the `Vector` class. So, in general, for classes that support it, use an `Iterator` over `Enumeration`. All classes that implement the `Collection` interface support iteration through the following method:

```
public Iterator iterator();
```

The classes that implement the `Collection` interface are `AbstractCollection`, `AbstractList`, `AbstractSet`, `ArrayList`, `BeanContextServicesSupport`, `BeanContextSupport`, `HashSet`, `LinkedHashSet`, `LinkedList`, `TreeSet`, and `Vector`. Some classes like `Hashtable` and `HashMap` (all the classes that implement the `Map` interface) indirectly support iteration by returning a `Collection` via the `values()` method. There are still some classes that only implement `Enumeration` like `StringTokenizer`, `java.security.Permissions`, and classes in Swing and JNDI.

Item 22: J2ME Performance and Pitfalls

The Java 2 Micro Edition (J2ME) is a subset of the Java platform created for developing applications on small footprint devices, like personal digital assistants (PDAs) and cell phones. These devices are significantly constrained in terms of processor speed, memory, and storage space. While the amounts vary between devices, you can easily expect several orders of magnitude difference between your desktop PC and these devices. There are three main classes of pitfalls associated with the J2ME platform: memory consumption pitfalls, performance pitfalls, and API differences. Both memory consumption and performance pitfalls have more to do with programming habit than the J2ME platform. The fact is our habits have developed around creating reusable, modular, and readable J2SE/J2EE programs where memory and processing speed are abundant. Unfortunately, those very habits deemed "good style" for J2SE/J2EE lead to either nonperforming or poor-performing J2ME applications. The API pitfalls relate to different method calls for classes implemented in both J2SE and J2ME.

My approach to exploring these J2ME pitfalls is to port a J2SE application to the J2ME platform. First, we take the most direct approach and then optimize our approach. Unfortunately, to demonstrate a nontrivial application (which we do) requires significant amounts of code. To reduce the size of the code listings, I have deleted simple, similar, and redundant code. Additionally, to avoid long runs of uninterrupted code, I have interspersed the code descriptions and explanations for all major methods instead of putting it all after the code.

The J2SE application we will port is called SwinginAmazonSearch. The main frame of the application is displayed in Figure 22.1.

The purpose of the application is to enable you to query Amazon.com's database using a Representational State Transfer (REST) approach. REST is an architectural style that describes how the World Wide Web (WWW) works. The term was coined by Roy Fielding in his Ph.D. thesis (available at http:// www.ics.uci.edu/~fielding/pubs /dissertation/top.htm). The Web uses Uniform Resource Identifiers (URIs) to retrieve representation and to change state by transferring to other representations. Amazon implemented this by encoding all query parameters in the URI and returning an XML document as the resulting representation.

The layout of the main interface in Figure 22.1 is divided into two halves: the top half to select and enter search terms and the bottom half to display the results after clicking "Search." After the results are displayed, you can examine more detail on a single entry by clicking "Details" or receive an additional "page" of results by clicking "Next Results." Amazon.com limits all queries to a single "page" of 10 hits and requires all queries to include a page number. Table 22.1 lists and defines the key parameters in an Amazon.com query.

Figure 22.1 A Swing-based Amazon search application.

Table 22.1 Amazon.com Query parameters

FIELD TYPE	FIELD NAME	FIELD VALUES	DEFINITION
Operation	KeywordSearch, BrowseNodeSearch, AsinSearch, UpcSearch, AuthorSearch, ArtistSearch, ActorSearch, DirectorSearch, ManufacturerSearch, ListManiaSearch, SimilaritySearch	Free text with keywords, authors, etc. corresponding to the search type.	The type of search to perform.
Mode	mode	baby, books, classical, dvd, electronics, garden, kitchen, magazines, music, pc-hardware, photo, software, toys, universal, vhs, videogames	A taxonomy of areas to search. Would be better to have been called category.
Return Type	type	lite, heavy	The DTD of the returned XML where the heavy version contains many more elements than the lite version.
Page #	page	An integer	The page of 10 hits to return (if available).
Format	f	xml or URI to an XSLT stylesheet	A return in XML or any format generated by the XSLT stylesheet.

Now let's examine the source code in Listing 22.1 that implements the application.

```
001: /* SwinginAmazonSearch.java */
002: package org.javapitfalls.item22;
003:
// - removed Import statements
015:
016: class SwinginAmazonSearch extends JFrame implements ActionListener
017: {
018:    public static final boolean debug;
019:
// - removed static block to set debug variable
```

Listing 22.1 SwinginAmazonSearch.java

```
035:
036:    public static final String CMD_SEARCH = "Search";
// - removed CMD_DETAILS, CMD_NEXT_TEN, and ELEMENT_PRODUCT_NAME Strings
040:
041:    JButton searchButton = new JButton(CMD_SEARCH);
042:    JButton detailsButton = new JButton(CMD_DETAILS);
043:    JButton nextResultsButton = new JButton(CMD_NEXT_TEN);
044:    JList results = new JList();
045:    JComboBox opsCombo = new JComboBox(AmazonHttpGet.legalOps);
046:    JComboBox modeCombo = new JComboBox(AmazonHttpGet.legalModes);
047:    JTextField targetTF = new JTextField(40);
048:    StatusPanel status = new StatusPanel();
049:    AmazonHttpGet getter = new AmazonHttpGet();
050:    int page = 1;
051:
052:    DocumentBuilderFactory dbf;
053:    DocumentBuilder db;
054:    Document doc;
055:    NodeList productNodes;
056:
```

Listing 22.1 *(continued)*

The class SwinginAmazonSearch extends a Swing JFrame and implements the Action Listener interface to receive events from the buttons on the main window. Lines 16-56 of the code contain the class declaration and class data members. The data members consist of some static constants, GUI components (`JButton`, `JList`, `JComboBox`), a reference to a class performing the HTTP networking functions called `AmazonHttpGet`, and references to XML parsing and the W3C Document class to manipulate the returned XML. It is important to note the use of static constants for all fixed `Strings` (like in line 36 and those deleted) in case the protocol changes at a later date. This then makes it easy to modify the protocol by only changing the string in one location instead of hunting down the occurrence of each `String` where it is used. Such "future proofing" is a good habit for J2SE/J2EE development, but one that wastes precious heap space in J2ME. We will see workarounds for this later. The constructor (below) creates and displays the main frame.

```
057:    public SwinginAmazonSearch() throws ParserConfigurationException
058:    {
059:       super("Amazon Search Tool");
060:
061:       dbf = DocumentBuilderFactory.newInstance();
062:       db = dbf.newDocumentBuilder();
063:
064:       // USE a vertical box for north Panel
065:       Box northWithTitle = new Box(BoxLayout.Y_AXIS);
066:       this.getContentPane().add("North", northWithTitle);
```

Listing 22.1 *(continued)*

```
067:
068:        // add label first
069:        // add label up north
070:        JPanel title = new JPanel(new FlowLayout(FlowLayout.CENTER));
071:        JLabel titleLabel = new JLabel("Amazon.com Search 
Assistant");
072:        titleLabel.setForeground(Color.green.darker());
073:        title.add(titleLabel);
074:        northWithTitle.add(title);
075:
076:        Box northPanel = new Box(BoxLayout.Y_AXIS);
077:        northWithTitle.add(northPanel);
078:        northPanel.setBorder(new TitledBorder(new EtchedBorder(), 
"Search Terms"));
079:
080:        // add operation drop down
081:        JPanel panel1 = new JPanel(new FlowLayout(FlowLayout.LEFT));
082:        northPanel.add(panel1);
083:        panel1.add(new JLabel("Operation:"));
084:        opsCombo.setEditable(false);
085:        panel1.add(opsCombo);
086:
// - removed adding most of the GUI components for brevity
// - removed adding status panel and WindowListener
132:
133:        this.setSize(600,400);
134:        this.setLocation(100,100);
135:        this.setVisible(true);
136:
137:     }
```

Listing 22.1 *(continued)*

The constructor (lines 57 to 137) instantiates the components, groups them in `JPanels`, adds them to `LayoutManagers`, and then sizes and shows the window. Unfortunately, the GUI components in the J2ME platform are different from (but similar to) those used in Swing:

```
139:     public void actionPerformed(ActionEvent aevt)
140:     {
141:        String command = aevt.getActionCommand();
142:        if (command.equals(CMD_SEARCH))
143:        {
144:           // check we have the valid parameters\
145:           String targets = targetTF.getText();
146:           if (targets.length() == 0)
147:           {
148:              status.setText("'Search For' text field cannot be 
empty.");
149:           }
```

Listing 22.1 *(continued)*

```
150:           else
151:           {
152:              try
153:              {
154:                 page = 1; // reset
155:                 doAmazonSearch(page, targets);
156:              } catch (MalformedURLException mue)
// - removed exception handling, simply displayed an error message
164:           }
165:        }
166:        else if (command.equals(CMD_DETAILS))
167:        {
168:           // popup a new window with the details for this product
169:           String selectedProductName = (String)
results.getSelectedValue();
170:
171:           // get the parent ELEMENT node of the node with a
ProductName with this Value
172:           if (productNodes != null)
173:           {
174:               Node n =
findNodeWithContent(productNodes,selectedProductName);
175:
176:               // get the Details element parent
177:               Node parent = n.getParentNode();
178:
179:               if (debug) System.out.println("parent: " +
parent.getNodeName());
180:               if (parent != null)
181:               {
182:                 // display a Details Window
183:                 new DetailsDialog(this, parent,
selectedProductName);
184:               }
185:           }
186:           else
187:           {
188:              status.setText("Internal error.  Try another search.");
189:           }
190:        }
// - removed handling NEXT_TEN for brevity
207:     }
```

Listing 22.1 *(continued)*

The `Action` event handler responds to the various button clicks generated by the GUI. Notice that I use the `String.equals()` method to determine which button was pressed. This will need to be changed, as string comparisons are slow. The two key events to respond to are the search request (line 142) and a details request (line 166). To respond to either event, some parameters are gathered (like the search term, line 145, or the XML node to report details on, line 174), and then `doAmazonSearch()` is invoked (line 155) or a new `DetailsDialog` window instantiated (line 183). Notice

the DetailsDialog is instantiated as an anonymous reference and not saved in a variable for reuse. Such an assumption of abundant memory and trust in the efficiency of the garbage collector is a pitfall for J2ME programming:

```
209:    private void doAmazonSearch(int page, String targets) throws
Exception
210:    {
211:       getter.newBaseURL(); // reset
212:
213:       // get the operation
214:       String op = (String) opsCombo.getSelectedItem();
215:
216:       // get the mode
217:       String mode = (String) modeCombo.getSelectedItem();
218:
219:       status.setText("Contacting Amazon.com...");
220:
221:       getter.addOperation(op, targets);
222:       getter.addMode(mode);
223:       getter.addType("lite");
224:       getter.addPage("" + page);
225:       getter.addFormat();
226:
227:       // GET it
228:       String response = getter.httpGet();
229:       if (response != null && response.length() > 0)
230:          status.setText("Received results! Formatting ...");
231:
232:       if (debug) System.out.println("response: " + response);
233:
234:       // parse the XML, extract ProductNames
235:       String [] productNames = null;
236:       ByteArrayInputStream bais = new
ByteArrayInputStream(response.getBytes());
237:       doc = db.parse(bais);
238:       if (doc != null)
239:          removeBlankNodes(doc.getDocumentElement());
240:
241:       productNodes =
doc.getElementsByTagName(ELEMENT_PRODUCT_NAME);
242:       if (productNodes != null)
243:       {
244:          int len = productNodes.getLength();
245:          productNames = new String[len];
246:          for (int i=0; i < len; i++)
247:          {
248:             Node n = productNodes.item(i);
249:             Node t = n.getFirstChild();
250:             if (t.getNodeType() == Node.TEXT_NODE)
251:                productNames[i] = t.getNodeValue();
252:          }
253:       }
```

Listing 22.1 *(continued)*

```
254:
255:       if (productNames != null && productNames.length > 0)
256:       {
257:          // populate the list
258:          results.setListData(productNames);
259:       }
// - removed else error condition handling for brevity
267:    }
// - removed utility method isBlank()
// - removed utility method removeBlankNodes()
// - removed utility method findNodeWithContent()
// - removed main() which merely Instantiates SwinginAmazonSearch
333: }
```

Listing 22.1 *(continued)*

The method `doAmazonSearch()` (lines 209 to 267) has four key functions:

- Format a URL.
- Send an HTTP GET request to xml.amazon.com.
- Parse the resulting XML to extract the product names.
- Populate the JList with the product names.

Both formatting the URL and "getting" it are performed in conjunction with the `AmazonHttpGet` class discussed later. It is important to note the modularity of the formatting operation for the URL displayed in lines 221 to 225. The formatting of a URL is broken into separate functions to assemble the URL in any order or length you want. Such modularity and the building of "generic code" will have to be sacrificed in our J2ME implementation for speed. At line 228, the `httpGet()` method is invoked which returns a `String` containing an XML document. Here is a portion of a sample return document:

```
<Details url="http://www.amazon.com/exec/obidos/redirect?tag=webservices-
20%26creative=D3AG4L7PI53LPH%26camp=2025%26link_code=xm2%26path=ASIN/047
1237515">
   <Asin>0471237515</Asin>
   <ProductName>More Java Pitfalls: 50 New Time-Saving Solutions and ⮐
Workarounds</ProductName>
   <Catalog>Book</Catalog>
   <Authors>
      <Author>Michael C. Daconta</Author>
      <Author>Kevin T. Smith</Author>
      <Author>Donald Avondolio </Author>
      <Author>W. Clay Richardson</Author>
   </Authors>
   <ReleaseDate>03 February, 2003</ReleaseDate>
   <Manufacturer>John Wiley &amp; Sons</Manufacturer>
   <ListPrice>$40.00</ListPrice>
   <OurPrice>$40.00</OurPrice>
</Details>
```

The XML response is then fed into the standard JDK XML Parser, and a Document Object Model (DOM) is constructed (the `org.w3c.Document` class) at line 237. All the elements with a tag of "ProductName" are retrieved (line 241) and their text content extracted. Since the XML document is guaranteed to be only 10 products at a time, this Swing application can comfortably construct a DOM even though it requires more memory than a SAX Parser or Pull Parser approach; however, this approach proves to use too much memory in our J2ME port. Several utility methods were created to search and manipulate the DOM, like `removeBlankNodes()` and `findNodeWithContent()`, but were deleted since they do not pertain to J2ME pitfalls:

```
335: class DetailsDialog extends JDialog
336: {
337:    JTextArea textArea = new JTextArea(5,60);
338:    public DetailsDialog(Frame f, Node detailsNode, String ⮐
productName)
339:    {
340:       super(f, "Details for " + productName, false);
341:
342:       getContentPane().setLayout(new BorderLayout(2,2));
343:       JScrollPane scroller = new JScrollPane(textArea);
344:       getContentPane().add("Center", scroller);
345:
346:       // initialize the text Area
347:       textArea.setEditable(false);
348:       textArea.setBackground(Color.lightGray);
349:
350:       NodeList children = detailsNode.getChildNodes();
351:       int len = children.getLength();
352:       for (int i=0; i < len; i++)
353:       {
354:          // display element children with a text node
355:          Node child = children.item(i);
356:          if (child.getNodeType() == Node.ELEMENT_NODE)
357:          {
358:             Node txt = child.getFirstChild();
359:             if (txt != null && txt.getNodeType() == Node.TEXT_NODE)
360:             {
361:                String label = child.getNodeName();
362:                String value = txt.getNodeValue();
363:
364:                if (value.length() > 0)
365:                   textArea.append("" + label + ": " + value +
"\n");
366:             }
367:          }
368:       }
369:
// - removed setting window size and location
372:       this.setVisible(true);
373:    }
374: }
```

Listing 22.1 *(continued)*

```
375:
// - removed StatusPanel Class for brevity
439:
```

Listing 22.1 *(continued)*

The `DetailsDialog` presents a frame with a single `JTextArea` where each tag name becomes the label (line 361) and each node value of the XML element is presented as the labels value separated by a colon (line 365). This approach relies on detailed knowledge of the XML format.

Now let's examine Listing 22.2 that presents the utility class, `AmazonHttpGet`, that performs the networking operations of the application.

```
001: package org.javapitfalls.item22;
002:
003: import java.net.*;
004: import java.io.*;
005:
006: class AmazonHttpGet
007: {
008:    public static final boolean debug;
009:
// - removed static block that sets debug variable
025:
026:    public static final String DEVTAG = "YOUR-DEV-TAG-HERE";
027:    public static final String [] legalOps = { "KeywordSearch",
// - removed other keywords listed In Table 22.1
"SimilaritySearch", };
031:
032:    public static final String OP_KEYWORD_SEARCH = "KeywordSearch";
033:    public static final String OP_BROWSE_NODE_SEARCH =
"BrowseNodeSearch";
// - removed remaining "operation" constants
043:
044:    public static final String [] legalModes = { "baby", "books",
// - removed other keywords listed In Table 22.1
047:                                  "videogames", };
048:
049:    public static final String MODE_BABY = "baby";
050:    public static final String MODE_BOOKS = "Books";
// - removed remaining "mode" constants
065:
066:    public static final String KEYWORD_MODE = "mode";
// - removed other KEYWORD constants
069:
070:    public static final String TYPE_LIGHT = "lite";
071:    public static final String TYPE_HEAVY = "heavy";
072:
// - removed stringExists() utility method
092:
093:    private StringBuffer urlBuf;
```

Listing 22.2 AmazonHttpGet.java *(continued)*

This class was designed with modularity and reusability in mind. The class data members consist mostly of static constants (though many have been removed for brevity). The key data members are the DEVTAG provided when you register at Amazon.com and the urlBuf StringBuffer. In fact, this class does not go far enough in terms of composability of the various queries and additional checking of parameter combinations. Unfortunately, as you will see, most of this flexibility will be eliminated in the J2ME port to reduce the number of method invocations. Throughout this pitfall, you should notice the recurring theme of a mind set shift required to program for small devices. The StringBuffer contains a string representation of the URL we will GET:

```
// - removed urlBuf accessor method
097:
098:    public AmazonHttpGet()
099:    {
100:       newBaseURL();
101:    }
102:
103:    public void newBaseURL()
104:    {
105:       urlBuf = new StringBuffer("http://xml.amazon.com/
onca/xml?v=1.0&t=webservices-20&dev-t=" + DEVTAG);
106:    }
107:
108:    public boolean validOp(String op)
109:    {
110:       if (stringExists(op, legalOps, false))
111:          return true;
112:       else
113:          return false;
114:    }
115:
// - removed validMode() as it is similar to validOp()
// - removed validType() as it is similar to validOp()
// - removed validPage() as it is similar to validOp()
145:
146:    public void addOperation(String operation, String target)
throws MalformedURLException
147:    {
148:       // validate the operation
149:       if (validOp(operation))
150:       {
151:          urlBuf.append('&');
152:          urlBuf.append(operation);
153:          urlBuf.append('=');
154:          if (target != null)
155:          {
156:             target.trim();
157:             target = replaceString(target, " ", "%20", 0);
158:
159:             urlBuf.append(target);
```

Listing 22.2 *(continued)*

```
160:          }
161:          else
162:              throw new MalformedURLException("Invalid target");
163:       }
164:       else
165:          throw new MalformedURLException("Invalid operation.");
166:    }
167:
// - removed addMode() as it is similar to addOperation()
// - removed addType() as it is similar to addOperation()
// - removed addPage() as it is similar to addOperation()
206:
207:    public void addFormat()
208:    {
209:       urlBuf.append("&f=xml"); // TBD: allow XSLT stylesheet
210:    }
211:
// - removed replaceString() utility method
```

Listing 22.2 *(continued)*

The formatting of the URL involves validating and then appending name/value pairs to the `urlBuf StringBuffer`. The `addOperation()` (line 146), `addMode()`, and `addType()` methods add the parameters discussed in Table 22.1 to the `StringBuffer`. Two utility methods were removed for brevity: `stringExists()` checked for the existence of a `String` in an array of `Strings` and `replaceString()` replaces a portion of a `String` with supplied replacement text:

```
233:    public String httpGet() throws IOException
234:    {
235:       if (debug) System.out.println("URL: " + urlBuf.toString());
236:
237:       // Create a URL object
238:       URL url = new URL(urlBuf.toString());
239:       // get the connection object
240:       URLConnection con = url.openConnection();
241:
242:       con.setDoOutput(true);
243:       con.setUseCaches(false);
244:
245:       InputStream in = con.getInputStream();
246:       BufferedReader br = new BufferedReader(new
InputStreamReader(in));
247:
248:       // read response
249:       String line = null;
250:       StringWriter sw2 = new StringWriter();
251:       PrintWriter pw2 = new PrintWriter(sw2);
252:       while ( (line = br.readLine()) != null)
```

Listing 22.2 *(continued)*

```
253:       {
254:         pw2.println(line);
255:       }
256:
257:       String response = sw2.toString();
258:       return response;
259:    }
260:
// - removed main() method (used for testing)
323: }
```

Listing 22.2 *(continued)*

The `httpGet()` method instantiates a URL object from the `urlBuf StringBuffer` (line 238), opens a connection to the URL (line 240) and then reads lines of text from the Web server (line 252) and prints them to a `PrintWriter/StringWriter` buffer. Lastly, the contents of the `StringWriter` are retrieved as a single `String` (line 257).

To examine the API differences between J2SE and J2ME, we attempt a straightforward port of the code to J2ME, changing as little code as necessary. Listing 22.3 is our direct port of SwinginAmazonSearch.java to J2ME. We will examine specific API differences as we cover each section of code.

```
001: package org.javapitfalls.item22;
002:
003: import java.io.*;
004: import java.util.*;
005: import javax.microedition.midlet.*;
006: import javax.microedition.lcdui.*;
007: import org.kxml.*;
008: import org.kxml.kdom.*;
009: import org.kxml.parser.*;
010:
011: public class BadMicroAmazonSearch extends MIDlet implements
CommandListener
012: {
013:    public static final String CMD_SEARCH = "Search";
014:    public static final String CMD_DETAILS = "Details";
015:    public static final String CMD_NEXT_TEN = "More";
// - removed other static constant Strings
019:
020:    // commands
021:    private Command searchCommand;
022:    private Command detailsCommand;
// - removed other Command references for brevity
026:
027:    // display
028:    private Display display;
029:
030:    // screens
```

Listing 22.3 BadMicroAmazonSearch.java

```
031:    private Form searchScreen;
032:    private List resultsScreen;
033:    private TextBox detailsScreen;
034:
035:    // screen components
036:    ChoiceGroup opsChoice;
037:    ChoiceGroup modeChoice;
038:    TextField targetTF;
039:    BadMicroAmazonHttpGet getter = new BadMicroAmazonHttpGet();
040:    int page = 1;
041:
042:    Vector productNodes;
043:
044:    boolean debug = true;
045:    Timer ticker = new Timer();
046:
```

Listing 22.3 *(continued)*

The class `BadMicroAmazonSearch` extends `MIDlet` and implements `CommandListener` (instead of implementing `ActionListener`). A MIDlet is a Mobile Information Device Profile (MIDP) application. The `MIDlet` is an abstract class with three abstract methods that must be overridden by the subclass: `startApp()`, `pauseApp()`, and `destroyApp()`. In this sense, a MIDlet is similar to an applet in that it has a lifecycle controlled by an external Management application.

Before we examine the details of the class, notice the import statements: two packages are the same as J2SE packages (`java.io` and `java.util`), although they are limited in scope, and the two javax packages are new (`javax.midlet`, which has the `MIDlet` class, and `javax.lcdui`, which has the GUI classes). The `kxml` package is an open-source package available at http://kxml.enhydra.org/. We will discuss the `kxml` package in more detail later. Now let's examine the GUI differences apparent in the `BadMicroAmazonSearch` class definition. Since we are dealing with such a small screen size, as shown in Figure 22.2, I divided the single Swing Frame into three "displayable" screens (`Screen` objects).

Figure 22.2 MIDP screen size on a Motorola phone.

Motorola and the Motorola logo are trademarks of Motorola, Inc.

A major difference between Swing and J2ME GUIs is that there are no `Frames` in J2ME. In essence, there is only a single `Display` (line 28) and you can switch the `Display` to any subclass of `Screen`. Our `Screen` subclasses are a `Form` for the search screen, a `List` for the results screen, and a `TextBox` for the details screen. One other important difference is that there are no `JButton` or `Button` classes in J2ME; instead, there is a `Command` class. You add your commands to the appropriate screen. You will also notice that we shortened the `"Next Results"` label to `"More"` (line 15) to take into account the smaller screen size. The last difference is that `ComboBox` has been replaced by `ChoiceGroup` (lines 36 and 37).

```
047:    public BadMicroAmazonSearch()
048:    {
049:       // Create GUI components here...
050:       display = Display.getDisplay(this);
051:
052:       // commands
053:       searchCommand = new Command(CMD_SEARCH, Command.SCREEN, 1);
054:       detailsCommand = new Command(CMD_DETAILS, Command.SCREEN, 1);
// - removed the Instantiation of other Commands
058:
059:       // Create 3 screens: 1. Search, 2. Results, 3. Details
060:       // search form
061:       searchScreen = new Form("Search Terms");
062:       opsChoice = new ChoiceGroup("Operation:", Choice.EXCLUSIVE, 
BadMicroAmazonHttpGet.legalOps, null);
063:       searchScreen.append(opsChoice);
064:       targetTF = new TextField("Search For:", "", 20, 
TextField.ANY);
065:       searchScreen.append(targetTF);
066:       modeChoice = new ChoiceGroup("Category:", Choice.EXCLUSIVE, 
BadMicroAmazonHttpGet.legalModes, null);
067:       modeChoice.setSelectedIndex(1, true);
068:       searchScreen.append(modeChoice);
069:       searchScreen.addCommand(searchCommand);
070:       searchScreen.addCommand(exitCommand);
071:       searchScreen.setCommandListener(this);
072:
073:       // results list
074:       resultsScreen = new List("Results", List.EXCLUSIVE);
075:       resultsScreen.addCommand(detailsCommand);
// - removed adding other commands to resultsScreen
080:
081:       // details text box
082:       detailsScreen = new TextBox("Details", "", 1024, 
TextField.ANY);
// - removed adding commands to detailsScreen
086:    }
```

Listing 22.3 *(continued)*

The constructor for `BadMicroAmazonSearch` performs three key functions:

- Gets the `Display` object via `Display.getdisplay()` (line 50)

- Instantiates the `Commands` (lines 53 to 57) and `Screens` (lines 61, 74, and 82)
- Adds the `Commands` and subcomponents (for the `Form`) to the screens

There are many minor differences in a MIDP GUI compared to a J2SE GUI, such as not adding the screens to the `Display` (as we would to a `JFrame`); instead, we call `Display.setCurrent()` to a `Display` object (as we will in the `startApp()` method below):

```
088:    public void startApp()
089:    {
090:       display.setCurrent(searchScreen);
091:    }
092:
// - removed pauseApp() and destroyApp() as they did nothing
102:
103:    public void commandAction(Command c, Displayable s)
104:    {
105:       String cmd = c.getLabel();
106:       if (cmd.equals(CMD_EXIT))
107:       {
108:          destroyApp(false);
109:          notifyDestroyed();
110:       }
111:       else if (cmd.equals(CMD_SEARCH))
112:       {
113:           // check we have the valid parameters\
114:          String targets = targetTF.getString();
115:          if (targets.length() == 0)
116:          {
117:             display.setCurrent(new Alert("Search Error", "'Search 
For' text field cannot be empty.", null, AlertType.ERROR));
118:          }
119:          else
120:          {
121:             try
122:             {
123:                page = 1; // reset
124:                doAmazonSearch(page, targets);
125:             } catch (Exception e)
126:               {
127:                 e.printStackTrace();
128:                 display.setCurrent(new Alert("SearchError", 
"ERROR: reason: " + e.toString(),null, AlertType.ERROR));
129:               }
130:          }
131:       }
// - removed handling CMD_DETAILS for brevity
136:    }
```

Listing 22.3 *(continued)*

The `commandAction()` method is analogous to the `actionPerformed()` method in J2SE. Note that instead of the `getActionCmd()`, we call `getLabel()` to retrieve

the text label of the `Command`. Although this is similar to the method used in the Swing application, it is a classic pitfall, since it is much slower than comparing the `Command` references passed in with the references of our predefined `Command` objects (like `searchCommand`). We make this optimization in the next version of the program. The rest of the method is nearly identical to the `actionPerformed()` method, except the error reporting requires the creation of an Alert screen. Although here in this "bad" version we create temporary objects (lines 117 and 128), hopefully, you noticed this waste and the opportunity for optimization:

```
138:    private void doAmazonSearch(int page, String targets) throws
Exception, IOException
139:    {
140:        ticker.reset("Started Timer in doAmazonSearch()");
141:        getter.newBaseURL(); // reset
142:
143:        // get the operation
144:        int idx = opsChoice.getSelectedIndex();
145:        String op = (String) opsChoice.getString(idx);
146:
// - removed getting the mode as it is similar to getting the op
150:
// - removed getter.addXXX methods -- no change.
156:
157:        // GET it
158:        byte [] response = getter.httpGet();
159:        System.out.println("Have response. Size is: " +
response.length);
160:
161:        // parse the XML, extract ProductNames
162:        // Kxml required ~ 200k for a full parse.
163:        String [] productNames = null;
164:        ByteArrayInputStream bais = new
ByteArrayInputStream(response);
165:        InputStreamReader isr = new InputStreamReader(bais);
166:        XmlParser parser = new XmlParser(isr);
167:        Document doc = new Document();
168:        doc.parse(parser);
169:
170:        productNodes = new Vector();
171:        getProductNames(doc.getRootElement(), productNodes);
172:        if (productNodes != null)
173:        {
174:           int len = productNodes.size();
```

Listing 22.3 *(continued)*

```
175:          System.out.println("# of products found: " + len);
176:          productNames = new String[len];
177:          for (int i=0; i < len; i++)
178:          {
179:             Node n = (Node) productNodes.elementAt(i);
180:             productNames[i] = n.getText();
181:          }
182:       }
183:
184:       if (productNames != null && productNodes.size() > 0)
185:       {
186:          // populate the list
187:          for (int i=0; i < productNames.length; i++)
188:             resultsScreen.append(productNames[i], null);
189:
190:          // set the display to the results
191:          display.setCurrent(resultsScreen);
192:       }
// - removed the else block for brevity
200:       ticker.printStats("Method doAmazonSearch()");
201:    }
202:
203:    public void getProductNames(Node root, Vector v)
204:    {
205:       int cnt = root.getChildCount();
206:       for (int i=0; i < cnt; i++)
207:       {
208:          Object o = root.getChild(i);
209:          if (o instanceof Node)
210:          {
211:             Node n = (Node) o;
212:             String name = n.getName();
213:             if (name.equals(ELEMENT_PRODUCT_NAME))
214:             {
215:                v.addElement(n);
216:             }
217:
218:             // element?
219:             if (n.getChildCount() > 0)
220:                getProductNames(n, v);
221:          }
222:       }
223:    }
224: }
```

Listing 22.3 *(continued)*

The doAmazonSearch()method is similar to its Swing counterpart with a few exceptions. For example, you cannot directly get a selected item (like with the getSelectedItem() method) from a ChoiceGroup or List; instead, you must get the index (line 144) and then call getString() (line 144). Such minor API changes can be frustrating, though in this case the purpose is to eliminate the need for casting (in this direct port it was accidentally left in but is corrected in the next version). On line 158 notice that the httpGet() method returns a byte array (which is required by the kxml parser). Lines 164 to 167 represent the parsing of the XML document using the kxml package. At line 171, we call the utility method, getProductNames(), to recursively traverse the document tree and extract the product Nodes. Unfortunately, this was necessary because the kdom package does not have a getElementsByTagName() method.

Like the minor API changes in the javax.microedition packages, the kdom package has a slightly different API than the w3c DOM package. Such API changes only cause a serious pitfall when such a change eliminates any implicit guarantee of the former abstraction. For the DOM, a tree of Nodes represents the "flattened view" where every object is of type Node. This uniformity makes traversal easy and consistent. Unfortunately, kxml breaks the metaphor by mixing Nodes and other objects (like Strings). This nonuniformity led to runtime ClassCastExceptions (due to the assumption of uniformity—a classic pitfall) and required explicit testing (line 209). Additionally, the kxml changed the method names from getNodeName() to getName()and from getNodeValue() to getText().

Figure 22.3 displays the Network Monitor application, which is part of Sun Microsystems J2ME Wireless Toolkit. This toolkit allows you to emulate J2ME on a personal computer or workstation and enabled the writing and testing of this pitfall. Sun Microsystems did a real service to developers in delivering such a high-quality emulation environment for Java programmers. You can download the kit from http://java.sun.com/products/j2mewtoolkit/.

The Network Monitor application captures all communication between your MIDlet and the Web server. The left pane shows a tree with all HTTP requests and responses. Clicking on a request or response displays the details of the communication in the right pane in both hexadecimal and ASCII. Now we can examine the port of AmazonHttpGet to the J2ME platform in Listing 22.4.

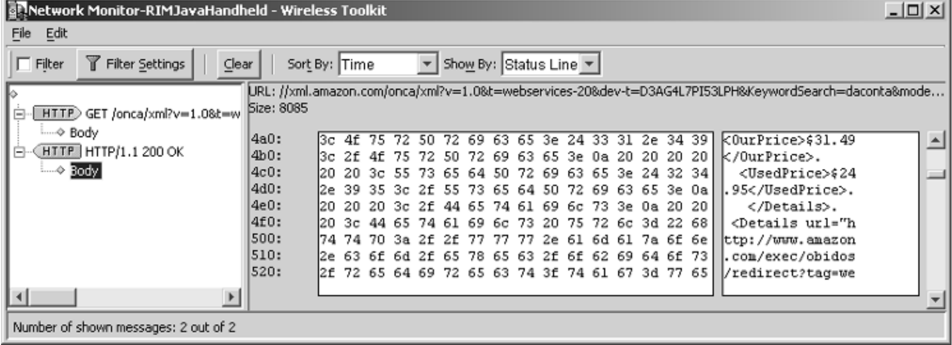


Figure 22.3 The J2ME Wireless Toolkit Network Monitor.

```
001: /* BadMicroAmazonHttpGet.java */
002: package org.javapitfalls.item22;
003:
004: import javax.microedition.io.*;
005: import java.io.*;
006: import java.util.*;
007:
008: public class BadMicroAmazonHttpGet
009: {
// - deleted static constants -- No change
056:
057:    static Timer ticker = new Timer();
058:
// - deleted stringExists() method -- No change
076:
077:    private StringBuffer urlBuf;
078:
079:    public StringBuffer getUrlBuf()
080:    { return urlBuf; }
081:
082:    public BadMicroAmazonHttpGet()
083:    {
084:       newBaseURL();
085:    }
086:
// - deleted method newBaseURL() -- No change
// - deleted all validation methods -- No change
// - deleted all addXXX methods -- No change
// - deleted replaceString() -- No change
216:
217:    public byte [] httpGet() throws IOException
218:    {
219:       ticker.reset("Started Timer in httpGet()");
220:       // get the connection object
221:       String surl = urlBuf.toString();
222:       surl.trim();
223:       System.out.println("url: " + surl);
224:
225:       HttpConnection con = (HttpConnection) Connector.open(surl);
226:       int respCode = con.getResponseCode();
227:       System.out.println("Response code: " + respCode);
228:
229:       InputStream in = con.openInputStream();
230:       ByteArrayOutputStream baos = new ByteArrayOutputStream();
231:
232:       // read response
233:       int b = 0;
234:       while ( (b = in.read()) != -1)
235:       {
```

Listing 22.4 BadMicroAmazonHttpGet.java *(continued)*

```
236:         baos.write(b);
237:       }
238:
239:       ticker.printStats("Method httpGet()");
240:       return baos.toByteArray();
241:    }
// - deleted main() method -- No change.
```

Listing 22.4 *(continued)*

In the `BadMicroAmazonHttpGet` class, all of the URL formatting methods remained unchanged (and thus were removed for brevity); however, the `httpGet()` method underwent significant changes. The porting changes are as follows:

- There is no `java.net package`; instead, the networking classes are in `javax.microedition.io` (line 4). Not only does this confuse it with the J2ME version of `java.io`, this limits future differentiation between IO and networking support in the platform. This is a prime example of change for the sake of change that slows productivity by forcing a context switch without good reason.
- There is no `MalformedURLException` or URL class; instead, the `HttpConnection` class accepts a `String` (line 225).
- There is no `URLConnection` class; instead, you use an `HttpConnection` (line 225).
- There is no `getInputStream()` method for the connection; instead, you use `openInputStream()`. Another example of useless incompatibility.
- There was no `BufferedReader` class, so instead we read in bytes (instead of Strings) and wrote into a `ByteArrayOutputStream` (line 236). This then led to returing a byte array (line 240).

With the memory limit set at 128 KB, a run of BadMicroAmazonSearch produces:

```
Started Timer in doAmazonSearch(). Free Memory: 73076
Started Timer in httpGet(). Free Memory: 71628
url: http://xml.amazon.com/onca/xml?v=1.0&t=webservices-20&dev-
t=D3AG4L7PI53LPH&KeywordSearch=daconta&mode=books&type=lite&page=1&f=xml
Response code: 200
Method httpGet(): 5678. Free Memory: 63924
Have response. Size is: 8085
java.lang.OutOfMemoryError at
javax.microedition.lcdui.Display$DisplayAccessor.commandAction(+165)
        at com.sun.kvem.midp.lcdui.EmulEventHandler$EventLoop.run(+459)
```

When I increased the memory to 200 KB the program was able to run and produced the following:

```
Started Timer in doAmazonSearch(). Free Memory: 141096
Started Timer in httpGet(). Free Memory: 139648
url: http://xml.amazon.com/onca/xml?v=1.0&t=webservices-20&dev-
t=D3AG4L7PI53LPH&KeywordSearch=daconta&mode=books&type=lite&page=1&f=xml
Response code: 200
Method httpGet(): 5718. Free Memory: 139152
Have response. Size is: 8085
# of products found: 8
Method doAmazonSearch(): 36082. Free Memory: 46240
Started Timer in doAmazonSearch(). Free Memory: 65044
```

Even though the BadMicroAmazonSearch ran within 200 KB, you should also notice the poor performance of the method (a noticeably long pause after selecting the Search command). The output of the run shows that the method took 36,082 milliseconds to run. The Wireless Toolkit also provides a Memory Monitor application, as shown in Figure 22.4. As you can see, the large peak in the memory graph occurs when the `kxml` package is parsing the XML document.

Unfortunately, we could not afford to allow the application to run within 200 KB in order to run it in the Palm emulator. At the time of this writing, the Palm emulator only allowed a Java application to have 64 KB of memory. Figure 22.5 shows our goal with the final code running under the Palm emulator.

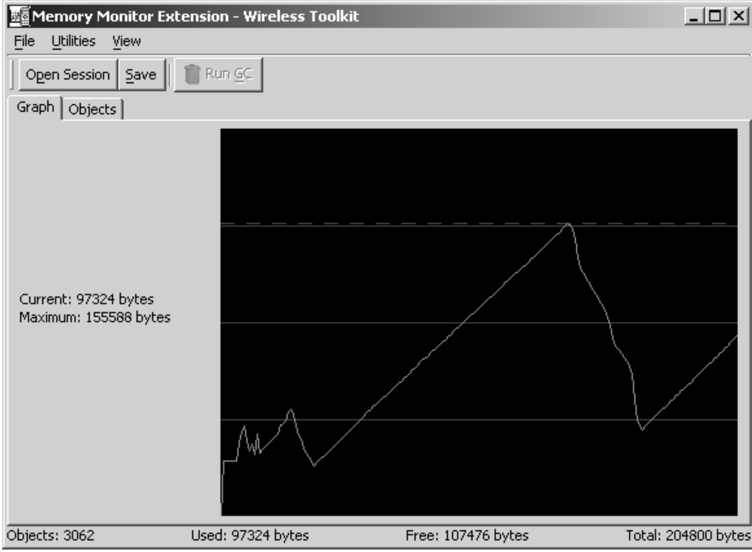


Figure 22.4 The Wireless Toolkit Memory Monitor.

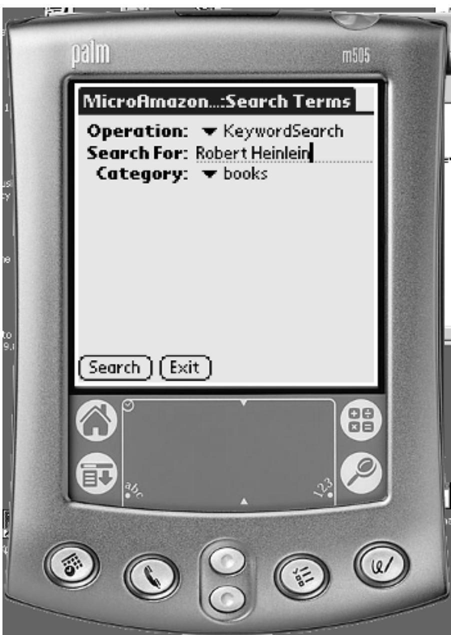


Figure 22.5 MicroAmazonSearch running in a Palm emulator.
© 2002 Palm, Inc. All rights reserved.

Now we are ready to optimize our J2ME application to get it to have both adequate performance and memory consumption. Listing 22.5 is the optimized code for MicroAmazonSearch.java. We will not discuss the functionality of MicroAmazonSearch, since that has been covered in the preceding pages; instead, we will focus only on the optimizations.

```
001: /* MicroAmazonSearch.java */
002: package org.javapitfalls.item22;
003:
004: import java.io.*;
005: import java.util.*;
006: import javax.microedition.midlet.*;
007: import javax.microedition.lcdui.*;
008:
009: public class MicroAmazonSearch extends MIDlet implements
CommandListener
010: {
011:    public static final int MAX_RECORDS = 10;
012:
```

Listing 22.5 MicroAmazonSearch.java

```
013:    // commands
014:    private Command searchCommand;
015:    private Command detailsCommand;
// - removed additional Command references for brevity
019:
020:    // Alerts
021:    Alert searchAlert;
022:    Alert detailAlert;
// - removed other Alert references for brevity
025:
026:    // display
027:    private Display display;
028:
029:    // screens
030:    private Form searchScreen;
031:    private List resultsScreen;
032:    private TextBox detailsScreen;
033:
034:    // screen components
035:    ChoiceGroup opsChoice;
036:    ChoiceGroup modeChoice;
037:    TextField targetTF;
038:    int page = 1;
039:
040:    Vector products;
041:    String xmlBuf;
042:    int [] detailIndexes;
043:    boolean updateIndexes;
044:
045:    boolean debug = true;
046:    Timer ticker = new Timer();
```

Listing 22.5 *(continued)*

Here are the optimizations in the class definition:

Guess the Size of `Vectors`. Resizing `Vectors` is expensive. This is demonstrated by using the constant in line 11.

Use Local Variables. Local variables are accessed faster than class members. You will notice that we have eliminated all of the public static `Strings`. This optimization could be used further in this code.

Avoid `String` Comparisons. Notice in lines 14 to 18 that the `Command` references are declared as class members, and these will be compared against in the event handler instead of comparing `Strings`.

Avoid Temporary Objects. In lines 21 to 24 we use class data members for the Alert references and thus reuse the objects after instantiating them lazily.

```
048:    public MicroAmazonSearch()
049:    {
050:       final String [] legalOps = { "KeywordSearch", 
"BrowseNodeSearch", "AsinSearch",
051:                             "UpcSearch", "AuthorSearch", 
"ArtistSearch",
052:                             /* reduce to work in 64k ...
053:                             "ActorSearch", "DirectorSearch", 
"ManufacturerSearch",
054:                             "ListManiaSearch", "SimilaritySearch",
055:                            */
056:                            };
057:
058:       final String [] legalModes = { "baby", "books", 
"classical", "dvd", "electronics",
059:                             /* reduce to conserve memory (< 64k)....
060:                             "garden", "kitchen", "magazines", 
"music", "pc-hardware",
061:                             "photo", "software", "toys", 
"universal", "vhs",
062:                             "videogames",
063:                             */
064:                            };
065:       final String CMD_SEARCH = "Search";
// - removed remaining final Strings for brevity
070:
071:       // Create GUI components here...
072:       display = Display.getDisplay(this);
073:
074:       // commands
075:       searchCommand = new Command(CMD_SEARCH, Command.SCREEN, 1);
076:       detailsCommand = new Command(CMD_DETAILS, Command.SCREEN, 
1);
077:       nextResultsCommand = new Command(CMD_NEXT_TEN, 
Command.SCREEN, 1);
078:       backCommand = new Command(CMD_BACK, Command.SCREEN, 2);
079:       exitCommand = new Command(CMD_EXIT, Command.SCREEN, 2);
080:
081:       // Create 3 screens: 1. Search, 2. Results, 3. Details
082:       // search form
083:       searchScreen = new Form("Search Terms");
// removed the construction of the searchScreen — no change.
094:
095:       // other screens, lazy instantiated
096:    }
```

Listing 22.5 *(continued)*

The MicroAmazonSearch constructor has three optimizations:

Use Local Variables. The arrays for operations (lines 50 to 63) and modes were moved to become local variables so the memory is reclaimed at method return.

Declare Variables and MethodsFinal. Final references are accessed faster and declaring both final and static is the fastest. Both the arrays and line 65 demonstrate this.

UseLazy Instantiation. Line 95 no longer contains the other screens, as we wait until they are needed by the user to create them.

```
098:    public void startApp()
099:    {
100:       display.setCurrent(searchScreen);
101:       // clean up as User decides what to do
102:       System.gc();
103:    }
104:
// - removed pauseApp() and destroyApp as they do nothing
114:
115:    public void commandAction(Command c, Displayable s)
116:    {
117:       String targets = null;
118:       try
119:       {
120:          if (c == exitCommand)
121:          {
122:             destroyApp(false);
123:             notifyDestroyed();
124:          }
125:          else if (c == searchCommand)
126:          {
127:              // check we have the valid parameters\
128:             targets = targetTF.getString();
129:             if (targets.length() == 0)
130:             {
131:                // lazy instantiation!
132:                if (searchAlert == null)
133:                   searchAlert = new Alert("Search Error", 
"'Search For' text field cannot be empty.", null, AlertType.ERROR);
134:                display.setCurrent(searchAlert);
135:             }
136:             else
137:             {
138:                page = 1; // reset
139:                xmlBuf = null;
140:                updateIndexes = true;
141:                // memory intensive, so get as much as we can
142:                System.gc();
143:                doAmazonSearch(page, targets);
144:                System.gc();
145:             }
146:          }
147:          else if (c == detailsCommand)
148:          {
149:             // get item selected in list
```

Listing 22.5 *(continued)*

```
150:             int selected = resultsScreen.getSelectedIndex();
151:             if (selected >= 0)
152:             {
153:                String product = 
resultsScreen.getString(selected);
154:                showDetails(product, xmlBuf);
155:             }
156:             else
157:             {
158:                if (detailAlert == null)
159:                   detailAlert = new Alert("Error", "Must 
select a product to see details.", null, AlertType.ERROR);
160:                display.setCurrent(detailAlert);
161:             }
162:          }
// - removed handling of backCommand and nextResultsCommand for brevity
182:       } catch (Throwable t)
183:         {
184:           if (debug) t.printStackTrace();
185:           if (genericAlert == null)
186:             genericAlert = new Alert("Error", "ERROR: reason: 
" + t.toString(),null, AlertType.ERROR);
187:           else
188:             genericAlert.setString("ERROR: reason: " + 
t.toString());
189:           display.setCurrent(genericAlert);
190:           if (t instanceof OutOfMemoryError)
191:           {
192:              Runtime r = Runtime.getRuntime();
193:              long free = 0, freed = 0;
194:              int trys = 0;
195:              while ( (freed += (r.freeMemory() - free)) > 0 && 
trys < 20)
196:              {
197:                 free = r.freeMemory();
198:                 System.gc();
199:                 trys++;
200:              }
201:              if (debug) System.out.println("Freed " + freed + 
" bytes.");
202:
203:           }
204:         }
205:    }
```

Listing 22.5 *(continued)*

The event handler demonstrates three optimizations:

Reduce `String` Comparisons. Lines 120 and 125 demonstrate comparing against a reference instead of using `String` comparison.

Lines 132 and 185 again demonstrate using lazy instantiation.

Handle OutOfMemoryError. This error is much more common in small footprint devices and must be handled explicitly, like in lines 190 to 200.

```
207:    private final void showDetails(String product, String xmlBuf)
208:    {
209:       final String ELEMENT_DETAILS = "Details";
210:       final String ELEMENT_AUTHORS = "Authors";
211:
212:       // lazy instantiation
213:       if (detailsScreen == null)
214:       {
215:          // details text box
216:          detailsScreen = new TextBox("Details", "", 1024, 
TextField.ANY);
// - removed adding the Commands to detailsScreen — no change.
220:       }
221:
222:       ticker.reset("Started Timer in showDetails()");
223:
224:       // clear the text box
225:       detailsScreen.delete(0, detailsScreen.size());
226:
227:       // display tagName : value
228:       // first, find the product
229:       int prodIdx = xmlBuf.indexOf(product);
230:       if (prodIdx >= 0)
231:       {
232:          int productCount = products.size();
233:          if (updateIndexes)
234:          {
235:             if (detailIndexes == null)
236:                detailIndexes = new int[MAX_RECORDS];
237:             int tmpIdx = 0;
238:             // this loops needs to count up
239:             for (int i=0;i < productCount; i++)
240:             {
241:                String tgt = "<" + ELEMENT_DETAILS;
242:                detailIndexes[i] = xmlBuf.indexOf(tgt, tmpIdx);
243:                tmpIdx = detailIndexes[i] + 1;
244:             }
245:          }
246:
247:          updateIndexes = false;
248:          int detailIdx = -1;
249:          for (int i=productCount-1; i >= 0; i—)
250:          {
251:             if (detailIndexes[i] < prodIdx)
252:             {
253:                detailIdx = i;
254:                break;
255:             }
256:          }
```

Listing 22.5 *(continued)*

```
257:
258:          int startIdx = detailIndexes[detailIdx];
259:          int endIdx = ( (detailIdx + 1) < detailIndexes.length
)? detailIndexes[detailIdx + 1] : xmlBuf.length();
260:
261:          int traverseIdx = startIdx + 1;
262:          while (traverseIdx < endIdx)
263:          {
264:             // find a tag
265:             int tagStartIdx = xmlBuf.indexOf('<', traverseIdx);
266:             int tagEndIdx = xmlBuf.indexOf('>', tagStartIdx);
267:             String tag = xmlBuf.substring(tagStartIdx+1,
tagEndIdx);
268:             if (tag.equals("/" + ELEMENT_DETAILS))
269:                break;
270:
271:
272:             // now get the tag contents
273:             int endTagStartIdx = xmlBuf.indexOf("</" + tag,
tagEndIdx);
274:             String contents = xmlBuf.substring(tagEndIdx + 1,
endTagStartIdx);
275:
276:             if (!tag.equals(ELEMENT_AUTHORS))
277:             {
278:                detailsScreen.insert(tag + ":",
detailsScreen.size());
279:                detailsScreen.insert(contents + "\n",
detailsScreen.size());
280:             }
281:
282:             traverseIdx = endTagStartIdx+1;
283:          }
284:
285:          // set the display to the results
286:          display.setCurrent(detailsScreen);
287:       }
288:       ticker.printStats("Method showDetails()");
289:    }
```

Listing 22.5 *(continued)*

The method showDetails() contains five optimizations:

Line 207 demonstrates declaring methods as final for faster access.

Line 213 again demonstrates lazy instantiation.

Use ArraysInstead of Objects. Line 236 uses an integer array instead of a `Vector` to store indexes, and it also guesses the maximum size so as to not have to resize the array. These optimizations could be used further in the code to gain additional speed and memory conservation.

Iterate Loops Down to Zero. Comparing against zero is the fastest, so coding loops to count down instead of up is more efficient. Lines 249 and 259 demonstrate this.

Only Code the Necessary Functionality. We have abandoned the kxml DOM implementation for performing the minimal number of string comparisons and substrings that we need. You will see this again used in `doAmazonSearch()`.

```
291:    private final void doAmazonSearch(int page, String targets) ⮐
throws Exception, IOException
292:    {
293:        final String ELEMENT_PRODUCT_NAME = "ProductName";
294:        final String LITE_FORMAT = "lite";
295:
296:        ticker.reset("Started Timer in doAmazonSearch()");
297:
298:       // get the operation
299:       int idx = opsChoice.getSelectedIndex();
300:       String op = opsChoice.getString(idx);
301:
302:       // get the mode
303:       idx = modeChoice.getSelectedIndex();
304:       String mode = modeChoice.getString(idx);
305:       // static method is fastest
306:       String sURL = MicroAmazonHttpGet.createURL(op, targets, ⮐
mode, LITE_FORMAT, "" + page);
307:
308:       // GET it via static method
309:       xmlBuf = MicroAmazonHttpGet.httpGet(sURL);
310:
311:       // very lazy instantiation
312:       if (resultsScreen == null)
313:       {
314:          // results list
315:          resultsScreen = new List("Results", List.EXCLUSIVE);
// removed adding Commands to resultsScreen — no change.
321:       }
322:
323:       String [] productNames = null;
324:       if (products == null)
325:       {
326:          products = new Vector(10); // Amazon returns 10 entries
327:       }
328:       else
329:       {
330:          products.setSize(0);
331:          int rcnt = resultsScreen.size();
332:          if (rcnt > 0)
333:          {
334:             // clear it
335:             for (int i=rcnt - 1; i >= 0; i—)
336:                resultsScreen.delete(i);
337:          }
338:       }
339:
340:       int index = 0;
```

Listing 22.5 *(continued)*

```
341:       String productName = null;
342:       while ( (index = xmlBuf.indexOf(ELEMENT_PRODUCT_NAME,
index)) > 0)
343:       {
344:          int endIdx = xmlBuf.indexOf(ELEMENT_PRODUCT_NAME,
index + 1);
345:          if (endIdx > index)
346:          {
347:             productName = xmlBuf.substring(index +
ELEMENT_PRODUCT_NAME.length() + 1, endIdx - 2);
348:             products.addElement(productName);
349:          }
350:          index = endIdx + 1;
351:       }
352:       productName = null;
353:
354:       int productCount = products.size();
355:       if (products != null && productCount > 0)
356:       {
357:          // populate the list
358:          for (int i=productCount - 1; i >= 0; i—)
359:             resultsScreen.append((String)products.elementAt(i),
null);
360:
361:          // set the display to the results
362:          display.setCurrent(resultsScreen);
363:       }
// - removed the else block for brevity
370:       ticker.printStats("Method doAmazonSearch()");
371:    }
372: }
```

Listing 22.5 *(continued)*

The method `doAmazonSearch()` contains five optimizations:

Line 291 declares the method final for faster access.

Lines 293 and 294 use local variables.

Line 326 sets the size of the `Vector`.

Line 330 avoids instantiating a new `Vector` by reusing it.

Set References to Null. This will assist the garbage collector in more efficiently reclaiming unused memory. Line 352 makes the `productName String` available for reclamation.

Listing 22.6 presents the optimized version of `BadMicroAmazonHttpGet`. This version has changed drastically to increase performance and conserve memory. There are even further improvements available, like ensuring that the data returned from the Web server can be stored in memory (or pared down to do so).

```
001: /* MicroAmazonHttpGet.java */
002: package org.javapitfalls.item22;
003:
// - removed Import statements -- no change.
007:
008: public class MicroAmazonHttpGet
009: {
010:    public static final String DEVTAG = "D3AG4L7PI53LPH";
011:    static Timer ticker = new Timer();
012:
013:    // Memory saving but not thread safe
014:    private static StringBuffer urlBuf;
015:    private static ByteArrayOutputStream baos;
016:    private static byte [] buf;
017:
```

Listing 22.6 MicroAmazonHttpGet.java

The `Class` definition of MicroAmazonHttpGet has two optimizations. First, most of the static `Strings` have been eliminated to conserve memory. Second, all of the data members have been declared as class data members (to eliminate instantiation) but declared as static for fast access. This means that the class is no longer thread safe, but this is okay because only a single thread uses it. In fact, another improvement may be to just eliminate the class and roll the methods into MicroAmazonSearch:

```
018:    public static final String createURL(String operation, String 
target, String mode, String type,
019:                          String page) throws Exception
020:    {
021:       final String KEYWORD_MODE = "mode";
022:       final String KEYWORD_TYPE = "type";
023:       final String KEYWORD_PAGE = "page";
024:
025:       if (urlBuf == null)
026:       {
027:          urlBuf = new 
StringBuffer("http://xml.amazon.com/onca/xml?v=1.0&t=webservices-20& 
dev-t=");
028:          urlBuf.append(DEVTAG);
029:       }
030:       else
031:       {
032:          urlBuf.setLength(0);
033:
urlBuf.append("http://xml.amazon.com/onca/xml?v=1.0&t=webservices- 
20&dev-t=");
034:          urlBuf.append(DEVTAG);
```

Listing 22.6 *(continued)*

```
035:        }
036:
037:        urlBuf.append('&');
038:        urlBuf.append(operation);
039:        urlBuf.append('=');
040:        if (target != null)
041:        {
042:           target.trim();
043:           target = replaceString(target, " ", "%20", 0);
044:
045:           urlBuf.append(target);
046:        }
047:        else
048:            throw new Exception("Invalid target");
049:
050:        // add Mode
051:        urlBuf.append('&');
052:        urlBuf.append(KEYWORD_MODE);
053:        urlBuf.append('=');
054:        urlBuf.append(mode);
055:
// removed code for adding Type and Page -- no change, just Inlined
069:    }
```

Listing 22.6 *(continued)*

The `createURL` method is a brand-new method that replaced all of the `addXXX` methods in the previous class. This class demonstrates six optimizations:

Line 18 demonstrates declaring a method both final and static for the fastest access.

Lines 21 to 23 demonstrate using local variables.

Lines 25 to 29 demonstrate lazy instantiation.

Manually Inline Methods. Lines 51 to 54 were previously in the `addMode()` method, and instead we inlined the method within the `createURL` method.

Minimize Method Calls. Inlining all of the `addXXX` methods demonstrates minimizing method calls. This should especially be followed for any method calls inside of loops (like a call to `check length()` or `size()`).

```
// - deleted replaceString() — No Change.
091:
092:    public static final String httpGet(String sURL) throws
IOException
093:    {
094:       ticker.reset("Started Timer in httpGet()");
095:       // get the connection object
```

Listing 22.6 *(continued)*

```
096:        System.out.println("url: " + sURL);
097:
098:        HttpConnection con = (HttpConnection) Connector.open(sURL);
099:        int respCode = con.getResponseCode();
100:        System.out.println("Response code: " + respCode);
101:        InputStream in = con.openInputStream();
102:
103:        // lazy instantiate!
104:        if (baos == null)
105:           baos = new ByteArrayOutputStream(1024);
106:        else
107:           baos.reset();
108:        if (buf == null)
109:           buf = new byte[1024];
110:        String response = null;
111:        int cnt=0;
112:        while ( (cnt = in.read(buf)) != -1)
113:        {
114:           baos.write(buf,0,cnt);
115:       }
116:        response = baos.toString();
117:        ticker.printStats("Method httpGet()");
118:        return response;
119:    }
120: }
```

Listing 22.6 *(continued)*

The `httpGet()` method demonstrates three optimizations:

Line 92 declares the method final and static.

Lines 104 and 108 demonstrate lazy instantiation. Also, lines 105 and 109 guess the size of objects to avoid resizing—though `ByteArrayOutputStream` could be better sized with better sampling of average query sizes.

Read More than 1 Byte from a Stream at a Time. All network connections will buffer more than a single byte so reading larger chunks of data is more efficient.

Here are some additional optimization tips:

Avoid `String` Concatenation. `String` concatenation has been proven extremely slow, so use `StringBuffers` or streams to avoid this. This can be better taken advantage of in this application.

Avoid Synchronization. Synchronization is slow because of the overhead of implementation the thread controls.

If adding by 1, int++ is the fastest operation. This is faster than expressions like "I = I + 1;".

Improve Perceived Performance. Use progress meters to inform the user that the computer is active.

Only Include Necessary Classes. To conserve memory required to store your application, only include necessary classes in the deployment archive.

Use Shift Operator to Multiply by 2. The shift operator is faster than the multiplication operator.

Avoid Casting. Casting is expensive, so have methods return only one type.

Use `int` as Much as Possible. Other types like `byte`, `short`, and `character` are promoted to an `int`. So eliminate the promotion by using `int`s directly.

Avoid Using Exceptions. Exceptions require additional checking by the VM, so your code will be faster using more traditional procedural programming (with status returns).

The new, optimized code runs extremely well within 128 KB. Here is a run of the code demonstrating that by performing three queries and multiple showing of details:

```
Started Timer in doAmazonSearch(). Free Memory: 108760
Started Timer in httpGet(). Free Memory: 102160
url: http://xml.amazon.com/onca/xml?v=1.0&t=webservices-20&dev-
t=D3AG4L7PI53LPH&KeywordSearch=daconta&mode=books&type=lite&page=1&f=xml
Response code: 200
Method httpGet(): 8572. Free Memory: 35372
Method doAmazonSearch(): 8773. Free Memory: 32792
Started Timer in showDetails(). Free Memory: 75040
Method showDetails(): 20. Free Memory: 68768
// - removed second query for brevity
Started Timer in doAmazonSearch(). Free Memory: 63284
Started Timer in httpGet(). Free Memory: 60692
url: http://xml.amazon.com/onca/xml?v=1.0&t=webservices-20&dev-
t=D3AG4L7PI53LPH&AuthorSearch=Robert%20Heinlein&mode=books&type=lite&pag
e=1&f=xml
Response code: 200
Method httpGet(): 8502. Free Memory: 23512
Method doAmazonSearch(): 9143. Free Memory: 21244
Started Timer in showDetails(). Free Memory: 63736
Method showDetails(): 1082. Free Memory: 57560
Execution completed successfully
1578454 bytecodes executed
750 thread switches
324 classes in the system (including system classes)
2491 dynamic objects allocated (344132 bytes)
600 garbage collections (283784 bytes collected)
Total heap size 131072 bytes (currently 56276 bytes free)
```

Figure 22.6 displays MicroAmazonSearch being emulated on a BlackBerry device.

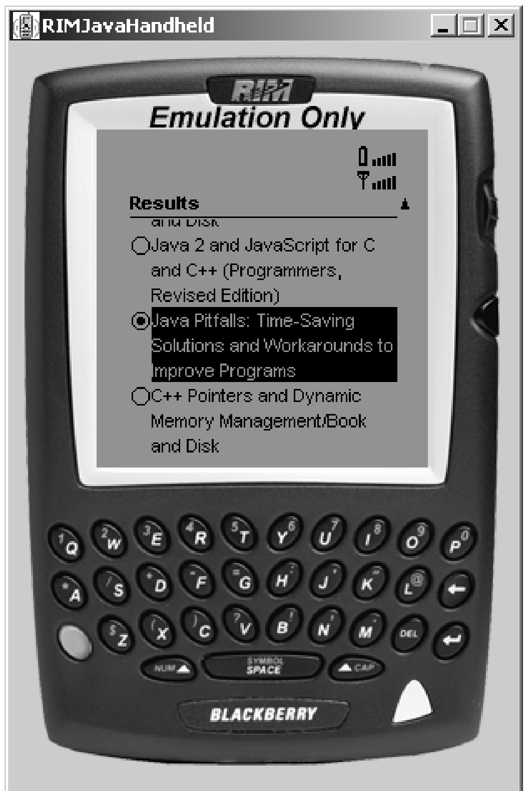
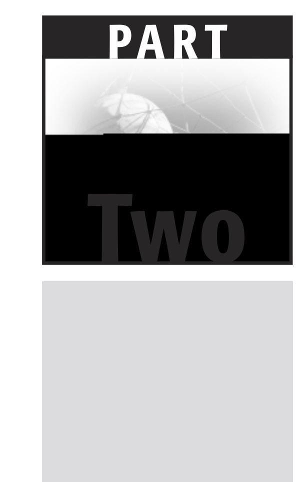


Figure 22.6 MicroAmazonSearch emulated on a BlackBerry.

Copyright © 2002 Research In Motion Limited.

In conclusion, this pitfall has ported a Swing application to the J2ME platform in order to reveal two categories of pitfalls: a bias to programming for the J2SE platform and pitfalls caused by API differences. Lastly we covered and demonstrated over 20 techniques for optimizing J2ME code.



The Web Tier

"The original impulse for modularity came from a desire for flexibility, in particular how to subdivide a sizeable program text into 'modules' . . . But the emphasis has shifted from such mere replaceability to the question of how to break down the whole task most effectively: the demands are such that elegance is no longer a dispensable luxury, but decides between success and failure."

Edsger W. Dijkstra, "On a Cultural Gap"

It is difficult to overstate the transforming nature of Java on the server side. While the adoption of Enterprise JavaBeans and other parts of the J2EE platform has been successful, there is no mistaking how transforming Java has been in the Web tier space.

The accessibility of Java has provided many developers and would-be developers with the opportunity to jump right in to building Web applications with the Java platform. The strong technological advantages contained in servlets, filters, JSPs, and tag libraries have caused many developers to become early adopters.

The early adoption and ease of use of the Java technologies is at the root of many of the Web tier pitfalls. Many developers have found things that work for them and have not understood the underlying implementation concerns. Furthermore, the nature of Web applications having multiple concurrent users is not readily apparent to many developers. Some notable pitfalls in this section:

Cache, It's Money (Item 23). One of the overriding principles of Web applications is the latency of the data being served by these applications. Many Web apps are built to query a database on every request despite the fact that the data has not changed.

JSP Design Errors (Item 24). JavaServer Pages are very powerful—combining the flexibility of scripting and the power of compilation. However, this can have unseen impact in the areas of readability, maintenance, and reuse.

When Servlet HTTPSessions Collide (Item 25). The nature of the Web is such that users jump from site to site with impunity. This can cause issues with systems that rely on the integrity of variables in the `HTTPSession` class. Collisions between these can cause interesting issues in Web applications.

When Applets Go Bad (Item 26). Any developer that has developed and deployed applet-based Web applications is well aware of the nightmares involved. Java Web Start has proven to be a terrific redesign for all of the problems of applets.

Transactional LDAP—Don't Make That Commitment (Item 27). The delivery of personalized Web content to authenticated users is a complicated process that involves tough questions about proper profile data storage operations needed to perform this activity. This pitfall addresses the decision to store profile attributes in a Relational Database Management System (RDBMS), an LDAP directory, or a combination of both.

Problems with Filters (Item 28). The Servlet 2.3 specification introduced a new Web component called the filter. This pitfall addresses problems encountered in the use of this new component.

Some Direction about JSP Reuse and Content Delivery (Item 29). JavaServer Pages are important visualization components because of their ability to be reused in Web applications. An important aspect of this reusability is their ability to swap in dynamic content from both default application contexts as well as remote contexts. This pitfall will show you how to do both.

Form Validation Using Regular Expressions (Item 30). A new feature of the JDK 1.4 is support for regular expressions. This new support greatly extends the capabilities of Web applications to perform validation of entered data.

Item 23: Cache, It's Money

Every developer's nightmare is developing code on a local development environment and migrating to a different enterprise deployment platform, only to discover substandard latency times during back-end document queries. This problem often occurs because of disparities between development and deployment systems, and because software developers and database administrators tend to work separately from one another. Unfortunately, this "Big Bang" theory, where all forces are expected to come together during integration and work as expected, is a common delusion for most projects.

A recent development effort we were part of exemplified this when we were confronted with latency issues on our front-end portal application that performed SOAP requests and XSLT translations on data received from a back-end database that was managed by a different contractor. Our customer's concerns about protracted query results forced us to take a fresh look at the problem. We knew that the database had documents that were updated roughly every month and had been relatively stable, and the user community was estimated to be between 1,000 and 5,000 concurrent users.

Our analysis led us to the conclusion that our document queries had to be cached, which would alleviate the strains that a relatively large user community might place on our database. This was possible because of the nature of our data that remained stable

until the end of every month. The downside of this solution was our understanding that the relevance of the document data could be diminished if older, less significant data was cached and rendered on user queries. Our final solution involved the implementation of two innovative open-source applications: the OSCache tag library from OpenSymphony and the JMeter application from the Apache Software Foundation. OSCache tags were used to cache database queries, and JMeter was used to emulate user requests.

Our aim was to cache the sections of our code where database queries were performed so that repeat trips back to the database could be avoided, thereby increasing performance. During normal operations, any new documents that were added to the database repository would be cached upon insertion. The caching process would be facilitated by implementing JMeter scripts with URL parameters that emulated all the different combinations of user responses. We determined that our entire site could be cached in 45 minutes every month.

To demonstrate this process, some example JavaServer Pages (JSPs) and JMeter XML `ThreadGroup` scripts were developed to cache query data from a simple MySQL database. The Web page shown in Figure 23.1 demonstrates two portlet-like tables with information items: Bike Trails, Gyms, and so on, and the 50 different U.S. states where these information items can be found, which would total 550 different link combinations. The caching process could have been performed manually by hitting each page with the OSCache tags, but this would take too much time and effort. We felt that a more efficient process would involve the creation of a test generation application and the implementation of JMeter to run those tests.

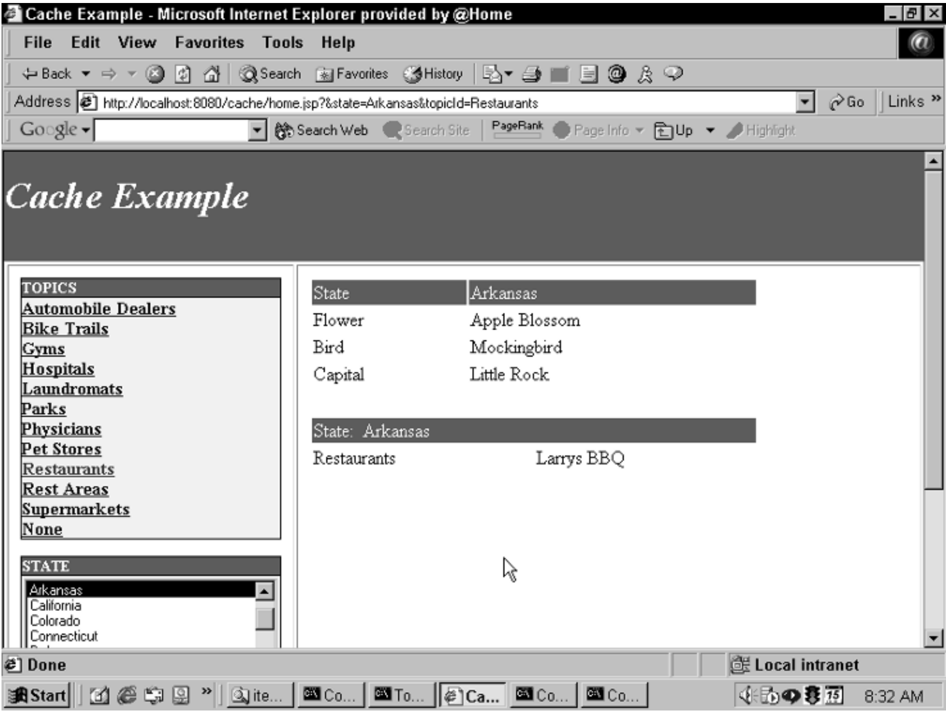


Figure 23.1 Cache example application.

The main focus of our implementation (see Listing 23.1) is a JSP named home.jsp because it renders dynamic query information to the user display and uses the OSCache tag library to cache that data. In the home.jsp code below, the tag library OSCache is specified on line 07. The URI for this class is specified in the deployment descriptor (web.xml). Please note that the `<cache:cache></cache:cache>` tags on lines 31 to 40 surround the content.jsp page include, which actually performs the database query. On line 31, we set the cache duration using the ISO-8601 format (YYYY-MM-DD). We could have also set the caching duration using the `time` attribute, whose default value is 3600 seconds, which is 1 hour.

```
01: <%@page import="java.util.*" %>
02: <%@page import="java.io.*" %>
03: <%@page import="java.sql.*" %>
04:
05: <jsp:useBean id="cacheHelper"
06:  class="org.javapitfalls.item23.cacheHelper" scope="request"/>
07: <%@ taglib uri="OSCache" prefix="cache" %>
08:
09: <%
10: String topicId = request.getParameter("topicId");
11: String state = request.getParameter("state");
12: if (topicId == null) topicId = "";
13: if (state == null) state = "";
14: %>
15:
16: <title>
17: Cache Example
18: </title>
19:
20: <jsp:include page="header.jsp" >
21:  <jsp:param name="topicId" value="<%= topicId %>" />
22:  <jsp:param name="state" value="<%= state %>" />
23: </jsp:include>
24:
25: <table border="1" width="100%">
26: <tr valign="top">
27:  <td width="25%" valign="top">
28:         <jsp:include page="leftNav.jsp" />
29:  </td>
30:  <td width="75%" valign="top">
31:  <cache:cache scope="session" duration="2002-01-31">
32:         <% try { %>
33:                 <jsp:include page="content.jsp"/>
34:                 <jsp:param name="topicId" value="<%= topicId %>" />
35:                 <jsp:param name="state" value="<%= state %>" />
36:                 </jsp:include>
37:         <% } catch (Exception e) { %>
```

Listing 23.1 home.jsp

```
38:                 <cache:usecached />
39:         <% } %>
40:  </cache:cache>
41:  </td>
42: </tr>
43: </table>
44:
45:
46: <jsp:include page="footer.jsp" />
47:
48:
```

Listing 23.1 *(continued)*

The OSCache tag library implementation includes a properties file that is installed in the /WEB-INF/classes directory, which allows the user to set attributes for operational preferences. We've included only the properties that are pertinent to our implementation in Listing 23.2. The `cache.path` property points to the location where we want to place our cache files. The `cache.debug` property specifies that we want to see debugging messages, and the `cache.unlimited` property ensures that the cache disk space is unlimited.

```
# CACHE DIRECTORY
01:#
02:# This is the directory on disk where caches will be stored.
03:# it will be created if it doesn't already exist, but OSCache
04:# must be able to write to here.
05:#
06: cache.path=c:\\cachetagscache
07:
08:# DEBUGGING
09:#
10:# set this to true if you want to see log4j debugging messages
11:#
12:cache.debug=false
13:
14:
15:# CACHE UNLIMITED DISK
16:# Use unlimited disk cache or not
17:cache.unlimited_disk=false
```

Listing 23.2 oscache.properties

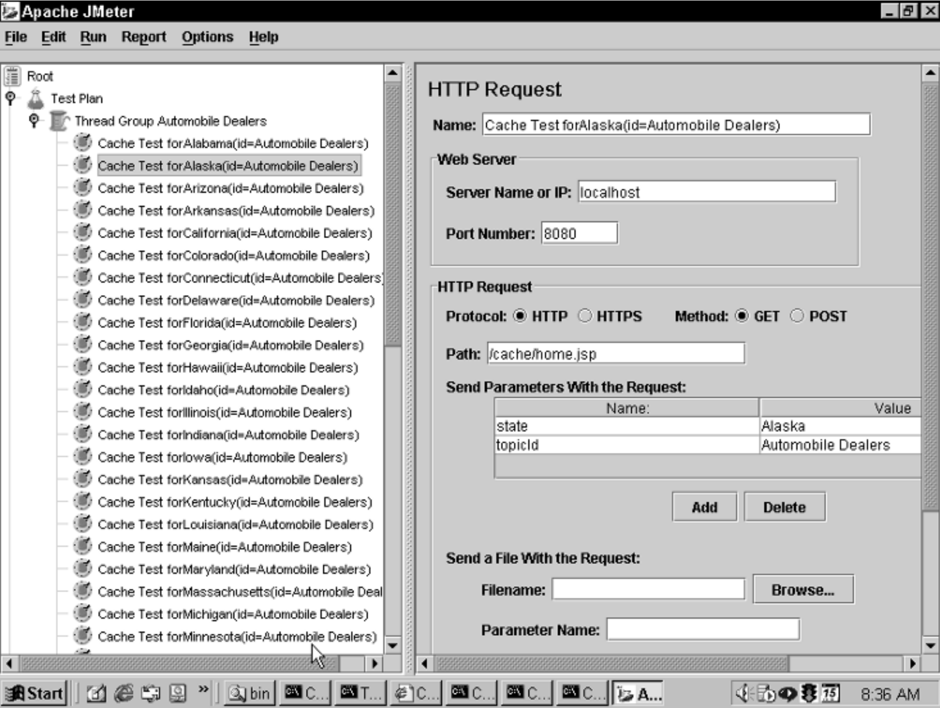


Figure 23.2 Run of Apache JMeter.

After properly inserting the cache tag library tags in the source code, we generated JMeter `ThreadGroup` scripts to replicate a user hitting each page with the `state` and `topicId` parameters.

The JMeter application, as shown in Figure 23.2, is a Java desktop tool that performs automated load testing and user activity measurements, and it comes with visualization tools that provide test feedback and performance metrics. Our application uses this tool to hit all possible user selections so that these pages could be cached. By caching these pages, user queries do not have to go to the back end to draw back data.

Users can generate these `ThreadGroup` tests manually using the JMeter GUI, but we felt that a more efficient option was to write a program called generateTests.java to build these tests automatically. Listing 23.3 hard-codes the `state` and `topicId` data, but an optimal solution would use a file or a database to store these values so that the data would not be tightly coupled with the application and changes could be accommodated more easily. In the generateTests.java program, lines 81 to 85 show a constant timer tag that will kick off a test every second or 1,000 milliseconds. Once these tests are generated, users can use the JMeter GUI to execute these tests, or they can run them manually from the command line using the nongui script:

```
prompt> nongui -o my_test.jmx -h <servername> -p <port #>
```

```
01: import java.io.*;
02: import java.util.*;
03:
04: class generateTests
05: {
06:  public static void main(String[] args) throws IOException
07:  {
08:   if (args.length != 0) {
09:    System.out.println("USAGE: java generateTests");
10:   } else {
11:    String[] states = { "Alabama", "Alaska", "Arizona", "Arkansas",
12:         "California", "Colorado", "Connecticut", "Delaware",
13:         "Florida", "Georgia", "Hawaii", "Idaho", "Illinois",
14:         "Indiana", "Iowa", "Kansas", "Kentucky", "Louisiana",
15:         "Maine", "Maryland", "Massachusetts", "Michigan",
16:         "Minnesota", "Mississippi", "Missouri", "Montana",
17:         "Nebraska", "Nevada", "New Hampshire", "New Jersey",
18:         "New Mexico", "New York", "North Carolina", "North Dakota",
19:         "Ohio", "Oklahoma", "Oregon", "Pennsylvania",
20:         "Rhode Island", "South Carolina", "South Dakota",
21:         "Tennessee", "Texas", "Utah", "Vermont", "Virginia",
22:         "Washington", "West Virginia", "Wisconsin", "Wyoming" };
23:
24:    String[] topicIds = { "Automobile Dealers", "Bike Trails",
25:            "Gyms", "Hospitals", "Laundromats", "Parks",
26:            "Physicians", "Pet Stores", "Restaurants",
27:            "Rest Areas", "Supermarkets" };
28:
29:    for (int x=0; x < topicIds.length; x++) {
30:
31:     PrintWriter pw = new PrintWriter(new
32:          FileOutputStream("Cache_Test_" + topicIds[x] + ".jmx"));
33:     pw.write("<?xml version=\"1.0\"?>\n");
34:     pw.write("<TestPlan>\n");
35:     pw.write("<threadgroups>\n");
36:     pw.write("<ThreadGroup name=\"Thread Group " + topicIds[x] + "\"
37:             numThreads=\"1\" rampUp=\"0\">");
38:     pw.write("<controllers>\n");
39:     pw.write("<LoopController
40:              type=\"org.apache.jmeter.control.LoopController\"
41:              name=\"Loop Controller\" iterations=\"1\">");
42:     pw.write("<configElements>\n");
43:     pw.write("</configElements>\n");
44:     pw.write("<controllers>\n");
45:
46:     for (int y=0; y < states.length; y++) {
```

Listing 23.3 generateTests.java *(continued)*

```
47:       pw.write("<HttpTestSample
48:       type=\"org.apache.jmeter.protocol.http.control.HttpTestSample\"
49:       name=\"Cache Test for" + states[y] +
50:       "(id=" + topicIds[x] + ")" + "\" getImages=\"false\">\n");
51:       pw.write("<defaultUrl>\n");
52:       pw.write("<ConfigElement  type=\"
53:       org.apache.jmeter.protocol.http.config.MultipartUrlConfig\">
54:       \n");
55:       pw.write("<property name=\"port\">8080</property>\n");
56:       pw.write("<property name=\"PROTOCOL\">http</property>\n");
57:       pw.write("<property name=\"domain\">localhost</property>\n");
58:       pw.write("<property name=\"arguments\">\n");
59:       pw.write("<Arguments>\n");
60:       pw.write("<argument name=\"state\"> " + states[y] +
61:                "</argument>\n");
62:       pw.write("<argument name=\"topicId\">" + topicIds[x] +
63:                "</argument>\n");
64:       pw.write("</Arguments>\n");
65:       pw.write("</property>\n");
66:       pw.write("<property name=\"path\">
67:                /cachePage/home.jsp</property>\n");
68:       pw.write("<property name=\"method\">GET</property>\n");
69:       pw.write("</ConfigElement></defaultUrl>\n");
70:       pw.write("<configElements>\n");
71:       pw.write("</configElements>\n");
72:       pw.write("<controllers>\n");
73:       pw.write("</controllers>\n");
74:       pw.write("</HttpTestSample>\n");
75:      }
76:
77:      pw.write("</controllers>\n");
78:      pw.write("</LoopController>\n");
79:      pw.write("</controllers>\n");
80:      pw.write("<timers>\n");
81:      pw.write("<Timer type=\"org.apache.jmeter.timers.ConstantTimer\"
82:                name=\"Constant Timer\">\n");
83:      pw.write("<delay>1000</delay>\n");
84:      pw.write("<range>0.0</range>\n");
85:      pw.write("</Timer>\n");
86:      pw.write("</timers>\n");
87:      pw.write("<listeners>\n");
88:      pw.write("</listeners>\n");
89:      pw.write("</ThreadGroup>\n");
90:      pw.write("</threadgroups>\n");
91:      pw.write("<configElements>\n");
92:      pw.write("</configElements>\n");
```

Listing 23.3 *(continued)*

```
93:     pw.write("</TestPlan>\n");
94:     pw.close();
95:     System.out.println("Finished writing: " + topicIds[x]);
96:    }
97:   }
98:  }
99: }
```

Listing 23.3 *(continued)*

Figure 23.3 demonstrates what occurs in our Web application when we use the JMeter test scripts to cache our database queries. These scripts ping the database with the query operations indicated by the number 1 in the figure and cache the pages to the cache repository so that future database queries hit the cached scripts as indicated by the number 2.

In the end, our scripts allowed us to cache our entire site and reduce query result times, which pleased our customer. Additionally, the knowledge we acquired from our JMeter implementation allowed us to engage our test personnel earlier in our next program, which facilitated our integration efforts between our front-end and database developers.

In all Web development efforts, it is paramount that developers pay some consideration to the implementation of a caching strategy so that pertinent data can be delivered in a timely fashion and database server overloads can be avoided. If your site serves up document artifacts that don't change regularly, it will serve you well. Consider that at the JavaOne 2002 Conference there was significant discussion of the JCACHE specification (JSR 107), whose purpose is to standardize caching of Java objects. Feature enhancements with existing tag libraries like OSCache, along with new implementations like JCACHE, will continue to produce faster Web page response times and make Web content queries a much more pleasant experience.

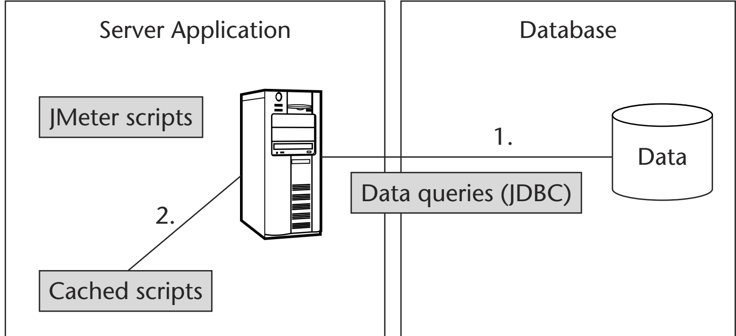


Figure 23.3 Cache architecture.

Item 24: JSP Design Errors

The evolution of Web applications followed two different paths: scripts and compiled executables. Servlets provided a strong improvement on the compiled code examples but suffered their own limitations in terms of presentation. The arrival of JSP caused many people to use them as self-compiling scripts. They saw the flexibility of script programming combined with the power of compiled code. Unfortunately, this can cause significant problems in maintenance, reuse, and flexibility.

My development team was assigned a task of building a Web-based workflow system that persisted information to a conventional relational database. Having built systems like this with a few different technologies, including most recently a servlet implementation of a similar system, we were eager to try JSP to eliminate our endless `println` statements, as well as eliminate the fundamental maintenance nightmare of trying to maintain HTML code inside our servlets.

JSP turned out to be a set of challenges in its own right. Before discussing the blow by blow, let's quickly review the basics of Web application development.

Request/Response Paradigm

Web applications, independent of technological implementation, come down to a few fundamental concepts. Having built quite a few of these systems with a number of technological solutions, we've found that it boils down to a simple process:

- Parse parameters from request.
- Apply business logic (query/update the database in our case).
- Present results to the user in the response.

Listing 24.1 is an example of what this looks like in a servlet implementation.

```
01: public void doPost(HttpServletRequest request, HttpServletResponse
02:                      response) throws ServletException, IOException {
03:
04:     // Parse parameters from the request
05:     String salary = request.getParameter("salary");
07:
08:     // Execute Business Logic (query a DB in this case)
09:     try {
11:     statement.setFloat(1, Float.parseFloat(salary));
12:
13:     // statement is a PreparedStatement initialized in the init() ⮐
method
14:     ResultSet results =
15:         statement.executeQuery();
```

Listing 24.1 doPost() from SalaryServlet

```
16:
17:
18:     //  Present the results to the user in the response
19:     response.setContentType("text/html");
20:
21:     PrintWriter out = response.getWriter();
22:     out.println("<html>");
23:     out.println("<head><title>SalaryServlet</title></head>");
24:     out.println("<body>");
25:     out.println("<table>");
26:     out.println("<tr>");
27:     out.println("<td><b>Employee</b></td></tr>");
28:
29:     while (results.next()) {
30:       out.println("<tr>");
31:       out.println("<td>");
32:       out.println(results.getString(1));
33:       out.println("</td>");
34:       out.println("</tr>");
35:     }
36:
37:     out.println("</table>");
38:     out.println("</body></html>");
40:     } catch ( SQLException sqle ) {
41:
42:       sqle.printStackTrace();
44:     }
45:
46:   }
```

Listing 24.1 *(continued)*

Maintaining State

The other issue that comes up in building Web applications is that HTTP is a stateless protocol—that is, one request is independent of the next. Therefore, some method must be chosen to determine how to maintain information about what was done prior to and after this request. A classic example of maintaining state is the user login. A user should not need to provide a login credential with each request. Therefore, there must be some way of keeping track of whether or not this user has been authenticated. There are a few options for how to do this:

Passing parameters. Obviously, the Web application can pass the parameters with each request and embed "hidden" form elements in the response, so that the user doesn't need to re-enter these parameters. The obvious problem is that this can become quite burdensome, involving continuously passing information that is essentially passed through and not really relevant to this particular screen.

Cookies. Cookies provide the ability to store pieces of information on the remote machine; therefore, the application can check to determine if it has previously set information on that machine. However, obviously, major concerns arise out of whether the user will allow such information to be set on his or her machine and how long such information can be counted on being available.

Session variables. HTTP actually provides the ability to put variables in the HTTP session; this was the mechanism that HTTP uses for its basic authentication scheme. Session variables can be helpful but can be problematic—especially with regard to maintaining the session and avoiding other variables with the same name.

JSP the Old Way

Let's take a second to discuss the evolution of Web applications. Originally, Web applications came in two forms: compiled executables (usually written in C or C++) and server scripts (usually written in Perl). They both used the Common Gateway Interface (CGI) to execute requests. To make Web applications easier to build, other scripting languages evolved like server-side JavaScript, JScript, and VBScript. The choice seemed to boil down to power (compiled code was always faster) versus ease of use.

The servlet implementation of my Web application brings me a number of advantages. It is compiled, so I get a performance increase. It operates in a shared process, the servlet engine, so it scales particularly well. Also, the shared process allows for the creation of shared resources like database connections. This is a particular advantage over previously compiled options, which would create new processes (and resources) to handle each request. This caused problems with scalability and with security (the command executed in its own space, without any managing thread to contain malignant or runaway code).

The problem with compiled code is that it is just that, compiled. Notice that I have embedded a great deal of HTML code into my servlet. That means any changes to the presentation require a recompile of the code. Furthermore, anyone who has created these `out.println()` commands of HTML understands the pain of escape sequences.

Along comes JSP, which allows the developer to invert his or her servlet and embed the logic into the presentation code. This proves very helpful to script writers who suffer from limited-capability scripting languages and convoluted programming structures (like conditional loops and iterators). Furthermore, a JSP is compiled into a servlet, so you receive the benefits of compiled code in a script-driven programming environment.

Listing 24.2 is an example JSP implementation of the previous salary servlet.

```
01: <%@ page import="java.util.*"%>
02: <%@ page import="java.sql.*"%>
03: <HTML>
04: <HEAD>
05: <TITLE>
06: Salary Jsp
```

Listing 24.2 BadSalaryJsp.jsp

```
07: </TITLE>
08: </HEAD>
09: <BODY>
10: <H1>
11: Here are the people who make over $<%=
request.getParameter("salary") %>:
12: <%
13:     // Database config information
14:     String driver = "oracle.jdbc.driver.OracleDriver";
15:     String url = "jdbc:oracle:thin:@joemama:1521:ORACLE";
16:     String username = "scott";
17:     String password = "tiger";
18:
19:     String salary = request.getParameter("salary");
20:
21:     // Establish connection to database
22:     try {
23:       Class.forName(driver);
24:       Connection connection =
25:         DriverManager.getConnection(url, username, password);
26:
27:         PreparedStatement statement
28:             = connection.prepareStatement("SELECT ename FROM emp
WHERE sal > ?");
29:
30:       statement.setFloat(1, Float.parseFloat(salary));
31:
32:       ResultSet results =
33:             statement.executeQuery();
34: %>
35: </H1>
36: <table>
37: <tr>
38: <td><b>Employee</b></td>
39: </tr>
40:
41: <%    while (results.next()) {  %>
42: <tr>
43: <td>
44: <%=   results.getString(1) %>
45: </td>
46: </tr>
47: </table>
48: <%    }
49:
50:     } catch(ClassNotFoundException cnfe) {
51:       System.err.println("Error loading driver: " + cnfe);
52:
53:     } catch(SQLException sqle) {
```

Listing 24.2 *(continued)*

```
54:            sqle.printStackTrace();
55:      }
56:
57: %>
58: </BODY>
59: </HTML>
60:
```

Listing 24.2 *(continued)*

This approach looks a lot like the servlet pulled inside out. In fact, to demonstrate how close this is to reality, Listing 24.3 shows a snippet of the source generated by Apache Tomcat to compile the JSP.

```
01:             // HTML // begin
[file="/BadSalaryJsp.jsp";from=(0,31);to=(1,0)]
02:                 out.write("\r\n");
03:
04:             // end
05:             // HTML // begin
[file="/BadSalaryJsp.jsp";from=(1,30);to=(10,35)]
06:                out.write("\r\n<HTML>\r\n<HEAD>\r\n<TITLE>\r\nSalary
Jsp\r\n</TITLE>\r\n</HEAD>\r\n<BODY>\r\n<H1>\r\nHere are the people who
make over $");
07:
08:             // end
09:             // begin
[file="/BadSalaryJsp.jsp";from=(10,38);to=(10,70)]
10:                 out.print( request.getParameter("salary") );
11:             // end
12:             // HTML // begin
[file="/BadSalaryJsp.jsp";from=(10,72);to=(11,0)]
13:                 out.write(":\r\n");
14:
15:             // end
16:             // begin
[file="/BadSalaryJsp.jsp";from=(11,2);to=(33,0)]
17:
18:                     // Database config information
19:                     String driver =
"oracle.jdbc.driver.OracleDriver";
20:                     String url =
"jdbc:oracle:thin:@joemama:1521:ORACLE";
21:                     String username = "scott";
22:                     String password = "tiger";
```

Listing 24.3 Apache Tomcat-generated BadSalaryJsp.jsp

```
23:
24:                     String salary = request.getParameter("salary");
25:
26:                     // Establish connection to database
27:                     try {
28:                       Class.forName(driver);
29:                       Connection connection =
30:                         DriverManager.getConnection(url, username,
password);
31:
32:                         PreparedStatement statement
33:                             = connection.prepareStatement("SELECT
ename FROM emp WHERE sal > ?");
34:
35:                       statement.setFloat(1,
Float.parseFloat(salary));
36:
37:                       ResultSet results =
38:                             statement.executeQuery();
39:             // end
40:             // HTML // begin
[file="/BadSalaryJsp.jsp";from=(33,2);to=(40,0)]
41:
out.write("\r\n</H1>\r\n<table>\r\n<tr>\r\n<td><b>Employee</b></td>\r\n</
tr>\r\n\r\n");
42:
43:             // end
44:             // begin
[file="/BadSalaryJsp.jsp";from=(40,2);to=(40,32)]
45:                     while (results.next()) {
46:             // end
```

Listing 24.3 *(continued)*

So what is wrong with that? The user gets the ease of scripting combined with the power of compilation. This seems like the best of both worlds. However, the example JSP shows the most obvious problem. Why do I want to put database configuration information in each of my pages? This is a maintenance nightmare. A fundamental of good software development, particularly object-oriented development, is the separation of concerns. The database is a common piece that should be accessible to numerous JSP pages.

While it is possible to do some of this by creating an independent JSP, this kind of issue—accessing an enterprise resource—screams for a programmatic implementation. The solution to this came almost immediately with the JSP specification: Use JavaBeans—now known as the Model 1 Architecture—to encapsulate your business logic.

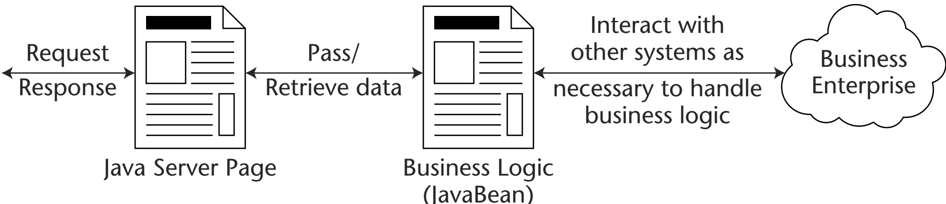


Figure 24.1 Model 1 Architecture.

JSP Development with Beans (Model 1 Architecture)

Figure 24.1 shows the JSP Model 1 Architecture.

The architecture features the following:

- Beans provide encapsulated data structure and logic.
- External resources (databases, Web services, etc.) are abstracted from presentation layer.

Although this is an improvement, the following drawbacks still exist:

- Control logic is still tied to presentation.
- Maintenance is better but still poor—too closely coupled to presentation logic.

Listing 24.4 shows the logic from the previous JSP in a bean.

```
01: public class SalaryServletBean {
02:   private String salary = "30000";
03:   private LinkedList nameList;
04:   private Connection connection;
05:   private PreparedStatement statement;
06:
07:   public SalaryServletBean() {
08:
09:     // Database config information
10:     String driver = "oracle.jdbc.driver.OracleDriver";
11:     String url = "jdbc:oracle:thin:@joemama:1521:ORACLE";
12:     String username = "scott";
13:     String password = "tiger";
14:
15:     // Establish connection to database
16:     try {
17:       Class.forName(driver);
18:       connection =
19:         DriverManager.getConnection(url, username, password);
20:
21:         PreparedStatement statement
```

Listing 24.4 SalaryServletBean.java

```
22:              = connection.prepareStatement("SELECT ename FROM emp ⮐
WHERE sal > ?");
23:
24:     } catch(ClassNotFoundException cnfe) {
25:       System.err.println("Error loading driver: " + cnfe);
26:
27:     } catch(SQLException sqle) {
28:           sqle.printStackTrace();
30:     }
31:   }
32:
33:   /**Retrieve the List of Names*/
34:   public LinkedList getNameList() {
36:     return nameList;
37:   }
38:
39:   /**Specify the salary level for getting the list of names*/
40:   public void setSalary(String newValue) {
41:     if (newValue!=null) {
42:       salary = newValue;
43:     }
44:
45:     statement.setFloat(1, Float.parseFloat(salary));
46:
47:     ResultSet results =
48:         statement.executeQuery();
49:
50:     nameList.clear();
51:
52:     while (results.next()) {
53:       nameList.add(results.getString(1));
54:     }
56:   }
57: }
58:
```

Listing 24.4 *(continued)*

Listing 24.5 shows the accompanying JSP that uses the bean.

```
01: <%@ page import="java.util.*"%>
02: <HTML>
03: <HEAD>
04: <jsp:useBean id="mySalaryServletBean" scope="session" ⮐
class="SalaryServletBean" />
```

Listing 24.5 SalaryJsp.jsp *(continued)*

```
05: <jsp:setProperty name="mySalaryServletBean" property="*" />
06: <TITLE>
07: Salary Servlet Jsp
08: </TITLE>
09: </HEAD>
10: <BODY>
11: <H1>
12: Salary Jsp
13: </H1>
14: <table>
15: <tr>
16: <td><b>Employee</b></td>
17: </tr>
18:
19: <%
20:       LinkedList myList = mySalaryServletBean.getNameList();
21:       ListIterator li = myList.listIterator();
22:       while (li.hasNext()) {
25: %>
26: <tr>
27: <td>
28: <%=      (String)li.next() %>
29: </td>
30: </tr>
31: <% } %>
32: </table>
33:
34: </BODY>
35: </HTML>
36:
```

Listing 24.5 *(continued)*

Another way to abstract business logic out of the JSP code is through the use of JSP custom tag libraries. This approach allows for component reuse of particular code and also allows content developers to have access to complex Java code programming logic in a form that is familiar to them: HTML-like tags.

Listing 24.6 is an example of our salary business logic in a custom tag library. Notice this tag library has not been designed for the purpose of reuse, but rather to show the same logic expressed in a different manner.

```
01: package mypackage;
02: import javax.servlet.jsp.tagext.TagSupport;
03: import javax.servlet.jsp.tagext.BodyContent;
```

Listing 24.6 SalaryTag.java

```
04: import javax.servlet.jsp.JspException;
05: import javax.servlet.jsp.JspTagException;
06: import javax.servlet.jsp.JspWriter;
07: import javax.servlet.jsp.PageContext;
08: import javax.servlet.ServletRequest;
09: import java.io.PrintWriter;
10:
11: public class salarytag extends TagSupport
12: {
13:   /*
14:   tag attribute: salary
15:   */
16:
17:   private String salary = "30000";
18:
19:   /**
20:    * Method called at start of tag.
21:    * @return SKIP_BODY
22:    */
23:   public int doStartTag() throws JspException
24:   {
25:     try
26:     {
27:       JspWriter out = pageContext.getOut();
28:       SalaryServletBean mySalaryBean = new SalaryServletBean();
29:       mySalaryBean.setSalary(salary);
30:       LinkedList myList = mySalaryBean.getNameList();
31:       ListIterator li = myList.listIterator();
32:       while (li.hasNext()) {
33:
34:         out.println("<tr><td>");
35:         out.println((String)li.next());
36:         out.println("</td></tr>");
37:       }
38:
39:     }
40:     catch(Exception e)
41:     {
42:       e.printStackTrace();
43:     }
44:
45:     return SKIP_BODY;
46:   }
47:
49:   /**
50:    * Method called at end of tag.
51:    * @return SKIP_PAGE
52:    */
53:   public int doEndTag()
```

Listing 24.6 *(continued)*

```
54:   {
55:     return SKIP_PAGE;
56:   }
57:
58:   public void setSalary(String value)
59:   {
60:     salary = value;
61:   }
62:
63:   public String getSalary()
64:   {
65:     return salary;
66:   }
67:
68: }
69:
```

Listing 24.6 *(continued)*

To deploy this tag library, you need to define it using a tag library descriptor (see Listing 24.7).

```
01: <?xml version = '1.0' encoding = 'windows-1252'?>
02: <!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag
Library 1.1//EN" "http://java.sun.com/j2ee/dtds/web-
jsptaglibrary_1_1.dtd">
03: <taglib>
04:    <tlibversion>1.0</tlibversion>
05:    <jspversion>1.1</jspversion>
06:    <shortname>salary</shortname>
07:    <uri>javapitfalls </uri>
08:    <info>Shows how to encapsulate the salary business logic in a
taglib.</info>
09:    <tag>
10:       <name>salarytag</name>
11:       <tagclass>mypackage.SalaryTag</tagclass>
12:       <bodycontent>empty</bodycontent>
13:       <attribute>
14:          <name>salary</name>
15:          <required>true</required>
16:          <rtexprvalue>true</rtexprvalue>
17:       </attribute>
18:    </tag>
19: </taglib>
20:
```

Listing 24.7 Salary.tld

However, neither of these shows the tag library in action. All of the complexity is gone, so your developer has an easy task. Listing 24.8 is an example of a JSP using our tag library.

```
01: <%@ page contentType="text/html;charset=windows-1252"%>
02: <%@ page import="java.util.*"%>
03: <%@ taglib uri="javapitfalls" prefix="jp" %>
04: <HTML>
05: <HEAD>
06: <TITLE>
07: Salary Servlet Jsp
08: </TITLE>
09: <META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=windows-⮐
1252">
10: </HEAD>
11: <BODY>
12: <H1>
13: Salary Jsp
14: </H1>
15: <table>
16: <tr>
17: <td><b>Employee</b></td>
18: </tr>
19:
20: <!-- Down to this simple line -->
21:
22: <jp:salary salary="<%= request.getParameter("salary") %>" />
23:
24: </table>
25:
26: </BODY>
27: </HTML>
28:
```

Listing 24.8 SalaryTag.jsp

All of this modularization has not addressed the fundamental problem we found in our workflow system. While we successfully separated logic from presentation, we had not separated out the flow control of the application. This is critical in most Web systems, but particularly in our workflow system.

The solution to this problem is the Model 2 Architecture.

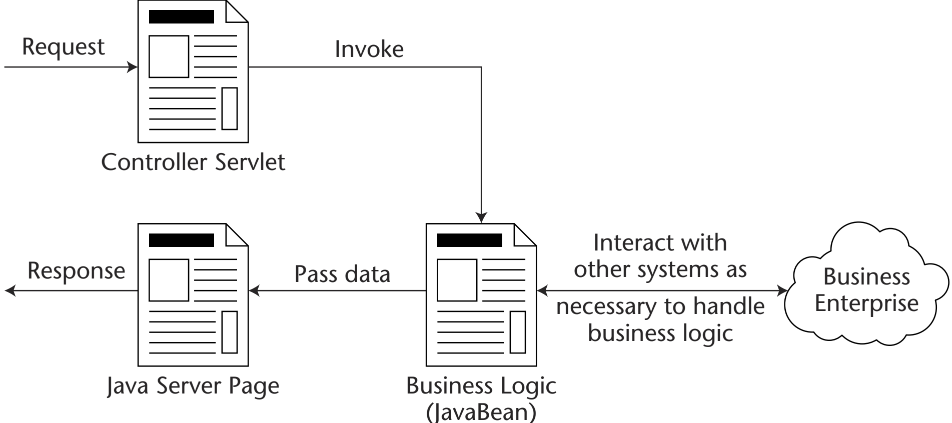


Figure 24.2 Model 2 Architecture.

JSP Development in the Model 2 Architecture

Figure 24.2 shows the Model 2 Architecture. This architecture is a JSP/servlet implementation of the popular and powerful Model-View-Controller pattern. Included is the controller, which is a servlet that responds to a request and dispatches it to the appropriate business logic component. The JavaBean wraps the business logic that needs to be executed to handle the request. From there, the bean hands off to a JavaServer Page, which controls the presentation returned in the response.

It seems like a great idea, but is it realistic to expect Web applications to build this entire infrastructure, especially considering the history of ad hoc Web development? The good thing is that you don't have to build this infrastructure. The Apache Software Foundation's Jakarta project has a rather sophisticated implementation of the Model 2 Architecture, called the Struts framework (http://jakarta.apache.org/struts).

Item 25: When Servlet HttpSessions Collide

Picture this scenario. A user is on the Internet, shopping at Stockman's Online Hardware Store, where there is an electronic commerce user interface with "shopping cart" functionality. As he browses the Web site, he decides to put a belt sander, a drill press, and an air compressor in his shopping cart. Instead of checking out what he has put in his shopping cart, he decides that he'd better buy something for his wife to ease the pain when she discovers his hardware purchases. He quickly goes to "Lace Lingerie Online," where they also have an e-commerce interface. There, he builds a shopping cart consisting of "Sensual Bubble Bath," the "Romantic Lunch Box Package for Two," and the "Lace Gift Certificate." He checks out of the lingerie store, enters his credit card information, and leaves the Web site. Now that his conscience is cleared, the

user goes back to the hardware store Web site. He clicks on the "Check out" button, but, to his surprise, his shopping cart at the Stockman Hardware store is filled with "Sensual Bubble Bath," the "Romantic Lunch Box Package for Two," and the "Lace Gift Certificate." The user is confused. Frankly, the folks processing orders at the hardware store are baffled and are calling the programmers who designed the Web site. What could have happened?

Believe it or not, that could be a feasible user scenario when this pitfall relating to the `HttpSession` class is introduced into Java servlets. `HttpSession` is a wonderful class for persisting state on the server. It is an interface that is implemented by services to provide an association (session) between a browser and the Web server's servlet engine over a period of time. Using the `HttpSession` class, you can store a great deal of information without the use of browser client-side cookies and hidden fields. In fact, you can store very complex data structures of information with the API of the `HttpSession`, which is shown in Table 25.1.

Table 25.1 HttpSession Interface

METHOD	DESCRIPTION
`long getCreationTime()`	Returns the time at which this session was created.
`String getId()`	Returns the unique identifier assigned to this session.
`long getLastAccessedTime()`	Returns the last time the current client requested the session.
`int getMaxInactiveInterval()`	Returns the maximum interval between requests that the session will be kept by the server.
`Object getValue(String)`	Returns a data object stored in the session represented by the parameter `String`. See `putValue()`.
`String[] getValueNames()`	Returns an array of all names of data objects stored in this session.
`void invalidate()`	Causes this session to be invalidated and removed.
`boolean isNew()`	Returns `true` if the session has been created by the server but the client hasn't acknowledged joining the session; otherwise, it returns `false`.
`void putValue(String, Object)`	Assigns (binds) a data object to correspond with a `String` name. Used for storing session data.

(continues)

Table 25.1 HttpSession Interface *(Continued)*

METHOD	DESCRIPTION
void removeValue(String)	Removes the data object bound by the String-represented name created with the putValue() method.
void setMaxInactiveInterval()	Sets the maximum interval between requests that the session will be kept by the server.

The API of the interface is quite simple. The most-used methods are getValue() and putValue(), where it is possible to save any Java object to the session. This is very helpful if you are developing an application that needs to save state information on the server between requests. In discussing this pitfall, we will discuss the use of this class in depth.

How does a servlet get access to the HttpSession object? The servlet's request object (HttpRequest) that is passed into the servlet's doGet() and doPost() methods contains a method called getSession() that returns a class that implements HttpSession. Listing 25.1 shows a good example of the doGet() method in a servlet using the HttpSession class. This block of code originates from our hardware store scenario discussed at the beginning of this pitfall. Notice that in line 6 of the listing, the servlet calls getSession() with the boolean parameter true. This creates an HttpSession if it doesn't already exist. On line 13, the user checks to see if the session is a new one (or if the client has never interacted with the session) by calling the isNew() method on HttpSession.

```
01:  public void doGet(HttpServletRequest request,
02:   HttpServletResponse response)
03:  throws ServletException, IOException
04:  {
05:   PrintWriter out;
06:   HttpSession session = request.getSession(true);
07:   Vector shoppingcart = null;
08:
09:   response.setContentType("text/html");
10:   out = response.getWriter();
11:   out.println("<HTML><TITLE>Welcome!</TITLE>");
12:   out.println("<BODY BGCOLOR='WHITE'>");
13:   if (session.isNew())
14:   {
15:    out.println("<H1>Welcome to Stockman Hardware!</H1>");
16:    out.println("Since you're new.. we'll show you how ");
17:    out.println(" to use the site!");
```

Listing 25.1 Block of servlet code using HttpSession

```
18:    //...
19:   }
20:   else
21:   {
22:     String name = (String)session.getValue("name");
23:     shoppingcart = (Vector)session.getValue("shoppingcart");
24:     if (name != null && shoppingcart != null)
25:     {
26:       out.println("<H1>Welcome back, " + name + "!</H1>");
27:       out.println("You have " + shoppingcart.size() + " left "
28:                   + " in your shopping cart!");
29:      //...
30:    }
31:   }
32:  //more code would follow here..
32: }
```

Listing 25.1 *(continued)*

On line 23, we see that the getValue() method is called on HttpSession to retrieve a String representing the name of the user and also a vector representing the user's shopping cart. This means that at one time in the session, there was a scenario that added those items to the session with session.putValue(), similar to the following block of code:

```
String myname="Scott Henry";

Vector cart = new Vector();
cart.add("Belt sander ID#21982");
cart.add("Drill press ID#02093");
cart.add("Air compressor ID#98983");

session.putValue("name", myname);
session.putValue("shoppingcart", cart);
```

In fact, the preceding block of code was made to follow the scenario we discussed at the beginning of this pitfall. Everything seems to follow the way the documentation describes the HttpSession API in Table 25.1. What could go wrong?

As we discussed earlier, an HttpSession exists between a browser and a servlet engine that persists for a period of time. If the values name and shoppingcart are placed to represent data objects in the HttpSession, then this session will exist *for every servlet-based application* on the server. What could this mean? If there are multiple servlet applications running on your server, they may use values such as name and shoppingcart to store persistent data objects with HttpSession. If, during the same session, the user of that session visits another servlet application on that server that uses

the same values in HttpSession, bad things will happen! In the fictional scenario where we discussed the "switched shopping carts," where lingerie items appeared in the hardware store shopping cart, it just so happened that both e-commerce sites resided on the same server and the visits happened during the same HTTP session.

"Isn't this probability slim?" you may ask. We don't think so. In the present Internet environment, it is not uncommon for e-commerce sites to exist on the same server without the end user knowing about it. In fact, it is not uncommon for an Internet hosting company to host more than one e-commerce site on the same server without the developers of the applications knowing about it!

What does this mean for you as a programmer? When using the HttpSession class, try to make certain that you will not collide with another application. *Avoid using common names storing types in* HttpSession *with the* putValue() *method.* Instead of name and shoppingcart, it may be useful to put your organization, followed by the name of the application, followed by the description of the item you are storing. For example, if you are ACME.COM, and you are developing the e-commerce application for the Stockman Hardware example, perhaps com.acme.StockmanHardware.shoppingcart would be a better choice for storing your data.

Keeping the idea of collisions in mind, look at Listing 25.2, which is used for a shopping cart "checkout" to finalize a transaction on the "Lace Lingerie" e-commerce site. Can you find anything that could cause unexpected behavior to happen? As you can see in the code listing in line 16, the programmer is using a better naming convention for the shopping cart. However, in a new scenario, when the user finishes his purchase at Lace Lingerie and returns to Stockman Hardware, his shopping cart is empty. How could this happen?

```
01:  public void checkout(PrintWriter out, HttpSession session)
02:  {
03:  /*
04:  * Call the chargeToCreditCard() method, passing the session
05:  * which has the user's credit card information, as well as the
06:  * shopping cart full of what he bought.
07:  */
08:  chargeToCreditCard(session);
09:
10:  out.println("<H2>");
11:  out.println("Thank you for shopping at Lace Lingerie Online!")
12:  out.println("</H2>");
13:  out.println("<B>The following items have been charged to ");
14:  out.println("your credit card:</B><BR>");
15:  Vector cart =
16:       session.getValue("com.acme.lacelingerie.shoppingcart");
17:
18:  Iterator it = cart.iterator();
19:
20:  while (it.hasNext())
21:  {
22:    out.println("<LI>" + it.next());
```

Listing 25.2 A checkout()portion of a Web site

```
23:  }
24:
25:  out.println("<H2>Have a nice day!</H2>");
26:
27:  session.invalidate();
28:}
```

Listing 25.2 *(continued)*

The problem lies in line 27 in Listing 25.2. As we showed in Table 25.1, calling the `invalidate()` method of the `HttpSession` interface eliminates the session and all of the objects stored in the session. Unfortunately, this will affect the *user's session for every application on the server*. In our e-commerce shopping scenario, if the user returns to another online store on the server that keeps any information in an `HttpSession`, that data will be lost.

What is the solution? Avoid the `invalidate()` method in `HttpSession`. If you are worried about leaving sensitive data in the `HttpSession` object, a better solution is shown in Listing 25.3.

```
01:  public void checkout(PrintWriter out, HttpSession session)
02:  {
03: /*
04:  * Call the chargeToCreditCard() method, passing the session
05:  * which has the user's credit card information, as well as the
06:  * shopping cart full of what he bought.
07:  */
08:  chargeToCreditCard(session);
09:
10:  out.println("<H2>");
11:  out.println("Thank you for shopping at Lace Lingerie Online!")
12:  out.println("</H2>");
13:  out.println("<B>The following items have been charged to ");
14:  out.println("your credit card:</B><BR>");
15:
16:  Vector cart =
17:       session.getValue("com.acme.lacelingerie.shoppingcart");
18:
19:  Iterator it = cart.iterator();
20:
21:  while (it.hasNext())
22:  {
23:    out.println("<LI>" + it.next());
24:  }
25:
```

Listing 25.3 Better alternative for checkout( ) *(continued)*

```
26:  out.println("<H2>Have a nice day!</H2>");
27:
28:  /*
29:   * Delete all Information related to this transaction,
30:   * because It Is confidential and/or sensitive!
31:   */
32:  session.removeValue("com.acme.lacelingerie.customername");
33:  session.removeValue("com.acme.lacelingerie.shoppingcart");
34:  session.removeValue("com.acme.lacelingerie.creditcard");
35:  session.removeValue("com.acme.lacelingerie.bodydimensions");
36:
37: }
```

Listing 25.3 *(continued)*

In lines 32 to 35 of Listing 25.3, the programmer deleted all of the objects specific to his application from the `HttpSession`. This can be a better alternative to the `invalidate()` method.

What other collision pitfalls could you encounter with this class? If you look back at the code in Listing 25.1, note the `else` clause in lines 20 to 31. The code assumed that since the `isNew()` method returned `false` on line 13, the user had previously visited that site. Now we know better. When `isNew()` returns `false`, it means that there exists a session between the browser and the server that has persisted over a matter of time. It does not mean that the user has established a session with the current servlet. The better way to write the block of code in Listing 25.1 is shown in Listing 25.4.

Listing 25.4 sends the user to the `showNewbieTheSite()` method if the `isNew()` method of `HttpSession` returns `true` on line 13. Also, it tests to see if the customer's name is in the `HttpSession` on lines 19 to 21. If the `getValue()` method returns `null`, then we know that although the session is not new, the user has not set up an account with the current e-commerce application.

```
01:  public void doGet(HttpServletRequest request,
02:   HttpServletResponse response)
03:  throws ServletException, IOException
04:  {
05:   PrintWriter out;
06:   HttpSession session = request.getSession(true);
07:   Vector shoppingcart = null;
08:
09:   response.setContentType("text/html");
10:   out = response.getWriter();
11:   out.println("<HTML><TITLE>Welcome!</TITLE>");
12:   out.println("<BODY BGCOLOR='WHITE'>");
13:   if (session.isNew())
```

Listing 25.4 A smarter way for assuming past usage

```
14:   {
15:         showNewbieTheSite(out);
16:   }
17:   else
18:   {
19:     String name =(String)session.getValue(
20:                      "com.acme.stockmanhardware.customername"
21:                     );
22:      if (name == null)
23:      {
24:          /* Here, the person might have an existing session,
25:           * but not with us!
26:           */
27:          showNewbieTheSite(out);
28:          return;
29:      }
30:      else
31:      {
32:        /* NOW we can assume that they've visited the site! */
33:        out.println("<H1>Welcome back, " + name + "!</H1>");
34:        shoppingcart = (Vector)session.getValue("shoppingcart");
35:        if (shoppingcart != null)
36:        {
37:        out.println("You have " + shoppingcart.size() +
38:                   " left in your shopping cart!");
39:        //...
40:      }
41:   }
42:   //more code would follow here..
43: }
```

Listing 25.4 *(continued)*

Finally, lines 32 to 40 can assume that the user has used the e-commerce application before!

In conclusion, be careful about your use of `HttpSession` in servlets. Be aware of the collision pitfalls that could await you in regard to naming data objects with `putValue()`, terminating the session with `invalidate()`, and testing for first-time use of the session with the `isNew()` method.

Item 26: When Applets Go Bad

In the pitfall "J2EE Architecture Considerations" (Item 37) in Part Three, we discuss a software project where we had proposed a solution that can be viewed like Figure 26.1. In that pitfall, we discuss the several scenarios for the client-side behaviors.

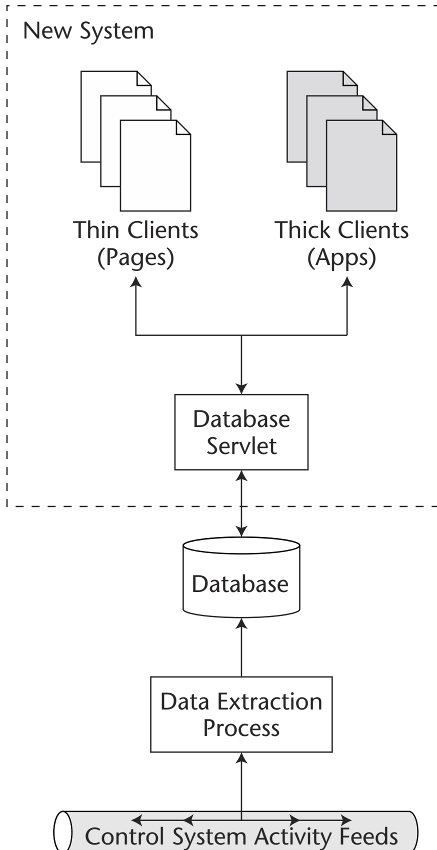


Figure 26.1 The proposed system.

Analyzing the requirements, we found there were really two types of users envisioned for this system. There were analyst personnel who needed a rich toolset by which they could pour through this voluminous set of data. Also, management personnel needed an executive-level summary of system performance. Therefore, there were truly two different clients that needed to be Web-enabled.

The analyst client needed functionality that included mapping, time lines, and spreadsheets. This was going to be the primary tool used for these personnel to perform their job and was expected to perform like the rest of the applications on their desktop machine. They wanted to be able to print reports, save their work, and most other things that users have come to expect from their PCs.

The manager client was meant to show some commonly generated displays and reports. Essentially, this would be similar to portfolio summary and headlines views. They didn't want anything more involved than pointing their Web browser at a Web site and knowing the latest information.

It was critical that the application needed to be centrally managed. Since the analysts were a widely distributed group, there could not be any expectation of doing desktop support or ensuring that the proper version or update was installed.

So we built an applet for the analyst's toolkit. Immediately, we noticed a number of problems with the applet:

It was approximately 3 MB in size. No matter what we looked at, the time line, graphing, and mapping components were simply too large to allow us to reduce the size any more. We tried a phased loading approach, but that didn't really help much; since this was a visualization suite, we couldn't really background-load anything.

The JVM versions were problematic. Moving to the Java plug-in was better, but we still had small subtleties that bothered us. Applets run within the context of the browser; even with the plug-in, they are still at the mercy of the browser's handling.

The security model was buggy and convoluted. Trying to assign permissions inside something that runs within the context of another application is fraught with issues. We had problems with saving and printing from the applet. The biggest showstopper was the fact that our Web server ran on a separate host than the database server. So if we couldn't get the permissions problem to work out, we would not be able to attach to the DB server at all. We intended to use a database access servlet, but we didn't want our hand forced this way over something that should work.

The application ran into responsiveness problems. It had buggy behavior about being unable to find a class that was not repeatable. However, it seemed these problems went away when it was run from an application.

Most of the problems regarding the security model and browser versioning were nuisances that could be worked around. However, we could not get past the long download time of the application. Since the users were widely dispersed and had varying degrees of network communications, this could be an unbearable wait.

The solution to this problem is Java Web Start. Java Web Start is Sun's implementation of the Java Network Launching Protocol (JNLP) and is packaged with JDK 1.4. Java Web Start essentially consists of a specialized configuration file (with the "jnlp" extension) associated with a special MIME type, combined with the Java class loader to allow deployment of Java class files over HTTP. Since the class loading and Java Runtime Environment are configured in accordance with the JNLP, the class files can be cached and downloaded as needed. This provides a solution for centrally managing Java applications and allowing clients to cache Java classes locally. When a patch or change is released, it is propagated automatically out to the clients without your having to download all of the classes again.

Furthermore, Java Web Start provides the ability to update the Java Runtime Environment versions as necessary to suit the applications being deployed. Also, the Java Security model is consistent and built into Java Web Start. The "sandbox" is in effect by default, and the user must grant permissions as appropriate. An interesting thing to note is that you get the security sandbox in a full application context (i.e., a main method) without having to use the indirection execution of an applet.

Examining the JNLP file format provides a great deal of insight into the capabilities of Java Web Start. Listing 26.1 is an example file.

```
01: <?xml version="1.0" encoding="utf-8"?>
02: <!-- JNLP File for Analyst's Toolkit Demo Application -->
03: <jnlp
04:   spec="1.0+"
05:   codebase="http://www.javapitfalls.org/apps"
06:   href="toolkit.jnlp">
07:   <information>
08:     <title>Analyst's Toolkit</title>
09:     <vendor>McDonald-Bradley, Inc.</vendor>
10:     <homepage href="docs/help.html"/>
11:     <description>Analyst's Toolkit Application</description>
12:     <description kind="tooltip">The Analyst's Toolkit</description>
13:     <icon href="http://www.javapitfalls.org/images/toolkit.jpg"/>
14:     <offline-allowed/>
15:   </information>
16:   <security>
17:       <all-permissions/>
18:   </security>
19:   <resources>
20:     <j2se version="1.4"/>
21:     <j2se version="1.3"/>
22:     <jar href="lib/toolkit.jar"/>
23:   </resources>
24:   <application-desc main-class="Toolkit"/>
25: </jnlp>
26:
```

Listing 26.1 toolkit.jnlp

This file shows the configuration of the Java Web Start software. There are four different subsections in the JNLP file: information, security, resources, and application.

The information element provides the display information for the user to view the application before having to run it. It provides descriptions, icons, vendor, and title. However, there is one more interesting tag in the information element: `<offline-allowed/>`. This means that the Web Start application can be run without being connected to the network! This is another advantage over applets.

The security element defines the permissions needed to run this application. If the `<all permissions>` tag is not used, then the application runs within the sandbox by default (as it would if the user chose not to grant permissions). Note that if the `<all-permissions>` tag is used, then all JAR files must be signed.

The resources element specifies how the system should download resources like JAR files and native libraries. Also, it specifies the version of the JRE that is required to run the application, as well as the preference order by which compatible ones should be used. For instance, in the example above, both the JRE 1.4 and 1.3 are allowed as compatible JREs, and JRE 1.2 is not. It prefers JRE 1.4 but will settle for JRE 1.3. The native libraries can be specified in terms of the operating systems with which they are compatible. Also, all resources can be specified as to whether they should be downloaded in an eager or lazy mode—that is, whether they should be downloaded immediately or as allowable.

Figure 26.2 is an example of what happens when a client requests a Java Web Start application.

The question becomes how to check on whether or not JNLP is installed. The way to handle this is to use a client-side script to ask the browser whether it can handle the `application/x-java-jnlp-file` MIME type. Listing 26.2 is the example provided by Sun in the Java Web Start developer's kit (http://java.sun.com/products/javawebstart/docs/developersguide.html):

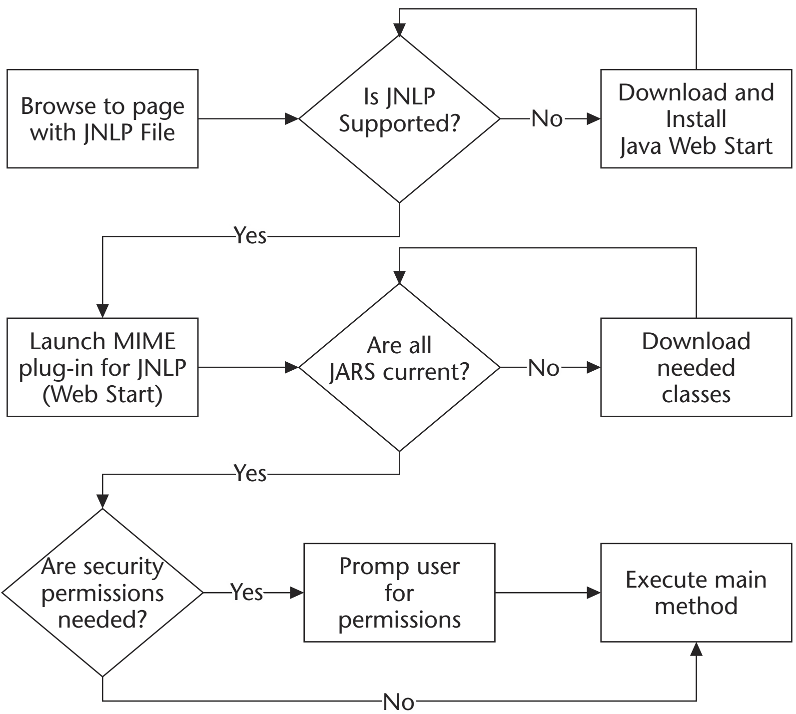


Figure 26.2 Java Web Start application invocation.

```
01:     <SCRIPT LANGUAGE="Javascript">
02:        var javawsInstalled = 0;
03:        isIE = "false";
04:
05:        if (navigator.mimeTypes && navigator.mimeTypes.length) {
06:           x = navigator.mimeTypes['application/x-java-jnlp-file'];
07:           if (x) javawsInstalled = 1;
08:        } else {
09:           isIE = "true";
10:        }
11:
12:        function insertLink(url, name) {
13:         if (javawsInstalled) {
14:              document.write("<a href=" + url + ">"  + name + "</a>");
15:           } else {
16:              document.write("Need to install Java Web Start");
17:           }
18:        }
19:    </SCRIPT>
20:
21:    <SCRIPT LANGUAGE="VBScript">
22:        on error resume next
23:        If isIE = "true" Then
24:          If Not(IsObject(CreateObject("JavaWebStart.IsInstalled"))) ⮐
Then
25:                javawsInstalled = 0
26:          Else
27:                javawsInstalled = 1
28:          End If
29:        End If
30:    </SCRIPT>
31:
```

Listing 26.2 Script to check for Java Web Start

Figure 26.3 is an example of how the deployment of a Java Web Start application is handled.

So, Java Web Start is really cool and makes great sense as a mechanism for centrally managing cross-platform applications, but there is also a full set of APIs that allow the application developer to control the Java Web Start functionality in his or her application.

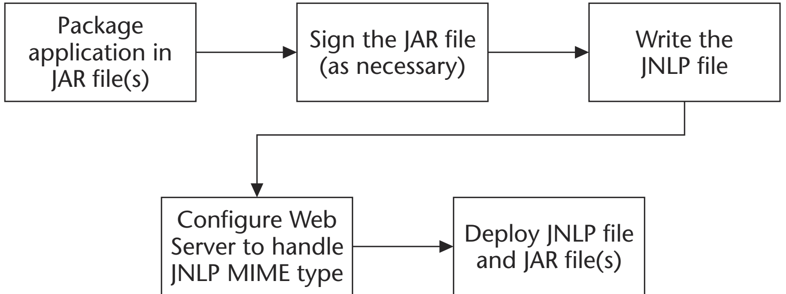


Figure 26.3 Java Web Start application deployment.

Here are some examples of the services in the `javax.jnlp` package:

`BasicService`. This service provides functionality like `AppletContext`, which allows the user to do things like kicking off the environments default browser to display a URL.

`ClipboardService`. This service provides access to the system's clipboard. If the user is operating inside the sandbox, he will be warned about the security risk.

`DownloadService`. This service manages the downloading and caching of Web Start JAR files.

`FileOpenService`. Browsers have the functionality in Web pages to allow the user to browse for a file to open. You most often see this in file upload forms. This service does a similar thing even from inside the sandbox.

`FileSaveService`. This replicates the "Save as" functionality of Web browsers, even within the sandbox.

`PersistenceService`. This can be thought of like a cookie, which allows certain information to be stored even within the browser security model.

`PrintService`. This allows the user to print files from the Web Start application, after accepting the Print Service prompt.

Looking at the JNLP API, we note all of these services are defined as interfaces. Listing 26.3, from Sun's documentation (http://java.sun.com/products/javawebstart/1.2/docs/developersguide.html), shows how you can get an object that implements these interfaces and also demonstrates using it for controlling the `DownloadService`.

```
01: import javax.jnlp.*;
02:     ...
03:
04:     DownloadService ds;
05:
06:     try {
07:         ds = 
(DownloadService)ServiceManager.lookup("javax.jnlp.DownloadService");
08:     } catch (UnavailableServiceException e) {
09:         ds = null;
10:     }
11:
12:     if (ds != null) {
13:
14:         try {
15:             // determine if a particular resource is cached
16:             URL url =
17:                     new 
URL("http://java.sun.com/products/javawebstart/lib/draw.jar");
18:             boolean cached = ds.isResourceCached(url, "1.0");
19:             // remove the resource from the cache
20:             if (cached) {
21:                 ds.removeResource(url, "1.0");
22:             }
23:             // reload the resource into the cache
24:             DownloadServiceListener dsl = 
ds.getDefaultProgressWindow();
25:             ds.loadResource(url, "1.0", dsl);
26:         } catch (Exception e) {
27:             e.printStackTrace();
28:         }
29:     }
30:
```

Listing 26.3 Example using DownloadService

Looking at these services, we see that it begins to look like Java Web Start is a pseudo Web browser. Consider the evolution of applets. Many saw them as the answer to centrally managed applications deployed over HTTP in a Web browser. However, this model of running within the context of a browser became very inconsistent and buggy. Therefore, the answer was to reengineer the entire solution, looking at the disadvantages and determining how to effectively handle this problem. Java Web Start and the Java Native Launching Protocol is the answer to this problem.

Item 27: Transactional LDAP—Don't Make that Commitment

The advent of distributed Web applications has created new channels for people to perform business transactions and to communicate with one another. These online dealings spawned the need for authentication mechanisms to ensure end users were properly recognized so that they would be privy to proprietary content. Eventually, this led to the creation of profile files that captured user roles inside and outside of an organization so that Web site content could be targeted properly during operations.

The practice of role assignments and efforts to define user communities evolved into a process called *personalization*. Personalization was predicated on an understanding of a user's preferences and role information. Back-end matching algorithms were developed so that pertinent data could be rendered to users based on their preferences and roles.

Profile data often constituted generic data about users (name, address, phone number), and role information often indicated his or her position inside and outside a corporation. When users did not belong to an organization, they were afforded the status of guest.

These authentication and personalization activities raised many questions about data exposure and integrity, along with proper delivery and storage mechanisms for that information. Some, specifically, database vendors, suggested that role and preference information should be stored in a database. Others suggested that this information should be collected in an LDAP directory server. For enterprise system deployments, both can be implemented, but the proper manner to store and retrieve profile and personalization information would be a combination of both. An LDAP directory should be used to house fairly static information—namely, items that don't change all that often—and user role information. Databases should be used for dynamic data, which means data that changes often.

From an architectural perspective, Relational Database Management Systems (RDBMSs) are flat tables with no mechanisms that reflect their organizational structure, while LDAP directories render data in treelike structures that allow users to retrieve organized data and view their spatial relationships. LDAP directories were designed for high-speed read operations and high availability through replication. Migration of dynamic data to an LDAP directory server would be a bad idea because that would be forcing the protocol to perform operations that it was not designed to do.

Two differences between LDAP directories and database systems are how they retain data and perform security activities. RDBMSs typically use table-oriented read/write operations, while LDAP directories concentrate on attribute-oriented security. LDAP directories are constructed to work with multivalue attributes, while traditional relational databases would have to perform SQL join operations to achieve the same functionality. This join-ing of tables in database operations generally hampers performance.

Since LDAP directories were designed for low-throughput and high-speed operations as well as high availability through replication, they've been known to produce incorrect answers from their repositories because of transactional consistencies during write and replication activities. Aberrations like these can be tolerated for storage and retrieval of simple queries, but not with mission-critical data, and that is why data transactions in LDAP directories should be avoided.

RDBMS support for ACID transactions (i.e., transactions that support atomicity, consistency, isolation, and durability) make databases the preferred choice for handling important data. Their implementation ensures predictability and reliability of data transactions. The hierarchical nature of LDAP directories often models real-world data better than flat files in relational databases. It just makes sense to store dynamic data in RDBMSs because they are more flexible and reliable in handling transactions, especially commit and rollback operations.

Personalization is generally considered a dynamic and personalized content delivery system that promotes self-service activities, which in turn strengthens user relationships. The objective of personalization is to deliver pertinent and targeted content to users so that user retention can be maintained and strengthened. When users talk about personalization, they often confuse it with customization. Customization should be considered a subset of personalization, because personalization dictates what content the user has access to and customization allows users to whittle down and modify the presentation of that data. On many applications, specifically, portals, users can customize visualization preferences (what portlets do I want to view? where I want to view them on the page?). More often than not, personalization is based on predetermined role information or previous clickstream detections. In some instances, personalization is based on information explicitly provided by users who are confident in their application's security to provide personal information like salary information or buying preferences. With personalized data, applications can create user communities and matching agents can target content to users based on that information.

To demonstrate how an LDAP directory and an RDBMS can be used in tandem to deliver personalized content, a sample application was developed that retrieves relatively stable profile data in an LDAP directory and stores more dynamic data in a relational data store. The authentication and personalization process is illustrated in Figure 27.1.

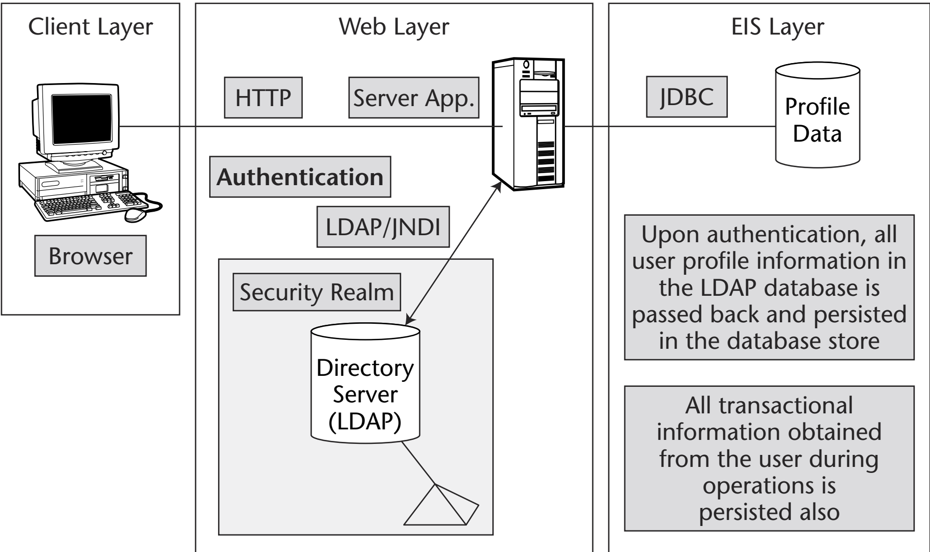


Figure 27.1 LDAP authentication.

The ControllerServlet class in Listing 27.1 implements the Front Controller pattern to control all requests through a single servlet application and dispatches requests based upon user role information derived from an LDAP directory server. If a user is recognized as a valid entry in the LDAP directory, that user will be forwarded to the Web page that reflects his or her role.

```
001: package org.javapitfalls.item27;
002:
003: import java.io.*;
004: import java.util.*;
005: import javax.servlet.*;
006: import javax.servlet.http.*;
007: import org.javapitfalls.item27.*;
008:
009: public class ControllerServlet extends HttpServlet {
010:
011:   String direction;
012:   String pageNumber;
013:   String username;
014:   String password;
015:   boolean loggedIn = false;
016:   ldapAuthenticate la;
017:   authenticateHelper helper;
018:
019:   public void service(HttpServletRequest req,
020:                        HttpServletResponse res)
021:                       throws ServletException, java.io.IOException {
022:
023:     direction = req.getParameter("direction");
024:     pageNumber = req.getParameter("page");
025:     username = req.getParameter("username");
026:     password = req.getParameter("password");
027:     if (loggedIn) {
028:
029:       if ( (username != la.getUsername()) ||
030:            (password != la.getPassword()) ) {
031:         loggedIn = false;
032:       }
033:     }
034:     if (!loggedIn) {
035:       if ((username != null) && (password != null) &&
036:            (!username.equals("")) && (!password.equals(""))) {
037:
```

Listing 27.1 ControllerServlet.java *(continued)*

On line 38, an instance of the authentication application named `ldapAuthenticate` is passed the username and password entered by the user from the login.jsp file dispatched on line 83. When valid entries have been made by the user, the profile data collected from the LDAP authentication and query will be passed along to the database through the helper class `authenticateHelper.java`. This process is shown in Figure 27.1.

```
038:         la = new ldapAuthenticate(username, password);
039:
040:         if ( (la.validUser()) && (la.getSearchCount() > 0) ) {
041:
042:           loggedIn = true;
043:           helper.setDB(la.getEmail());
044:
045:         } else {
046:
047:           System.out.println("ERROR: Invalid user.");
048:           getServletConfig().getServletContext().
049:             getNamedDispatcher("InvalidUser").forward(req, res);
050:         }
051:       }
052:
053:     }
054:
055:     if (loggedIn) {
056:
057:       if (pageNumber == null) {
058:
059:         pageNumber = "Page1";
060:
061:       } else {
062:
```

Listing 27.1 *(continued)*

The `isUserInRole` method checks for authenticated users in the application so that proper navigation can take place. This role information is passed from the LDAP directory and the JNDI realm that was implemented to expose those roles. If a user is determined to possess an admin role, that user will be forwarded to the Page4.jsp.

```
063:         if (req.isUserInRole("admin")) {
064:
065:           pageNumber = "Page4";
066:
067:         } else if (req.isUserInRole("manager")) {
068:
069:           pageNumber = "Page3";
```

Listing 27.1 *(continued)*

```
070:
071:         } else if (req.isUserInRole("tester")) {
072:
073:           pageNumber = "Page2";
074:
075:         } else {
076:
077:           pageNumber = "Page1";
078:
079:         }
080:
081:       }
082:
083:       getServletConfig().getServletContext().
084:         getNamedDispatcher(pageNumber).forward(req, res);
085:
086:     } else {
087:
088:       System.out.println("Login error...");
089:       getServletConfig().getServletContext().
090:         getNamedDispatcher("Login").forward(req, res);
091
092:     }
093:
094:   }
095:
```

Listing 27.1 *(continued)*

The Front Controller pattern was employed in this application indicated by the `Servlet` controller, the dispatcher process shown on Lines 48, 83, and 89. Also part of that pattern implementation is a helper routine shown in the `init()` method on lines 97 and 98. This one-time call determines the database properties and performs connection and query update operations on profile data that needs to be pushed to the back-end data store and predetermined and user-defined personalization preferences.

```
095:   public void init() throws ServletException {
096:
097:     helper = new authenticateHelper();
098:     helper.getDBProperties();
099:
100:   }
101:
102: }
103:
```

Listing 27.2 authenticateHelper.java

The `authenticateHelper` class shown in Listing 27.3 loads the database.properties file. It then extracts the key properties and stores them in instance variables:

```
01: package org.javapitfalls.item27;
02:
03: import java.io.*;
04: import java.net.*;
05: import java.util.*;
06: import javax.servlet.*;
07: import javax.servlet.http.*;
08:
09: public class authenticateHelper extends HttpServlet {
10:
11:  String driver;
12:  String dbname;
13:  String username;
14:  String password;
15:
16:  authenticateHelper() {}
17:
18:  public void getDBProperties() {
19:
20:   URL url = null;
21:   Properties props = new Properties();
22:
23:   try {
24:    url = this.getClass().getClassLoader().
25:                     getResource("database.properties");
26:    props.load( new FileInputStream(url.getFile()) );
27:    // Get properties
28:    driver = props.getProperty("driver");
29:    dbname = props.getProperty("dbname");
30:    username = props.getProperty("username");
31:    password = props.getProperty("password");
32:   }
33:   catch(Exception e) {
34:    System.out.println("ERROR:" + e.toString());
35:   }
36:
37:  }
38:
39:  public void setDB(String s) {
40:
41:   System.out.println("INSIDE setDB()...email= " + s);
42:   // UPDATE entry in database with email address.
43:  }
44: }
```

Listing 27.3 authenticateHelper

The ldapAuthenticate routine in Listing 27.4 receives the username and password from the login.jsp and searches the LDAP directory to determine if the user is a valid user that should be authenticated.

```
001: package org.javapitfalls.item27;
002:
003: import java.io.*;
004: import java.net.*;
005: import java.util.*;
006: import java.text.*;
007: import java.util.Date;
008: import javax.naming.*;
009: import javax.naming.directory.*;
010:
011: public class ldapAuthenticate {
012:
013:  String username;
014:  String password;
015:  String firstname;
016:  String lastname;
017:  String email;
018:  int searchCount;
019:
020:    public ldapAuthenticate(String u, String p)
021:    {
022:      username = u;
023:      password = p;
024:      setUsername(u);
025:      setPassword(p);
026:    }
027:
028:    public boolean validUser() {
029:
030:      try {
031:        search();
032:      }
033:      catch(Exception e) {
034:        System.out.println("ERROR: " + e.toString());
035:        return false;
036:      }
037:      return true;
038:    }
039:
040:    public boolean search() {
041:
```

Listing 27.4 ldapAuthenticate.java *(continued)*

The `DirContext` class allows our application to search the directory for the username specified in the login script. This operation is followed by the `Attribute` operation on the `DirContext` object, which is used for the username lookup.

```
042:       try {
043:          DirContext ctx =  getDirContext();
044:          Attributes matchAttrs = new BasicAttributes(true);
045:          matchAttrs.put(new BasicAttribute("uid", username));
046:
047:          NamingEnumeration result = ctx.search("dc=avey",
048:                                                    matchAttrs);
049:          int count = 0;
050:          while (result.hasMore()) {
051:            SearchResult sr = (SearchResult)result.next();
052:            System.out.println("RESULT:" + sr.getName());
053:            printAttributes(sr.getAttributes());
054:            count++;
055:          }
056:          System.out.println("Search returned "+ count+ " results");
057:          setSearchCount(count);
058:          ctx.close();
059:       }
060:       catch(NamingException ne) {
061:         System.out.println("ERROR: " + ne.toString());
062:         return false;
063:       }
064:       catch(Exception e) {
065:         System.out.println("ERROR: " + e.toString());
066:         return false;
067:       }
068:       return true;
069:     }
070:
071:     public void printAttributes(Attributes attrs)throws Exception {
072:
073:        if (attrs == null) {
074:          System.out.println("This result has no attributes");
075:        } else {
076:
077:        try {
078:
079:          for (NamingEnumeration enum = attrs.getAll();
080:                                                   enum.hasMore();) {
081:            Attribute attrib = (Attribute)enum.next();
082:
083:            for(NamingEnumeration e = attrib.getAll();e.hasMore();) {
084:              String s = e.next().toString();
085:              System.out.println("attrib = " + s + "\n");
086:
```

Listing 27.4 *(continued)*

```
087:             if (attrib.getID().equals("mail")) {
088:               setEmail(s);
089:             } else if (attrib.getID().equals("givenName")) {
090:                setFirstname(s);
091:              } else if (attrib.getID().equals("surName")) {
092:                setLastname(s);
093:           }
094:         }
095:       }
096:
097:     } catch (NamingException ne) {
098:       System.out.println("ERROR: " + ne.toString());
099:     }
100:
101:   }
102:
103:  }
104:
105:  public DirContext getDirContext() throws Exception {
106:
```

Listing 27.4 *(continued)*

The `InitialDirContext` class is the starting context for performing directory operations. The `Hashtable` items are the environment properties that allow the application to construct a directory context to be searched.

```
107:    Hashtable env = new Hashtable(11);
108:    env.put(Context.INITIAL_CONTEXT_FACTORY,
109"            "com.sun.jndi.ldap.LdapCtxFactory");
110:    env.put(Context.PROVIDER_URL, "ldap://localhost:389");
111:    env.put(Context.SECURITY_PRINCIPAL, username);
112:    env.put(Context.SECURITY_CREDENTIALS, password);
113:    DirContext ctx = new InitialDirContext(env);
114:    return ctx;
115:
116:  }
166: }
167:
```

Listing 27.4 *(continued)*

Customarily, the personalization process is incorporated after several iterations of a program, because it is a difficult process to capture. Personalization preferences and matching algorithms often vacillate, which hampers their implementation. That is why role information, which is relatively stable, should be stored in a directory server and personalization preferences should be part of a database that can handle transactions in a more efficient manner.

Consider the following. An LDAP directory is a lightweight database that acts as a central repository for object management and is searched far more often than it should be written to. Its real strength lies in its ability to replicate data across networks, which comes at the expense of transactional protections. Because of this, an RDBMS should be deployed to manage dynamic data, like personalization preferences, which often change regularly.

When you are building distributed business systems, keep in mind that transactional data should be handled with a database, and role information and static user profile information should be managed by an LDAP directory. LDAP directories serve as a complementary technology to database systems, because they can be used to organize and search for relatively static user information in an efficient manner, while database systems can accommodate frequent fluctuations in data.

Item 28: Problems with Filters

Most Web developers want to craft applications that handle user requests, accommodate modifications, and render pertinent content in an efficient manner. Often this means migration to a controller that processes requests and renders different views based on user selections. More often than not, the Model-View-Controller (MVC) pattern is used because it allows components to be separated from application interdependencies, which enables modifications to be made in an easy fashion.

The three elements of the MVC pattern are as follows:

- A *model* that represents the application data and business logic that dictates the availability and changes of the application data.
- A *view* that renders the model data and forwards user input to the controller.
- A *controller* that determines presentation views and dispatches user requests.

Applications that adhere to the MVC design pattern decouple application dependencies, which means that component behavior can be separated and modifications to one layer, say the model tier, can be made and will not affect the view and controller tiers. This improves application flexibility and reusability because changes don't reverberate through the entire system. The MVC paradigm also reduces code duplication and makes applications easier to maintain. It also makes handling data easier, whether you are adding new data sources or changing data presentation, because business logic is kept separate from data.

The MVC solution seems simple enough, but the implementation of simple solutions can often be a different matter. Application controller implementations can assume many forms. Sometimes they involve having several applications pass requests among each other with hard-coded navigation context paths. In many JavaServer Page solutions, a left navigation page is incorporated into a home page that includes header, footer, and content pages, and requests propagate between all of these pages. Problems arise when modifications need to be made to those pages that share a link, perhaps a header and a footer page, because corrections need to be made to both pages. Additionally, the management of parameters between applications becomes cumbersome, and changes don't always propagate properly.

Depending on your situation, the JSP solution described above could be right for your deployment, but for enterprise deployments, the appropriate solution is a single servlet or JavaServer controller page that serves as a common entry point for all application requests. When a single servlet controller is implemented, it makes examining and monitoring all process requests easy.

In addition to servlet controller implementations in MVC applications, filters are needed to enhance request processing because of their ability to transform requests and forward responses. Filters also allow applications to log, audit, and perform security role administration operations prior to request forwarding, which previously had to be performed by additional code and requests.

Filters are generally used to modify request headers and data. More importantly, filters allow requests to be processed in a chainlike fashion. Filter chain implementations locate chain items sequentially through the `doFilter` method and pass along request and response objects through the chain. When the last filter in the filter chain has been processed, the target servlet is summoned.

Figure 28.1 visually demonstrates how a Web controller implementation should be developed. All configuration data should be aggregated in the deployment descriptor (web.xml), including servlet, filter, and JSP mappings. With this architecture, all requests are “filtered” prior to being passed on to the servlet controller that passes control to the JSP visualization applications. The persistence layer on the outer perimeter interacts with both the servlet and JSP applications.

The J2EE specification defines a deployment descriptor as an XML file that describes components between the application assembler and the deployer. In Listing 28.1 all JSP and servlet mappings are migrated to the web.xml file. All controller management emanates from the web.xml file, which simplifies the application’s deployment and facilitates modifications.

As part of the deployment described below, the Front Controller pattern is applied to the controller implementation in the ControllerServlet.java program. With this pattern, a controller component manages the user requests, and a dispatcher component manages navigation and user views.

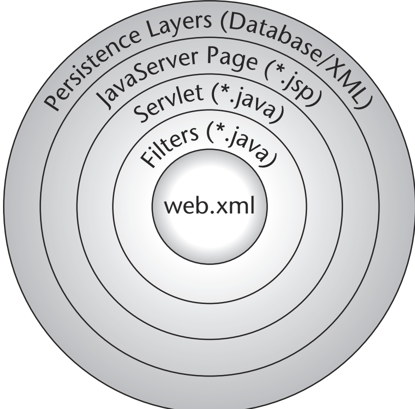


Figure 28.1 Web controller architecture.

```
01: package org.javapitfalls.item28
02: import java.io.*;
03: import java.util.*;
04: import javax.servlet.*;
05: import javax.servlet.http.*;
06:
07: public class ControllerServlet extends HttpServlet {
08:
09:   public void service(HttpServletRequest req,
10:                       HttpServletResponse res)
11:                       throws ServletException, java.io.IOException {
12:
13:     String direction = req.getParameter("direction");
14:     String pageNumber = req.getParameter("page");
15:     if (pageNumber == null)
16:       pageNumber = "Page1";
17:     else {
18:       if (direction.equals("next")) {
```

Listing 28.1 ControllerServlet.java

Line 19 in Listing 28.1 illustrates how J2SDK 1.4 regular expressions can be used to parse string variables. The code parses the first four characters of the page request variable with `\\w{4}` and extracts the fifth character and, depending on the direction parameter, increases or decreases that value and resets the page mapping request parameter with the new value.

```
19:        int iPage = 
Integer.parseInt(pageNumber.replaceAll("^(\\w{4})(\\d{1})", "$2")) + 1;
20:        if (pageNumber.equals("Page4"))
21:          pageNumber = "Page1";
22:        else
23:          pageNumber = "Page" + String.valueOf(iPage);
24:       }
25:       else {
26:         int iPage = 
Integer.parseInt(pageNumber.replaceAll("^(\\w{4})(\\d{1})", "$2")) - 1;
27:         if (pageNumber.equals("Page1"))
28:           pageNumber = "Page4";
29:         else
30:           pageNumber = "Page" + String.valueOf(iPage);
31:       }
32:     }
33:     // forward page request: Page1, Page2, Page3, Page4
34:     // works too
35:     // RequestDispatcher rd =
36:     //       req.getRequestDispatcher("/test2/jsp/second.jsp");
37:     // rd.forward(req, res);
```

Listing 28.1 *(continued)*

```
38:     // works too
39:     // String contextPath = req.getContextPath();
40:     // res.sendRedirect( contextPath + "/test2/jsp/second.jsp" );
41:     getServletConfig().getServletContext().
42:                 getNamedDispatcher(pageNumber).forward(req, res);
43:   }
44:
45: }
46:
```

Listing 28.1 *(continued)*

On line 35 of Listing 28.1 the `getRequestDispatcher` method gets a `RequestDispatcher` instance from its request object so that it can dispatch the JSP component to the given URI path `"/test2/jsp/second.jsp"`. In other words, this method takes a user-specified path that is relative to the servlet's root context and wraps it with a `RequestDispatcher` object to forward a user request. The relative pathname cannot extend outside the current servlet context, so if the path begins with a "/", it is interpreted as relative to the current context root. The forward method of the `RequestDispatcher` interface emulates the JSP directive `<jsp:forward>`, which will throw the `IllegalStateException` if the application's response buffer has data that has not been committed to the user requests. The forward operation delegates all processing of the request to the target application. With filters, applications can now perform preprocessing prior to the forward operation, which includes logging, security, and request header modifications.

The migration of JSP, filter, and servlet configuration mappings to the deployment descriptor allows the `getNamedDispatcher` method on line 41 to map the appropriate JSP page in the `pageNumber` variable so that the request can be forwarded properly. The web.xml file in Listing 28.6 specifies all the JSP pages (`Page1`, `Page2`, `Page3`, `Page4`) that are available for invocation.

An important result of moving the business logic into the servlet controller is that scriptlet code is reduced in the JSP user interface components and maintenance is facilitated.

```
01:
02: <form method="post">
03:
04:   [ content goes here. ]
05:
06:   <input type="hidden" name="page" value="Page1">
07:   <input type="submit" name="direction" value="prev">
08:   <input type="submit" name="direction" value="next">
09:
10:
11:   </form>
```

Listing 28.2 This is a sample test page test1.jsp

The FileServlet application in Listing 28.3 is invoked after the filter applications are processed. In our sample application the FileServlet application processes the `filename` parameter in the URL and displays the content data in HTML and XML format based on that XSL stylesheet filename and the XML filename affiliated with that request. Figure 28.2 shows the HTML presentation based on the filename htmlStates.xsl being applied to the states.xml file. The filename (states.xml) that has the user-specified XSL stylesheet applied to it is specified in line 31 of the web.xml file, which is a lot easier to maintain and modify than the property file (controller.properties) shown on lines 19 to 22.

```
01: package org.javapitfalls.item28;
02:
03: import java.net.URL;
04: import java.io.*;
05: import java.util.*;
06: import javax.servlet.*;
07: import javax.servlet.http.*;
08:
09: public class FileServlet extends HttpServlet {
10:
11: private String filename = "";
12:
13: public void doGet (HttpServletRequest req,
14:                    HttpServletResponse res)
15:                    throws ServletException, IOException {
16:   PrintWriter out = res.getWriter();
17:   try {
18:     /*
19:     Properties resource = new Properties();
20:     URL url = this.getClass().getClassLoader().
21:                   getResource("controller.properties");
22:     resource.load( new FileInputStream(url.getFile()) );
23:     */
24:     // use filename from init() method
25:     // File file = new File(getServletContext().
26:     //            getRealPath(resource.getProperty("filename")));
27:     File file = new
28:             File(getServletContext().getRealPath(filename));
29:     BufferedReader reader =
30:             new BufferedReader(new FileReader(file));
31:     while(reader.ready()) out.println(reader.readLine());
32:   } catch (Exception e){
33:       out.println("ERROR: " + e.toString());
```

Listing 28.3 FileServlet.java

```
34:   }
35:   out.close();
36: }
37:
38: public void init() throws ServletException {
39:   filename = getInitParameter("filename");
40: }
41:
42: }
43:
```

Listing 28.3 *(continued)*

The Web display in Figure 28.2 is the result of the XSL stylesheet displayed in Listing 28.7 called htmlStates.xsl being applied to the xml.states file. A different view would be rendered if the alternative sample stylesheet xmlState.xsl was specified.

The XSLTransformFilter application in Listing 28.4 performs a transformation on the states.xml file with the user-specified XSL stylesheet in the `filename` parameter in the URL request.

Figure 28.2 XSLT output.

```
01: package org.javapitfalls.item28;
02:
03: import java.io.*;
04: //import java.util.logging.*;
05: import javax.servlet.*;
06: import javax.servlet.http.*;
07: import javax.xml.transform.*;
08: import javax.xml.transform.stream.*;
09:
10: public class XSLTransformFilter implements Filter {
11:
12:   private FilterConfig filterConfig = null;
13:   // Reference below renders a compile-time error [reference to
14:   // Filter is ambiguous] because javax.servlet.Filter and
15:   // java.util.logging.Filter conflict
16:   // private static java.util.logging.Logger logger =
17:   // java.util.logging.Logger.getLogger(XSLTFilter.class.getName());
18:
19:   public void doFilter(ServletRequest request,
20:                        ServletResponse response, FilterChain chain)
21:                        throws IOException, ServletException {
22:
23:     // commented out because of Filter conflict
24:     // logger.setLevel(Level.ALL);
25:     // logger.info("[XSLTFilter]");
26:
27:     String filename = request.getParameter("filename");
28:     String contentType = "text/html";
29:     String styleSheet = "/data/" + filename;
30:     if (filename.startsWith("xml"))
31:         contentType = "text/plain";
32:
33:     response.setContentType(contentType);
34:     String stylePath = filterConfig.getServletContext().
35:                                         getRealPath(styleSheet);
36:     Source styleSource = new StreamSource(stylePath);
37:     PrintWriter out = response.getWriter();
38:     Wrapper wrapper = new Wrapper((HttpServletResponse)response);
39:
40:     chain.doFilter(request, wrapper);
41:
42:     StringReader sr = new StringReader(wrapper.toString());
43:
44:     StreamSource xmlSource = new StreamSource(                    ↩
filterConfig.getServletContext().getRealPath("/data/states.xml") );
45:
46:     try {
47:       TransformerFactory transformerFactory =                     ↩
                TransformerFactory.newInstance();
```

Listing 28.4 XSL TransformFilter.java

```
48:       Transformer transformer =
49:             transformerFactory.newTransformer(styleSource);
50:       ByteArrayOutputStream baos = new ByteArrayOutputStream();
51:       StreamResult result  = new StreamResult(baos);
52:       transformer.transform(xmlSource, result);
53:       response.setContentLength(baos.toString().length());
54:       out.write(baos.toString());
55:
56:     } catch(Exception ex) {
57:       out.println(ex.toString());
58:       out.write(wrapper.toString());
59:     }
60:   }
61:   public void init(FilterConfig filterConfig) {
62:    this.filterConfig = filterConfig;
63:   }
64:   public void destroy(){
65:    this.filterConfig = null;
66:   }
67: }
68:
```

Listing 28.4 *(continued)*

The filter's `FilterConfig` object on line 61 has access to the `ServletContext` of the Web application, so the container can pass information to the application and state information can be stored.

In Listing 28.4, lines 04 and 17 attempt to use the `Logger` class that is part of the J2SDK 1.4 implementation but renders a compile-time error (reference to Filter is ambiguous) because the `javax.servlet.Filter` class conflicts with the `java.util.logging.Filter` class.

```
01: package org.javapitfalls.item28;
02:
03: import java.io.*;
04: import java.util.*;
05: import javax.servlet.*;
06: import javax.servlet.http.*;
07:
08: public final class ControllerFilter extends HttpServlet
09:                                                    implements Filter {
10:
11:   private ServletContext ctx;
12:
```

Listing 28.5 ControllerFilter.java *(continued)*

```
13:   public void doFilter(ServletRequest req,
14:                          ServletResponse res,
15:                          FilterChain chain)
16:                          throws IOException, ServletException {
17:
18:     PrintWriter out = res.getWriter();
19:     Wrapper wrapper = new Wrapper((HttpServletResponse)res);
20:     chain.doFilter(req, wrapper);
21:     String filename = req.getParameter("filename");
22:     String wts = wrapper.toString();
23:
24:     if (filename != null && filename.startsWith("html")) {
25:       StringBuffer sb = new StringBuffer();
26:       sb.append("<html><body>");
27:       sb.append(wts);
28:       sb.append("</body></html>");
29:       res.setContentLength(sb.toString().length());
30:       out.write(sb.toString());
31:     } else {
32:       out.write(wrapper.toString());
33:     }
34:     out.close();
35:   }
36:
37:   public void destroy() {}
38:   public void init(FilterConfig config) throws ServletException {
39:     ctx = config.getServletContext();
40:   }
41: }
42:
```

Listing 28.5 *(continued)*

Because the filters' applications are run sequentially prior to the invocation of the requested servlet, the `doFilter()` method of `XSLTFilter` will be run prior to the same method in the ControllerFilter application. The invocation sequence can be seen in Listing 28.6 on lines 7 and 13.

```
01: <?xml version="1.0" encoding="ISO-8859-1"?>
02: <!DOCTYPE web-app
03:     PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
04:     "http://java.sun.com/dtd/web-app_2_3.dtd">
05: <web-app>
06:   <filter>
07:     <filter-name>XSLTranformFilter</filter-name>
```

Listing 28.6 Web.xml

```
08:     <filter-class>
09:      org.javapitfalls.item28.XSLTransformFilter
10:     </filter-class>
11:   </filter>
12:   <filter>
13:     <filter-name>ControllerFilter</filter-name>
14:     <filter-class>
15:       org.javapitfalls.item28.ControllerFilter
16:     </filter-class>
17:   </filter>
18:   <filter-mapping>
19:      <filter-name>XSLTranformFilter</filter-name>
20:      <servlet-name>FilteredFileServlet</servlet-name>
21:   </filter-mapping>
22:   <filter-mapping>
23:     <filter-name>ControllerFilter</filter-name>
24:     <url-pattern>/ControllerFilter</url-pattern>
25:   </filter-mapping>
26:   <servlet>
27:     <servlet-name>FilteredFileServlet</servlet-name>
28:     <servlet-class>
29:       org.javapitfalls.item28.FileServlet
30:     </servlet-class>
31:     <init-param>
32:       <param-name>filename</param-name>
33:       <param-value>/data/states.xml</param-value>
34:     </init-param>
35:   </servlet>
36:   <servlet>
37:         <servlet-name>Page1</servlet-name>
38:         <jsp-file>/test1/jsp/first.jsp</jsp-file>
39:   </servlet>
40:
41:   <!--- second.jsp, third.jsp, fourth.jsp excluded -- >
42:
56:   <servlet-mapping>
57:       <servlet-name>FilteredFileServlet</servlet-name>
58:       <url-pattern>/FileServlet</url-pattern>
59:   </servlet-mapping>
60:
61: </web-app>
62:
```

Listing 28.6 *(continued)*

With respect to the Filter environment, initialization parameters can be associated with a filter using the `init-params` element in the deployment descriptor (web.xml). The names and values of those parameters are available to the filter at runtime using the `getInitParameter` and `getInitParameterNames` methods.

```
01: <?xml version="1.0" ?>
02: <xsl:stylesheet version="1.0"
03:  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
04:
05: <xsl:output method="html"/>
06: <xsl:template match="/">
07: <xsl:choose>
08: <xsl:when test="//state">
09:
10: <table border="0" width="100%">
11:
12:  <tr>
13:  <td bgcolor="#dcdcff" width="25%">Name</td>
14:  <td bgcolor="#dcdcff" width="25%">Flower</td>
15:  <td bgcolor="#dcdcff" width="25%">Bird</td>
16:  <td bgcolor="#dcdcff" width="25%">Capital</td>
17:  </tr>
18:
19: <xsl:for-each select="//state">
20:  <xsl:if test="position() mod 2 != 0">
21:  <tr>
22:  <td bgcolor="#eeeeee" width="25%">
23:    <xsl:value-of select="name"/></td>
24:  <td bgcolor="#eeeeee" width="25%">
25:    <xsl:value-of select="flower"/></td>
26:  <td bgcolor="#eeeeee" width="25%">
27:    <xsl:value-of select="bird"/></td>
28:  <td bgcolor="#eeeeee" width="25%">
29:    <xsl:value-of select="capital"/></td>
30:  </tr>
31:  </xsl:if>
32:  <xsl:if test="position() mod 2 = 0">
33:  <tr>
34:  <td width="25%"><xsl:value-of select="name"/></td>
35:  <td width="25%"><xsl:value-of select="flower"/></td>
36:  <td width="25%"><xsl:value-of select="bird"/></td>
37:  <td width="25%"><xsl:value-of select="capital"/></td>
38:  </tr>
39:  </xsl:if>
40: </xsl:for-each>
41:
42: </table>
43:
44: </xsl:when>
45: </xsl:choose>
46: </xsl:template>
47: </xsl:stylesheet>
48:
```

Listing 28.7 htmlstates.xsl

Design patterns serve as building blocks for propagating best practices in software development. In our sample application, the MVC and Front Controller patterns demonstrate how to avoid programming pitfalls that inevitably reveal themselves in superfluous code that is hard to maintain and deploy. The migration of our configuration mappings to our deployment descriptor helped our application management, and the `getNamedDispatcher` method helped avoid the hard-coding of context paths in our application. The dispatcher component manages the application navigation and display views. Applications and program paths change during normal development lifecycles, and when business logic is spread across your application, they become very difficult to maintain and modify.

Additionally, when control logic is moved into a controller class, processing is facilitated because all requests are coordinated and an application's business logic is centralized. A controller servlet application is also employed by a popular Apache Software Foundation offering called the Struts framework, mentioned in Item 24, which is a much more sophisticated application than the one demonstrated in our example and has been well received by the development community. Servlet controllers and XML transform filters can be a powerful combination when processing user requests, specifically SOAP requests. One shortcoming that needs mention is that a single servlet controller can become unmanageable when too much business logic is transferred into the application, which would warrant additional controllers.

Lastly, filter applications are important architectural considerations that need to be included in all enterprise development design decisions. Essentially, they are building blocks that allow applications to layer logic and transform user requests, which allow developers to promote reuse.

Item 29: Some Direction about JSP Reuse and Content Delivery

Your boss just asked you to storyboard his latest and greatest business plan so that it could be shown to local and remote management teams at next week's status meeting. Your previous development experience involved crafting Java Swing components for the accounting department and building some static HTML pages for your office lottery selections.

You've determined that in order to satisfy your requirement to serve both local and remote users, your application should run in a browser, and that JavaServer Pages would be a perfect vehicle to serve up dynamic content. When you asked your boss about the content needed for the presentation, his reply was that it was still being worked on, and it was possible that it could be published by one of those remote management teams that will participate in next week's meeting. His response has led you to believe that a simple content management application might be needed to accommodate the inclusion of dynamic content from this remote source.

Your task appears to be a lot of work to deliver in one week's time, but with a better understanding of JSP page fragmentation and dynamic content inclusion, you should have a pretty good chance of making yourself and your boss look good. JSP page fragmentation will allow the application to be reusable and fairly maintainable, which

means that additional functionality can be added without having to rewrite the entire application. The most difficult aspect of this task will be obtaining content from a different context than the application being executed, but we'll look into that later.

Certainly, every development effort should start with a simple storyboard to envision what the application might look like. Figure 29.1 is a start.

Not only will your storyboard show spatial relationships and interface behaviors, it will also allow you to organize your files into subdirectories to make them easier to identify and maintain. In our case, we'll need two file structures, one for the default application, and another for the remote location to drop in content.

Now that your look and feel has been determined, we'll add our page fragments in a file called home.jsp, displayed in Listing 29.1. The JSP `include` directive on line 06 reads, translates, and pastes the specified file (header.jsp) into your JSP page one time only. If any modifications are made to your header.jsp code, your Web container will need to retranslate the code to reflect these changes. The JSP `include` action on line 74 includes the footer.jsp page during request time. The `include` action is more robust than the directive because it performs an automatic update dynamically.

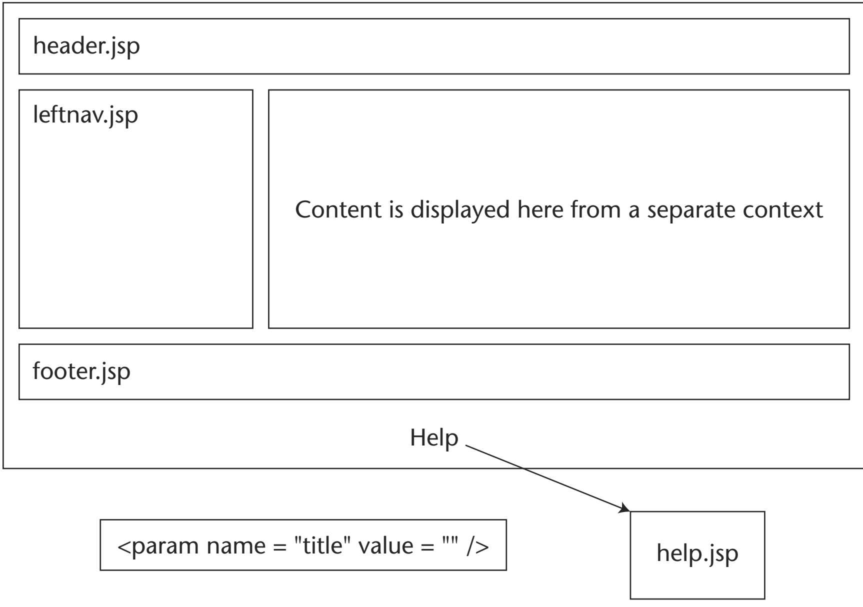


Figure 29.1 Application storyboard.

```
01: <%@ taglib uri="http://java.sun.com/jstl/ea/core" prefix="c" %>
02: <%@page import="java.net.*" %>
03:
04: <title>Fruit Stand</title>
05:
06: <jsp:include page="header.jsp" flush="true" />
07:
08: <table border="0" width="100%">
09: <tr valign="top">
10: <td width="25%" align="center" valign="top">
11: <jsp:include page="leftnav.jsp" flush="true" />
12: </td>
13: <td width=75%" align="center" valign="top">
14:    <%
15:      String sMode = request.getParameter("mode");
16:      String sTopicID = request.getParameter("topic");
17:    %>
18:
19:    <table width="100%" border="0" cellspacing="0" cellpadding="0"
20:                                                        valign="top">
21:     <tr>
22:     <td bgcolor="#ffffff" align="center" class="portletBoxTitle">
23:     <b><font face="Verdana" size="2" color="#FFFFFF">                 ⮐
INFORMATION</font></b>
24:     </td>
25:     </tr>
26:     <tr>
27:     <td bgcolor="#ffffff" align="center" class="portletBoxBody">
28:     <font face="Verdana" size="2"><br>
29:    <%
30:     StringBuffer strbuf = new StringBuffer();
31:     strbuf.append("/default/" + sTopicID);
32:     strbuf.append("/index.html");
33:
34:     String s = strbuf.toString();
35:
36:     // Obtain the custom view context
37:     ServletContext scTemp = getServletConfig().getServletContext();
38:     ServletContext scHome =  getServletConfig().getServletContext();
39:     ServletContext scRemote = scTemp.getContext("/newstuff");
40:
```

Listing 29.1 home.jsp *(continued)*

```
41:     // Create the absolute url for c:import tag to use
42:     String sURL = request.getScheme() + "://" +
43:           request.getServerName() + ":" + request.getServerPort() +
44:           "/newstuff/" + s;
45:     URL urlRsrcFile = null;
46:
47:     try {
48:       // Try to obtain resource from remote context
49:       urlRsrcFile = scRemote.getResource((String)s);
50:     } catch (Exception e) {
51:       scTemp.log("\n\nError: home.jsp - error occured when trying
to obtain a resource from the 'REMOTE' context. Please verify that
the server is configured to allow cross-context access.\n\n", e);
52:     }
53:
54:     if (urlRsrcFile != null) {
55:     %>
56:        <c:import url="<%= sURL %>" />
57:     <%--
58:      This does not work.
59:      <jsp:include page="<%= sURL %>" />
60:     --%>
61:     <%
62:     }
63:     %>
64:
65:     </font>
66:     </td>
67:     </tr>
68:  </table>
69:
70:  </td>
71: </tr>
72: </table>
73:
74: <jsp:include page="footer.jsp" />
```

Listing 29.1 *(continued)*

All appears to be okay until you attempt to add the content using the `include` directive as shown on line 59 from the [newstuff] context. Why? Apparently, JSP `include` operations add resources that are relative to their context, and since the application is being run from the [customviews] context, it looks there for the resource inclusion. To rectify this problem, we'll need to use the Java Standard Template Library (JSTL) implementation of the `import` tag because it allows for the retrieval of absolute URLs, which we'll need to obtain data from the [newstuff] context. Additionally, an

entry has to be made to set the context path for your remote inclusion. With Tomcat, this means a context path inclusion in the server.xml file:

```
<Context path="/newstuff" docBase="newstuff" debug="0"
crossContext="true"/>
```

which will allow the customviews application to access the newstuff pages.

In the home.jsp script, the `import` tag library is used on line 56 to import a resource from a directory that lies outside the context of the current application [customviews]. With the `import` tag, a simple drop-off mechanism like a WebDAV client can be used to drop content into the [newstuff] location outside of your deployment area in [customviews]. This simple content management process will enable you to work uninterrupted, and without concern that someone outside of your office will corrupt your development platform.

Lastly, we'll look at the `forward` tag and its implementation in our application. In Listing 29.2, the forward action `<jsp:forward...>` on line 12 passes the request to the help.jsp page. Normally, if a response has already been partially sent to the browser, the forwarding operation will throw an `IllegalStateException` error. This could occur if a user specifies a page buffer equal to `"none"`, as shown on line 02. A buffer size is generally set to `"none"` on Web pages that are slow and to which the developer wants to feed the content as it comes in, rather than waiting for it to reach its buffer maximum or for the page to load to render data. This error can be captured by inserting a try/catch exception handler like the one shown in lines 10 to 17 below. When the page buffer directive is omitted, a default value of 8 KB is allocated and the request will be forwarded properly.

```
01: <%--
02: <%@ page buffer="none" %>   // bad idea. throws an
IllegalStateException error.
03: --%>
04:
05: <%
06: String mode = request.getParameter("mode");
07: if (mode==null) mode="";
08:
09: if (mode.equals("forward")) {
10:  try {
11: %>
12:  <jsp:forward page="help.jsp" >
13:  <jsp:param name="title" value="Help Screen Topic passed from ⮐
[footer.jsp]" />
14:  </jsp:forward>
15: <%
16:  }
17:  catch (IllegalStateException e) { System.out.println("ERROR: " + ⮐
e.toString()); }
18: }
19: %>
20:
```

Listing 29.2 footer.jsp *(continued)*

```
21: <table width="100%" border="0" cellspacing="0" cellpadding="0">
22:   <tr bgcolor="#FFFFFF">
23:     <td height="4"></td>
24:   </tr>
25:   <tr bgcolor="#3399CC">
26:     <td height="2"></td>
27:   </tr>
28:   <tr class="largetext">
29:     <td height="19">
30:    
31:     </td>
32:   </tr>
33:   <tr bgcolor="#CCCC99">
34:     <td height="2"></td>
35:   </tr>
36:   <tr>
37:     <td height="2"> </td>
38:   </tr>
39:   <tr>
40:     <td>
41:       <div align="center"><font color="#3399CC">
42:         <b><font size="2" face="Arial, Helvetica, sans-serif"><a
href="footer.jsp?mode=forward">Help</a></font></b>
43:       </div>
44:     </td>
45:   </tr>
46: </table>
```

Listing 29.2 *(continued)*

A finished prototype of your design could look like that shown in Figure 29.2.

When developing JSP Web pages, you need to understand when to use `include` *directives*, which are included during the JSP-to-servlet translation phase, and when to use `include` *actions*, which are included during request time. In addition, keep in mind that `forward` operations should always test for `IllegalStateException` errors. Lastly, it is imperative that you implement the `c:import` tag to retrieve remote URLs in your JSP Web pages, rather than attempting to use JSP `include` directives/actions for pages that differ in context from the application being run.

Hopefully, our simple example has shown you proper JSP development strategies that will prevent you from realizing these pitfalls and will allow you to deliver robust JSP applications and dynamic storyboards in a timely manner. It is important to consider that the architecture you have chosen adheres to the Model 1 design, which is considered “page-centric” because application flow is controlled by the JSP page logic. Understand that this could present maintenance problems because of the tight coupling of the flow and the logic, but it seems preferable to those with little experience in Web development, and your tight delivery schedule. A better solution would incorporate a Model 2 architecture that implements a mediating application that decouples hard-coded Web page references.

Figure 29.2 Completed prototype.

Item 30: Form Validation Using Regular Expressions

The latest Merlin release (Java SDK 1.4) introduced regular expressions so that developers can manipulate character text in an easier fashion than previously released string handling methods. This enhancement has strengthened the maturing language and facilitated error checking and text manipulation, which is so important to Web application components.

Search and search-and-replace activities are among the more common uses of regular expressions, but they can also be used to perform Boolean tests on text patterns and data streams. Anyone familiar with Unix should recognize regular expressions and their powerful capabilities because of their prevalence in Unix tools and commands. I like to use regular expressions to parse form text and perform validation and replacement activities in my JavaBean components. In Figure 30.1, a Web form demonstrates user validation on input fields and error text that is rendered to the user display when improper data is submitted by the end user. The code that follows will make obvious how important regular expressions can be for Web developers and how the Java language is the "programming language that keeps on giving."

The form above takes user inputs and validates the data prior to passing the application on to the next application in the workflow. The validation bean in Listing 30.1 reads and remembers the user input using the Memento pattern and checks to see if valid entries have been submitted. Improper entries are tagged and sent back to the user display to indicate what the proper input format should be.

Figure 30.1 A Web form.

Telephone numbers are typically character strings separated by delimiters for readability, so our regular expression should capture digits only ranging from 0 to 9 and be linked by dash characters. The telephone number pattern shown on line 36 ensures that a 10-digit value is input by the user and that an optional dash delimiter can be used for input. Several string manipulations would be needed to perform the same operation that is performed in just one line. Note the double backslash notation in the regular expression `"\\d"`, which differs from the single backslash notation used in Perl, because it differentiates escape character values from regular expressions. The brace notation "{}" indicates the number of decimals values that should be found in the string literal being examined.

```
001: package org.javapitfalls.item30;
002:
003: import java.util.*;
004: import java.text.*;
005: import java.util.Date;
006: import java.util.regex.*;
007:
008: public class validateBean {
009:  private String title;
010:  private String marriedFlag;
```

Listing 30.1 validateBean.java

```
011: private String hobbies[];
012: private String colors[];
013: private String ageGroup;
014: private String telephoneNumber;
015: private String birthDate;
016: private String ssn;
017: private String email;
018: private String comments;
019: private SimpleDateFormat dateFormat;
020:     private String DATE_FORMAT_PATTERN;
021:
022: private Hashtable errors;
023:
024: public boolean validate() {
025:      boolean errorsFound=false;
026:      if (title.equals("")) {
027:         errors.put("title","Please enter a valid title");
028:         errorsFound=true;
029:       }
030:
031:       if (telephoneNumber.equals("")) {
032:       errors.put("telephoneNumber","Please enter a valid telephone ⮐
#");
033:       errorsFound=true;
034:       }
035:       else {
036:       if (!(telephoneNumber.matches("\\+?([0-9]+-)+([0-9]+-     ⮐
)+[0-9]+"))) {
037:          errors.put("telephoneNumber","Please enter a valid     ⮐
telephone format ### - ### - ####");
038:          errorsFound=true;
039:       }
040:  }
041:
```

Listing 30.1 *(continued)*

Social security numbers are typically nine-digit strings separated by dash delimiters, so our regular expression needs to capture the digit value and size constraints in a similar fashion as the telephone number pattern.

```
042:     if (ssn.equals("")) {
043:    errors.put("ssn","Please enter a valid Social Security #");
044:    errorsFound=true;
045:    }
046:    else {
047:    if (!(ssn.matches("(\\d{3}\\-?)+(\\d{2}\\-?)+\\d{4}+"))) {
```

Listing 30.1 *(continued)*

```
048:        errors.put("ssn","Please enter a valid SSN: ### - ## - ####");
049:        errorsFound=true;
050:     }
051: }
052:
```

Listing 30.1 *(continued)*

Date of birth formats come in many different varieties, but our application will validate user input on entries that adhere to the "YYYY-MM-DD" format. The commented text shown in lines 63 to 77 was implemented previously for DOB validation. Again, the ability to do more with less is evident in this code segment.

```
053:     if (birthDate.equals("")) {
054:        errors.put("birthDate","Please enter a valid date");
055:        errorsFound=true;
056:     }
057:     else {
058:
059:     if (!(birthDate.matches("(\\d{4}\\-?)+(\\d{2}\\-?)+\\d{2}+"))) {
060:         errors.put("birthDate","Please enter a valid date format: 
(yyyy-mm-dd)");
061:         errorsFound=true;
062:     }
063:     /*
064:     Date date=null;
065:     try
066:     {
067:        dateFormat.applyPattern(DATE_FORMAT_PATTERN);
068:         // dateFormat.setLenient(false);
069:          date = dateFormat.parse(birthDate);
070:     }
071:     catch(ParseException parseexception) { }
072:     if (date==null) {
073:        errors.put("birthDate","Please enter a valid date format: 
(yyyy-mm-dd)");
074:        birthDate="";
075:        errorsFound=true;
076:      }
077: */
078: }
```

Listing 30.1 *(continued)*

Email account validation is achieved with the pattern described on line 85. The pattern text checks for alphanumeric text, with underlines and periods, that is separated by the "at sign"(@) and terminated with a 3-byte extension.

```
080:     if (email.equals("")) {
081:       errors.put("email","Please enter a valid email address");
082:       errorsFound=true;
083:     }
084:     else {
085:     if (!(email.matches("[-A-Za-z0-9_.]+@[-A-Za-z0-9_.]+\\.[-A-Za-
z]{2,}"))) {
086:       errors.put("email","Please enter a valid email address: ex.
name@company.org");
087:       errorsFound=true;
088:      }
089: }
```

Listing 30.1 *(continued)*

A pet peeve of mine is being forced to fill out an inordinate amount of information about myself in order to gain access to something I'm interested in. This annoyance typically leads to puerile inputs, which become more vulgar the longer I have to keep typing. I know that I'm not the only one doing this because most sites have some kind of dirty word checking to ensure that improper language is not propagated to their database.

The code shown between lines 94 and 96 checks user input to ensure that inappropriate language is captured and replaced with more mild, less offensive text. The regular expression `(darn | shoot ....)` is run against the comment text input by the user, and if a match is found, replaced by the `replaceAll(text)` method of the `Matcher` class.

```
090:    if (comments.equals("")) {
091:       errors.put("comments","Please enter a comment");
092:           errorsFound=true;
093:    } else {
094:           Pattern pattern =
Pattern.compile("(darn|shoot|damn|jerk|stupid|dummy)");
095:            Matcher match = pattern.matcher(comments);
096: if (match.find()) { comments = match.replaceAll("#%&@"); }
097:         }
098: return errorsFound;
099: }
```

Listing 30.1 *(continued)*

The following code illustrates how to use parenthesized subexpressions to replace text and return a new string value. In the example below, my intention is to cut off all of the digits after the decimal point with the exception of the first two. This can be accomplished by matching the first two digits after the decimal point with the "`\\.\\d\\d`" pattern. If the code segment below is run, then testAnswer will print out `111.63`.

```
String test = "111.63642343422";
String testAnswer = test.replaceAll("(\\.\\d\\d?)\\d+", "$1"); // 111.63
System.out.println("testAnswer=" + testAnswer);
```

Another simple example that performs grouping operations is a code snippet that performs pig Latin string manipulation shown below. In pig Latin, all strings that start with a nonvowel character move the first character to the end of the string and add the "ay" characters to the end of the string. Since *Eagles* starts with E, the string operation will be skipped. All of the other strings will be replaced with their Pig latin string manipulation.

```
String testAnswer="";
String[] s = { "PigLatin", "Eagles", "Redskins", "Giants" };
for (int i=0; i < s.length; i++) {
  testAnswer = s[i].replaceAll("^([^aeiouAEIOU])(.+)", "$2$1ay");
  System.out.println("testAnswer=" + testAnswer);
}
```

A big problem for developers with regular expressions on Unix systems is inconsistent behavior with its tools, specifically ed, ex, vi, sed, awk, grep, and egrep. Different conventions often lead to unpredictable behavior that requires lots of patience and work to better understand the pattern syntax of their regular expression libraries. Users often have problems when describing patterns and recognizing the context in which they appear. These same problems exist in Java applications that use third-party regular expression libraries. Hopefully, the implementation of regular expressions in the latest Java SDK will address this and provide pattern consistency across applications and different platforms.

The regular expression libraries that shipped with the Merlin release were a long-awaited addition to an already powerful enterprise programming language. Pattern matching and replacement should unleash a great deal of flexibility in Java development efforts and will facilitate Web development in the future. With regular expressions, metacharacter implementations in patterns will improve text range matching and make text processing a much more pleasant experience.

According to the published Java 2 Standard Edition APIs, the Java Regular Expressions API does not support the following operations that are supported by the Perl 5 scripting language:

- *Conditional constructs.* (`?{X}`).
- *Embedded code constructs.* (`?{code}`).
- *Embedded comment syntax.* To parse comments from a string, the `?#comment` is used.
- *Preprocessing operations.* This includes the implementation of the "`\l \L \u \U`" constructs. To perform lowercase and uppercase operations on an entire string, the `\L` and `\U` are used. To perform lowercase and uppercase on the next character in a string, the `\l` and `\u` constructs need to be used.

Also, the constructs that are supported by the Java Regular Expressions class that are not supported by the Perl 5 scripting language are as follows:

Possessive quantifiers. The inability to backtrack to another operation when a condition has been met. This results in a greedy match operation.

Character class operator precedence. Literal escape, Grouping, Range, Union, Intersection.

The validation code shown above is okay for parsing user input, but there are cases where you might want to parse text within a Web page. The code below is used to strip meta data from all HTML pages that are spidered. On line 166, the `Pattern` class is used to set up the string pattern to parse on the page. There are two tag elements that are part of the pattern, `meta name` and `content`. The `Matcher` class is given the Web page content in the `pageOutput` string, and all of the items are stripped out using the new `split` method of the `String` class.

```
163:    // strip out metadata -----------------------------------------
-------------------------
164:     StringBuffer metadata = new StringBuffer();
165:
166:     Pattern p = Pattern.compile("<meta name=\"XX.data1\"
(CONTENT|content)=\"(.*)");
167:     Matcher m = p.matcher(pageOutput);
168:     int z;
169:     if (m.find()) {
170:        String[] sw1 = m.group(0).split("[\"]");
171:        String[] data1 = sw1[3].split("[,]");
172:        for (z=0; z < data1.length; z++)
173:           metadata.append("<data1>" + data1[z] + "</data1>");
174:     }
175:
176:     p = Pattern.compile("<meta name=\"XX.data2\"
(CONTENT|content)=\"(.*)");
177:     m = p.matcher(pageOutput);
178:     if (m.find()) {
179:        String[] sw2 = m.group(0).split("[\"]");
180:        String[] data2 = sw2[3].split("[,]");
181:        for (z=0; z < data2.length; z++)
182:           metadata.append("<data2>" + data2[z] + "</data2>");
183:     }
184:
185:    p = Pattern.compile("<meta name=\"XX.data3\"
(CONTENT|content)=\"(.*)");
186:     m = p.matcher(pageOutput);
187:     if (m.find()) {
188:        String[] sw3 = m.group(0).split("[\"]");
189:        String[] data3 = sw3[3].split("[,]");
190:        for (z=0; z < data3.length; z++)
191:           metadata.append("<data2>" + data3[z] + "</data2>");
192:     }
193:
194:     System.out.println("metadata= " + metadata.toString());
```

Listing 30.1 *(continued)*

A new regular expression pattern is used to strip out additional links in the HTML pages to spider on subsequent levels. The `<a href></a>` pattern is the target expression to be parsed from the text. The `(a|A)` groupings are used so that both lowercase and uppercase expressions are matched.

```
Pattern pattern = Pattern.compile("<(a|A) href+[^<]*</(a|A)>");
 Matcher match = pattern.matcher(pageOutput);
 %>
 <tr bgcolor="#eeeeee">
<td align="center"> Links for next Level: <%= filename %></td>
 </tr>
 <%
  // display all the references found in the URI
  while (match.find()) {

  // split words
  String[] sw = match.group(0).split("[\"]");
  if ((sw.length > 1) && (sw[1].startsWith("http://")) &&
(sw[1].endsWith("html"))) {
  %>
 <tr><td>
 <%
 out.println("webpage=" + sw[1]);
 vRef.addElement(sw[1]);
 %>
 </td></tr>
 <%
 }
 }
```

The Web page displayed in Figure 30.2 is the result of the spidering action of the code snippet above, which resides in the regexpTest.jsp application. The results shown below the Submit button are the links parsed from the Web page requested in the URI field.

Java Regular Expressions are a powerful new language construct that strengthens string manipulation activities for developers. With Java Regular Expressions, Java developers can realize in their text processing activities what Perl programmers have been lauding about for years.

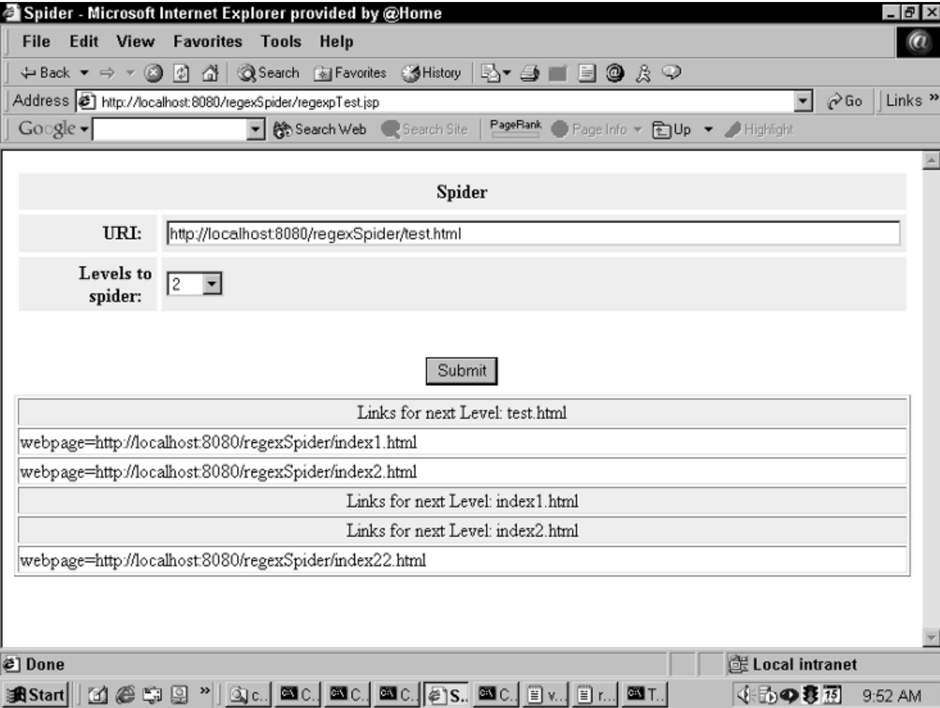


Figure 30.2 Spider results.

Item 31: Instance Variables in Servlets

A common trap that new servlet developers find themselves in revolves around the use of instance variables. Unfortunately, the symptoms of this problem are not easy to diagnose until the last minute. The developer writes the servlet, and it goes through standalone testing just fine. When it is load-tested (or when it goes into production with many concurrent users), however, strange things start to occur on an ad hoc basis: garbled strings of nonsense begin appearing in Web browsers, users of the enterprise Web system begin receiving other users' information, and seemingly "random" errors appear in the application. What went wrong?

A simple example of this situation can be seen in Listing 31.1, where we have an example application that serves as a library of technical resources. Our fictional "Online Technobabble Library" is a document repository where multiple users can check out, check in, and read multiple documents with a lot of technobabble. The servlet takes a parameter, `userid`, which tells the servlet where to get the user's information.

```
001: package org.javapitfalls.item31;
002: import java.io.*;
003: import java.text.*;
004: import java.util.*;
005: import javax.servlet.*;
006: import javax.servlet.http.*;
007: /**
008: * This example demonstrates using instance variables
009: * in a servlet.. The example features a fictional
010: * "TechnoBabble Library", where users can check out
011: * and check in technical documentation.
012: */
013: public class BadTechnobabbleLibraryServlet
014: extends HttpServlet
015: {
016:
017:   PrintWriter     m_out    = null;
018:   String          m_useridparam  = null;
019:
020:
021:   /**
022:    * doGet() method for a HTTP GET
023:    *
024:    * @param  request   the HttpServletRequest object
025:    * @param  response  the HttpServletResponse object
026:   */
027:   public void doGet(HttpServletRequest request,
028:                     HttpServletResponse response)
029:   throws ServletException, IOException
030:   {
031:     String title = "Online Technobabble Library";
032:     response.setContentType("text/html");
033:     m_out = response.getWriter();
034:     m_useridparam = request.getParameter("userid");
035:
036:     m_out.println("<HTML>");
037:     m_out.println("<TITLE>" + title + "</TITLE>");
038:     m_out.println("<BODY BGCOLOR='WHITE'>");
039:     m_out.println("<CENTER><H1>" + title +
040:                   "</H1></CENTER>");
041:     m_out.println("<HR>");
042:
043:     //This will put the user's personal page in..
044:     putInUserData();
045:     m_out.println("<HR>");
046:     m_out.println("</BODY></HTML>");
047:   }
048:
049:
```

Listing 31.1 A bad example!

```
050:   /**
051:    * doPost() method for a HTTP PUT
052:    *
053:    * @param  request   the HttpServletRequest Object
054:    * @param  response  the HttpServletResponse Object
055:    */
056:   public void doPost(HttpServletRequest request,
057:                      HttpServletResponse response)
058:   throws ServletException, IOException
059:   {
060:     doGet(request, response);
061:   }
062:
063:
064:   /**
065:    * This method reads the user's data from the filesystem
066:    * and writes the data to the browser screen.
067:    */
068:   private void putInUserData() throws IOException
069:   {
070:
071:     BufferedReader br = null;
072:     String fn = m_useridparam + ".html";
073:     String htmlfile =
074:     getServletContext().getRealPath(fn);
075:
076:     System.out.println("debug: Trying to open "
077:                        + htmlfile);
078:
079:     File htmlSnippetFile = new File(htmlfile);
080:     try
081:     {
082:       String  line;
083:
084:       //Check to see if it exists first
085:       if (!htmlSnippetFile.exists())
086:       {
087:         m_out.println("File " + fn + "not found!");
088:         return;
089:       }
090:
091:       br = new BufferedReader(new FileReader(htmlfile));
092:
093:       /*
094:        * Now, let's read it..
095:        * Since finding the bad behavior in this pitfall
096:        * revolves around timing, we will only read 2
097:        * characters at a time so that the bad behavior
098:        * can be easily seen.
```

Listing 31.1 *(continued)*

```
099:         */
100:
101:        char[] buffer = new char[2];
102:        int count = 0;
103:        do
104:        {
105:          m_out.write(buffer, 0, count);
106:          m_out.flush();
107:          count = br.read(buffer, 0, buffer.length);
108:        }
109:        while (count != -1);
110:      }
111:      catch (Exception e)
112:      {
113:        m_out.println(
114:                     "Error in reading file!!"
115:                     );
116:        e.printStackTrace(System.err);
117:      }
118:      finally
119:      {
120:        if (br != null)
121:          br.close();
122:      }
123:
124:    }
125: }
126:
127:
```

Listing 31.1 *(continued)*

Looking at the code in lines 17 and 18 of Listing 31.1, we have two instance variables. The `PrintWriter m_out` and the `String m_useridparam` are assigned in the `doGet()` method on lines 33 and 34. After the `doGet()` method initializes these variables, the `putInUserData()` method is called to read the user-specific HTML files and print these out to the browser screen. Tested alone, a screen capture of the browser window looks fine, as shown in Figure 31.1.

However, during load testing, multiple users log in and many see the screen shown in Figure 31.2. The result seems like a combination of many screens and looks like nonsense. What happened?

Because instance variables are set in the `doGet()` method in Listing 31.1 and are used later in the servlet in the `putInUserData()` method, the servlet is not thread-safe. Because many users access the servlet at the same time, and because the instance variables are written to and referenced by multiple areas of the servlet, the values of the `m_out` variable and the `m_useridparams` variables have the potential to be clobbered!

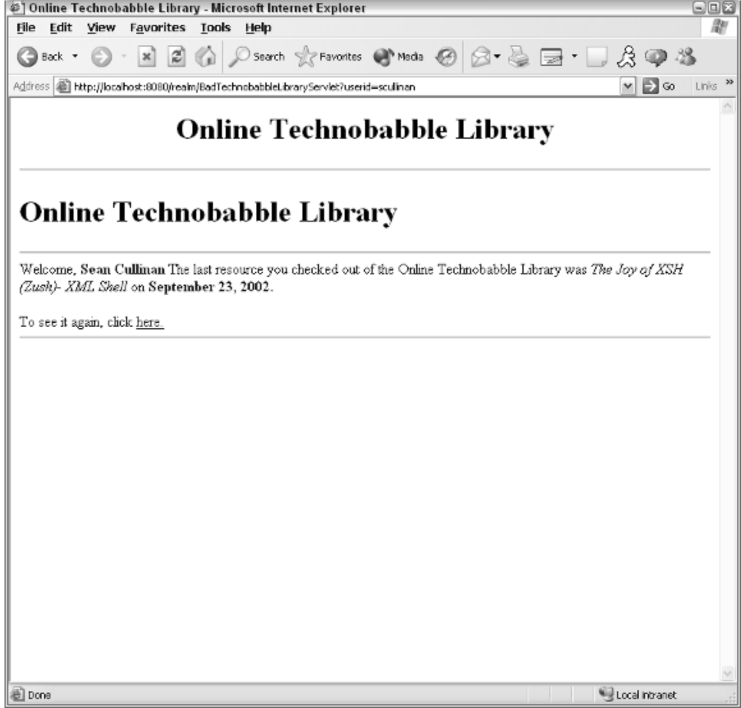


Figure 31.1 Our example with one concurrent user.

For example, when one user runs his servlet, the servlet could be setting the `m_out` variable for his session while another user's session is writing with the `m_out` variable.

Figure 31.3 shows us a time line of how this strange behavior occurred in our example from Listing 31.1. In the time line, we have two fictitious users of the system, Alice and Bob. At time t0, the servlet engine first instantiates the servlet, where the instance variables are declared in lines 17 and 18 of Listing 31.1. At time t1, the servlet engine calls the servlet's `init()` method. At time t2, Alice loads the servlet, which calls the servlet's `doGet()` method. At time t3, the instance variables `m_out` and `m_useridparam` are set just for Alice. At the same time, Bob loads the servlet, which calls the servlet's `doGet()` method. At time t4, Alice's servlet gets to the point where the `putInUserData()` method is called, which loads her information and begins printing to the `PrintWriter` variable, `m_out`. At the same time, Bob's servlet is in the execution of `doGet()`, where the instance variables `m_out` and `m_useridparam` are set.

Time t5 in Figure 31.3 is where everything seems to go nuts. Since Bob reset the servlet's instance variable `m_out`, Alice continues to print her information, but it goes to Bob's browser! When Bob also begins printing to `m_out`, he sees a combination of his information and Alice's information. The result is something like the screen shot that we showed in Figure 31.2.

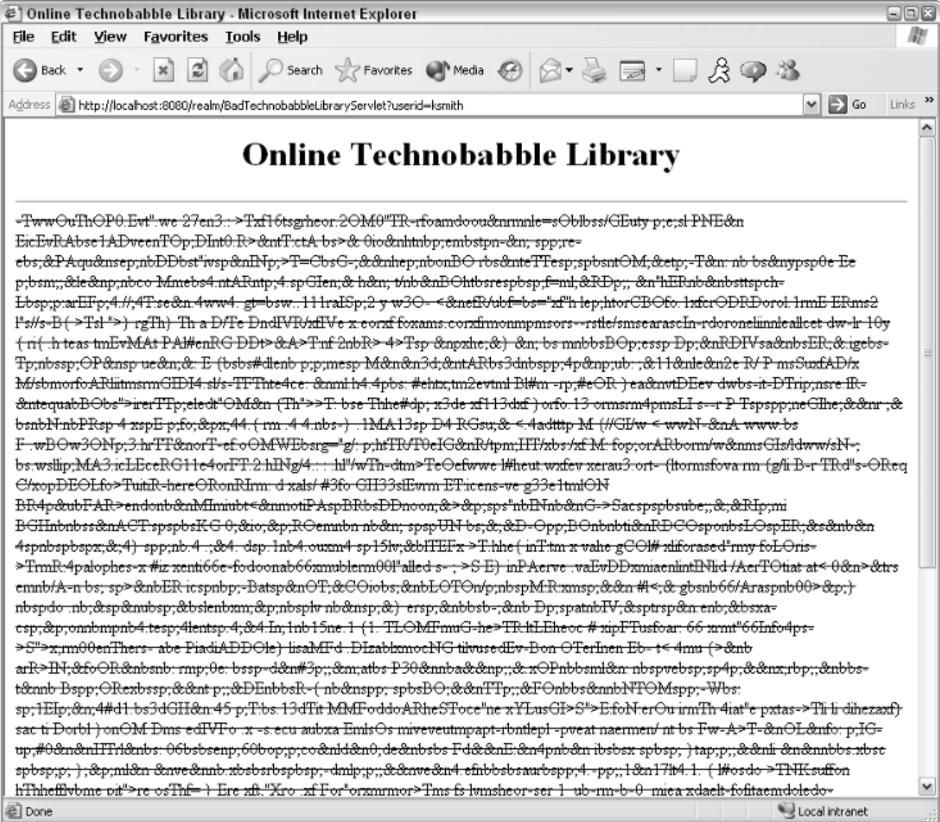


Figure 31.2 Chaos: Our example with concurrent users.

In fact, we chose the example so that this strange behavior could be shown easily. Each servlet client (or user Web browser) will need its own `PrintWriter`, and because each servlet is a thread in the server's virtual machine, `m_out` can be trampled on whenever a new user loads the servlet, producing scary output. In practice, these types of errors can be difficult to detect until load testing, because errors occur with timing when multiple clients hit the server at once. In fact, we couldn't see any tangible ramifications of assigning the variable `m_useridparam`, because the timing has to be right to actually see the effects. We have seen one situation where a customer's requirements were to provide sensitive and confidential data to its users. The software developers used instance variables in their servlets for printing information gathered from each user's database connection. When the developers were testing it alone, the system seemed to work fine. When the system went in for load testing with several different users, the testers saw the other users' confidential data.

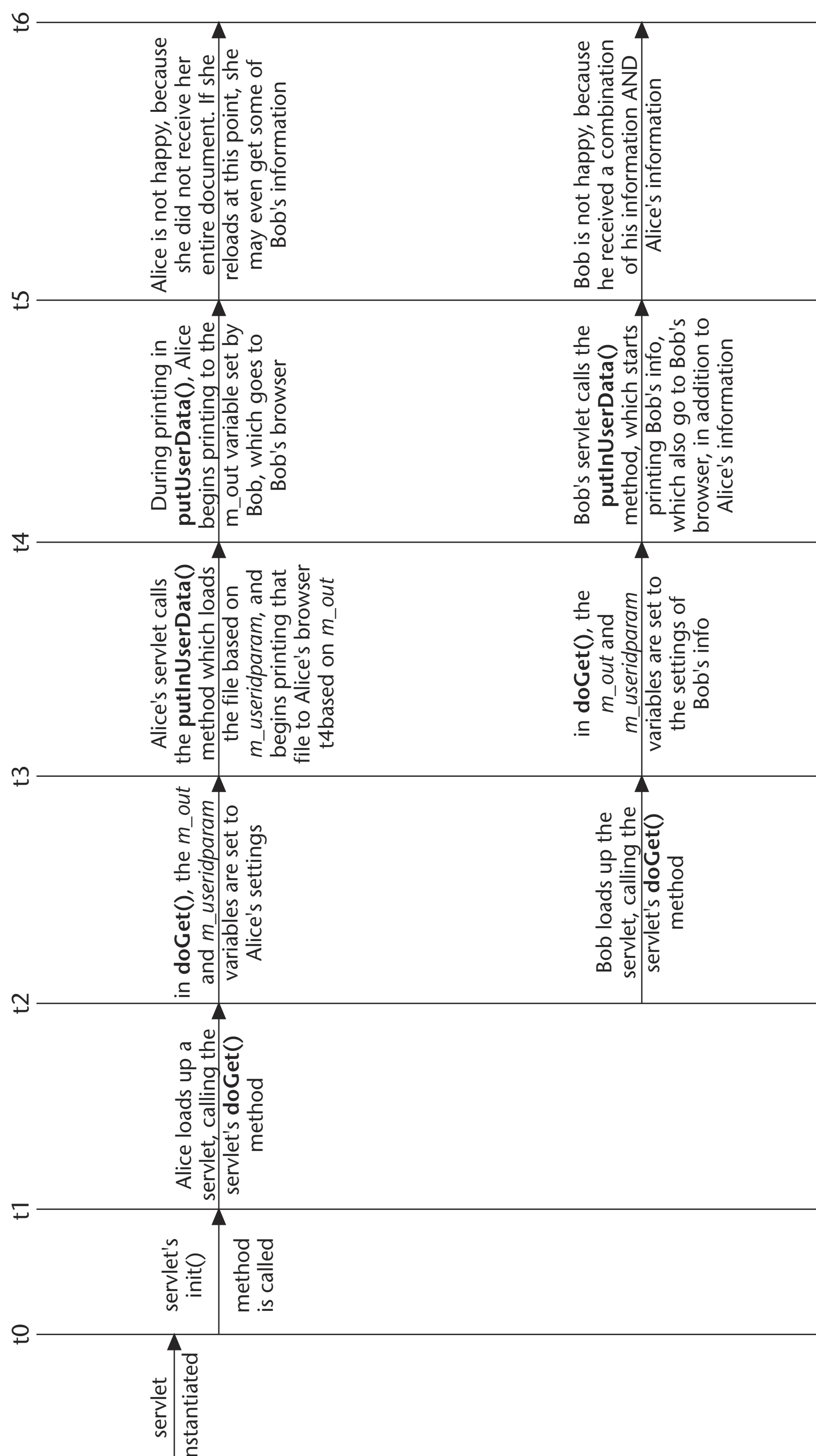


Figure 31.3 Time line of data corruption.

What if you use instance variables but only set them at instantiation? In this case, you may not run into any concurrency issues. However, this could lead to other pitfalls if you do not keep the lifecycle of the servlet in mind. Here is a bad example that we saw recently:

```
//Here Is an Instance variable that we will be using later

ServletContext m_sc = getServletContext();
```

The programmer who wrote that snippet of code assumed that since the variable was set at instantiation and never set again, no concurrency issues would arise. That is true. Unfortunately, because the `init()` method sets the servlet's `ServletContext` object after instantiation, the servlet ran into a big problem: The value of the instance variable `m_sc` was null, resulting in a `NullPointerException` later on in the servlet. Because the instance variable was set at instantiation of the servlet, and not after the `init()` method was called, the servlet had big problems.

So what is the best solution to this pitfall? Be very hesitant in using instance variables in servlets. Certainly, you could continue to use instance variables and synchronize them whenever you need to access or set the instance variable, but that could create code that is very complex-looking, inflexible, and ugly. Instead, try to pass the variables and objects that you will need to your other methods. If it gets to the point where you believe that you have too many variables to pass around, create a class that serves as a container for these variables. Instantiate the class with the variables you need, and pass the object around.

Listing 31.2 shows a better approach to our earlier "Online Technobabble Library" example. Instead of having instance variables, we create the variables that we need in the `doGet()` method in lines 32 and 33, and we pass them to the `putInUserData()` method. The result is code that is thread-safe.

```
001: package org.javapitfalls.item31;
002: import java.io.*;
003: import java.text.*;
004: import java.util.*;
005: import javax.servlet.*;
006: import javax.servlet.http.*;
007: /**
008: * This example demonstrates using instance variables
009: * in a servlet.. The example features a fictional
010: * "TechnoBabble Library", where users can check out
011: * and check in technical documentation.
012: */
013: public class GoodTechnobabbleLibraryServlet extends
014: HttpServlet
015: {
016:
017:   /**
018:    * doGet() method for a HTTP GET
```

Listing 31.2 A better application solution

```
019:     *
020:     * @param  request   the HttpServletRequest object
021:     * @param  response  the HttpServletResponse object
022:     */
023:    public void doGet(HttpServletRequest request,
024:                      HttpServletResponse response)
025:    throws ServletException, IOException
026:    {
027:      PrintWriter out;
028:      String userid;
029:      String title = "Online Technobabble Library";
030:
031:      response.setContentType("text/html");
032:      out = response.getWriter();
033:      userid = request.getParameter("userid");
034:
035:      out.println("<HTML>");
036:      out.println("<TITLE>" + title + "</TITLE>");
037:      out.println("<BODY BGCOLOR='WHITE'>");
038:      out.println("<CENTER><H1>" + title +
039:                  "</H1></CENTER>");
040:      out.println("<HR>");
041:
042:      //This will put the user's personal page in..
043:      putInUserData(out, userid);
044:      out.println("<HR>");
045:      out.println("</BODY></HTML>");
046:    }
047:
048:
049:    /**
050:     * doPost() method for a HTTP PUT
051:     *
052:     * @param  request   the HttpServletRequest Object
053:     * @param  response  the HttpServletResponse Object
054:     */
055:    public void doPost(HttpServletRequest request,
056:                       HttpServletResponse response)
057:    throws ServletException, IOException
058:    {
059:      doGet(request, response);
060:    }
061:
062:
063:    /**
064:     * This method reads the user's data from the filesystem
065:     * and writes the data to the browser screen.
066:     *
067:     * @param out  the printwriter we are using
```

Listing 31.2 *(continued)*

```
068:    * @param userid  the userid of the accessing user
069:    */
070:   private void putInUserData(PrintWriter out,
071:                              String userid)
072:   throws IOException
073:   {
074:     BufferedReader br = null;
075:     String fn = userid + ".html";
076:     String htmlfile =
077:       getServletContext().getRealPath(fn);
078:
079:     System.out.println("debug: Trying to open "
080:                        + htmlfile);
081:
082:     File htmlSnippetFile = new File(htmlfile);
083:     try
084:     {
085:       String  line;
086:
087:       //Check to see if it exists first
088:       if (!htmlSnippetFile.exists())
089:       {
090:         out.println("File " + fn + "not found!");
091:         return;
092:       }
093:
094:       br = new BufferedReader(new FileReader(htmlfile));
095:
096:       /*
097:        * Now, let's read it..
098:        * Since finding the bad behavior in this pitfall
099:        * revolves around timing, we will only read 2
100:        * characters at a time so that the bad behavior
101:        * can be more easily seen.
102:        */
103:
104:       char[] buffer = new char[2];
105:       int count = 0;
106:       do
107:       {
108:         out.write(buffer, 0, count);
109:         out.flush();
110:         count = br.read(buffer, 0, buffer.length);
111:       }
112:       while (count != -1);
113:     }
114:     catch (Exception e)
115:     {
116:       out.println(
```

Listing 31.2 *(continued)*

```
117:                  "Error in reading file!!"
118:                  );
119:        e.printStackTrace(System.err);
120:      }
121:      finally
122:      {
123:        if (br != null)
124:          br.close();
125:      }
126:
127:    }
128: }
129:
130:
```

Listing 31.2 *(continued)*

Now that we have eliminated instance variables in Listing 31.2, we have fixed our problems with thread safety! Be hesitant in using instance variables in servlets. If there is a better way, do it.

Item 32: Design Flaws with Creating Database Connections within Servlets

Connecting to a database is a convenient way for generating Web content. That being said, there can be many performance issues with creating database connections in servlets. It is imperative that a Web-enabled application be able to scale to the demand of its users. For that reason, preparation for a large amount of users is a necessity.

We will present a sample scenario where we are developing a Java Servlet-based system for a local shop, "Lavender Fields Farm." The decision makers on the project have decided that we will need to use a database for the online purchases and transactions. At the same time, we will need to use that database to keep track of inventory. This example will focus on the development of that "inventory servlet."

Listing 32.1 shows a servlet that queries the database to show inventory. In lines 52 to 75, in the servlet's `doPost()` method, we establish a connection to the database, and we create an HTML table showing the name of each item available, the description, and the amount that the shop has in stock. Figure 32.1 shows a screen capture of the result. In testing, everything with this example works wonderfully. When many users begin to use the system, the database becomes a bottleneck. Finally, after intense usage of the Web application, the database begins refusing new connections. What happened?

```
01: package org.javapitfalls.item32;
02: import java.io.*;
03: import java.sql.*;
04: import java.text.*;
05: import java.util.*;
06: import javax.servlet.*;
07: import javax.servlet.http.*;
08: public class BadQueryServlet extends HttpServlet
09: {
10:   /**
11:    *  simply forwards all to doPost()
12:    */
13:   public void doGet(HttpServletRequest request,
14:                     HttpServletResponse response)
15:   throws IOException, ServletException
16:   {
17:       doPost(request,response);
18:   }
19: 
20:   /**
21:    * The main form!
22:    */
23:   public void doPost(HttpServletRequest request,
24:                      HttpServletResponse response)
25:   throws IOException, ServletException
26:   {
27:       PrintWriter out = response.getWriter();
28:       out.println("<TITLE>Internal Inventory Check</TITLE>");
29:       out.println("<BODY BGCOLOR='white'>");
30:       out.println("<H1>Lavender Fields Farm Internal Inventory</H1>");
31: 
32:       //show the date.
33:       SimpleDateFormat sdf =
34:           new SimpleDateFormat ("EEE, MMM d, yyyy h:mm a");
35:       java.util.Date newdate = new
36:       java.util.Date(Calendar.getInstance().getTime().getTime());
37:       String datestring = sdf.format(newdate);
38: 
39:       out.println("<H3>Inventory as of: " + datestring + "</H3>");
40: 
41:       out.println("<TABLE BORDER=1>");
42:       out.println("<TR><TD BGCOLOR='yellow'><B><CENTER>Name</CENTER>"+
43:                  "</B></TD><TD BGCOLOR='yellow'><B><CENTER>" +
44:                  "Description</CENTER></B></TD><TD BGCOLOR='yellow'>" +
45:                  "<B><CENTER>Inventory Amount</CENTER></B></TD></TR>");
46: 
47:       //Load the inventory from the database.
48: 
49:       try
50:       {
```

Listing 32.1 Specifying connection in a servlet

```
51:
52:             Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
53:             String connect = "jdbc:odbc:Lavender";
54:
55:             Connection con = DriverManager.getConnection(connect);
56:
57:             Statement stmt = con.createStatement();
58:            ResultSet rs = stmt.executeQuery("select * from Inventory");
59:
60:             while (rs.next())
61:             {
62:                 String amtString = "";
63:                 int amt = rs.getInt("Amount");
64:                 if (amt < 50)
65:                     amtString ="<TD><CENTER><FONT COLOR='RED'>" + amt +
66:                                 "</FONT></CENTER></TD>";
67:                 else
68:                     amtString ="<TD><CENTER>" + amt + "</CENTER></TD>";
69:
70:                 out.println("<TR><TD><CENTER>" + rs.getString("Name") +
71:                             "</CENTER></TD><TD><CENTER>" +
72:                             rs.getString("Description") + "</CENTER>" +
73:                             "</TD>" + amtString + "</TR>");
74:             }
75:             rs.close();
76:             out.println("</TABLE><HR>Items in <FONT COLOR='red'>RED" +
77:                     "</FONT> denote low inventory levels. Click " +
78:                    "Here to Contact <A HREF='mailto:mgmt@localhost'>" +
79:                    "MANAGEMENT</A> to order more supplies.");
80:         }
81:         catch (Exception e)
82:         {
83:           out.println("There were errors connecting to the database." +
84:                       " See your systems administrator for details.");
85:             e.printStackTrace();
86:         }
87:
88:
89:     }
90:
91:
92:}
```

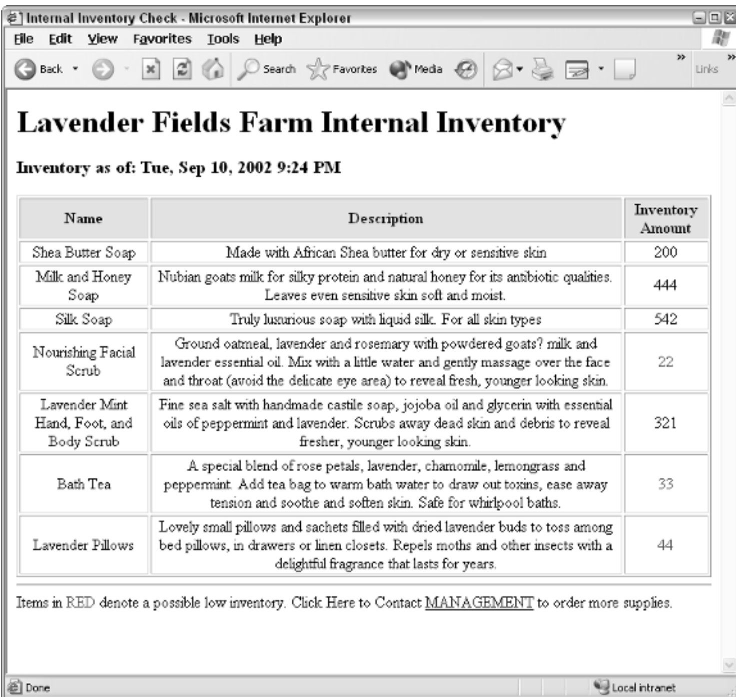
Listing 32.1 *(continued)*

The problem is twofold: opening a new connection to the database is a computationally expensive operation, and there is a finite number of open connections that a database can have. In line 55 in Listing 32.1, we open a new connection on line 55 every time a user loads the Web page. On line 75, we close the result set, resulting in the eventual termination of the connection. When there are hundreds (or thousands) of users connecting to that online store, that strategy will simply not suffice.

What is a better solution? This is where the specifics of your application impact your decision. If this servlet were the only one connecting to the database, you could design it so that the servlet shares connections with itself. That is, a set of connections could be shared, reused, recycled, and managed by this servlet. If, however, the entire Web server shared connections between servlets, you should design it so that all the servlets share the management of connections with each other.

This is where *connection pooling* comes into play. Connection pooling involves allocating database connections in advance, along with the reuse and management of the connections. You could write your own connection pool package that your servlet could use, but many are available on the Internet. For our next example, we used an open-source connection broker called `DDConnectionBroker`, from http://opensource.devdaily.com/. This package offers the basics that any connection pooling class should—including the pre-allocation, reuse, and management of database connections.

In Listing 32.2, we use the connection broker within the servlet. In our `init()` method, we instantiate the connection broker, specifying the details of our database connection and setting our broker to be the instance variable in line 13. Since the `init()` method is only called once (right after instantiation), our connection pool is initialized once. We specify that the connection pool will have a maximum number (10) of database connections. This number is dependent on the configuration of your database. After setting up the connection broker in the `init()` method, the only further change to the servlet is that instead of creating the connection in the `doPost()` method, we call the method `m_broker.getConnection()` in line 95 and call the method `m_broker.freeConnection()` in line 119. The result is that every thread going through this servlet (every user loading the Web page) will use a connection pool in getting to the database.



Lavender Fields Farm Internal Inventory

Inventory as of: Tue, Sep 10, 2002 9:24 PM

Name	Description	Inventory Amount
Shea Butter Soap	Made with African Shea butter for dry or sensitive skin	200
Milk and Honey Soap	Nubian goats milk for silky protein and natural honey for its antibiotic qualities. Leaves even sensitive skin soft and moist.	444
Silk Soap	Truly luxurious soap with liquid silk. For all skin types	542
Nourishing Facial Scrub	Ground oatmeal, lavender and rosemary with powdered goats? milk and lavender essential oil. Mix with a little water and gently massage over the face and throat (avoid the delicate eye area) to reveal fresh, younger looking skin.	22
Lavender Mint Hand, Foot, and Body Scrub	Fine sea salt with handmade castile soap, jojoba oil and glycerin with essential oils of peppermint and lavender. Scrubs away dead skin and debris to reveal fresher, younger looking skin.	321
Bath Tea	A special blend of rose petals, lavender, chamomile, lemongrass and peppermint. Add tea bag to warm bath water to draw out toxins, ease away tension and soothe and soften skin. Safe for whirlpool baths.	33
Lavender Pillows	Lovely small pillows and sachets filled with dried lavender buds to toss among bed pillows, in drawers or linen closets. Repels moths and other insects with a delightful fragrance that lasts for years.	44

Items in RED denote a possible low inventory. Click Here to Contact MANAGEMENT to order more supplies.

Figure 32.1 Screen capture of inventory page.

```
000: package org.javapitfalls.item32;
001: import com.devdaily.opensource.database.DDConnectionBroker;
002: import java.io.*;
003: import java.sql.*;
004: import java.text.*;
005: import java.util.*;
006: import javax.servlet.*;
007: import javax.servlet.http.*;
008:
009:  public class BetterQueryServlet extends HttpServlet
010: {
011:    //only set in the init() method, so concurrency
012:    //issues should be fine.
013:    private DDConnectionBroker m_broker = null;
014:
015:    public void init()
016:    {
017:        String driver         = "sun.jdbc.odbc.JdbcOdbcDriver";
018:        String url            = "jdbc:odbc:Lavender";
019:        String uname = "";
020:        String passwd = "";
021:
022:        int minConnections = 1;
023:        int maxConnections = 10;
024:        long timeout        = 100;
025:        long leaseTime      = 60000;
026:        String logFile        = "c:/tmp/ConnectionPool.log";
027:
028:        try
029:        {
030:            m_broker = new DDConnectionBroker(driver,
031:                                                url, uname, passwd,
032:                                                minConnections,
033:                                                maxConnections,
034:                                                timeout,
035:                                                leaseTime,
036:                                                logFile);
037:        }
038:        catch (SQLException se)
039:        {
040:            System.err.println( se.getMessage() );
041:        }
042:
043:
044:    }
045:    /**
046:     *  simply forwards all to doPost()
047:     */
048:    public void doGet(HttpServletRequest request,
```

Listing 32.2 Sharing a connection within a servlet *(continued)*

```
049:                        HttpServletResponse response)
050:     throws IOException, ServletException
051:     {
052:         doPost(request,response);
053:     }
054:
055:     /**
056:      * The main form!
057:      */
058:     public void doPost(HttpServletRequest request,
059:                         HttpServletResponse response)
060:     throws IOException, ServletException
061:     {
062:         PrintWriter out = response.getWriter();
063:
064:         if (m_broker == null)
065:         {
066:             out.println("<B>There are currently database problems. " +
067:                       "Please see your administrator for details.</B>");
068:             return;
069:         }
070:        out.println("<TITLE>Internal Inventory Check</TITLE>");
071:        out.println("<BODY BGCOLOR='white'>");
072:        out.println("<H1>Lavender Fields Farm Internal Inventory</H1>");
073:
074:         //show the date.
075:         SimpleDateFormat sdf =
076:            new SimpleDateFormat ("EEE, MMM d, yyyy h:mm a");
077:         java.util.Date newdate =
078:       new java.util.Date(Calendar.getInstance().getTime().getTime());
079:         String datestring = sdf.format(newdate);
080:
081:         out.println("<H3>Inventory as of: " + datestring + "</H3>");
082:
083:         out.println("<TABLE BORDER=1>");
084:         out.println("<TR><TD BGCOLOR='yellow'>" +
085:       "<B><CENTER>Name</CENTER></B></TD>" +
086:       "<TD BGCOLOR='yellow'><B><CENTER>Description</CENTER></B></TD>" +
087:       "<TD BGCOLOR='yellow'><B><CENTER>Inventory Amount</CENTER></B>" +
088:          "</TD></TR>");
089:
090:         //Load the inventory from the database.
091:
092:         try
093:         {
094:
095:            Connection con = m_broker.getConnection();
096:
097:            Statement stmt = con.createStatement();
```

Listing 32.2 *(continued)*

```
098:          ResultSet rs = stmt.executeQuery("select * from Inventory");
099:
100:           while (rs.next())
101:           {
102:               String amtString = "";
103:               int amt = rs.getInt("Amount");
104:              if (amt < 50)
105:                  amtString ="<TD><CENTER><FONT COLOR='RED'>" + amt + 
                          "</FONT></CENTER></TD>";
106:               else
107:                   amtString ="<TD><CENTER>" + amt + "</CENTER></TD>";
108:              out.println("<TR><TD><CENTER>" + rs.getString("Name") +
109:                          "</CENTER></TD><TD><CENTER>"      + 
                            rs.getString("Description") + "</CENTER>" +
110:                          "</TD>" + amtString + "</TR>");
111:           }
112:          rs.close();
113:          out.println("</TABLE><HR>Items in 
                          <FONTCOLOR='red'>RED</FONT>"
114:                       +" denote a possible low inventory. Click Here to
115:                         " Contact <A HREF='mailto:mgmt@localhost'>" +
116:                         "MANAGEMENT</A> to order more supplies.");
117:
118:          //Free the connection!
119:          m_broker.freeConnection( con );
120:
121:       }
122:       catch (Exception e)
123:       {
124:           out.println("There were errors connecting to the database.
125:                        "See your systems administrator for details.");
126:           e.printStackTrace();
127:       }
128:
129:   }
130:
131:
132:}
```

Listing 32.2 *(continued)*

The effect of Listing 32.2 is that every user that runs the servlet BetterQueryServlet will share the connection broker object, so that connections will be shared and reused. Even better, our `DDConnectionBroker` object is instantiated in the `init()` method, where it pre-allocates connections before they are requested. This will make performance of this servlet better, and it is a good method of database connection management when that servlet is the only application talking to your database.

In our scenario, however, we said that customers at our online store will be connecting to the database as well. This means that it will be wise to share the database connections across all servlets. How could we do this? One of the best ways to do this is to use the Gang of Four's Singleton design pattern.[1] A very convenient design pattern, a Singleton is used when there should be only one instance of a class in a virtual machine. In our scenario, it would be great to get an instance of a connection broker from any servlet and be able to use one of the connections. Listing 32.3 shows a simple Singleton that also acts as an adapter to a few methods of the connection broker. This class, `LavenderDBSingleton`, will be our single point of entry to the database for our "Lavender Fields Farm" example. In our private constructor in lines 17 to 50, we instantiate our connection pool. In our `getInstance()` method in lines 55 to 63, you can see that this class will only be instantiated once. Finally, `freeConnection()` and `getConnection()` are simply wrappers to the methods in the `DDConnectionBroker` class.

```
01: package org.javapitfalls.item32;
02:
03: import com.devdaily.opensource.database.DDConnectionBroker;
04: import java.io.*;
05: import java.sql.*;
06:
07:   /**
08:   /* This is our class that will be shared across all of the
09:    * servlets for the 'Lavender' database. It is a singleton,
10:    * and also works as an adapter to the connection broker
11:    * class that we are using.
12:   */
13:   public class LavenderDBSingleton
14:   {
15:
16:     private DDConnectionBroker m_broker;
17:     private static LavenderDBSingleton m_singleton = null;
18:
19:     private LavenderDBSingleton()
20:     {
21:        /*
22:         * We will put all of our database-specific information
23:         * here. Please note that we could have read this
24:         * information from a properties file.
25:         */
26:
27:          String driver         = "sun.jdbc.odbc.JdbcOdbcDriver";
28:          String url            = "jdbc:odbc:Lavender";
29:          String uname = "";
30:          String passwd = "";
```

Listing 32.3 Singleton class for sharing a connection pool

[1]Gamma, Helm, Johnson, Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* 1995. Reading, Mass.: Addison-Wesley.

```
31:
32:         int minConnections  = 1;
33:         int maxConnections  = 10;
34:         long timeout        = 100;
35:         long leaseTime      = 60000;
36:         String logFile        = "c:/tmp/ConnectionPool.log";
37:
38:         try
39:         {
40:             m_broker = new DDConnectionBroker(driver,
41:                                                 url, uname, passwd,
42:                                                 minConnections,
43:                                                 maxConnections,
44:                                                 timeout,
45:                                                 leaseTime,
46:                                                 logFile);
47:         }
48:         catch (SQLException se)
49:         {
50:             System.err.println( se.getMessage() );
51:         }
52:     }
53:     /**
54:      *  getInstance() returns the class, instantiating it
55:      *  if there is not yet an instance in the VM.
56:      */
57:     public synchronized static LavenderDBSingleton getInstance()
58:     {
59:         if (m_singleton == null)
60:         {
61:             m_singleton = new LavenderDBSingleton();
62:         }
63:
64:         return (m_singleton);
65:     }
66:
67:     /*
68:      * calls getConnection() on the broker class
69:      */
70:     public synchronized Connection getConnection() throws Exception
71:     {
72:         if (m_broker == null)
73:         {
74:             throw new Exception("Can't get Connection broker!");
75:         }
76:         return (m_broker.getConnection());
77:     }
78:
79:     /*
```

Listing 32.3 *(continued)*

```
80:      * frees the connection from the broker class
81:      */
82:     public synchronized void freeConnection(Connection con)
83:      throws Exception
84:     {
85:         if (m_broker == null )
86:         {
87:             throw new Exception("Can't get Connection broker!");
88:         }
89:         m_broker.freeConnection(con);
90:     }
91:   }
92:
93:
```

Listing 32.3 *(continued)*

Listing 32.4 shows the final version of our servlet with our `Singleton` class in action. In the `init()` method, we get the instance of our `Singleton` class on line 23. On line 72, we call the `getConnection()` method of our Singleton, and finally, on line 107, we free the connection. If the other servlets in our example use the Singleton that does connection pooling and database connection management, we will maximize the efficiency of our servlet, reducing the overhead of creating connections for every client.

```
001: package org.javapitfalls.item32;
002:
003: import java.io.*;
004: import java.sql.*;
005: import java.text.*;
006: import java.util.*;
007: import javax.servlet.*;
008: import javax.servlet.http.*;
009:
010: public class BestQueryServlet extends HttpServlet
011: {
012:   //only set in the init() method, so concurrency
013:   //issues should be fine.
014:   private LavenderDBSingleton m_dbsingleton = null;
015:
016:   public void init()
017:   {
018:     /*
019:      * This will instantiate it within the Servlet's
020:      * virtual machine if it hasn't already. If it
021:      * has, we have the instance of it.
022:      */
023:     m_dbsingleton = LavenderDBSingleton.getInstance();
```

Listing 32.4 Servlet Sharing Connection Pool Across Server

```
024:   }
025:   /**
026:    *  simply forwards all to doPost()
027:    */
028:   public void doGet(HttpServletRequest request,
029:                     HttpServletResponse response)
030:   throws IOException, ServletException
031:   {
032:     doPost(request,response);
033:   }
034:
035:   /**
036:    * The main form!
037:    */
038:   public void doPost(HttpServletRequest request,
039:                      HttpServletResponse response)
040:   throws IOException, ServletException
041:   {
042:     PrintWriter out = response.getWriter();
043:
044:     out.println("<TITLE>Internal Inventory Check</TITLE>");
045:     out.println("<BODY BGCOLOR='white'>");
046:     out.println("<H1>Lavender Fields Farm Internal Inventory</H1>");
047:
048:     //show the date.
049:     SimpleDateFormat sdf =
050:     new SimpleDateFormat ("EEE, MMM d, yyyy h:mm a");
051:     java.util.Date newdate =
052:     new java.util.Date(
053:       Calendar.getInstance().getTime().getTime()
054:       );
055:     String datestring = sdf.format(newdate);
056:
057:     out.println("<H3>Inventory as of: " + datestring + "</H3>");
058:
059:     out.println("<TABLE BORDER=1>");
060:     out.println("<TR><TD BGCOLOR='yellow'>" +
061:                 "<B><CENTER>Name</CENTER></B></TD>" +
062:                 "<TD BGCOLOR='yellow'><B>" +
063:                 "<CENTER>Description</CENTER></B></TD>" +
064:                 "<TD BGCOLOR='yellow'><B>" +
065:                 "<CENTER>Inventory Amount</CENTER></B></TD></TR>");
066:
067:     //Load the inventory from the database.
068:
069:     try
070:     {
071:
072:       Connection con = m_dbsingleton.getConnection();
073:       if (con == null)
```

Listing 32.4 *(continued)*

```
074:       {
075:         out.println("<B>There are currently database problems. " +
076:                      "Please see your administrator for details.</B>");
077:         return;
078:       }
079:
080:
081:       Statement stmt = con.createStatement();
082:       ResultSet rs = stmt.executeQuery("select * from Inventory");
083:
084:       while (rs.next())
085:       {
086:         String amtString = "";
087:         int amt = rs.getInt("Amount");
088:         if (amt < 50)
089:           amtString ="<TD><CENTER><FONT COLOR='RED'>" +
090:                          amt + "</FONT></CENTER></TD>";
091:         else
092:           amtString ="<TD><CENTER>" +
093:                          amt + "</CENTER></TD>";
094:
095:         out.println("<TR><TD><CENTER>" + rs.getString("Name") +
096:                       "</CENTER></TD><TD><CENTER>" +
097:                       rs.getString("Description") +
098:                       "</CENTER></TD>" + amtString + "</TR>");
099:       }
100:       rs.close();
101:       out.println("</TABLE><HR>Items in <FONT COLOR='red'>RED</FONT>"
102:                   + " denote a possible low inventory. Click Here to " +
103:                     " contact <A HREF='mailto:mgmt@localhost'>" +
104:                     "MANAGEMENT</A> to order more supplies.");
105:
106:       //Free the connection!
107:       m_dbsingleton.freeConnection( con );
108:
109:     }
110:     catch (Exception e)
111:     {
112:       out.println("There were errors connecting to the database." +
113:                   " Please see your systems administrator for details.");
114:       e.printStackTrace();
115:     }
116:
117:   }
118:
119:
120: }
121:
122:
```

Listing 32.4 *(continued)*

Using this Singleton class, you would simply need to do the following in your servlet:

```
LavenderDBSingleton singleton = LavenderDBSingleton.getInstance();
Connection con = singleton.getConnection();
try
{
  Statement stmt = con.createStatement();
  ResultSet rs = stmt.executeQuery("select * from Inventory");
  //do the rest...
  singleton.freeConnection(con);
}
catch (Exception e)
{
  //...
}
```

If the other servlets on our Web server VM use the Singleton in this manner, we will maximize the efficiency of our servlet, reducing the overhead of creating connections for every client (and for every thread!). The initial performance overhead may be the initial instantiation of the Singleton. For that purpose, it may be wise to instantiate it in the servlet's init() method. For the purposes of brevity, we have not included a full program example of this in the book. The Web site, however, will have a code listing that you can use.

In this pitfall, we discussed a few of the performance pitfalls that can arise when servlets communicate directly to a database. We presented two methods of connection pooling with servlets—one where a servlet shares a connection pool with its clients and one where all servlets share a connection pool by calling a Singleton class that will reside in memory in the virtual machine.

It should be noted, however, that there are ways of abstracting the database connection away from the user interface (servlet) model. Using servlets as the front end in a J2EE architecture where Enterprise JavaBeans (EJBs) worry about database connections is a good way for accomplishing this abstraction. While this pitfall was meant for developers who build applications where servlets connect to the database, there will be other pitfalls in this book that will discuss the use of EJBs.

For more information about other methods of connection pooling, see Item 45 in Part Three, "The Enterprise Tier."

Item 33: Attempting to Use Both Output Mechanisms in Servlets

If you've done a lot of servlet programming, you probably recognize this pitfall. The Servlet API provides two mechanisms for printing out a response: PrintWriter and ServletOutputstream. This pitfall discusses problems that may occur in using these two objects and will demonstrate an example.

In Listing 33.1, we have created a simple servlet that takes a quick voting poll on the Internet. We use this servlet along with a helper object called VoterApp, which has

methods that tally votes, create an HTML-formatted “Poll of the Day,” and create a graphical image representing the current tally of today’s votes. Our servlet either shows the HTML Poll and creates a form for the user to vote or it shows the user a graph of the current tally. If the `vote` parameter in our servlet is `null`, it will create the poll, as shown in Listing 33.1 on line 29. In lines 33 to 40 of our servlet’s `doGet()` method, we create an HTML form, with most of the contents returned from the `getPollOfTheDay()` method on the `VoterApp` object.

```
01: package org.javapitfalls.item33;
02: import java.io.*;
03: import java.text.*;
04: import java.util.*;
05: import javax.servlet.*;
06: import javax.servlet.http.*;
07: /* Bad Voter Servlet Example */
08: public class BadVoterServlet extends HttpServlet
09: {
10: 
11:     public void doGet(HttpServletRequest request,
12:                       HttpServletResponse response)
13:     throws IOException, ServletException
14:     {
15:         doPost(request,response);
16:     }
17: 
18: 
19:     public void doPost(HttpServletRequest request,
20:                        HttpServletResponse response)
21:     throws IOException, ServletException
22:     {
23:         String vote = request.getParameter("vote");
24: 
25:         PrintWriter out = response.getWriter();
26: 
27:         VoterApp voter = VoterApp.getInstance();
28: 
29:         if ( vote == null )
30:         {
31:             //Let's print out the Poll of the Day!
32:             response.setContentType("text/html");
33:             out.println("<TITLE>Poll of the Day!</TITLE>");
34:             out.println("<FORM METHOD='POST' ACTION='" +
35:                         request.getRequestURI() + "'>");
36: 
37:             out.println(voter.getPollOfTheDay());
38: 
39:             out.println("<INPUT TYPE='SUBMIT' VALUE='Vote Now!'>");
40:             out.println("</FORM>");
41:         }
```

Listing 33.1 BadVoterServlet.java

```
42:         else
43:         {
44:             //Have our voter object tally up the results
45:             voter.addToPollResults(vote);
46:
47:             //Get the generated poll results graph
48:             byte[] generatedGraph = voter.generateImageBytes();
49:             if ( generatedGraph == null )
50:             {
51:                 response.setContentType("text/html");
52:                out.println("<B>Technical difficulties.. Please see" +
53:                             " your administrator for details.</B>");
54:                 return;
55:             }
56:             else
57:             {
58:                //We need to get the outputstream to write binary data
59:
60:                 ServletOutputStream os = response.getOutputStream();
61:                 response.setContentType("image/gif");
62:
63:                 os.write(generatedGraph, 0, generatedGraph.length);
64:                 os.flush();
65:             }
66:         }
67:     }
68:
69: }
70:
```

Listing 33.1 *(continued)*

The first time we run this servlet (with no parameters), the browser shows the poll of the day, as seen in Figure 33.1. Once we vote, however, we see an error message that appears on our browser:

```
java.lang.IllegalStateException: Writer is already being used for this
request at
org.apache.tomcat.facade.HttpServletResponseFacade.getOutputStream(HttpS
ervletResponseFacade.java:156)
     at BadVoterServlet.doPost(BadVoterServlet.java:63)
```

What went wrong? The problem is that we requested the `PrintWriter` on line 25 of our servlet, and then we requested the `ServletOutputStream` object on line 60 of our servlet in Listing 33.1. Servlet documentation tells us that we should use `ServletOutputStream` for printing binary data and use `PrintWriter` for printing out character text. But our servlet writes both binary data *and* character text. What should we do?

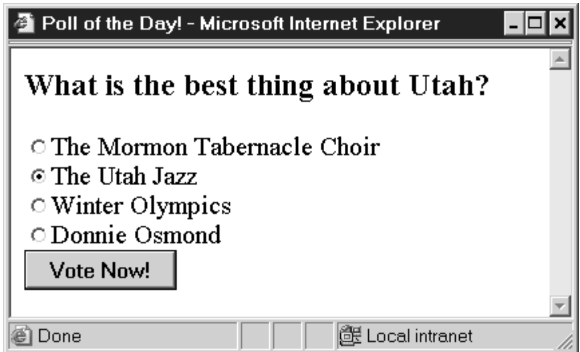


Figure 33.1 The first part works!

We can either use one or the other. `ServletOutputStream` is an abstract class that extends `java.io.OutputStream`, and `PrintWriter` is a class that extends `java.io.Writer`. The difference is that `PrintWriter` is character-based and `ServletOutputStream`, like `java.io.OutputStream`, is byte-based. To make life easier, `ServletOutputStream` adds `print()` and `println()` methods for primitive types and `Strings`, as shown in Table 33.1.

Table 33.1 javax.servlet.ServletOutputStream Methods

METHOD
`void print(boolean b);` `void println(boolean b);`
`void print(char c);` `void println(char c);`
`void print(double d);` `void println(double d);`
`void print(float f);` `void println(float f);`
`void print(int i);` `void println(int i);`
`void print(long l);` `void println(long l);`
`void print(String s);` `void println(String s);`
`void println();`

Because ServletOutputStream contains print() and println() methods, it is easy to see that you may print character-based data with the print() and println() methods, and binary data with the write() methods inherited from java.io.OutputStream. To mix binary and text data in a multipart response, you may use a ServletOutputStream and manage the character sections with the methods in Table 33.1. Still, the rule of thumb is that if you are only sending character data, use the PrintWriter object returned by getWriter().

If your servlet is used for both binary and character data, you may be wise to use ServletOutputStream, rather than getting one or the other in an if-then clause or a try-catch clause. We have seen instances where code asks for the ServletOutputStream right before printing binary data, but asks for the PrintWriter if an error occurs in processing. This is a bad idea and can introduce conditional complexity to your servlet code, making the lifetime of your servlet difficult to manage. Listing 33.2 shows a segment of servlet code that will compile but could introduce complexity and bugs as time goes on. In that listing, the developer gets the ServletOutputStream in the try{} segment, and if an error occurs, it asks for the PrintWriter in the catch() segment.

```
01: //This is a very bad idea!
02: PrintWriter pw;
03: ServletOutputStream out;
04: try
05: {
06:   //get binary data
07:   out = response.getOutputStream();
08:   //now write the binary data with out
09: }
10: catch (Exception e)
11: {
12:   pw = response.getWriter();
13:   pw.println("There was an error: " + e.getMessage())
14:}
```

Listing 33.2 Using control flow to determine output

Listing 33.2 is simple enough, but if the servlet is changed to include more and more conditionals, code management could be a nightmare as time goes on. What if more processing goes on after that segment of code? You would have to keep track of which one is used, and that is a bad idea. It is a lot easier to simply use the ServletOutputStream object alone, if there is a case where you are printing binary data.

The important thing to remember is that if you are sending different types of data to the client, make sure to set the content type with the HttpServletResponse object, as shown in Listing 33.1 on lines 51 and 61. Since our “Poll of the Day” servlet example

prints either binary or character-based data, we should simply ask for the `ServletOutputStream`. Listing 33.3 shows the good example.

```
01: import java.io.*;
02: import java.text.*;
03: import java.util.*;
04: import javax.servlet.*;
05: import javax.servlet.http.*;
06:
07: /* Good Voter Servlet Example */
08: public class GoodVoterServlet extends HttpServlet
09: {
10:
11:     public void doGet(HttpServletRequest request,
12:                       HttpServletResponse response)
13:     throws IOException, ServletException
14:     {
15:         doPost(request,response);
16:     }
17:
18:
19:     public void doPost(HttpServletRequest request,
20:                        HttpServletResponse response)
21:     throws IOException, ServletException
22:     {
23:         String vote = request.getParameter("vote");
24:
25:         ServletOutputStream out = response.getOutputStream();
26:
27:         VoterApp voter = VoterApp.getInstance();
28:
29:         if ( vote == null )
30:         {
31:             //Let's print out the Poll of the Day!
32:             response.setContentType("text/html");
33:             out.println("<TITLE>Poll of the Day!</TITLE>");
34:             out.println("<FORM METHOD='POST' ACTION='" +
35:                         request.getRequestURI() + "'>");
36:
37:             out.println(voter.getPollOfTheDay());
38:
39:             out.println("<INPUT TYPE='SUBMIT' VALUE='Vote Now!'>");
40:             out.println("</FORM>");
41:         }
42:         else
43:         {
44:             //Have our voter object tally up the results
45:             voter.addToPollResults(vote);
46:
```

Listing 33.3 GoodVoterServlet.java

```
47:             //Get the generated poll results graph
48:             byte[] generatedGraph = voter.generateImageBytes();
49:             if ( generatedGraph == null )
50:             {
51:                 response.setContentType("text/html");
52:                out.println("<B>Technical difficulties.. Please see " +
53:                             "your administrator for details.</B>");
54:                 return;
55:             }
56:             else
57:             {
58:
59:                 response.setContentType("image/gif");
60:
61:                 out.write(generatedGraph, 0, generatedGraph.length);
62:                 out.flush();
63:             }
64:         }
65:     }
66:
67: }
68:
```

Listing 33.3 *(continued)*

As you can see in Listing 33.3, we get the `ServletOutputStream` on line 25. We use the `println()` methods for character-based data in lines 32 to 40 after setting the content-type to "`text/html`" on line 32. When there is an error in getting the binary image, we use the same mechanism for printing binary data on lines 49 to 55. Finally, when we have binary data to produce, we set the content-type to `"image/gif"`, and write the binary data (or the graphed poll results in this example) to the user's browser with the methods inherited from `java.io.OutputStream` on lines 59 to 62.

This item showed the possible pitfalls that may lurk in your code when you try to use `PrintWriter` and `ServletOutputStream` together in a servlet. We showed you how to determine which object to use and showed how to eliminate this problem from your code.

Item 34: The Mysterious File Protocol

Many developers have built applications that read files from both the Internet and the local filesystem. Invariably, as developers get more seasoned, they discover that they are able to use the "file protocol" to reference local files as URLs. Because of this, a growing number of tools and APIs are beginning to simply accept URLs as references for files.

A lot of Java classes are overloaded to handle both the conventional local file syntax and also the URL syntax for files. Figure 34.1 shows an example of a simple Web browser, built with one of those classes, `JEditorPane`. Observe that it is a simple `JFrame` with a `JTextField` and a `JScrollPane` containing the `JEditorPane`. A URL is typed into the `JTextField`, which is then browsed by calling the `setPage(String url)` method on the `JEditorPane`.

With the exception of the HTML rendering being more primitive than the average Web browser user expects, it works quite well as a basic Web browser. However, when the user tries to take advantage of the "file protocol" to browse the local filesystem, he or she needs to be careful. If the user simply substitutes the file protocol for the HTTP protocol, thus creating a URL that starts with "file://" instead of the usual "http://", then the user will be surprised to find the error shown in Figure 34.2 popping up.

The developer feverishly checks to determine if the slashes must go the other way, if the colon after the drive letter must come out, if a pipe character needs to substitute for the colon—anything to make the exception shown in Listing 34.1 go away.

Figure 34.1 JEditorPane loading normally.

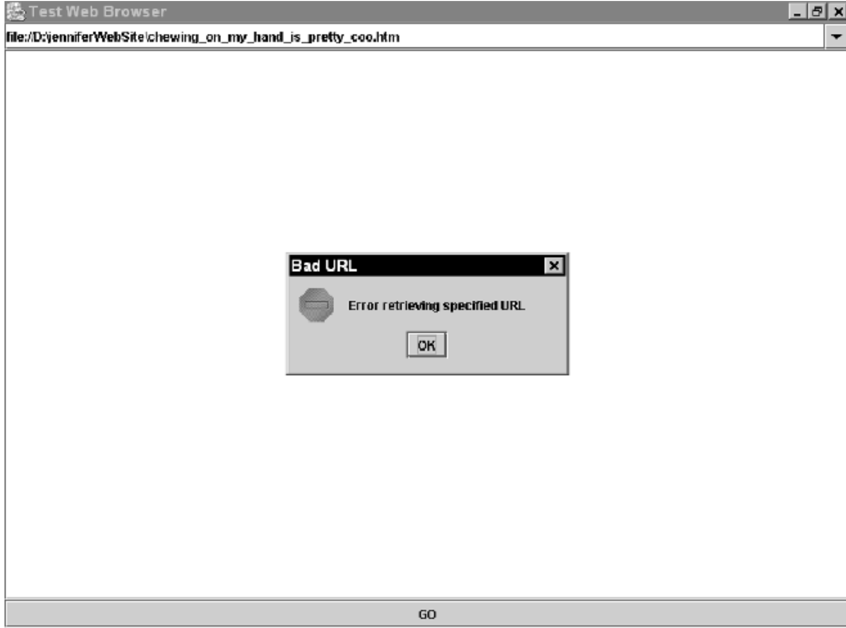


Figure 34.2 Loading with file protocol misapplied.

```
01: java.net.UnknownHostException: D
02:   at java.net.InetAddress.getAllByName0(InetAddress.java:571)
03:   at java.net.InetAddress.getAllByName0(InetAddress.java:540)
04:   at java.net.InetAddress.getByName(InetAddress.java:449)
05:   at java.net.Socket.<init>(Socket.java:100)
06:   at sun.net.NetworkClient.doConnect(NetworkClient.java:50)
07:   at sun.net.NetworkClient.openServer(NetworkClient.java:38)
08:   at sun.net.ftp.FtpClient.openServer(FtpClient.java:267)
09:   at sun.net.ftp.FtpClient.<init>(FtpClient.java:381)

20:   at [...]
21:
```

Listing 34.1 UnknownHostException

The developer starts thinking: "UnknownHostException! That cannot be right. Shouldn't it be a MalformedURLException?" Desperate for a workaround, other measures are thrown around, "What if I test for the 'file://' and then strip the rest out and load it as a local file? That won't work . . . now I need to find another method to set the page, because setPage deals with URLs."

The developer takes another look at the URL class and sees this gem:

Class URL represents a Uniform Resource Locator, a pointer to a "resource" on the World Wide Web. A resource can be something as simple as a file or a directory, or it can be a reference to a more complicated object, such as a query to a database or to a search engine. More information on the types of URLs and their formats can be found at http://archive.ncsa.uiuc.edu/SDG/Software/Mosaic/Demo/url-primer.html.

The developer quickly browses over to confirm his understanding of the proper format of the file protocol, and this is what he finds:

File URLs
Suppose there is a document called "foobar.txt"; it sits on an anonymous ftp server called "ftp.yoyodyne.com" in directory "/pub/files". The URL for this file is then:
file://ftp.yoyodyne.com/pub/files/foobar.txt
The toplevel directory of this FTP server is simply:
file://ftp.yoyodyne.com/
The "pub" directory of this FTP server is then:
file://ftp.yoyodyne.com/pub
That's all there is to it.[2]

"That's all there is to it!" If that isn't adding insult to injury! The information posted reminds the developer of two things: his understanding of the file protocol is correct and he is still at square one.

Desperate to make some sort of progress, the developer starts considering other options and inspecting the File class. In the File class there is a method called toURL(), which claims to convert the path and filename into a URL specifying the file protocol. The developer decides to add a JFileChooser into the mix and use the resulting selected file to create the URL:

```
try {

textField.setText(chooser.getSelectedFile().toURL().toString());

} catch (MalformedURLException malE) { malE.printStackTrace(); }
```

Figure 34.3 shows how that looks in the application.

After the developer gets this code up and running, the JFileChooser is now handling the URL specification for the application. This approach turns out to be a successful one, as shown in Figure 34.4.

[2]http://archive.ncsa.uiuc.edu/SDG/Software/Mosaic/Demo/url-primer.html

Test Web Browser

Open

Look In: jenniferWebSite

_private
Chewing on my hand is pretty cool_files
Crawling around on the floor here_files
images
As you may have noticed.htm
as_you_may_have_noticed.htm
astrbul1e.gif
Chewing on my hand is pretty cool.htm
chewing_on_my_hand_is_pretty_coo.htm
Code.xml
Crawling around on the floor here.htm
crawling_around_on_the_floor_her.htm
eatingWithDad.htm
eyes_open.htm
favorite.htm
favorite_sleeping_place.htm
index.htm
interest.htm

File Name: chewing_on_my_hand_is_pretty_coo.htm
Files of Type: All Files

Open Cancel
Open selected file

Load Files

Figure 34.3 The JFileChooser strategy in action.

Figure 34.4 Properly loaded local file.

Wanting to determine why the representation in the `JTextField` has only one slash and whether that would work if entered by hand, the developer performs some tests. He determines that it works without the slash at all and up to seven slashes—then realizes that the only number of slashes that doesn't work is the only number that should: two.

Why does this problem occur? Well, reviewing RFC2396 (URLs) more closely, we note a distinction is made between local and remote files. Absolute paths are annotated with single slashes, followed by the appropriate path segments, whereas remote paths are indicated by the double slashes. This is a common misunderstanding, driven out of the assumption that the file protocol specification refers only to the local filesystem. In fact, looking more closely at the previous example, we see the URL is actually pointing at a remote file server, an FTP server, and hence the two slashes. Note, however, that this subtle distinction is not made to the reader, nor is it prominent in the RFC.

Now that it has been established that the URL was wrong all along, why didn't a `MalformedURLException` get thrown when the developer was trying to create this URL? After all, this is an incorrect URL. Instead, unexpectedly an `UnknownHostException` is thrown. To understand this, the developer must consider the challenges of implementing such functionality. First, it is important to recognize that an `UnknownHostException` is the result of being unable to resolve the host part of the URL to an IP address. As such, variability must be built in to allow for both hosts that do not exist and hosts that are simply unavailable. After all, how can it be determined that "vol1" is incorrectly referring to a local volume, as opposed to a file server on the network known as "vol1"? This requires the inconvenient problem of receiving an `UnknownHostException` rather than a `MalformedURLException`.

However, this still does not explain why seven slashes works—in fact, any multiple of slashes other than two works. It appears that this is merely because the code checks for two slashes to be a remote host and resolves any number of slashes other than that to a path. This clearly does not strictly adhere to RFC2396, but this is not something developers frequently know or even care about until running across this problem.

Item 35: Reading Files from Servlets

When developers begin to write servlet code, they quickly come across one major reality: They are writing code that runs within the context of another application (a servlet engine). Since they are within the context of another application, they are forced to live within that context, or at least they should be. Unfortunately, this is not always the case. One major advantage of servlets is their access to the entire Java programming language, but it means that servlets can be written to do a number of ill-advised things within the servlet engine. This includes reading files from the local filesystem.

Listing 35.1 is an example of the problem. For the sake of simplicity, ignore the practice of hard-coding "header.html" into the example.

```
01: public class ReallyBadReadingServlet extends HttpServlet {
02:   private static final String CONTENT_TYPE = "text/html";
03:   private StringBuffer strBuf;
04:   private String header;
05:
06:   /**Initialize global variables*/
07: public void init(ServletConfig config) throws ServletException {
08:     super.init(config);
09:
10:     strBuf = new StringBuffer();
11:     try {
12:
13: BufferedReader bufRead = new BufferedReader(new ⮐
FileReader("header.html"));
14:
15:       while (bufRead.read() != -1){
16:         strBuf.append(bufRead.readLine());
17:       }
18:
19:       bufRead.close();
20:
21:     } catch (IOException ioe) {
22:       ioe.printStackTrace();
23:     };
24:
25:     header = strBuf.toString();
27:   }
28:
```

Listing 35.1 ReallyBadReadingServlet.java

The example shows a conventional `BufferedReader` buffering a `FileReader`. The servlet is reading a header file to include with output. This is a simplified example of reading and processing a file.

The major problem with this code is that the working directory, where it will seek the "header.html" file, cannot be reliably determined between servlet engine implementations. As an example, some vendors run their servlet engine in the same process as their HTTP daemon (to improve performance). In an effort to keep the implementation simple, a frustrating search for where to place the file begins, which ends up costing the developer a lot more time over a small subtlety in the servlet specification. It bites the developer again if he or she deploys the servlet into another servlet engine.

The servlet specification does have a couple of methods that can help avoid this problem. First, every servlet has a `ServletContext` object, which has a `getRealPath(String virtualPath)` method. This means that it will give the local filesystem path for the specified virtual path. Notice in the example in Listing 35.2 that `localPath` is prepended to the "header.html" filename.

```
01: public class StillBadReadingServlet extends HttpServlet {
02:   private static final String CONTENT_TYPE = "text/html";
03:   private StringBuffer strBuf;
04:   private String header;
05:
06:   /**Initialize*/
07:   public void init(ServletConfig config) throws ServletException {
08:     super.init(config);
09:
10:     ServletContext context = config.getServletContext();
11:
12:     String localPath = context.getRealPath("/myapp");
13:
14:     strBuf = new StringBuffer();
15:     try {
16:
17: BufferedReader bufRead = new BufferedReader(new FileReader(localPath ⮐
+ "header.html"));
18:
19:       while (bufRead.read() != -1){
20:         strBuf.append(bufRead.readLine());
21:       }
22:
23:       bufRead.close();
24:
25:     } catch (IOException ioe) {
26:       ioe.printStackTrace();
27:     };
28:
29:     header = strBuf.toString();
30:
31:   }
32:
```

Listing 35.2 StillBadReadingServlet.java

The problem with using `getRealPath()` occurs when the path you are attempting to get is within a Web Application Archive, also known as a WAR. The `getRealPath()` method will return `null` as its result. A WAR can be thought of as the Web equivalent of

a JAR file. It provides a deployable component to plug into a Web server, which contains all of the servlets, filters, JSPs, tag libraries, HTML pages, and images.

In the situation shown in Listing 35.3, the developer specifies an explicit path for the file in the local filesystem (e.g., `C:\pitfallsbook\code\header.html`). This definition is specified within parameters passed to the `ServletConfig` object, but the concept can be implemented in a number of ways. No longer using an implicit path definition, the developer decides to place the burden of the deployer to specify a path that is valid (and accessible). So, while this solves the first problem of not definitively knowing where files should be stored, it still doesn't help if we are using the WAR (as is a good practice).

```
01: public class OKReadingServlet extends HttpServlet {
02:   private static final String CONTENT_TYPE = "text/html";
03:   private StringBuffer strBuf;
04:   private String header;
05:
06:   /**Initialize global variables*/
07:   public void init(ServletConfig config) throws ServletException {
08:     super.init(config);
09:
10:     ServletContext context = config.getServletContext();
11:
12:     //Explicit specification of the path via file property
13:     String configFile = config.getInitParameter("Config-File");
14:
15:     strBuf = new StringBuffer();
16:     try {
17:
18:       BufferedReader bufRead = new BufferedReader(new ⮐
FileReader(configFile));
19:
20:       while (bufRead.read() != -1){
21:         strBuf.append(bufRead.readLine());
22:       }
23:
24:       bufRead.close();
25:
26:     } catch (IOException ioe) {
27:       ioe.printStackTrace();
28:     };
29:
30:     header = strBuf.toString();
33:   }
```

Listing 35.3 OKReadingServlet.java

To alleviate this issue, the servlet specification adopted the paradigm first introduced in the Applet. As shown in Listing 35.4, the solution is to call `getResource(String path)`, which returns a `URL` object of the file at the path—a relative URL to the file. This methodology coincides with the way applets retrieve resources from a JAR file.

```
01: public class GoodReadingServlet extends HttpServlet {
02:   private static final String CONTENT_TYPE = "text/html";
03:   private StringBuffer strBuf;
04:   private String header;
05:
06:   /**Initialize global variables*/
07:   public void init(ServletConfig config) throws ServletException {
08:     super.init(config);
09:
10:     ServletContext context = config.getServletContext();
11:
12:     strBuf = new StringBuffer();
13:     try {
14:         //  Using an init parameter
15:         URL headerURL = context.getResource(config ⮐
.getInitParameter("header"));
16:         //  Getting the Content directly
17:         strBuf.append(headerURL.getContent());
18:
19:     } catch (IOException ioe) {
20:       ioe.printStackTrace();
21:     };
22:
23:     header = strBuf.toString();
24:
25:   }
```

Listing 35.4 GoodReadingServlet.java

Notice how this allows the user to avoid having to create a reader and iterate through the reader to populate the header string. When the purpose is merely to read the file into a variable, the `getResource()` method provides the most direct way.

However, often the developer wants to process the file, not just load its content into a variable. To perform this function, the developer has a better way: call `getResourceAsStream()`. This method, shown in Listing 35.5, returns an `InputStream`, which can be enclosed in whatever reader the developer needs to handle the processing.

```
01: public class AnotherGoodReadingServlet extends HttpServlet {
02:   private static final String CONTENT_TYPE = "text/html";
03:   private StringBuffer strBuf;
04:   private String header;
05:
06:   /**Initialize global variables*/
07:   public void init(ServletConfig config) throws ServletException {
08:     super.init(config);
09:
10:     ServletContext context = config.getServletContext();
11:
12:     strBuf = new StringBuffer();
13:     try {
14:         //  Using an init parameter
15:         BufferedReader bufRead = new BufferedReader(
16:          new InputStreamReader(
17:            context.getResourceAsStream(
config.getInitParameter("header"))));
18:
19:         //line by line reading allows for any additional processing
to occur
20:         while (bufRead.read() != -1){
21:           strBuf.append(bufRead.readLine());
22:         }
23:
24:         bufRead.close();
28:     } catch (IOException ioe) {
29:       ioe.printStackTrace();
30:     };
31:
32:     header = strBuf.toString();
34:   }
```

Listing 35.5 AnotherGoodReadingServlet.java

Good software development practice demands avoiding the use of hard-coded configuration information. Because of this, strong consideration should be given to placing the information in the servlet engine's Web Application Deployment Descriptor.

This pitfall has introduced the concept of the Web Application Archive and briefly mentioned the Web Application Deployment Descriptor. Now we will discuss the Web Application Deployment Descriptor in greater detail.

Web Application Deployment Descriptors

A primary part of a Web application is its Web Application Deployment Descriptor. This is euphemistically known as the "web.xml", as it is stored in a file by that name. Also, the file is stored in the *mywebapp*/WEB-INF directory.

Listing 35.6 is an annotated version of a Web Application Deployment Descriptor for a common Web application. See the comments in Listing 35.6 for an explanation of each part.

```
001: <?xml version="1.0"?>
002: <!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web
Application 2.3//EN" "http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">
003: <web-app>
004:
005: <!--
006: Basic Web Application Description information.  Mostly used for
Tools to display the web app in a more user friendly manner.
007: -->
008:
009:     <display-name>example</display-name>
010:     <description>My example web application</description>
011:
012: <!--
013: Filters are new to the Servlet 2.3 Specification.
014:
015: This declares the clickstreamFilter, which is an instance of the
com.opensymphone.clickstream.ClickstreamFilter class.
016: This class should be located in the CLASSPATH, but the most common
place to put it is in the WEB-INF/classes or WEB-INF/lib
017: directories.
018:
019: Clickstream tracks all accesses to the site.  See
http://opensymphony.com for more details.
021: -->
022:
023:     <filter>
024:          <filter-name>clickstreamFilter</filter-name>
025:          <filter-
class>com.opensymphony.clickstream.ClickstreamFilter</filter-class>
026:     </filter>
027:
028: <!--
030: The filter-mapping applies the filter to all requests that match
the url-pattern.
032: -->
033:
034:     <filter-mapping>
035:          <filter-name>clickstreamFilter</filter-name>
036:          <url-pattern>*.jsp</url-pattern>
```

Listing 35.6 Web Application Deployment Descriptor (web.xml)

```
037:     </filter-mapping>
038:
039:     <filter-mapping>
040:          <filter-name>clickstreamFilter</filter-name>
041:          <url-pattern>*.html</url-pattern>
042:     </filter-mapping>
043:
044: <!--
046: The servlet listener is new to Servlet 2.3 also.  It also is up ⮐
to the developer to define a listener to monitor lifecycle events.
047:
048: In this case, the ClickstreamListener waits for an HTTPSession to ⮐
terminate, and then logs the HTTP Session information.
050: -->
051:
052:     <listener>
053:          <listener- ⮐
class>com.opensymphony.clickstream.ClickstreamListener</listener-class>
054:     </listener>
055:
056:
057: <!--
059: This declares the XYZServlet, which is in a class by the same name.⮐
By convention, it usually is in the WEB-INF/classes
060: directory or in a jar in the WEB-INF/lib directory
062: -->
063:
064:
065:     <servlet>
066:         <servlet-name>XYZServlet</servlet-name>
067:         <servlet-class>XYZServlet</servlet-class>
068:     </servlet>
069:
070:
071: <!--
073: This servlet mapping dictates that all urls that end in *.xyz will ⮐
be handled by the XYZServlet.
075: -->
076:
077:     <servlet-mapping>
078:         <servlet-name>
079:             XYZServlet
080:         </servlet-name>
081:         <url-pattern>
082:             /*.xyz
083:         </url-pattern>
084:     </servlet-mapping>
085:
086: <!--
```

Listing 35.6 *(continued)*

```
088: These files are the files that are loaded (in the listed order) if
they exist when the webapp is
089: called without a file specified (e.g. /examples/ )
091: -->
092:
093:     <welcome-file-list>
094:          <welcome-file>home.jsp</welcome-file>
095:          <welcome-file>index.jsp</welcome-file>
096:          <welcome-file>index.html</welcome-file>
097:     </welcome-file-list>
098:
099: <!--
101: These are the tag library definitions.  The taglib-uri gives a
unique identifier.
102: The taglib-location gives the location of the taglib's tag library
definition.
103:
104: These taglibs are the Jakarta IO, XTags, and DBTags all very good
libraries from
105: http://jakarta.apache.org/taglibs.
106:
107: The last one is from opensymphony.com, the oscache tag library,
which is an outstanding
108: tag library for caching web documents.
110: -->
111:
112:
113:     <taglib>
114:          <taglib-uri>http://jakarta.apache.org/taglibs/io-
1.0</taglib-uri>
115:          <taglib-location>/WEB-INF/taglibdefs/io.tld</taglib-location>
116:     </taglib>
117:
118:     <taglib>
119:         <taglib-uri>http://jakarta.apache.org/taglibs/xtags-
1.0</taglib-uri>
120:         <taglib-location>/WEB-INF/lib/xtags.tld</taglib-location>
121:     </taglib>
122:
123:     <taglib>
124:         <taglib-
uri>http://jakarta.apache.org/taglibs/dbtags</taglib-uri>
125:         <taglib-location>/WEB-INF/lib/dbtags.tld</taglib-location>
126:         </taglib>
127:
128:     <taglib>
129:         <taglib-uri>oscache</taglib-uri>
130:                <taglib-location>/WEB-INF/lib/oscache.tld</taglib-
location>
```

Listing 35.6 *(continued)*

```
131:     </taglib>
132:
133: <!--
135: This defines a resource reference that is available to the web ↩
application.
136:
137: It should be noted that it doesn't make the resource, just ↩
declares the reference.
138:
139: In Tomcat, you would still need to declare the resource factory in ↩
the server.xml.
141: -->
142:
143: <resource-ref>
144:
145:   <description>
147:     Resource reference to a javax.sql.DataSource
149:   </description>
151:   <res-ref-name>
153:     jdbc/myDB
155:   </res-ref-name>
157:   <res-type>
159:     javax.sql.DataSource
161:   </res-type>
163:   <res-auth>
165:     Container
167:   </res-auth>
169: </resource-ref>
172: <!--
174: This declares a security constraint on the web application.  First ↩
there is definition of the collection,
175: in this case the JSP that flushes the cache, and the auth- ↩
constraint defines the user and/or roles that
176: this constraint applies to.  In this case, the role "hero" is able ↩
to access this source.
177:
178: This facility is available programmatically through the ↩
request.isUserInRole(roleName) method.
180: -->
181:
182:
183:   <!-- Protect certain pages with a password -->
184:   <security-constraint>
185:     <web-resource-collection>
186:       <web-resource-name>Flush Cache</web-resource-name>
187:       <url-pattern>/flush.jsp</url-pattern>
188:     </web-resource-collection>
189:     <auth-constraint>
190:       <role-name>hero</role-name>
```

Listing 35.6 *(continued)*

```
191:      </auth-constraint>
192:    </security-constraint>
193:
194:    <!--
196:    This defines the way that user logs into the web application. ⮐
In this case, it uses a FORM, with the
197:    login page and error page defined.
199:     -->
200:
201:    <login-config>
202:      <auth-method>FORM</auth-method>
203:      <realm-name>Example Authentication</realm-name>
204:       <form-login-config>
205:          <form-login-page>/login.jsp</form-login-page>
206:          <form-error-page>/error.jsp</form-error-page>
207:      </form-login-config>
208:    </login-config>
209:
210:    <!--
212:    This is the place where environment entries can be made for the ⮐
web application.  This is quite preferable to a
213:    properties file in that this is part of the configuration.  To ⮐
read this environment entry you could use these
214:    lines of code:
215:
216:  String configDirectory = "";
217:  Context initCtx = new InitialContext();
218:      Context ctx = (Context) initCtx.lookup("java:comp/env");
219:      configDirectory = (String) ctx.lookup("configDirectory");
220:
221:    -->
224:          <env-entry>
225:          <env-entry-name>configDirectory</env-entry-name>
226:          <env-entry-value>C:/pitfallsBook</env-entry-value>
227:          <env-entry-type>java.lang.String</env-entry-type>
228:  </env-entry>
230: </web-app>
```

Listing 35.6 *(continued)*

Item 36: Too Many Submits

The primary purpose of the Web site today is to display dynamic content. Of course, that means at some point the user sends input to a Web application, the input is

processed, and the result is returned. Typically, operations on the back end run fast enough that under normal circumstances little can go wrong. However, occasionally, more time-consuming processing must take place—processing that takes more than a second or two. The problem of handling operations that run for long periods of time isn't a new one. Java provides a robust threading mechanism for supporting creating background tasks. Additionally, with the arrival of the EJB 2.0 specification, message-based EJBs can be used to perform background operations. The problem with both of these mechanisms is that they are designed to primarily handle asynchronous operations. You start a thread or background process, and then at some point, you are notified or you check for a result.

The too-many-submits problem occurs when an application is synchronous in nature but still somewhat long-running. Imagine the scenario where a concertgoer logs on to her favorite Web site to order tickets for a show that's just gone on sale. Under normal circumstances, the site performs fine, and our would-be concertgoer purchases her tickets and is on her way. However, when a heavy load occurs, the server slows down, and the buyer gets frustrated waiting for the site and thinks her request to purchase tickets failed. So she hits the Submit button again and again. Unfortunately, earlier requests didn't fail, they were just slow, and so each press of the Submit button ends up ordering *another* set of tickets.

There are many ways to handle the multiple-submit problem. Two of the most obvious is to prevent the user from submitting the same request over and over. The second is to somehow track that a user has previously submitted a request and revert to the previously submitted action. Figure 36.1 shows the output from a simple servlet that processes the input as it arrives and assigns a ticket number to each request.

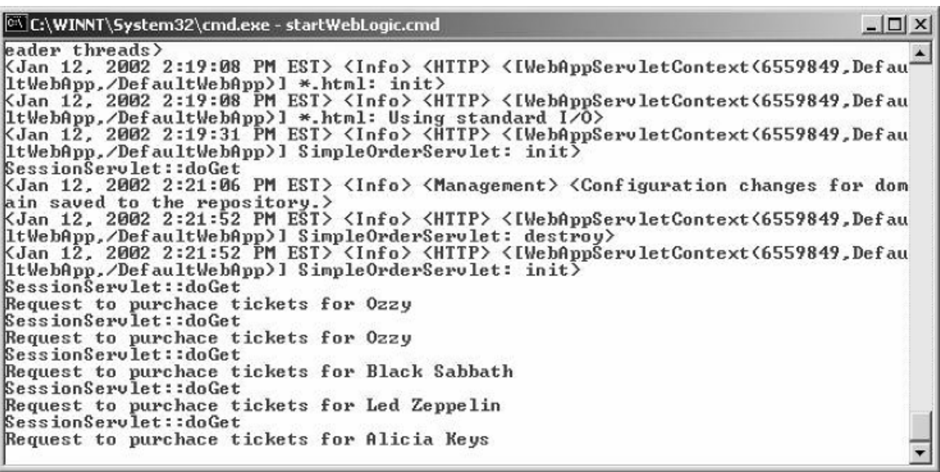


Figure 36.1 Processing simple submissions.

Preventing Multiple Submits

The first and most effective way to handle the multiple-submit problem is to not allow it to happen in the first place. Listing 36.1 shows the underlying HTML for a simple form that captures the name of a concert and submits it to a servlet to order tickets. The form works perfectly fine when the Web site performs adequately speedwise. However, if the Web site bogs down and the submit is not processed quickly enough, the user gets frustrated and processing such as that shown in Figure 36.2 results; the submit happens over and over.

```
01: <HTML>
02: <HEAD><TITLE>Online Concert Tickets</TITLE></HEAD>
03:
04: <CENTER><H1>Order Tickets</H1></CENTER>
05:
06: <FORM NAME="Order" ACTION="./SimpleOrder" METHOD="GET">
07:   <TABLE BORDER="2" WIDTH="50%" ALIGN="CENTER" BGCOLOR="CCCCCC">
08:     <TR><TD ALIGN="RIGHT" WIDTH="40%">Concert: </TD>
09:         <TD WIDTH="60%"><INPUT TYPE="TEXT" NAME="Concert" ⮐
VALUE=""></TD></TR>
10:
11:     <TR><TD COLSPAN="2" ALIGN="CENTER">
12:     <INPUT TYPE="submit" NAME="btnSubmit"
13:            VALUE="Do Submit"></TD></TR>
14:   </TABLE>
15: </FORM>
16: </BODY>
17: </HTML>
```

Listing 36.1 ConcertTickets.html form

```
C:\WINNT\System32\cmd.exe - startWebLogic.cmd
SessionServlet::doGet
Request to purchace tickets for ozzy
<Jan 12, 2002 2:43:19 PM EST> <Info> <HTTP> <[WebAppServletContext(6559849,Defau
ltWebApp,/DefaultWebApp)] SimpleOrderServlet: destroy>
<Jan 12, 2002 2:43:19 PM EST> <Info> <HTTP> <[WebAppServletContext(6559849,Defau
ltWebApp,/DefaultWebApp)] SimpleOrderServlet: init>
SessionServlet::doGet
Request to purchace tickets for ozzy
Sleeping for 15 seconds
SessionServlet::doGet
Request to purchace tickets for ozzy
Sleeping for 15 seconds
SessionServlet::doGet
Request to purchace tickets for ozzy
Sleeping for 15 seconds
SessionServlet::doGet
Request to purchace tickets for ozzy
Sleeping for 15 seconds
SessionServlet::doGet
Request to purchace tickets for ozzy
Sleeping for 15 seconds
SessionServlet::doGet
Request to purchace tickets for ozzy
Sleeping for 15 seconds
```

Figure 36.2 Repeated submissions.

As we said, the simplest way to handle the multiple-submit problem is to stop it from happening. Listing 36.2 shows a subtly different version of our concert order form—one that has a small amount of embedded Java script. The embedded Java script "remembers" if the Submit button was previously pressed, and if so, an alert pops up and the submit isn't processed. We short-circuit the normal submit processing by adding an `onClick` attribute to the Submit button. Every time the button is pressed, the code described in the `onClick` is processed. In our case this results in the JavaScript `checksubmitcount()` method being called. However, just calling a function doesn't really help. If we did no more then add the `onClick`, we'd get our popup alert box every time the Submit button was pressed, and then immediately the submit would happen. The user would be alerted that he or she made a mistake, and the request would be sent anyway. This is an improvement over our prior problem, but only from the user's perspective; the end result was the same: multiple submits.

We can solve the problem by going one step further and subtly changing the way our page works. Sharp readers might have noticed the one additional change. The type of our button, line 12, was originally "submit," but now it is replaced by "button." The look and feel of the page is identical. However, the default action associated with the form, shown on line 6 of Listing 36.1, to invoke the servlet, is no longer automatic. We can now programmatically choose to submit the form to our server, and our problem is solved. Or is it?

```
01: <HTML>
. . .<!-- repeated code removed //-->
12:     <INPUT TYPE="button" NAME="btnSubmit"
13:            VALUE="Do Submit"
14:            onClick="checksubmitcount();"></TD></TR>
15:   </TABLE>
16: </FORM>
17:
18: <SCRIPT LANGUAGE="JAVASCRIPT">
19: <!--
20:     var submitcount = 0;
21:     function checksubmitcount()
22:     {
23:         submitcount++;
24:         if (1 == submitcount )
25:         {
26:             document.Order.submit();
27:         }
28:         else
29:         {
30:          if ( 2 == submitcount)
31:                 alert("You have already submitted this form");
32:          else
33:                 alert("You have submitted this form "
34:                       + submitcount.toString()
35:                       + " times already");
```

Listing 36.2 Concert2.html form *(continued)*

```
36:         }
37:     }
38: //-->
39: </SCRIPT>
40: </BODY>
41: </HTML>
```

Listing 36.2 *(continued)*

Handling Multiple Submits

Listing 36.2 was certainly an improvement, but we've still got a ways to go. A number of issues still could go wrong. For example, what if the user pushes the back button and starts over? What if his or her browser has JavaScript disabled, or for some other reason, handling the processing in the browser cannot be used? We can still solve the problem, but now instead of preventing multiple submits, we need to handle them on the back end, via the servlet that processes the form.

To understand how to solve the multiple-submit problem, we must first understand how servlets work with respect to *sessions*. As everyone knows, HTTP is inherently a stateless protocol. To handle state, we need some way for the browser to communicate to the back end that the current request is part of a larger block of requests. Additionally, the server needs a way to manage the data for a given set of requests. The servlet *session* provides us a solution to this problem. The `HttpServlet` methods `doGet()` and `doPost()` are provided with two specific parameters: `HttpServletRequest` and `HttpServletResponse`. The servlet request parameter allows us to access what is commonly referred to as the *servlet session*. Servlet sessions have mechanisms for accessing and storing state information. But what exactly is a servlet session?

A servlet session is a number of things. It is:

- A set of states managed by the Web server and represented by a specific identifier
- Shared by all requests for a given client
- A place to store state data
- Defined, at least for `HttpServlets`, via the `HttpSession` interface

Before we look at how we can solve our problem with multiple submits with a server-side solution, we need to understand the servlet session lifecycle. As with EJBs and other server-side entities, servlet sessions go through a defined set of states during their lifetime. Figure 36.3 shows pictorially the lifecycle of a servlet session.

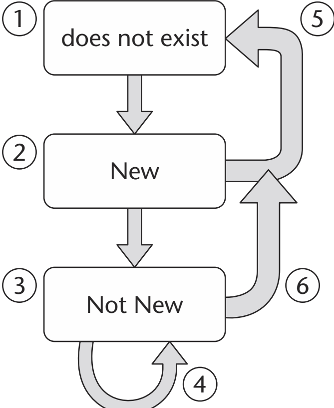


Figure 36.3 Servlet session lifecyle.

Examining how a session moves through its various states during its lifecycle will help us understand how we can solve the multiple-submit problem with a session. Servlets move through three distinct states: *does not exist*, *new*, and *not new* or *in-use*.

1. Initially, a servlet session *does not exist*. A session starts here or returns to this state for a number of reasons. The most likely are that the user has never accessed the state before or the session was invalidated because the user left the site (timed out) or explicitly left (logged out).
2. Sessions move from *does not exist* to *new* when the session is first created. The distinction between *new* and *not new* is important because the HTTP is stateless. Sessions cannot move to not new from being a prospective session to an actual session according to the servlet specification, until the client returns the session back to the server. Thus, sessions are new because the client does not yet know about the session or the client decides not to join the session.
3. When the session is returned back to the server from the client via a cookie or URL rewriting (more on URL rewriting in a moment), then the session becomes in-use or not new.
4. Continued use of the session, via its various get and set methods, results in the session remaining in use.
5. Transitions 5 and 6 happen when a session times out because inactivity of a session causes it to be explicated invalidated. Application servers handle timeouts in a variety of ways. BEA WebLogic Server handles timeouts by allowing the application deployer the ability to set the session timeout via a special deployment descriptor (weblogic.xml) packaged with the Web application.

Now that we understand the lifecycle of a session, how do we go about obtaining a session and using it to our advantage?

The `HttpServletRequest` interface offers two methods for obtaining a session:

- `public HttpSession getSession()`. Always returns either a new session or an existing session.
  - `getSession()` returns an existing session if a valid session ID was provided via a cookie or in some other fashion.
  - `getSession()` returns a new session if it is the client's first session (no ID), the supplied session has timed out, the client provided an invalid session, or the provided session has been explicitly invalidated.
- `public HttpSession getSession(boolean)`. May return a new session or an existing session or `null` depending on how the Boolean is set.
  - `getSession(true)` returns an existing session if possible; otherwise, it creates a new session
  - `getSession(false)` returns an existing session if possible; otherwise, it returns `null`.

NOTE At first glance, it appears that we should always use `getSession(true)`. **However, you should be careful in that an out-of-memory style of attack can be performed on your site by always creating new sessions on demand. An unscrupulous hacker could discover your site was creating sessions and keep pumping requests, each of which would result in a new session. By using** `getSession(false)` **and then redirecting to a login page when a session is not detected, you can protect against such attacks.**

There are a number of interesting methods on `HttpSession` objects such as `isNew()`, `getAttribute()`, `setAttribute()`, and so on. For an exhaustive review, see the Servlet specification or any of the excellent John Wiley & Sons publications.

Getting back to our problem at hand, we have still only solved half of our problem. We'd like to be able to use our sessions to somehow skip over the *session new* state and move to the *session in-use* stage automatically. We can achieve this final step by redirecting the browser back to the handling servlet automatically. Listing 36.3 combines servlet session logic with the ability to redirect, with a valid session, the client back to the handling servlet.

```
01: package org.javapitfalls.item36;
02:
03: import java.io.*;
04: import java.util.Date;
05: import javax.servlet.*;
06: import javax.servlet.http.*;
```

Listing 36.3 RedirectServlet.java

```
07:
08: public class RedirectServlet extends HttpServlet {
09:
10:  static int count = 2;
11:
12:  public void doGet(HttpServletRequest req, HttpServletResponse res)
throws ServletException, IOException {
13:
14:   HttpSession session = req.getSession(false);
15:   System.out.println("");
16:   System.out.println("----------------------------------------");
17:   System.out.println("SessionServlet::doGet");
18:   System.out.println("Session requested ID in request: " +
req.getRequestedSessionId());
19:
20:   if ( null == req.getRequestedSessionId() ) {
21:    System.out.println("No session ID, first call, creating new
session and forwarding");
22:    session = req.getSession(true);
23:    System.out.println("Generated session ID in Request: " +
session.getId());
24:    String encodedURL = res.encodeURL("/resubmit/TestServlet");
25:    System.out.println("res.encodeURL(\"/TestServlet\";=" + encodedURL);
26:    res.sendRedirect(encodedURL);
27:    //
28:    // RequestDispatcher rd = getServletContext()
.getRequestDispatcher(encodedURL);
29:    // rd.forward(req, res);
30:    //
31:    return;
32:   }
33:   else {
34:    System.out.println("Session id = " + req.getRequestedSessionId() );
35:    System.out.println("---------------------------------------------");
36:   }
37:
38:   HandleRequest(req, res);
39:   System.out.println("SessionServlet::doGet returning");
40:   System.out.println("--------------------------------");
41:   return;
42:  }
43:
44:  void HandleRequest(HttpServletRequest req, HttpServletResponse
res) throws IOException {
45:
46:   System.out.println("SessionServlet::HandleRequest called");
47:   res.setContentType("text/html");
48:   PrintWriter out = res.getWriter();
```

Listing 36.3 *(continued)*

```
49:   Date date = new Date();
50:   out.println("<html>");
51:   out.println("<head><title>Ticket Confirmation</title></head>");
52:   // javascript
53:   out.println("<script language=\"javascript\">");
54:   out.println("<!--");
55:   out.println("var submitcount = " + count + ";");
56:   out.println("function checksubmitcount()");
57:   out.println("{");
58:   out.println(" if ( 2 == submitcount )");
59:   out.println("   alert(\"You have already submitted this form.\");");
60:   out.println("   else");
61:   out.println("     alert(\"You have submitted this form " + count + 
" times already.\");");
62:   out.println(" document.Order.submit();");
63:   out.println("}");
64:   out.println("//-->");
65:   out.println("</script>");
66:   // body
67:   out.println("<body>");
68:   out.println("<center>");
69:   out.println("<form name=\"Order\" action=\"./RedirectServlet\" 
method=\"GET\">");
70:   out.println("<table border=\"2\" width=\"50%\" align=\"center\" 
bgcolor=\"#cccccc\">");
71:   out.println("<tr>");
72:   out.println("<td align\"right\" width=\"40%\">Concert:</td>");
73:   out.println("<td width=\"60%\"><input type=\"text\" 
name=\"Concert\" value=\"\"></td>");
74:   out.println("</tr>");
75:   out.println("<tr>");
76:   out.println("<td colspan=\"2\" align=\"center\">");
77:   out.println("<input type=\"button\" name=\"btnSubmit\" value=\"Do 
Submit\" onClick=\"checksubmitcount();\">");
78:   out.println("</td>");
79:   out.println("</tr>");
80:   out.println("</form>");
81:   // message
82:   out.println("<br>");
83:   out.println("<h1>The Current Date and Time Is:</h1><br>");
84:   out.println("<h3>You have submitted this page before</h3><br>");
85:   out.println("<h3>" + date.toString() + "</h3>");
86:   out.println("</body>");
87:   out.println("</html>");
88:
89:   count++;
90:
```

Listing 36.3 *(continued)*

```
91:   System.out.println("SessionServlet::HandleRequest returning.");
92:   return;
93:  }
94:
95: }
```

Listing 36.3 *(continued)*

Just how does Listing 36.3 solve our problem? If we examine the code closely, we see that on line 14 we try to obtain a handle to a session. On line 20 we identify that an active session exists by comparing the session to `null` or by checking for a valid session ID. Either method suffices. Lines 20 to 31 are executed if no session exists, and to handle our problem, we:

1. Create a session, as shown on line 22.
2. Use URL encoding to add the new session ID to our URL, as shown on line 24.
3. Redirect our servlet to the newly encoded URL, as shown on line 26.

Those readers unfamiliar with URL rewriting are directed to lines 18 and 25. The request parameter to an `HttpServlet` can do what is known as *URL rewriting*. URL rewriting is the process whereby a session ID is automatically inserted into a URL. The underlying application server can then use the encoded URL to provide an existing session automatically to a servlet or JSP. Note that depending on the application server, you may need to enable URL rewriting for the above example to work.

WARNING **Lines 28 and 29, while commented out, are shown as an example of something *not to do*. On first glance, `forward` seems to be a better solution to our problem because it does not cause a round-trip to the browser and back. However, `forward` comes at a price: The new session ID *is not* attached to the URL. Using `forward` in Listing 36.3 would cause the servlet to be called over and over in a loop and ultimately kill the application server.**

The JavaScript/servlet implementation described above is okay for many situations, but I've been on several programs that wanted to limit the amount of JavaScript used on their deployments, so I thought it would beneficial to include an example that satisfies that requirement. In the example below, a controller servlet will be used to prohibit multiple user form requests using the Front Controller pattern.

```
01: package org.javapitfalls.item36;
02:
03: import java.io.*;
```

Listing 36.4 ControllerServlet.java *(continued)*

```
04: import java.util.*;
05: import javax.servlet.*;
06: import javax.servlet.http.*;
07: import org.javapitfalls.item36.*;
08:
09: public class ControllerServlet extends HttpServlet {
10:
11: private static String SESSION_ID;
12:
13:  public void destroy() {}
```

Listing 36.4 *(continued)*

Our application reads an `id` tag and its initial value from the deployment descriptor embedded in the `param-name` and `param-value` tags in Listing 36.5 on lines 29 and 30. This read operation takes place in the `init()` method on line 17 of the controller servlet in Listing 36.4 and will be used to identify the user session. The controller application uses three parameters: `numTickets`, `stadiumTier`, and `ticketPrice`, as data items to process from the ticketForm application shown in Listing 36.4. The `getNamedDispatcher` forwards all requests by the name mappings specified in the deployment descriptor. The form request associates the `ticketForm.jsp` with the “form” label on line 44 of the web.xml in Listing 36.5. This method is preferred over dispatching requests by application path descriptions because this exposes the path information to the client, which could present a safety concern. Additionally, it is a good practice to migrate applications and their dependencies to the deployment descriptor so that modifications can be made more easily.

```
14:
15:  public void init() throws ServletException {
16:
17:     SESSION_ID = getInitParameter("id");
18:
19:  }
20:
21:  protected void doGet(HttpServletRequest req, HttpServletResponse
res) throws ServletException, IOException {
22:
23:   process(req, res);
24:
25:  }
26:
27:  protected void process(HttpServletRequest req,
28:                         HttpServletResponse res)
29:                         throws ServletException, IOException {
30:
```

Listing 36.4 *(continued)*

```
31:    HttpSession session = req.getSession(false);
32:    String numTickets = req.getParameter("numTickets");
33:    String stadiumTier = req.getParameter("stadiumTier");
34:    String ticketPrice = req.getParameter("ticketPrice");
35:    if(session == null) {
36:      if( (numTickets == null) || (stadiumTier == null) ||
37:                                  (ticketPrice == null) ) {
38:
39:        getServletConfig().getServletContext().
40:              getNamedDispatcher("form").forward(req, res);
41:
42:      } else {
43:        throw new ServletException("[form] Page Not Found");
44:      }
45:
46:    } else {
47:
48:       if ( (!numTickets.equals("Please enter a Ticket #")) &&
49:          (!stadiumTier.equals("Please enter a Stadium Tier")) &&
50:          (!ticketPrice.equals("Please enter a Ticket Price")) ) {
51:
```

Listing 36.4 *(continued)*

The `session.getAttribute` operation on line 52 reads the ID name captured in the `init` method on line 17 during the initialization of the controller servlet. This ID, `SESSION_ID`, will serve as the session identifier for the submit page. If the user has entered all the proper form information on the ticketForm page, and the session ID is not null, then the controller will remove the ID and forward the application to the successful completion page. When the form has been properly completed and the session ID is equal to null, then the user will be forwarded to the error page that indicates that the `ticketForm` has already been completed satisfactorily and cannot be resubmitted.

```
52:         String sessionValidatorID =
53:              (String)session.getAttribute(SESSION_ID);
54:         if(sessionValidatorID != null ) {
55:
56:           session.removeAttribute(SESSION_ID);
57:           getServletConfig().getServletContext().
58:              getNamedDispatcher("success").forward(req, res);
59:
60:         } else {
61:           getServletConfig().getServletContext().
62:             getNamedDispatcher("resubmit").forward(req, res);
63:         }
64:
```

Listing 36.4 *(continued)*

```
65:       } else {
66:
67:         getServletConfig().getServletContext().
68:                 getNamedDispatcher("form").forward(req, res);
69:       }
70:
71:     }
72:   }
73:
74: }
75:
```

Listing 36.4 *(continued)*

Lastly, the deployment descriptor exhibits the application's mappings that allow requests to be forwarded and processed by the controller. As mentioned earlier, the session ID token is read from the parameter tags on lines 25 and 26 of Listing 35.5. The JavaServer Pages that are used for presentation are shown on lines 42 to 55. When the controller uses the `getNamedDispatcher` method, a label is passed that is associated with a JSP script. When a user attempts to resubmit the ticketForm page, the `resubmit` label is passed through controller, which forwards control to the resubmit error page (resubmitError.jsp).

```
01: <?xml version="1.0"?>
02: <!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web ⮐
Application 2.3//EN" "http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">
03: <web-app>
04:
05:   <servlet>
06:     <servlet-name>RedirectServlet</servlet-name>
07:     <display-name>RedirectServlet</display-name>
08:     <description>RedirectServlet</description>
09:     <servlet-class>
10:       org.javapitfalls.item36.RedirectServlet
11:     </servlet-class>
12:   </servlet>
13:   <servlet>
14:     <servlet-name>SimpleOrder</servlet-name>
15:     <display-name>SimpleOrder</display-name>
16:     <description>SimpleOrder</description>
17:     <servlet-class>
18:       org.javapitfalls.item36.SimpleOrder
19:     </servlet-class>
20:   </servlet>
21:   <servlet>
22:     <servlet-name>ControllerServlet</servlet-name>
23:     <display-name>ControllerServlet</display-name>
```

Listing 36.5 web.xml

```
24:     <description>ControllerServlet</description>
25:     <servlet-class>
26:       org.javapitfalls.item36.ControllerServlet
27:     </servlet-class>
28:     <init-param>
29:     <param-name>id</param-name>
30:     <param-value>id</param-value>
31:     </init-param>
32:   </servlet>
33:
34:   <servlet>
35:     <servlet-name>TestServlet</servlet-name>
36:     <display-name>TestServlet</display-name>
37:     <description>TestServlet</description>
38:     <servlet-class>
39:       org.javapitfalls.item36.TestServlet
40:      </servlet-class>
41:   </servlet>
42:   <servlet>
43:         <servlet-name>form</servlet-name>
44:         <jsp-file>/ticketForm.jsp</jsp-file>
45:   </servlet>
46:
47:   <servlet>
48:         <servlet-name>success</servlet-name>
49:         <jsp-file>/success.jsp</jsp-file>
50:   </servlet>
51:
52:   <servlet>
53:         <servlet-name>resubmit</servlet-name>
54:         <jsp-file>/resubmitError.jsp</jsp-file>
55:   </servlet>
56:
57:   <servlet-mapping>
58:     <servlet-name>RedirectServlet</servlet-name>
59:     <url-pattern>/RedirectServlet</url-pattern>
60:   </servlet-mapping>
61:
62:   <servlet-mapping>
63:     <servlet-name>SimpleOrder</servlet-name>
64:     <url-pattern>/SimpleOrder</url-pattern>
65:   </servlet-mapping>
66:
67:   <servlet-mapping>
68:     <servlet-name>ControllerServlet</servlet-name>
69:     <url-pattern>/ControllerServlet</url-pattern>
70:   </servlet-mapping>
71:
72:   <servlet-mapping>
```

Listing 36.5 *(continued)*

```
73:     <servlet-name>TestServlet</servlet-name>
74:     <url-pattern>/TestServlet</url-pattern>
75:   </servlet-mapping>
76:
77: </web-app>
78:
```

Listing 36.5 *(continued)*

In this pitfall we discussed a number of solutions to the multiple-submit problem. Each solution, as with almost every solution, had its positive and negative aspects. When solving problems, we must clearly understand the various pros and cons of a solution so we can assess the value of each trade-off. Our final JavaScript example had the benefit of solving the problem at hand, but had the trade-off that we needed to make an extra round-trip to the client in order to make it work. The first JavaScript solution was perhaps the most elegant, but required that the client enables JavaScript for it to work. The final application would serve those well that opted to forego the use of JavaScript to avoid browser incompatibilities with the scripting language. As with any problem, there is often a world of solutions, each one with its own trade-offs. By understanding the trade-offs of a given solution, we can make the most informed choice for a given problem.

PART Three

The Enterprise Tier

"Machine capacities give us room galore for making a mess of it. Opportunities unlimited for fouling things up! Developing the austere intellectual discipline of keeping things sufficiently simple is in this environment a formidable challenge, both technically and educationally."

Edsger W. Dijkstra,
from "The Threats to Computing Science," EWD Manuscript #898

In that paper, which he wrote in 1984, Dijkstra suggests that in a world of incredibly fast computers and increasingly complex programming environments, simplicity is the key. "We know perfectly well what we have to do," he writes, "but the burning question is, whether the world we are a part of will allow us to do it."[1] I would contend that the frameworks present in Java, and specifically on the server side today, provide the beginnings of such a world—not that this will completely keep us from fouling things up! As this book shows you, there are always opportunities for that. However, when we do not have to worry about the management of services such as memory, threads, connection pooling, transactions, security, and persistence, things get simpler. When we allow the container to handle the details of these services for us, we can focus where Dijkstra thought we should focus: on the logic of our application itself.

The J2EE environment provides the atmosphere for Java programmers to focus on application logic. Because application servers focus on the "hard stuff," we can focus on the business logic of our applications. However, since programming in this environment represents a change in mind-set, new enterprise developers stumble into

[1] Dijkstra, Edsger. "The Threats to Computing Science." EWD manuscript #898. Available at http://www.cs.utexas.edu/users/EWD/ewd08xx/EWD898.PDF.

quite a few traps. Because the Java language is so flexible, there is the ability for programmers to add unneeded complexity (thread management, etc.) even when the container provides this support. Because of misunderstandings of the J2EE environment, design errors pop up. The use of database connections and different types of databases present other challenges, and the use of new Java Web services APIs, such as Java API for XML-based Remote Procedure Calls (JAX-RPC) and the Java API for XML Registries (JAXR), provides more opportunities for misunderstandings and traps to occur.

In this section of the book, we have tried to focus on areas of the enterprise tier where we have seen and experienced problems and confusion. Although some of the pitfalls could possibly be tied to the other tiers (JDBC pitfalls, for example), we placed them in this section because developers make use of them in the enterprise tier. Many of the pitfalls in this section revolve around Enterprise JavaBeans, some focus on APIs related to Web services, and some focus on design pitfalls. In this section, many pitfalls focus on "the big picture" in a multi-tier environment.

Here are highlights of pitfalls in this section:

J2EE Architecture Considerations (Item 37). J2EE has become a major player in enterprise IT solutions. Developers are often confused about how to apply this architecture to their solution. This pitfall discusses how those mistakes occur and considerations for avoiding them.

Design Strategies for Eliminating Network Bottleneck Pitfalls (Item 38). This pitfall focuses on pitfalls where network calls and bandwidth are a major factor and shows performance-tuning design strategies to use between tiers.

I'll Take the Local (Item 39). EJB 2.0 introduced the local interface as a mechanism for referencing EJBs within the same process. This pitfall discusses the why and how of local interfaces over the traditional remote interface.

Image Obsession (Item 40). Frequently, Enterprise Java applications are developed on one platform and deployed on another. This can cause some cross platform issues. This pitfall examines one of those issues regarding server generated images.

The Problem with Multiple Concurrent Result Sets (Item 41). When you have to interface with many different types of databases, your connections may be handled in different ways. This pitfall addresses a problem that can arise when you use multiple `ResultSet` objects concurrently.

Generating Primary Keys for EJB (Item 42). There are many ways to create unique primary keys for entity beans. Some of these techniques are wrong, some suffer performance problems, and some tie your implementation to the container. This pitfall discusses the techniques and offers solutions.

The Stateful Stateless Session Bean (Item 43). Developers frequently confuse the stateless nature of the stateless session bean. This pitfall provides an example of where this confusion can cause problems and clarifies just how stateful the stateless bean is.

The Unprepared PreparedStatement (Item 44). The `PreparedStatement` is a powerful capability in server side applications that interface with a database. This pitfall explores a common mistake made in the use of the `PreparedStatement`.

Take a Dip in the Resource Pool (Item 45). Container resource pools are frequently overlooked by developers, who instead rely on several techniques that predate this capability. This pitfall explores techniques currently used and how a container resource pool is the superior choice.

JDO and Data Persistence (Item 46). Data persistence is an important component for all enterprise systems. This pitfall introduces Java Data Objects as a new persistence mechanism that uses both Java and SQL constructs for data transactions.

Where's the WSDL? Pitfalls of Using JAXR with UDDI (Item 47). This item addresses pitfalls with using JAXR with UDDI. The advent of Web services brought us a new API—the Java API for XML Registries. Problems can occur when you are querying UDDI registries with JAXR. This pitfall discusses the problems in depth.

Performance Pitfalls in JAX-RPC Application Clients (Item 48). As JAX-RPC is adopted into the J2EE environment, it is important to know the gory details of how slow or fast your connections could be when using some of JAX-RPC's features on the client side. This pitfall demonstrates an example and shows speed statistics.

Get Your Beans Off My Filesystem! (Item 49) The EJB specification lists certain programming restrictions for EJB developers. This pitfall shows an example of how one of these restrictions is often violated and provides an alternative solution.

When Transactions Go Awry , or Consistent State in Stateful Session EJBs (Item 50). Data transactions are an important part of all distributed system applications. This pitfall shows how the `SessionSynchronization` interface and the Memento pattern can be employed as vehicles in enterprise applications to save and restore data.

Item 37: J2EE Architecture Considerations

I was assigned to be the software lead of a project to Web-enable a legacy XWindows database visualization application. This application consisted of a large database and a data-handling interface that captured data coming from various control systems, correlated it, and loaded it. The data from the control systems was immense, and the interface to them was customized and optimized for its purpose. It worked well, and the goal was merely trying to make this data interface available to a wider audience. How better than to make it a Web application? While database-driven Web applications had been around for a while, including relatively stable versions of servlets and JSPs, commercial J2EE containers were beginning to proliferate. Figure 37.1 shows the architecture of the "as is" system.

Analyzing the requirements, we found there were really two types of users envisioned for this system:

- Analyst personnel, who needed a rich toolset by which they could pour through this voluminous set of data
- Management personnel, who needed an executive-level summary of system performance

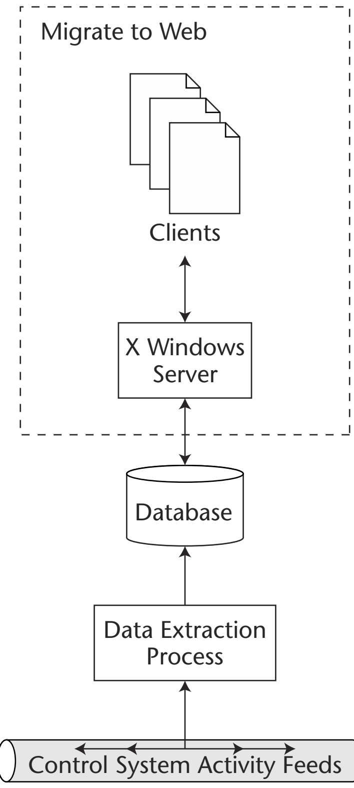


Figure 37.1 The current system.

Therefore, there were truly two different clients that needed to be Web-enabled.

The analyst client needed functionality that included mapping, time lines, and spreadsheets. This was going to be the primary tool used for these personnel to perform their job, and they had expectations that it would perform like the rest of the applications on their desktop machine. They wanted to be able to print reports, save their work, and perform most other tasks that users have come to expect from their PCs.

The manager client was meant to show some commonly generated displays and reports. Essentially, this would be similar to portfolio summary and headlines view. They didn't want anything more involved than essentially to point their Web browser at a Web site and view the latest information.

We proposed to solve the problem of two clients by building a servlet/JSP Web site for the managers with a restricted area that included a Java Web Start deployed application for the analysts (for further explanation of the Java Web Start versus applet decision, see Item 26, "When Applets Go Bad"). There would be a query servlet interface to the database, which would be reused by both clients, using XML as the wire data transmission format. (It should be noted that there were also restrictions on the network enforced by firewalls.) Figure 37.2 outlines the proposed solution.

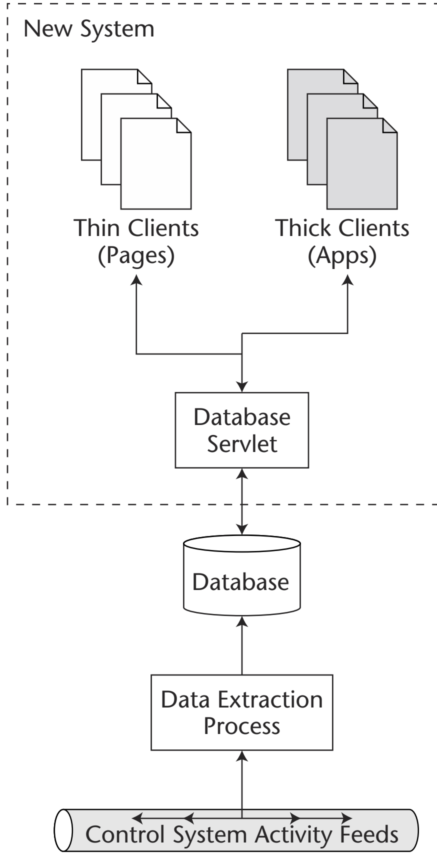


Figure 37.2 The proposed solution.

The customer accepted this as a sound approach and approved the project. Then senior management attended a demonstration on J2EE from an application server vendor, who gave an inspiring presentation. They used the very impressive-looking J2EE Blueprint picture, which is shown in Figure 37.3 (no longer in use by Sun). I pointed out to them that principles were entirely consistent with our design. However, they seemed stuck on the fact that the yellow tier was not present in our design. More specifically, they didn't understand how we could in good conscience not use EJB.

This is a textbook case of what Allen Holub calls "bandwagon-oriented programming" in his interesting article, "Is There Anything to JavaBeans but Gas?"[2] The central point of the article was that problems should not be solved in a "follow the herd" mentality. While I agree with this statement, Mr. Holub forgets that the inverse of that is also problematic. Just because an approach is popular does not mean that it is without merit.

[2] Holub, Allen. "Is There Anything to JavaBeans but Gas?" *C/C++ Users Journal*. Available at http://www.cuj.com/java/forum.htm?topic=java.

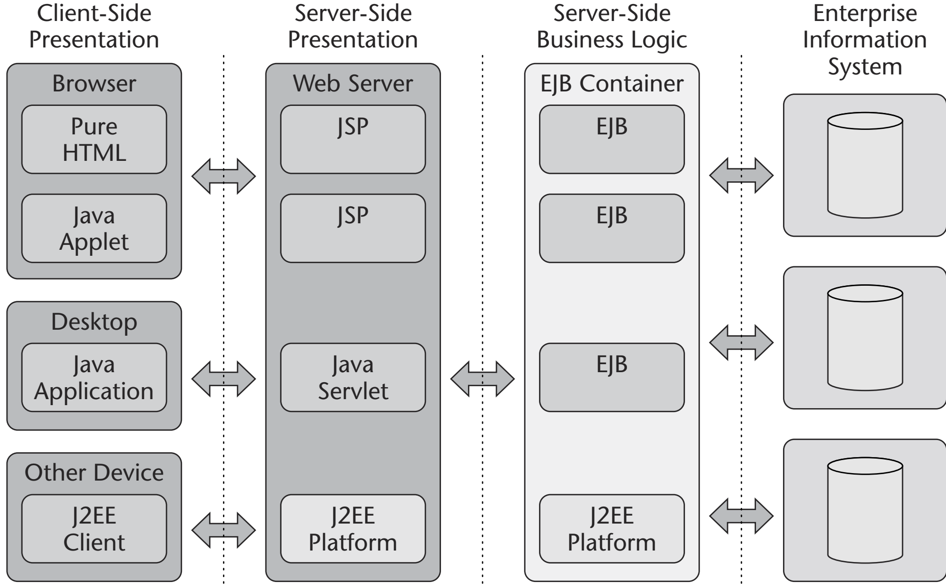


Figure 37.3 J2EE Blueprint diagram (old).

http://java.sun.com/blueprints/

Copyright 2002 Sun Microsystems, Inc. Reprinted with permission. Sun, Sun Microsystems, the Sun Logo, Java, J2EE, JSP, and EJB are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Enterprise JavaBeans, and the J2EE in general, are based on sound engineering principles and fill a major void in the enterprise application market (a platform-neutral standard for enterprise applications and services). So although we've now seen an example where J2EE is "bad," this doesn't necessarily mean that it can't be good.

Let's clarify. Both solutions deal with J2EE. I took issue with the use of EJB, and here is why. EJB is not an object-relational mapping (ORM) tool. The purpose of entity beans is to provide fault-tolerant persistent business objects. Corrupt data, particularly transactions like orders, bills, and payments, cost businesses money—*big* money. Such persistence costs resources and time. This application was a data visualization suite for lots of data, which would amount to having to handle negotiating "locks" from the database vernacular over a tremendous number of records. This would slow performance to incredibly slow levels.

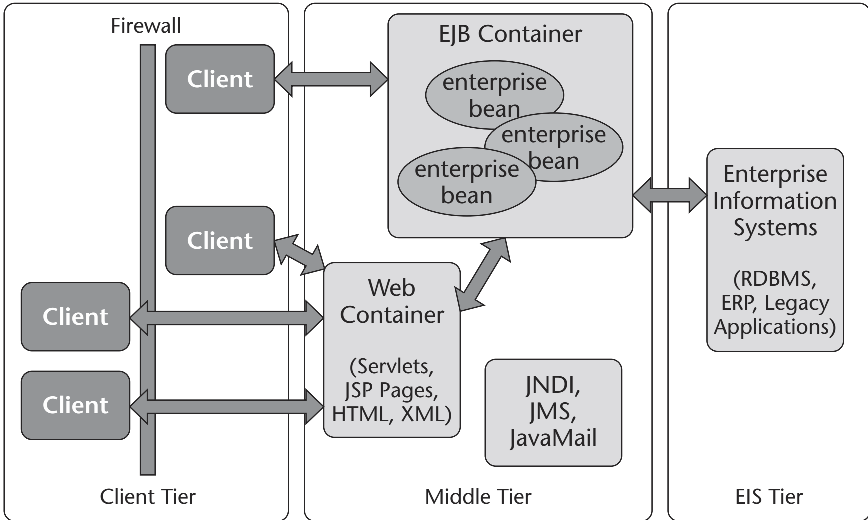


Figure 37.4 J2EE tiers (new).

http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/introduction/introduction3.html#1041147
Copyright 2002 Sun Microsystems, Inc. Reprinted with permission. Sun, Sun Microsystems, the Sun Logo, JSP, JNDI, and JavaMail are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

But the original solution was a J2EE solution, what became known as the "Web tier" by Sun. Figure 37.4 is a diagram representing the new vision that Sun has for different containers for the tier of solution needed.

As to whether the original specification intended to have the flexibility of providing simply Web tier solutions, it was not readily apparent from the initial blueprints. This is why Sun separated them—to clarify that all solutions did not need to center on EJB. This caused a lot of frustration on my team's part (and I know that we were not alone judging from the backlash against EJB I have read).

Figure 37.5 illustrates the various containers that the Java platform envisions. The handheld clients are not shown but are not really relevant relative to the server-side technologies. Note the distinction between the Web container and the EJB container. Also note how the Web container can serve as a façade for the EJB container.

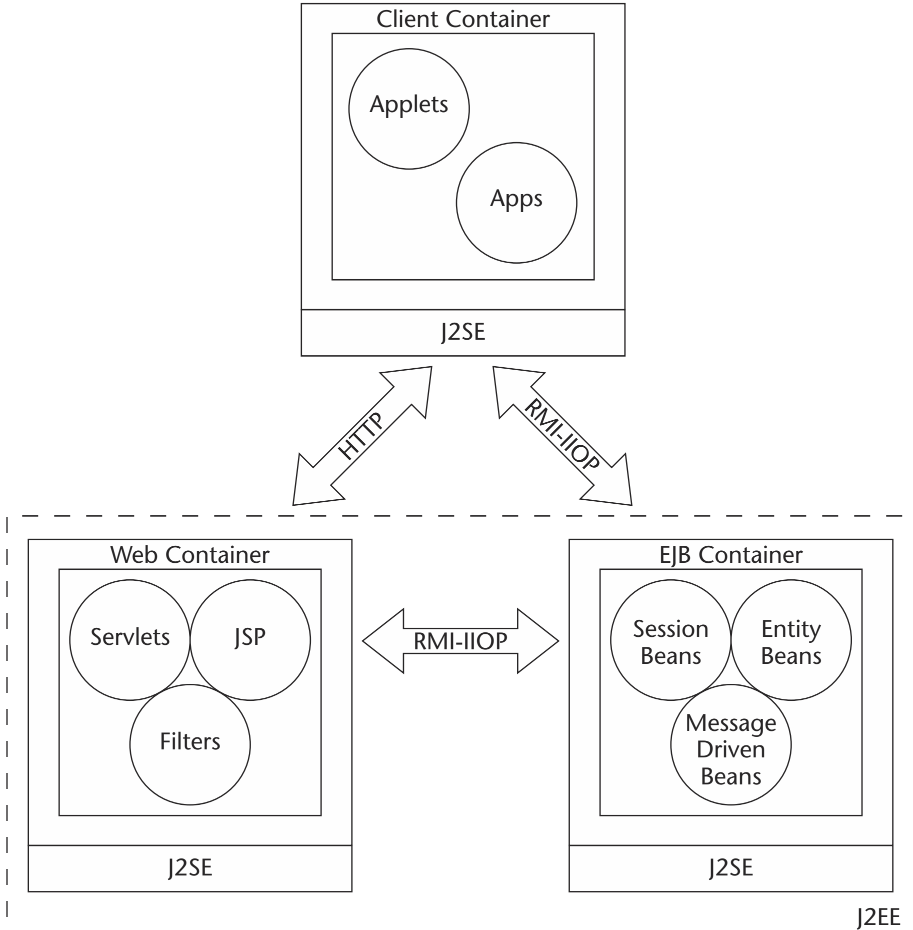


Figure 37.5 Java containers.

So how do we tell if the EJB tier (container) is necessary in our application? This boils down to looking at what EJB provides:

Scalability. Everyone wants scalability, but how scalable? Essentially, we want all of our apps to scale immensely and forever, because something in our system design mentality says, "It can handle anything." However, bigger is not always better, and benchmarks from 2000 show servlets having peak performance on the order of 400 requests per second. Now, you would want to look into your own benchmarking studies, especially concerning the specific vendor solutions you want, but it is not safe to assume that if you want a highly scalable system, you must use EJB.

Transactions. Transactions are a strong capability of EJB containers. This is probably a major selling point when you are dealing with distributed or complex transaction scenarios (multiple systems, phased transactions). However, a great

number of developers use transactions to ensure that a series of SQL statements execute together in the same RDBMS. If this is the level of complexity required, then the JDBC transaction capability is probably more than sufficient.

Security. If you have complex, role-based security or complicated access control lists, this is probably a good place for the EJB container. While the filtering mechanisms in the new servlet specification are very helpful for building good user authentication, it is pretty transparent in EJB.

Reliability. A lot of people define reliability as whether their system ever crashes. In fact, there are a number of semantic debates over reliability versus fault tolerance versus redundancy, but what we are talking about is how well the system responds to something going wrong. As previously mentioned, entity beans were designed around the old data processing nightmare, so if this is a major concern to you, then EJB is probably a good idea.

Componentization. J2EE revolves around the idea of being able to build and reuse components, allowing for the classic build versus buy engineering equation to be introduced into enterprise applications. It is easy to assume that the market for EJB container components is more vibrant than Web container components, but it is quite the opposite. There are tremendous numbers of Web container components available, including some very good open-source ones. A classic example is the Jakarta Taglibs or Struts framework (http://jakarta.apache.org/).

Clustering, Load Balancing, Failover. These are classic examples of the need for the EJB tier. When your applications need to be spread over several containers in a transparent way to support things like clustering (multiple cooperating instances to improve performance), load balancing (assigning requests to different instances to level the load among all instances), and failover (when one instance fails, another is able to pick up on it quickly).

Because it was the first cross-platform enterprise component architecture, J2EE has gained a tremendous amount of attention. While the attention caused a great amount of hype as well, it is wrong to assume J2EE is a technology without merit. J2EE is a collection of technologies, so it is almost always inappropriate to suggest it is a bad solution for any enterprise scenario, despite the care that must be taken in determining whether EJB is a helpful technology to your enterprise solution.

Item 38: Design Strategies for Eliminating Network Bottleneck Pitfalls

In a distributed world where, as Sun Microsystems says, “the network is the computer,” the network is also where a lot of performance bottlenecks can be found. If your software communicates over the network, your design must be flexible enough to tune your performance so that waiting for network traffic does not become the weakest link in your system. Whether you are programming with Sockets, Remote Method Invocation (RMI), CORBA, Enterprise JavaBeans, or using SOAP messaging for Web services using JAXM or JAX-RPC, it is important that you concentrate on your design so that

network latency is not a factor. At the same time, you want to design your system with a flexible architecture.

Although some Java programmers experience problems when designing parallel processing algorithms in network computing, most of the pitfalls we've experienced relate to call granularity in remote messages or method calls. That is, the more messages and calls sent over a network, the more network traffic may impact the performance of your application. In network programming, many developers design their systems where clients send multiple small messages over the network, giving the client more details in the specifics of the transaction but increasing the amount of network connections in an application session. Most of the time, systems that are designed without performance in mind fall into this "granular messaging" trap. In this pitfall, we provide a scenario where we discuss design patterns and strategies for general network programming, and we discuss examples for systems developed with Enterprise JavaBeans.

A Scenario

To demonstrate these challenges, let's look at a scenario for a network application for an automobile service center. An enterprise application for this auto shop supports troubleshooting for its vehicle service. When customers arrive at the auto shop, they tell the employees their customer number and the make and model of their automobile, and they describe their problem. This online application allows the employees and mechanics to look up information about their automobile, find troubleshooting solutions, and provide real-time assistance in fixing the problem. The use cases for the auto shop workflow are shown in Figure 38.1. As you can see from the figure, the auto shop employee enters the car trouble information (probably with the customer's information), gets the owner information from the database, gets the vehicle history from the database, gets possible solutions to the problem, and finally, gets part numbers related to those solutions that are provided.

Looking past the use case and into how data is stored, we see that the application needs to talk to *four databases:* the owner database with information about the owner's automobile and his or her service history, the "car trouble" database with common problems and solutions, the automobile database about makes and models of cars, and the parts database that keeps part numbers for different makes and models of cars.

General Design Considerations

Before we even look at the pitfalls that occur in each API, the following should be made abundantly clear: Although there are four databases full of information, it would be a bad decision to design your client in such a way as to query the databases directly, as shown in Figure 38.2. Such a design would put too much implementation detail in the client, and a very possible bottleneck will be in the management of the JDBC connections, as well as the amount of network calls happening. In Figure 38.2, for each network transaction, there is significant overhead in the establishment and setup of each connection, leading to performance problems. In addition, such a solution is an example of a very brittle software architecture: If the internal structure of one of these databases changes, or if two databases are combined, a client application designed as shown in Figure 38.2 will have to be rewritten. This causes a software maintenance nightmare.

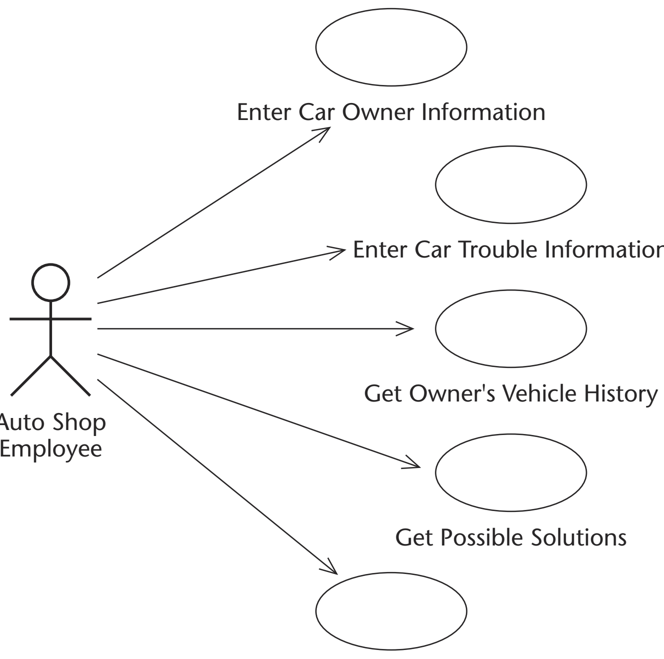


Figure 38.1 Use cases of auto shop scenario.

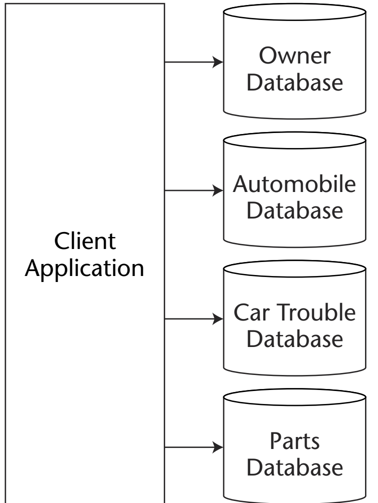


Figure 38.2 Bad client design: calling databases directly.

Figure 38.3 shows an approach that abstracts the implementation details about the four databases from the clients. Instead of the client communicating directly with the four databases, the client communicates with four Java objects. This is good because implementation-specific details in each database are abstracted from the client. The abstraction objects hide these details from the client, but the workflow logic still exists in the client. Added to that is that there is still a potential performance bottleneck: The solution still requires many network calls from the client. For example, if there were eight possible solutions to a car trouble problem, the solution in Figure 38.3 would require 11 network connections (one connection each to the owner, automobile, and car trouble abstractions, and eight connections asking for parts for possible solutions). If this system were live, the performance of this would be unacceptable.

Figure 38.4, on the other hand, seems to eliminate all of the problems that we previously discussed. In Figure 38.4, the client application sends one big message to a gigantic "object that talks to everything." This object talks to all four databases, encapsulates all the business logic, and returns the results back to the client application. On the surface, the application design shown in Figure 38.4 seems perfect—but is it? We have definitely eliminated the bottleneck of too many network connections, and we have eliminated the tight coupling between the client and the databases, but we have now a monstrous object that is tightly coupled to the databases! This design is still inflexible, similar to the design in Figure 38.1, but we have simply added another layer.

Finally, the diagram in Figure 38.5 shows a better solution. We have a client making one network call to an object, which then calls the abstractions to the databases. The database abstractions do the database-specific logic, and the object does the workflow for our use cases. The client application passes in the customer information and symptoms of car trouble to an object, which then delegates that work to abstractions to talk to the database. Finally, when the possible problems and their respective solution part numbers are returned to the object, this information is passed back to the client in one connection. Because only one network connection happens, this creates a positive impact on performance.

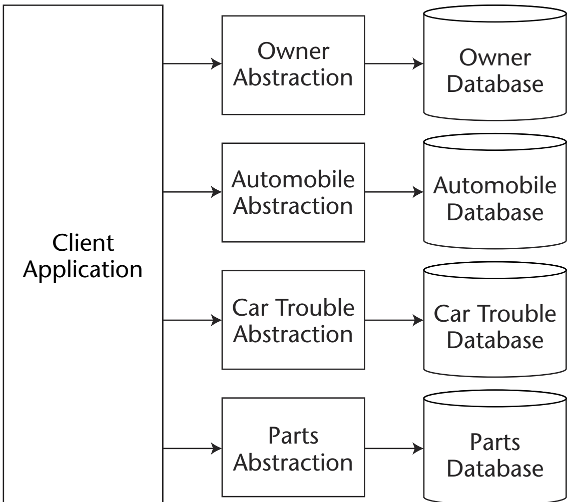


Figure 38.3 Bottleneck: too many network calls from client.

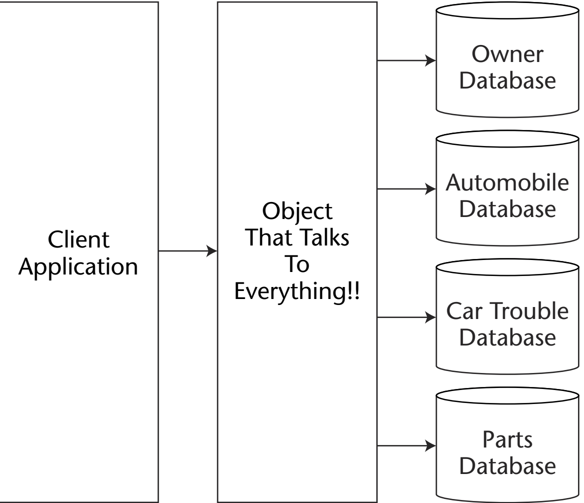


Figure 38.4 The extreme: better performance, but inflexible.

Looking at this scenario, we see that there could be many solutions to designing this system. Before we go into the pitfalls that could occur in each API, let's mention that the key goal for designing this system for this scenario is network performance.

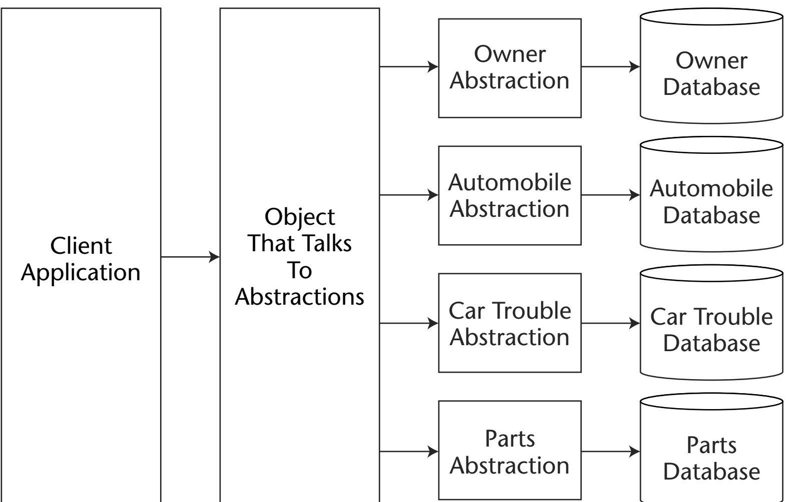


Figure 38.5 A more flexible solution.

EJB Design Considerations

Because we have discussed the potential design pitfalls in this scenario in depth, these pitfalls easily translate to design pitfalls using Enterprise JavaBeans. One problem similar to the poorly thought out designs shown in Figures 38.2 and 38.3 is the design solution where clients directly call entity beans. When this happens, clients become too tightly coupled to the logic of entity beans, putting unneeded logic and workflow in the client, creating multiple network transactions, and creating an architecture that is very unflexible. If the entity bean changes, the client has to change.

Luckily, session beans were made to handle transactional logic between your client and your entity beans. Instead of having your clients call your entity beans directly, have them talk to session beans that handle the transactional logic for your use case. This cuts down on the network traffic and executes the logic of a use case in one network call. This is an EJB design pattern called the Session Façade pattern and is one of the most popular patterns used in J2EE.[3] In our very simple use case for our example, we could have a stateless session bean handle the workflow logic, talking to our session beans, as shown in Figure 38.6.

Of course, you could also use the EJB Command pattern to accomplish the abstraction and to package your network commands into one network call. For more information about these design patterns and for similar patterns in Enterprise JavaBeans, read *EJB Design Patterns* by Floyd Marinescu. It's a great book. (In fact, I had to throw away several pitfall ideas because they were described so well in that book!)

In this pitfall, we discussed network programming design that can relate to any area of Java programming—RMI, Sockets, CORBA, Web Services, or J2EE programming. As you can see by our discussions related to EJB, the basic concepts of design and programming for network performance do not change. With J2EE, we discussed the Session Façade and the Command patterns, which were similar to the solutions we discussed in Figure 38.5. In fact, the J2EE architecture itself helps the flexibility and stability of our architecture.

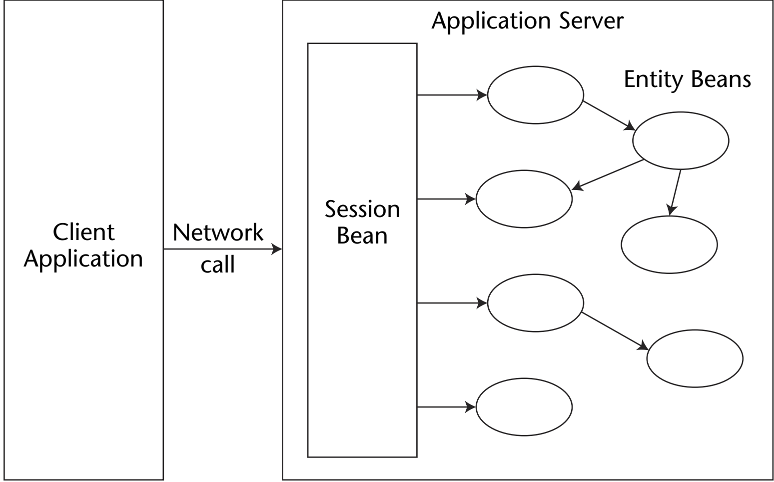


Figure 38.6 Using the session façade pattern.

[3]Marinescu. *EJB Design Patterns.* John Wiley & Sons, 2002.

In conclusion, use the following pieces of advice when designing network applications:

- Abstract details from the client.
- Abstract things in a multi-tiered solution.
- When the network has the potential to be a bottleneck, use the "plan and execute" strategy—it is often better to send one large message with lots of information than to send smaller messages over the network.

Item 39: I'll Take the Local

I have worked with a team that is providing EJBs available via the Web services protocols (SOAP over HTTP). They use a servlet to receive SOAP messages over HTTP. Based on the SOAP message, the appropriate EJB is referenced and called.

For the sake of clarity and simplicity, we will use a more stripped-down version to demonstrate the same concept (see Figure 39.1).

This is the source of the handler, which extends `JAXMServlet` and implements the `ReqRespListener` (a JAXM convention for handling requests that return a synchronous response).

This servlet:

1. Receives the SOAP message.
2. Parses a parameter from the SOAP message.
3. Looks up the `EJBHome` interface object.
4. Narrows the object into a `SOAPHome` object.
5. Calls `create()` on the `SOAPHome` object to create a `SOAPHandler` object.
6. Calls the `convert()` method on the `SOAPHandler` object.
7. Builds a response SOAP message.

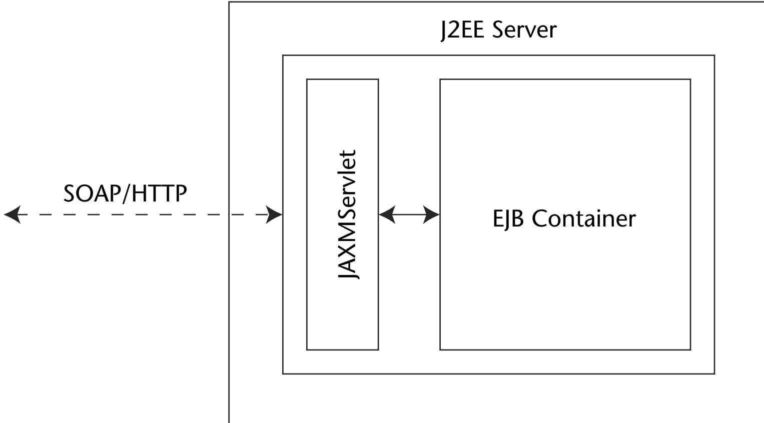


Figure 39.1 Architecture overview.

Listing 39.1 demonstrates this servlet.

```
001: /*
002:  * RemoteEJBServlet.java
003:  *
004:  */
005: 
006: import javax.servlet.*;
007: import javax.servlet.http.*;
008: import javax.xml.messaging.*;
009: import javax.xml.soap.SOAPMessage;
010: import javax.xml.soap.SOAPPart;
011: import javax.xml.soap.SOAPEnvelope;
012: import javax.xml.soap.SOAPHeader;
013: import javax.xml.soap.SOAPElement;
014: import javax.xml.soap.Name;
015: import javax.xml.soap.SOAPException;
016: 
017: import org.xml.sax.*;
018: import org.xml.sax.helpers.*;
019: import javax.xml.parsers.*;
020: 
021: import org.w3c.dom.*;
022: import javax.xml.transform.*;
023: import javax.xml.transform.dom.*;
024: import javax.xml.transform.stream.*;
025: 
026: import java.net.URL;
027: 
028: import javax.naming.*;
029: 
030: import javax.rmi.PortableRemoteObject;
031: 
032: public class RemoteEJBServlet extends JAXMServlet implements ⮐
ReqRespListener {
033: 
034:     /** Initializes the servlet.
035:      */
036:     public void init(ServletConfig config) throws ServletException {
037:         super.init(config);
038: 
039:     }
040: 
041:     /** Destroys the servlet.
042:      */
043:     public void destroy() {
044: 
045:     }
046: 
```

Listing 39.1 RemoteEJBServlet

```
047:
048:     /** Returns a short description of the servlet.
049:      */
050:     public String getServletInfo() {
051:         return "Example of a servlet accessing an EJB through the
remote interface.";
052:     }
053:
054:     /** This method receives the message via SOAP, gets a Local
EJB Home interface,
055:      *  creates the EJB, invokes the appropriate method on the
EJB, and sends
056:      *  the response in a SOAP message.
057:      */
058:     public SOAPMessage onMessage(SOAPMessage message) {
059:
060:         SOAPMessage respMessage = null;
061:
062:         try {
063:             //retrieve the number element from the message
064:             SOAPBody sentSB = message.getSOAPPart()
.getEnvelope().getBody();
065:             Iterator it = sentSB.getChildElements();
066:             SOAPBodyElement sentSBE = (SOAPBodyElement)it.next();
067:             Iterator beIt = sentSBE.getChildElements();
068:             SOAPElement sentSE = (SOAPElement) beIt.next();
069:
070:             //get the text for the number element to put in response
071:             String value = sentSE.getValue();
072:
073:             Context ctx = new InitialContext();
074:
075:             Object obj = ctx.lookup("mySOAPHome");
076:
077:             SOAPHome obj = (SOAPHome) PortableRemoteObject.narrow
(obj, SOAPHome.class);
078:
079:             SOAPHandler soapHandler = soapHm.create();
080:
081:             String responseData = soapHandler.convert(value);
082:
083:              //create the response message
084:             respMessage = fac.createMessage();
085:             SOAPPart sp = respMessage.getSOAPPart();
086:             SOAPEnvelope env = sp.getEnvelope();
087:             SOAPBody sb = env.getBody();
088:             Name newBodyName = env.createName("response",
089:                   "return", "http://www.javapitfalls.org");
090:             SOAPBodyElement response =
```

Listing 39.1 *(continued)*

```
091:                    sb.addBodyElement(newBodyName);
092:
093:                //create the orderID element for confirmation
094:                Name thingName = env.createName("thing");
095:                SOAPElement newThing =
096:                          response.addChildElement(thingName);
097:                newThing.addTextNode(responseData);
098:
099:                respMessage.saveChanges();
100:
101:            } catch (SOAPException soapE) {
102:                // log this problem
103:                soapE.printStackTrace();
104:            }
105:
106:            return respMessage;
107:        }
108:
109: }
110:
```

Listing 39.1 *(continued)*

What is wrong with this approach? Actually, there is nothing explicitly wrong. Countless systems have been built that use this servlet-to-EJB paradigm. However, in this case, both the Web container and the EJB container are running in the same process (in this case the JBoss Application Server). Therefore, this is what happens:

1. The client calls a local stub.
2. That stub performs the necessary translations (marshalling) to make the Java objects suitable for network transport.
3. The skeleton receives the stub's transmission and converts everything back to Java objects.
4. The actual `EJBObject` is called, and it does all of its necessary magic (why you built it).
5. The response is transformed back into a network neutral format.
6. The skeleton returns the response.
7. The stub converts back into Java stuff.

This is typical RMI stuff, right? So what is the big deal? Well, in this case we are operating in the same process. No one would ever make RMI calls to interact with their local objects, so why would you do it in EJB? It used to be that you had to do it; the EJB 1.x specifications gave you no other option. This was based on a basic principle in distributed objects: Design things to be independent of their location.

Furthermore, let's not forget that the Web server and EJB server running in different processes is not that uncommon. As a matter of fact, it is very common to see Web containers running on entirely different machines. However, in this case, we have provided a SOAP interface to our EJBs. That it communicates like a Web server, in other words, over HTTP, is not really the point. We are essentially writing a protocol wrapper to our business objects. It makes little sense to write a protocol wrapper that communicates in an entirely different protocol when it doesn't have to do so.

NOTE All signs point to J2EE 1.4 containing the Web services protocols as an inherent part of J2EE. However, this shows that it is not required in order to use SOAP right now.

So how do you solve this problem? EJB 2.0 offers a solution: local interfaces. When you use a local interface, you can interact directly with the object. This avoids all the network-related issues. This is a remarkable timesaver when you consider just how much is dependent on the size and complexity of the objects.

Listing 39.2 shows the code that changes to use the local interface.

```
1:              //get the text for the number element to put in response
2:              String value = sentSE.getValue();
3:              Context ctx = new InitialContext();
4:              SOAPHome obj = (SOAPHome)ctx.lookup("mySOAPHome");
5:              SOAPHandler soapHandler = soapHm.create();
6:              String responseData = soapHandler.convert(value);
7:
```

Listing 39.2 LocalEJBServlet

That doesn't look much different than our previous example—because it isn't. In this case, I have only removed the narrowing of the Home object. I left the names of the other interfaces the same. Why? Because the real difference is in the back end. Listing 39.3 is an example of a remote home interface.

```
01: package org.javapitfalls;
02:
03: /**
04:  * This is an example of SOAPHome as a remote interface.
07:  */
08:
09: public interface SOAPHome extends javax.ejb.EJBHome
10: {
```

Listing 39.3 SOAPHome (remote) *(continued)*

```
11:
12:   /**
13:    * This method creates the SOAPHandler Object
16:    */
17:
18:   SOAPHandler create() throws java.rmi.RemoteException, ⮐
javax.ejb.CreateException;
19:
20: }
21:
```

Listing 39.3 *(continued)*

Listing 39.4 is the exact same thing as a local interface.

```
01: package org.javapitfalls;
02:
03: /**
04:  * This is an example of SOAPHome as a local interface.
07:  */
08:
09: public interface SOAPHome extends javax.ejb.EJBLocalHome
10: {
12:   /**
13:    * This method creates the SOAPHandler Object
16:    */
17:
18:   SOAPHandler create() throws javax.ejb.CreateException;
20: }
21:
```

Listing 39.4 SOAPHome (local)

Notice that this extends `EJBLocalHome` instead of `EJBHome` as in the remote case. Furthermore, notice how the methods in the local version do not throw `RemoteException`. This is because it is local. It won't have problems with remote access, because it won't be dealing with things that are remote.

But this isn't all. Let's examine the actual business interface and notice its differences. Listing 39.5 is an example of a remote `SOAPHandler` object.

```
01: package org.javapitfalls;
02:
03: /**
04:  * This is an example of SOAPHandler remote interface.
07:  */
08:
09: public interface SOAPHandler extends javax.ejb.EJBObject
10: {
11:
12:   /**
13:    * This method converts the parameter into the response
16:    */
17:
18:   String convert(String param) throws javax.rmi.RemoteException;
20: }
```

Listing 39.5 SOAPHandler (remote)

Listing 39.6 is the exact same thing as a local `SOAPHandler` object.

```
01: package org.javapitfalls;
02:
03: /**
04:  * This is an example of SOAPHandler local interface.
07:  */
08:
09: public interface SOAPHandler extends javax.ejb.EJBLocalObject
10: {
12:   /**
13:    * This method converts the parameter into the response
16:    */
17:
18:   String convert(String param);
20: }
```

Listing 39.6 SOAPHandler (local)

There is something important to understand about this configuration. Since the servlet and EJBs need to be run in the process space, they need to be able to have each other in the same context. To accomplish this, you should place any Web

components that use the local interfaces in the same EAR (Enterprise ARchive) as the classes that implement them.

So, should you always use local interfaces? Not necessarily. There is really one major thing to consider: Are your Web container and EJB container running in the same process, or should they be? This all depends on your individual architecture, but here are some thoughts to consider:

- Do you need separate processes for the Web container and the EJB container? Most people believe that in order to load-balance the Web servers, they must be kept separate from the EJB servers. This is untrue; in fact, it is likely that clustering multiple "combined" application servers will provide just as good, if not better, results.
- Is the EJB container going to be a resource burden on the Web container, since they are running in the same process? If your architecture is light on EJBs and heavy on Web components, perhaps they should be refactored out of the equation. A Web tier solution may be best.

This example is done to illustrate a point about using local interfaces for Enterprise JavaBeans. The example itself is quite helpful, but it is sure to become moot with the release of J2EE 1.4. In this release, JAX-RPC will become another communication protocol for interacting with EJB containers, like RMI is right now. Also, JAXM will become another protocol for interacting with the Java Message Service. These enhancements will make the Web services protocols just another transparent mechanism for communicating with the Java 2 Enterprise Edition.

Item 40: Image Obsession

Java has been enormously successful on the server side, largely tracking with the rise of the Internet and distributed computing. This has given rise to many, many thin clients. One of the biggest issues in thin clients is trying to deliver more expressive user interfaces. This was the idea behind XWindows—a big, powerful machine generates user interface components for thin clients.

A classic example in Web interfaces is to dynamically generate images. We built a graphical reporting tool that generated images using the open-source JFreeChart package (http://www.object-refinery.com/jfreechart/) and the open-source Cewolf custom tag library (http://cewolf.sourceforge.net/).

Our application generates a pie chart showing the percentages of Web browser usage on this particular site. We developed this application on our Windows development boxes. Listing 40.1 is what it looks like both in code (simplified, see the Cewolf docs for more information) and on the screen.

```
01: <%@page import="java.util.*" %>
02: <%@page import="de.laures.cewolf.*" %>
03: <%@page import="com.jrefinery.data.*" %>
04:
```

Listing 40.1 Example code

```
05: <%@ taglib uri="http://cewolf.sourceforge.net/taglib/cewolf.tld" ⮐
prefix="cewolf" %>
06:
07: <%
08: class DatasetProducerImpl implements DatasetProducer {
09:     private Object obj;
10:
11:     public DatasetProducerImpl(Object obj) {
12:         this.obj = obj;
13:     }
14:
15:     public Object produceDataset(HashMap params) throws ⮐
DatasetProduceException {
16:             return obj;
17:     }
18: }
19: %>
20:
21: <html>
22:     <head>
23:         <title>Browser Graph</title>
24:     </head>
25:
26: <body>
27: <%
28: try {
29:
30:     DefaultPieDataset pieData = new DefaultPieDataset();
31:     pieData.setValue("Internet Explorer", new Integer(23405));
32:     pieData.setValue("Netscape", new Integer(12313));
33:     pieData.setValue("Mozilla", new Integer(11202));
34:  pieData.setValue("Opera", new Integer(1333));
35:
36:     DatasetProducer pieProducer = new DatasetProducerImpl(pieData);
37:     pageContext.setAttribute("browserPie", pieProducer, ⮐
PageContext.APPLICATION_SCOPE);
38:
39: %>
40:
41: <center>
42: <h1>Usage of Site by Browser</h1>
43: </center>
44: <p> 
45: <table border="0" cellpadding="2" align="center">
46: <tr>
47:     <td colspan="2">
48: <cewolf:chart
49:     id="browserPieChart"
50:     title="Browser Usage"
```

Listing 40.1 *(continued)*

```
51:     renderer="servlet/chart"
52:     width="400" height="300"
53:     type="pie"
54:     antialias="true">
55:     <cewolf:data>
56:         <cewolf:producer id="browserPie" />
57:     </cewolf:data>
58: </cewolf:chart>
59:     </td>
60: </tr>
61: </table>
62: <p>
63: <%
64:
65: } catch (Exception e) {
66:     e.printStackTrace();
67: }
68:
69: %>
70:
71: </body>
72: </html>
```

Listing 40.1 *(continued)*

Figure 40.1 shows our beautiful graph, easy to build and deploy. At this point, we start thinking that it cannot be this easy, so then we deploy our Web application on our production Solaris machine.

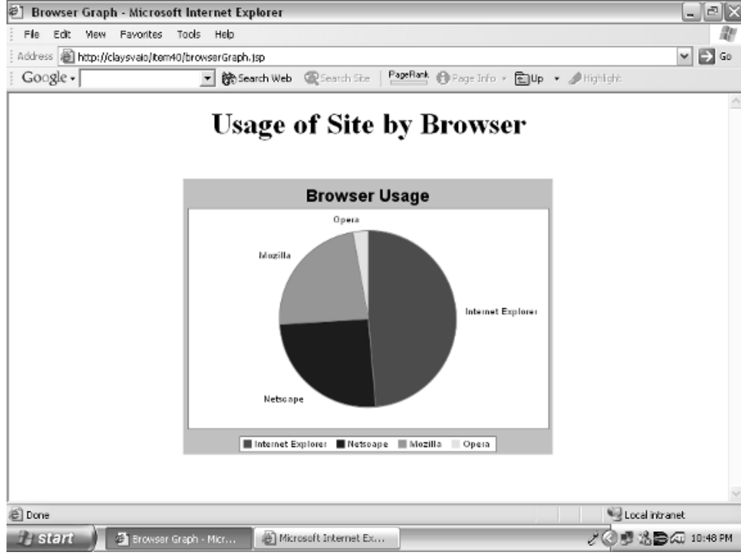


Figure 40.1 Rendered chart example (Windows).

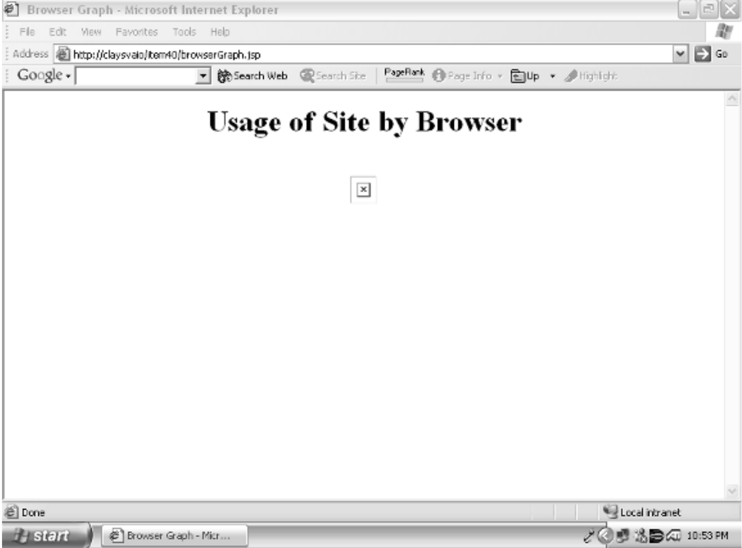


Figure 40.2 Example (Solaris 8).

After bragging about our new graph tool and showing it off on our development boxes, our stomach turned when we saw what showed up on our deployment systems. Figure 40.2 shows the source of our nightmare.

What happened to the image? We ran the exact same code with the same configuration. When we examine the stack trace, we see the problem, shown in Listing 40.2 (abridged).

```
01: 2002-08-15 17:53:43 StandardWrapperValve[chart]: Servlet.service() ⮐
for servlet chart threw exception
02: javax.servlet.ServletException: Servlet execution threw an exception
03:   at org.apache.catalina.core.ApplicationFilterChain ⮐
.internalDoFilter(ApplicationFilterChain.java:269)
04:   at org.apache.catalina.core.ApplicationFilterChain.doFilter ⮐
(ApplicationFilterChain.java:193)
05:   at org.apache.catalina.core.StandardWrapperValve.invoke ⮐
(StandardWrapperValve.java:243)
06:   at org.apache.catalina.core.StandardPipeline.invokeNext ⮐
(StandardPipeline.java:566)
07: {...}
08:   at org.apache.catalina.connector.http.HttpProcessor.run ⮐
(HttpProcessor.java:1107)
09:   at java.lang.Thread.run(Thread.java:536)
10: ----- Root Cause -----
11: java.lang.NoClassDefFoundError: com.jrefinery.chart.AxisConstants
12:   at com.jrefinery.chart.ChartFactory.createPieChart(Unknown Source)
```

Listing 40.2 Stack trace *(continued)*

```
13:   at de.laures.cewolf.DefaultChartRenderer.getChartInstance
(DefaultChartRenderer.java:61)
14:   at de.laures.cewolf.AbstractChartRenderer.renderChart
(AbstractChartRenderer.java:99)
15:   at de.laures.cewolf.CewolfRenderer.renderChart(CewolfRenderer
.java:85)
16:   at de.laures.cewolf.CewolfRenderer.doGet(CewolfRenderer.java:71)
17:   at javax.servlet.http.HttpServlet.service(HttpServlet.java:740)
18:   at javax.servlet.http.HttpServlet.service(HttpServlet.java:853)
19: {...}
20:
```

Listing 40.2 *(continued)*

We get a `NoClassDefFoundError`. We look to see if the class is not within the `CLASSPATH`. Searching through the JFreeChart JAR file, we find the named class, `AxisConstants`, is there. So we check to see if the jar is somehow not getting loaded. Searching through the Tomcat logs, we find the line that shows it mounts the jar just like the rest of the (working) JARs.

`NoClassDefFoundError` clearly doesn't make sense. After all, it means that the JVM cannot find the definition of a particular class, when, in fact, it can. Rather than continue to try to chase this inexplicable phenomenon, we follow the stack trace a bit further. It comes from rendering a pie chart in the `CewolfRenderer`. This causes us to think about it a little closer.

What has changed? The operating system has, but that isn't supposed to matter in Java! As it turns out, we have a small anomaly in how Windows and Unix handle their windowing environments. Windows, as the name implies, is tied to its windowing environment. However, Unix does not require a windowing environment and uses an X server to generate its user interface.

Actually, the heart of the problem has to do with the way the Abstract Windowing Toolkit (AWT) is implemented. The implementation expects to find an X server running when it is created, despite the fact that no user interface components are going to be used. This is relevant to our problem because certain classes out of the Java image classes use the AWT (e.g., `BufferedImage`).

So, there are a couple of solutions to this problem. First, attach a monitor and run the server as a logged-in user. (You should ignore any random images that pop up on the screen as you are working.) However, this is not a very elegant or useful solution.

The real solution depends on which version of the JDK you are using. If you are running on JDK 1.3 or earlier, you should download an X emulator. One you can download is called xvfb, which is at http://www.x.org.

However, JDK 1.4 comes with what is known as the "headless AWT." This allows the J2SE components to be run on the server side without an X server. To use this, you need to specify the following JVM option:

```
-Djava.awt.headless=true
```

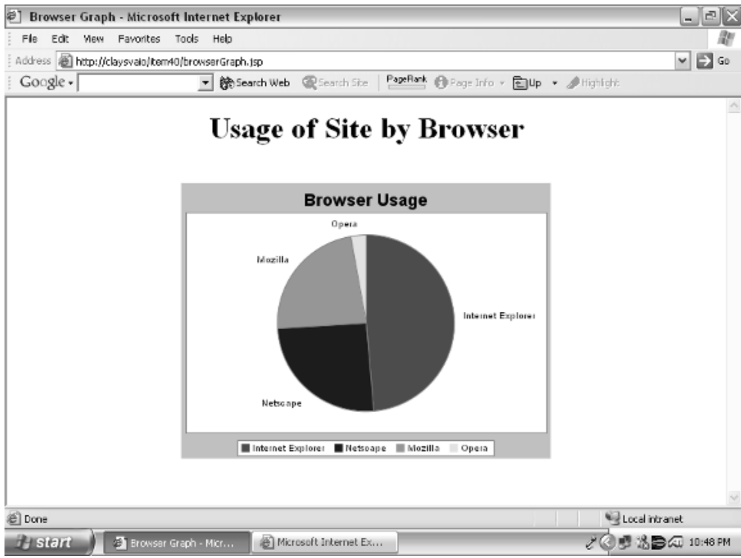


Figure 40.3 Working version.

Sun recommends if you are running a server-side application without any graphical user interface that you specify this option.

Figure 40.3 is the outcome of making this 24-character addition to our JVM invocation.

Item 41: The Problem with Multiple Concurrent Result Sets

Over the years, we've run into many situations where we have needed to write Java applications that have the capability of connecting to "any database." If there is a JDBC driver available for that database, this shouldn't be a problem. In those situations, we typically design our application, write the code, and initially test our application against one type of database. Requirements for the development of many applications beg the need for *query builders*—graphical user interfaces that allow users to click on a selection of tables and columns from the database, allowing them to create advanced queries. Of course, if you have created an application like this, you know that the `java.sql.DatabaseMetaData` class allows you to get column and table information from the database.

Before you build this query builder application, you need to write a simple method that will print this database information to the screen. Later, you will probably change this method to return a structure with all of the tables and columns of the database. Listing 41.1 shows a portion of your code, where you get the table information from the database and print out each table and column of the database.

```
01:  /**
02:   * this goes through the database and lists all the
03:   * tables and columns of the database.
04:   *
05:   * @throws exception
06:   */
07: public void printMetaDataStuff(Connection conn) throws Exception
08: {
09:      String[] tabletypes = {"TABLE"};
10:
11:      if (conn == null)
12:      {
13:           throw new Exception("Could not establish a connection!");
14:      }
15:
16:      DatabaseMetaData dmd  = conn.getMetaData();
17:
18:       ResultSet rs   = dmd.getTables(null,null,null,tabletypes);
19:
20:      if (rs == null)
21:      {
22:           throw new Exception("No metadata!");
23:      }
24:      while (rs.next())
25:       {
26:          String table = rs.getString("TABLE_NAME");
27:
28:          System.out.println("ResultSet 1: Got table " + table);
29:          ResultSet rs2  = dmd.getColumns(null,null,table,null);
30:
31:           if (rs2 == null)
32:           {
33:               throw new Exception ("No Metadata!");
34:          }
35:           while (rs2.next())
36:           {
37:               String col = rs2.getString("COLUMN_NAME");
38:               System.out.println("ResultSet2: Table " + table +
39:                                 " has column " + col);
40:           }
41:       }
42:   }
```

Listing 41.1 printMetaDataStuff( ) from BadResultSet.java

Initially, you test your code with one database, and it works fine. Many developers stop there and continue to build their application. However, just to be on the safe side, you decide to test this class against several databases running on different operating systems, including Microsoft Access, Microsoft SQL Server, MySQL, Sybase, and Oracle. Your class takes a database URL and driver class as an argument, builds a connection, and passes it to your `printMetaDataStuff()` function in Listing 41.1. You write a script that passes the JDBC URL and the driver class to the application, shown in Listing 41.2.

```
echo testing Access

java BadResultSet jdbc:odbc:TestAccessDB sun.jdbc.odbc.JdbcOdbcDriver

echo testing SQL Server
java BadResultSet jdbc:odbc:TestSQLServer sun.jdbc.odbc.JdbcOdbcDriver
testuser testpass

echo testing MySQL

java BadResultSet jdbc:mysql://testmachine:3306/metadotdb
org.gjt.mm.mysql.Driver testuser testpass

echo testing Sybase

java BadResultSet jdbc:sybase:Tds:testmachine:3000/TestDB
com.sybase.jdbc2.jdbc.SybDriver testuser testpass

echo testing ORACLE

java BadResultSet jdbc:oracle:thin:@testmachine:1521:dppo
oracle.jdbc.driver.OracleDriver testuser testpass
```

Listing 41.2 Script for testing

As you can see in Listing 41.2, you are testing this application against different databases using different drivers. The output is shown in Listing 41.3. As you can see, many of these databases (MS Access, MySQL, and Sybase) were able to run `printMetaDataStuff()` from Listing 41.1 with no problems. However, the SQL Server driver throws a `java.lang.SQLException`, saying that the "Connection is busy with the results of another hstmt." The Oracle database throws an ORA-1000 error, saying that you have exceeded your maximum open cursors. What happened?

```
testing Access

ResultSet 1: Got table EQP
ResultSet2: Table EQP has column ACQUIRED_CC
ResultSet2: Table EQP has column ACQUIRED_DATE
ResultSet2: Table EQP has column ACTIVITY
ResultSet 1: Got table EQP_AFLD
ResultSet2: Table EQP_AFLD has column ARRESTING_BARRIER
ResultSet2: Table EQP_AFLD has column ARRESTING_CABLE
ResultSet2: Table EQP_AFLD has column BAYS_QTY
...

testing SQL Server

ResultSet 1: Got table AuthorizedUsers
java.sql.SQLException: [Microsoft][ODBC SQL Server Driver]Connection is
busy with results for another hstmt
     at sun.jdbc.odbc.JdbcOdbc.createSQLException(JdbcOdbc.java:6031)
     at sun.jdbc.odbc.JdbcOdbc.standardError(JdbcOdbc.java:6188)
     at sun.jdbc.odbc.JdbcOdbc.SQLColumns(JdbcOdbc.java:2174)
     at sun.jdbc.odbc.JdbcOdbcDatabaseMetaData.getColumns
(JdbcOdbcDatabaseMetaData.java:2576)
     at BadResultSet.printMetaDataStuff(BadResultSet.java:92)
     at BadResultSet.main(BadResultSet.java:127)

testing MySQL

ResultSet 1: Got table addressbook
ResultSet2: Table addressbook has column ciid
ResultSet2: Table addressbook has column fname
ResultSet2: Table addressbook has column mname
ResultSet2: Table addressbook has column lname
ResultSet2: Table addressbook has column nname
ResultSet 1: Got table channel
ResultSet2: Table channel has column cid
ResultSet2: Table channel has column isa
ResultSet2: Table channel has column name
ResultSet2: Table channel has column description
--

testing Sybase

ResultSet 1: Got table PERSONNEL
ResultSet2: Table PERSONNEL has column PERSONNEL_ID
ResultSet2: Table PERSONNEL has column SOURCE_ID
ResultSet2: Table PERSONNEL has column PERSONNEL_NUMBER
```

Listing 41.3 Output of script

```
ResultSet 1: Got table UNIT_STATUS
ResultSet2: Table UNIT_STATUS has column UNIT_ID
ResultSet2: Table UNIT_STATUS has column STATUS_TIME
ResultSet2: Table UNIT_STATUS has column SOURCE_ID
...

testing ORACLE

ResultSet 1: Got table DR$CLASS
ResultSet2: Table DR$CLASS has column CLA_ID
ResultSet2: Table DR$CLASS has column CLA_NAME
ResultSet2: Table DR$CLASS has column CLA_DESC
ResultSet2: Table DR$CLASS has column CLA_SYSTEM
ResultSet 1: Got table DR$DELETE
ResultSet2: Table DR$DELETE has column DEL_IDX_ID
ResultSet2: Table DR$DELETE has column DEL_DOCID
ResultSet 1: Got table DR$INDEX
ResultSet2: Table DR$INDEX has column IDX_ID
ResultSet2: Table DR$INDEX has column IDX_OWNER#
ResultSet2: Table DR$INDEX has column IDX_NAME
java.sql.SQLException: ORA-01000: maximum open cursors exceeded
     at oracle.jdbc.dbaccess.DBError.throwSqlException(DBError.java:168)
     at oracle.jdbc.ttc7.TTIoer.processError(TTIoer.java:208)
     at oracle.jdbc.ttc7.Oopen.receive(Oopen.java:118)
     at oracle.jdbc.ttc7.TTC7Protocol.open(TTC7Protocol.java:466)
     at oracle.jdbc.driver.OracleStatement.<init>(OracleStatement.java:413)
     at oracle.jdbc.driver.OracleStatement.<init>(OracleStatement.java:432)
     at oracle.jdbc.driver.OraclePreparedStatement
.<init>(OraclePreparedStatement.java:182)
     at oracle.jdbc.driver.OraclePreparedStatement
.<init>(OraclePreparedStatement.java:165)
     at oracle.jdbc.driver.OracleConnection.privatePrepareStatement
(OracleConnection.java:604)
     at oracle.jdbc.driver.OracleConnection.prepareStatement
(OracleConnection.java:485)
     at oracle.jdbc.OracleDatabaseMetaData.getColumns
(OracleDatabaseMetaData.java:2533)
     at BadResultSet.printMetaDataStuff(BadResultSet.java:92)
     at BadResultSet.main(BadResultSet.java:127)
```

Listing 41.3 *(continued)*

Why did some of the databases run through your code without any problems? Why did your queries to the other databases throw exceptions? It revolves around the use of multiple concurrent result sets. Some databases, depending on their configurations,

can handle having more than one result set open at once. From your output in Listing 41.3, you can see that while SQL Server driver threw an exception immediately when the `getColumns()` method was called on the `DatabaseMetaData` object, the Oracle database itself continued to run for a while until its maximum cursors threshold was exceeded.

To write code that is portable across databases, you unfortunately shouldn't make assumptions about what databases can handle having multiple `ResultSet` objects open at a time. Lucky for you, you went through this testing process by writing the script in Listing 41.2 before you wrote your main query builder application. Other developers could have tested it against one database and deployed the application, only to discover that the application couldn't be ported to a few databases.

How can we solve this problem? A simple change in our design will work. Listing 41.4 shows our new `printMetaDataStuff()` method that uses an initial `ResultSet` object to get the table names of the database in line 13 and puts each database table in a `LinkedList` object in lines 21 to 23. Finally, the initial result set is closed in line 26. In lines 30 to 45, we loop through the linked list of table names that we created, opening up one `ResultSet` object at a time and printing out the column names for each table.

```
01:   public void printMetaDataStuff() throws Exception
02:   {
03:       LinkedList ll = new LinkedList();
04:       String[] tabletypes = {"TABLE"};
05:
06:       if (conn == null)
07:       {
08:          throw new Exception("Could not establish a connection!");
09:       }
10:
11:       DatabaseMetaData dmd  = conn.getMetaData();
12:
13:       ResultSet rs   = dmd.getTables(null,null,null,tabletypes);
14:
15:       if (rs == null)
16:       {
17:           throw new Exception("No metadata!");
18:       }
19:       while (rs.next())
20:       {
21:           String table = rs.getString("TABLE_NAME");
22:           ll.add(table);
23:           System.out.println("ResultSet 1: Got table " + table);
24:
25:       }
26:       rs.close();
27:
28:       ListIterator li = ll.listIterator(0);
29:
```

Listing 41.4 A better printMetaDataStuff( ) method

```
30:       while (li.hasNext())
31:       {
32:           String table = li.next().toString();
33:           ResultSet rs2  = dmd.getColumns(null,null,table,null);
34:
35:           if (rs2 == null)
36:           {
37:               throw new Exception ("No Metadata!");
38:           }
39:           while (rs2.next())
40:           {
41:               String col = rs2.getString("COLUMN_NAME");
42:               System.out.println("ResultSet2: Table " + table +
43:                                     " has column " + col);
44:           }
45:           rs2.close();
46:       }
47:   }
```

Listing 41.4 *(continued)*

After you replace your `printMetaDataStuff()` method, you run the same script shown in Listing 41.2. Now your connection to each database prints out all tables and columns without any errors, and you are ready to create your query builder application.

This pitfall showed how important it is to be aware of the problem of using more than one `ResultSet` at once. While many JDBC drivers will support the use of multiple concurrent `ResultSet` objects, many will not. Many databases will exceed the number of maximum open cursors. To achieve maximum portability, rethink your designs so that you will only have one open at a time.

Item 42: Generating Primary Keys for EJB

In a distributed environment, it is imperative to be able to distinguish different instances of objects. With the distributed model of Enterprise JavaBeans, it is important to be able to distinguish instances of entity beans from each other. A common trap that many EJB developers fall into relates to the methods of generating primary keys for entity beans. This item provides an example scenario, gives several examples of how an EJB developer could fall into such a trap, and provides solutions to the problem.

A Simple Scenario

A large video store chain wants to provide national access to accounts throughout the United States. When a customer brings her identity card to the store, the store would like to look up account information and allow the customer to check out videos—no

matter where the customer started the account. It was decided early on that the J2EE model would provide a flexible solution for this video chain. In a high-level design phase, they decided that their JSP would eventually talk to a session bean called `SignOnVideoBean`, which would create an entity bean called `VideoUser`, as shown in Figure 42.1. The question is this: How should the primary key for `VideoUser` be created? We will look at many approaches and discuss pitfalls that lie in each solution.

A "Client Control" Approach

In many small systems, it is commonplace to have the client application (or sometimes the user) generate a unique identifier that represents a new user. An example of this could be an application where a user chooses a user identifier (userid) for access into a portal. In these types of applications, a servlet or JSP from the Web tier initiates a transaction that eventually triggers the creation of an entity bean, which uses the passed-in user identifier as its primary key. This is a valid practice, and depending on the back-end database constraints for primary keys, strings that are guaranteed to be unique (such as email addresses) can be used as the unique identifiers. If there is a collision, the EJB will throw a `javax.ejb.CreateException`. The difficulty is finding a recipe for the creation of these primary keys—throwing and passing exceptions over a network can be bandwidth costly and pretty annoying.

To prevent primary key collisions, they chose a recipe for creating a primary key in such a way that the date, time, store number, clerk ID, and customer name compose the primary key. For example, if "Matt Van Wie" were to start an account at the second Mechanicsville, Virginia, branch of the video store on March 1, 2003, at 5:30 P.M., and the clerk was "Todd Tarkington," the primary key would be "MattVanWie-03-01-2003-530pm-MechanicsvilleVA2-ToddTarkington". This approach is shown in Figure 42.2.

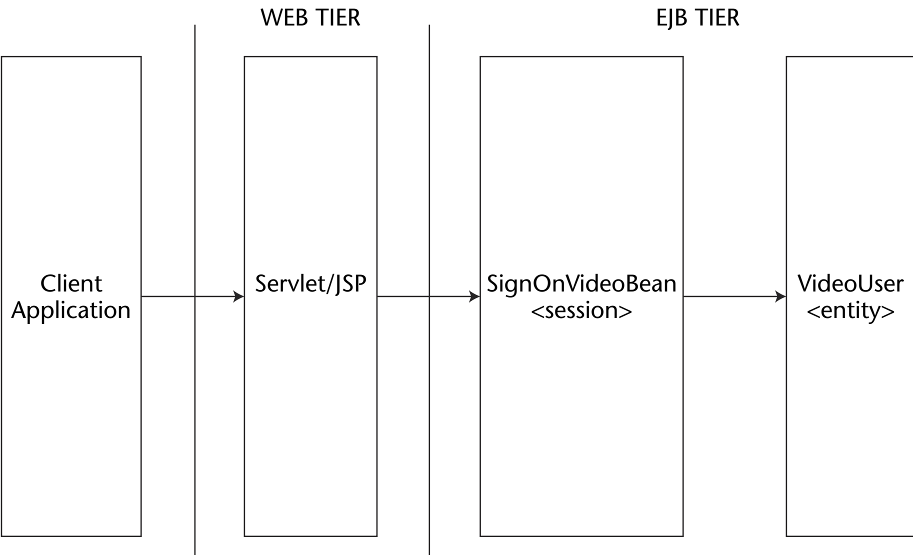


Figure 42.1 Creating video user accounts.

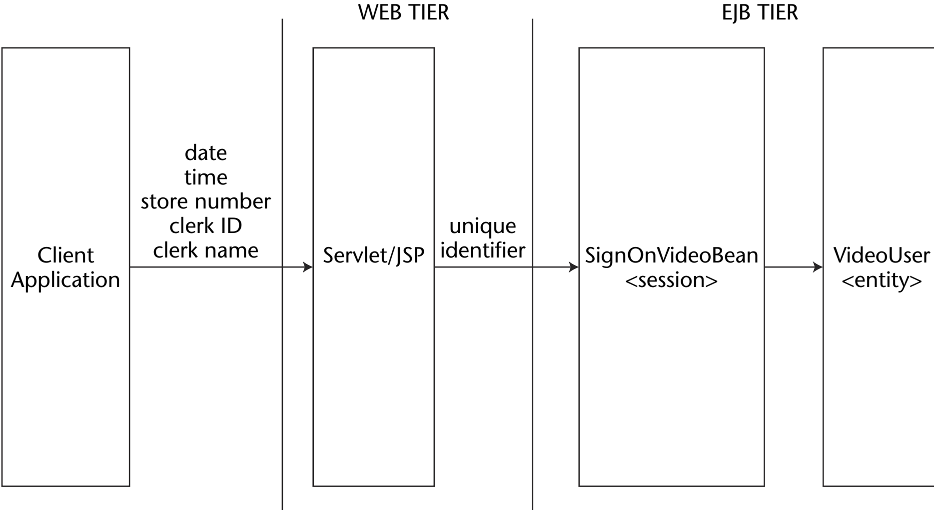


Figure 42.2 Client control solution.

In Figure 42.2, the client application passes the system date, time, store number, clerk identifier, clerk ID, and client name to the servlet/jsp, which creates the unique identifier and creates a reference to `SignOnVideoBean`, which is the session bean that is used to access customer information and create users. `SignOnVideoBean` creates a `VideoUser` entity bean, which is the entity bean that represents the video store customer. This particular approach has many problems that could lead to collisions and chaos:

- For owner names and clerk names that are commonplace, such as "Steve Jones" and "Kevin Smith," there is potential for primary key collisions.
- The solution assumes that a clerk will be only be logged in to the system at one location. If, in the scenario, Todd is the manager and logs in to all of the client workstations at the beginning of the day, this could increase the possibility of name collisions.
- This solution assumes that all client workstations have a way to synchronize date and time. A company policy must be written to specify how time synchronization must work on each store basis. Even if the Web container in this solution generated the system date and time, it is possible that many Web containers at remote locations may be communicating with the EJB tier, and time there must be synchronized.
- The solution assumes that each branch of the video store has a unique name. This requires the assignment of unique store names at a central location, and when new stores are added and removed, just the control of unique store names causes much management overhead.
- The format of the primary key identifier is dependent on the database schema, making one more constraint between the client/Web tier and the EJB tier.

This scenario should demonstrate that allowing the client application to choose the primary key from a certain “recipe” can cause an administrative nightmare. For one, it puts more of a burden on the logic of the client and end users. Second, the burden for creating and enforcing policy that fits into the primary “recipe” will put a huge burden on the manager of the system. Finally, each J2EE system may have a large number of entity beans, and even if you have a great recipe for generating primary keys for one entity bean, you will probably run out of ideas. This is something that may work great for username account generation on a Web site, but you will eventually find that you will need a solution on the server side.

ASSESSMENT: This approach can become unmanageable and burdensome, and it is not recommended.

The Singleton Approach

After the software designers for the video store applications decided that the “Client Control” solution was too unmanageable, they decided that the use of the Singleton design pattern (discussed in Item 15) could be helpful. They decided to create a Singleton that implements a counter, so that it would be the central place to generate primary keys. The team made a simple Singleton called `UIDSingleton`, with a synchronized `getNextKey()` method to generate the key, shown in Figure 42.3.

Satisfied that this would solve the EJB primary key problem, the solution was deployed. Unfortunately, what they didn’t understand is that the use of “pure Singletons” in J2EE is not recommended. A Singleton ensures that there is one instance of a class per virtual machine. With most J2EE servers, there are multiple VMs involved. As a result, the result is primary key chaos, resulting in primary key collisions—and one very flawed system.

ASSESSMENT: Don’t do it.

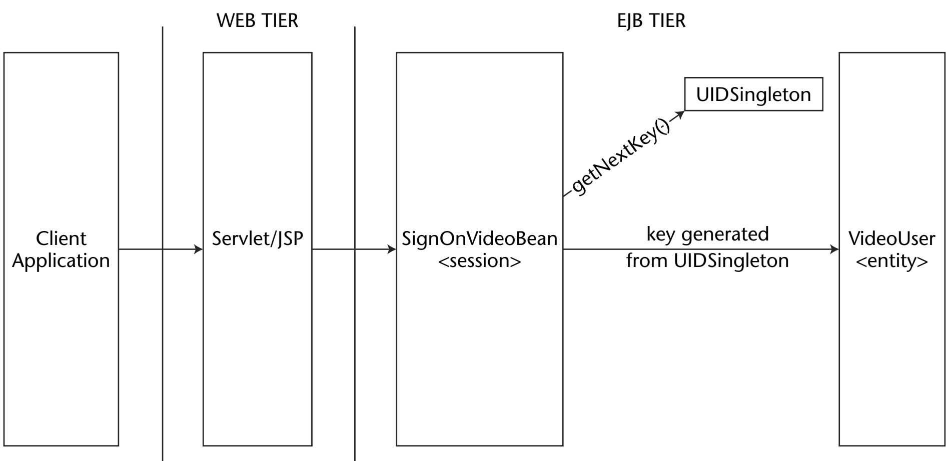


Figure 42.3 Attempting to use a Singleton for PK generation.

The Networked Singleton Approach

Convinced that they were on the right track with the Singleton approach, the team decides to ensure that there is one Singleton for the J2EE system by giving it a network identifier. In this approach, the software developers decided to create a Singleton that is callable via RMI and found by using JNDI. This Singleton implements a synchronized counter and ensures that there are no primary key collisions.

Satisfied that this solution was better than the earlier solution, the software team quickly implemented it. This was easy because each entity bean calls it via RMI and calls `getNextKey()`. In functionality testing, the system worked wonderfully. When it was load-tested with thousands of users, however, the system slowed to a crawl. What happened?

The solution here—and the problem—revolves around the nature of a Singleton. Because a unique network identifier ensures that there is a single object that generates primary keys, this is also a single point of failure. Every entity bean in the system must make a JNDI lookup and then must call the Singleton over the network. While this may work in small-scale solutions, this approach will not scale.

ASSESSMENT: It works, but performance will not scale on large projects.

An Application Server-Specific Approach

At this point, the developers writing the video store application notice that their J2EE application server provides an easy solution to the entity bean/primary key problem. It is a proprietary solution, but it seems to work great. The team develops the entire application, and it works beautifully. However, when it is mandated that they need to change application servers, the team is in trouble. Their entire video store application is now tied to their app server vendor, and it will take a long time to rewrite everything.

In many cases, application servers provide a simple solution to create primary keys for your entity beans. A danger here is nonportability. When you tie your solution to a particular application server with features like these, your solution will undoubtedly work great. Unfortunately, porting your J2EE application to other application servers will be a real headache. Keep in mind that when you go down this route, you may be stuck with the app server you use.

ASSESSMENT: May work great, but at what cost?

Database Autogeneration Approaches

As the designers of the video store solution research new approaches, they decide to use the autogeneration facilities that their chosen database vendor provides. The database vendor supports a counter for the primary key that is incremented every time a new record is created. This is powerful because it provides a centralized mechanism for generating primary keys, guaranteeing uniqueness. In the solution, the `VideoUser` entity bean manages its own persistence. A potential problem here is the nonportability of this solution. If there is a need to switch to another database vendor that does not support autogeneration, the application will be in trouble. Luckily, many database vendors do support autogeneration, so developers may consider this to be a minor concern.

Of the approaches we've discussed so far, this seems to have the most merit. In the past, it has been difficult to make SQL calls to get autogenerated IDs in a nonproprietary way. It was also a challenge to get the autogenerated key right after the insert, because there was not a way to retrieve the generated key from the resulting `ResultSet`. Luckily, with the release of JDK 1.4, an inserted row can return the generated key using the `getGeneratedKeys()` method in `java.sql.Statement`, as can be seen in the code segment below.

```
int primkey = 0;
Statement s = conn.prepareStatement();

s.execute("INSERT INTO VIDEOUSERS" +
   "(name,phone,address,creditcard)" +
   "VALUES ('Matt Van Wie', '555-9509', '91 Habib Ave',
     '208220902033XXXX')",
    Statement.RETURN_GENERATED_KEYS);
ResultSet rs = s.getGeneratedKeys();
if (rs.next())
{
  primkey = rs.getInt(1);
}
```

Many developers using an earlier version of the JDK (or JDBC 2.0 or earlier drivers) that cannot take advantage of this feature use stored procedures in SQL that can be called from Java with the `CallableStatement` interface. The "Stored Procedures for Autogenerated Keys" EJB Design Pattern from Floyd Marinescu's book, *EJB Design Patterns*, provides such a solution that does an insert and returns the autogenerated key.[4]

ASSESSMENT: This is a good solution. With the release of JDK 1.4 and the ability to get generated keys returned from the `java.sql.Statement` *interface, this provides a sound mechanism for solving the EJB primary key problem.*

Other Approaches

It is important to note that *EJB Design Patterns* (Marinescu, 2002), discussed in the previous section, provides a few design patterns to solve this problem. One worth mentioning is a Universally Unique Identifier (UUID) pattern for EJB, which is a database-independent server-side algorithm for generating primary keys. Another pattern, called Sequence Blocks, creates primary keys with fewer database accesses, using a combination of an entity bean that serves as a counter and a stateless session bean that caches many primary keys from the entity bean at a time.

[4] Marinescu, Floyd. *EJB Design Patterns*. John Wiley & Sons, 2002.

ASSESSMENT: These are good design patterns for solving the problem and are described in detail in the book. You can also check out discussions on this topic at http://www.theserverside.com/.

There are many traps that a J2EE architect can fall into when attempting to generate unique identifiers for primary keys. There are also many approaches to solving this problem. This pitfall showed several approaches and provided assessments for each. By avoiding some of the traps discussed in this pitfall item, you will save yourself time and headaches.

Item 43: The Stateful Stateless Session Bean

A developer approached me the other day with an interesting problem. He had an existing API that he wanted to use (in a bean) that builds an index once and then provides search access to it. It seems a pretty simple idea in objected-oriented parlance. There is an object with a member variable, the index, which is built at initialization, and then methods that provide access to it (in this case, search). Figure 43.1 is a simplified example of what the class would look like.

There was concern about how to build this. This developer had some experience with EJB and understood the basic concepts. There are three types of Enterprise Java Beans: session beans, entity beans, and message-driven beans. In addition, there are two flavors of session beans: stateful and stateless. The idea is that stateful session beans maintain their state across invocations, and stateless session beans do not. Entity beans provide real-time persistence of business objects.

The developer had developed EJBs prior to this, but his work had been confined to new development—that is, from whole cloth—and dealt with basic problems. This was the first time where he was trying to use something else in the EJB paradigm.

To examine this dilemma, we reviewed the different options.

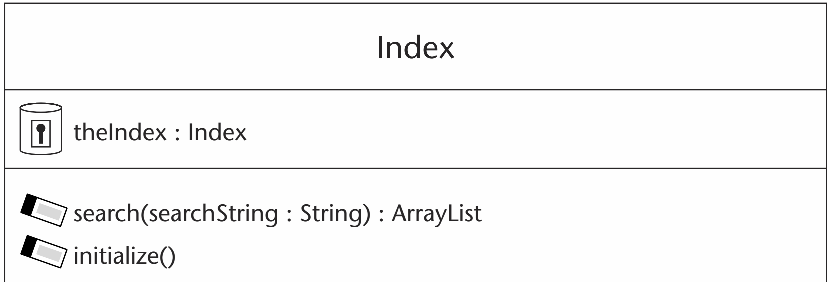


Figure 43.1 The index class diagram.

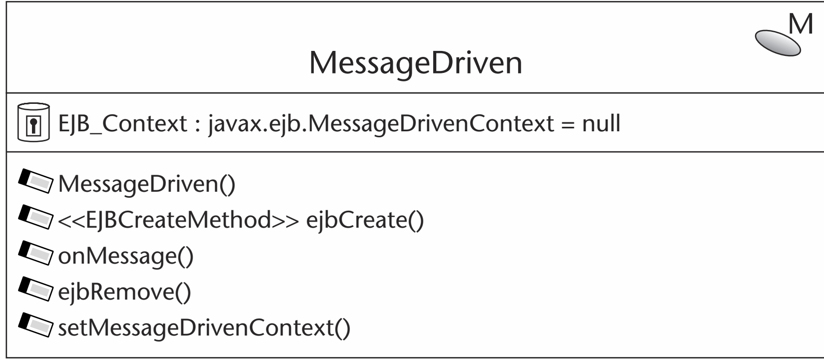


Figure 43.2 The MessageDriven EJB class diagram.

Message-Driven Beans

The container creates message-driven beans to handle the receipt of an asynchronous message. Figure 43.2 shows a message-driven EJB called `MessageDriven`. Notice that there are no interfaces from which client applications can directly invoke the bean. This means that the only access to this bean would be to send a message to the queue that this message-driven bean would handle. When that happens, the container calls the `onMessage()` method. All of the business logic would be contained within that `onMessage()` method.

This is clearly problematic for two reasons. First, reinitializing the index on every invocation is completely inconsistent with our purposes (a shared instance). Second, the means of communicating with our EJB is not supposed to be asynchronous. So, clearly this option was not seriously considered, but for the sake of completeness, we had to look at it.

Entity Bean

Next, we considered using an entity bean. Thinking about it, we realized that the index could be considered a persistent business object. Figure 43.3 shows the Entity EJB class diagram.

Notice that the `EntityHome` interface contains the pertinent `create()` and `findByPrimaryKey()` methods. The remote interface, `Entity`, has the `search()` method within it. Notice that all of the clients would have to try the `findByPrimaryKey()` method to see if the index was created, and if not, create it. Once the reference has been grabbed, the client could call the `search()` method.

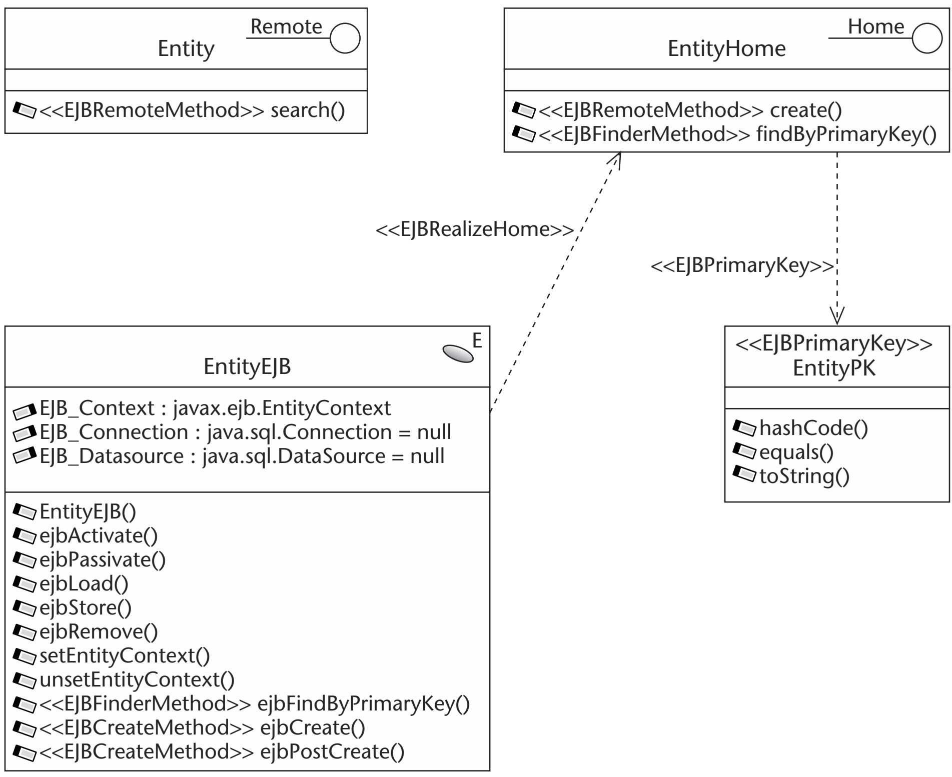


Figure 43.3 The Entity EJB class diagram.

Seems to be a good idea, right? Well, this ignores the fact that entity beans are notoriously slow and it is considered bad practice to interact with them directly, instead of referencing them through session beans (known commonly as the Session Façade pattern). Of course, even if using the Session Façade, this is a classic example of using a sledgehammer to drive a tack—that is, overkill. It could be counterproductive in terms of the additional processing time required to maintain the index in persistence and it could cost more time than re-creating the index on each invocation. After all, the purpose of the entity bean is to ensure that the object survives system failure. Also, if the system fails, then the index can be re-created upon reinitialization of the system. Since we are now persisting this object, we will need to have a methodology to propagate changes to the index so that it does not become a permanently configured instance. This requires additional functionality to handle a case that is not really a requirement for this problem (persistence).

Furthermore, writing the persistence code for something that does not map very congruently with a database (an object) can be a really big pain. Even if the effort is not as bad as it appears, it is still more than is necessary.

Therefore, we saw some real drawbacks in using the entity bean solution. So we decided to review further ideas, which brought us to session beans.

Stateful Session Bean

The stateful session bean provides an interesting alternative. The stateful session bean maintains state across client invocations but does not have the performance overhead of attempting to maintain persistence to survive system crashes. If the system crashes, then the state is lost and must be reestablished.

Figure 43.4 is the class diagram for the stateful session bean, called Stateful. The diagram shows the remote and home interfaces for the stateful session bean. Notice the `create()` method and the requisite index object `theIndex`. Also, the remote interface contains the `search()` method for searching the index.

However, the purpose of the stateful session bean is to maintain the conversation state between the client and server across invocation, essentially tracking the interaction. As such, this means that there is a separate context for each client conversation. This in turn means that every client would create its own index object at the beginning of interaction. Then the client would invoke the search method, and the interaction would end. This is not a shared instance, as the performance hit from initialization comes with each client.

So, this method would not work for us. This left us with one option left for an EJB solution. The stateless session bean seemed like the least likely solution, because the name implies that it lacks state. Or does it?

Stateless Session Bean

By process of elimination, we were left with the stateless session bean approach. The stateless session bean does not track any context between requests. The practical use is to allow the EJB container to provide a highly scalable solution to handling client requests. The container creates a pool of beans to handle the number of requests being received by the container.

The key concept is that the container can assign a stateless session bean to handle any client request, without guaranteeing that the next request by the same client will get the same bean. This allows for much more efficient resource allocation.

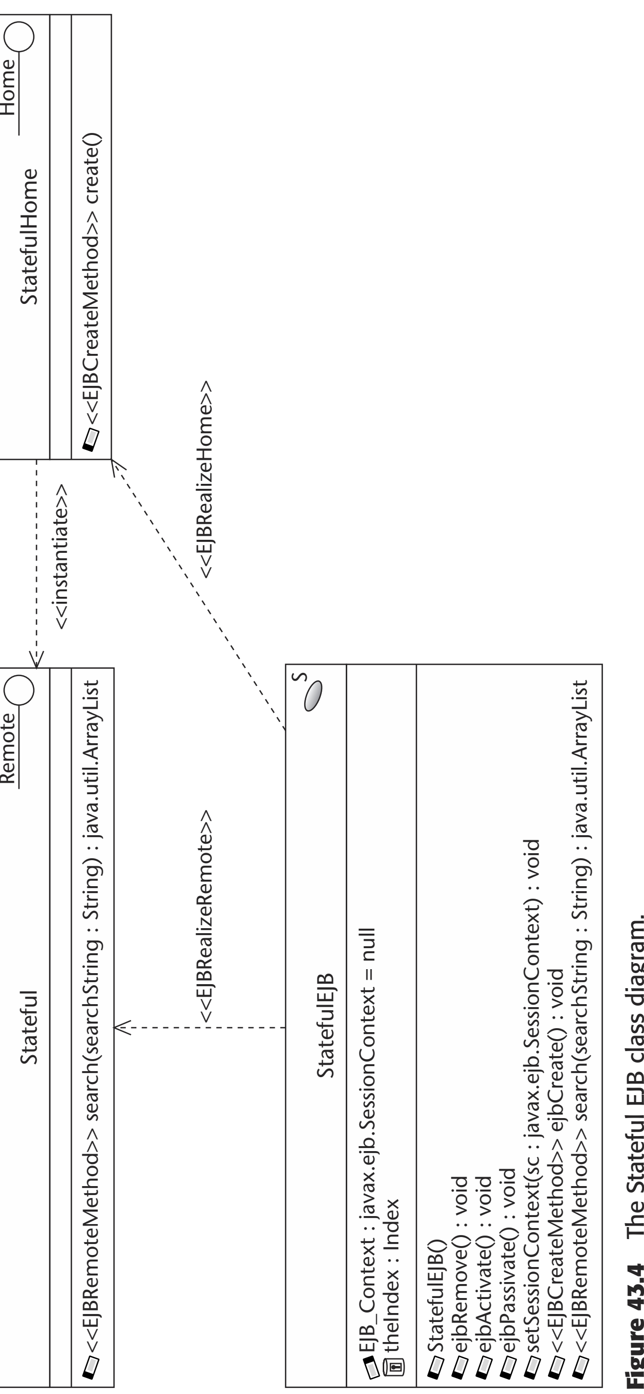


Figure 43.4 The Stateful EJB class diagram.

Frequently overlooked in this scenario is that a stateless session bean can contain member variables that are independent of the client request. Figure 43.5 is a class diagram of a stateless session diagram, called Stateless.

This shows the member variable called `theIndex`, which is populated at initialization when the `create()` method is called on the home interface. After that, requests can be executed by calling the `search()` method on the remote interface.

But the obvious question arises. Isn't every client calling the `create()` method? If so, doesn't this give us the same problem? Actually, this is a common misconception. Because EJB relies on indirection through the container, frequently there is confusion with regard to what a given method on the EJB home and remote interfaces actually does before it gets to the EJB implementation. In fact, the call to `create()` doesn't necessarily mean that the EJB container will create a new stateless session bean. It probably will not, unless this request actually causes the container implementation to determine that it is necessary to create another bean.

This raises an interesting point, though. There is no way to predict or determine how many stateless session beans will be created. Any bean using the member variables needs to take this into account in its design. This becomes even more critical when the container uses clustering.

This means that if we are trying to create a Singleton, or any pattern that demands a resource exists once and only once, this method cannot be used safely.

However, the Singleton pattern and other resource pooling patterns are generally used to ensure the number of instances do not grow out of control. That design goal is achieved implicitly by the design of stateless session beans. They use resource-pooling algorithms to determine the number of session beans that should exist.

Note that this example is not a database connection member variable. In fact, there are certain connection pools that are specified in the EJB 2.0 specification. Actually, it is a called a *resource manager connection factory*, and it entails more than just pooling (deployment, authentication). However, bean developers are required to use the standard ones for JDBC, JMS, JavaMail, and HTTP connections. The container developers are required to provide implementations of these factories.

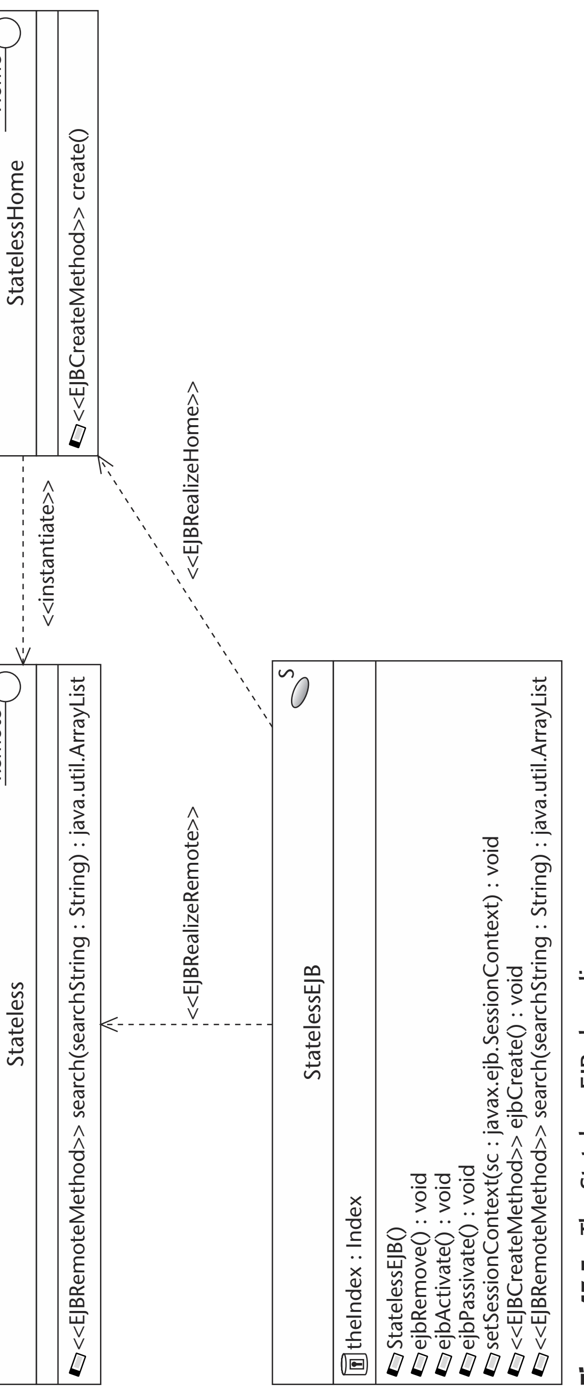


Figure 43.5 The Stateless EJB class diagram.

NOTE These resource manager connection factory objects are part of the movement to the J2EE Connector Architecture, thus fulfilling the third piece of the J2EE paradigm (connectors, along with the current containers and components).

A subtlety in the EJB session bean specification is often overlooked. A large number of developers believe that stateless session beans can maintain no state information at all. This causes confusion on how to maintain instance information that is neutral to any particular request.

This pitfall explored how this misunderstanding can lead the developer into a real bind. By looking at how to solve this particular issue, the developer can gain insight on how each of the beans work behind the scenes and can gain a better appreciation of what is really happening in the container, since it serves as the intermediary in all EJB development.

Item 44: The Unprepared PreparedStatement

JDBC is one of the most popular APIs in the Java platform. Its power and ease of use combined with its easy integration in the Java Web application APIs has caused a proliferation of database-driven Web applications. The most popular of these Web applications is the classic HTML form driving a SQL query to present the data back to the user in HTML. Figure 44.1 is an example of one such Web application. Figure 44.2 shows the response.

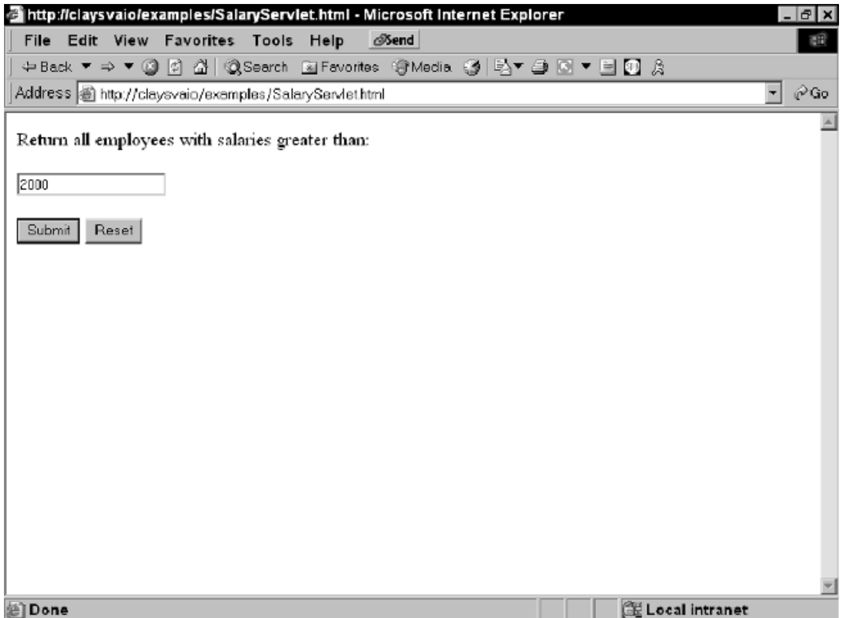


Figure 44.1 Salary HTML form.

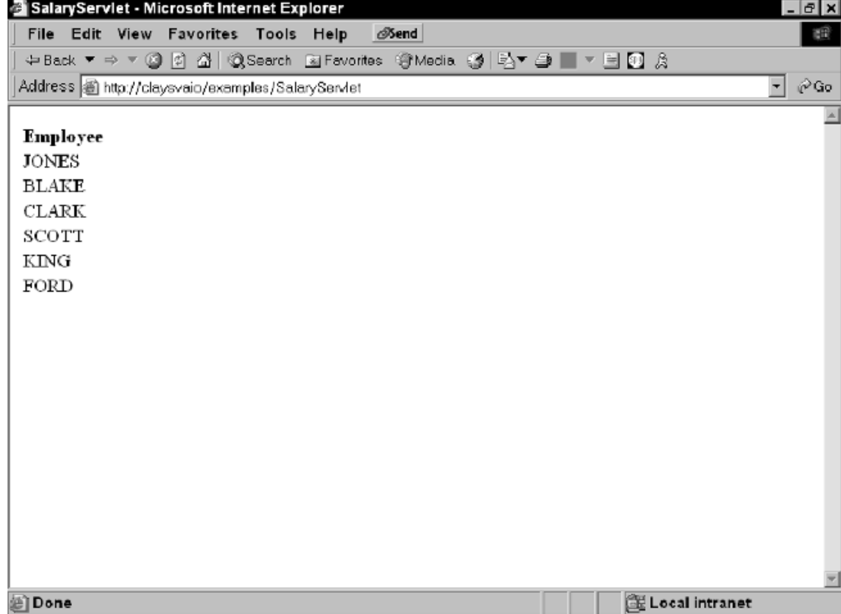


Figure 44.2 HTML form results.

As we look at the code for this example, there are a few things to note. First, it is a very simplified example meant to show all the pieces together in one class. Second, this is an example of a servlet running in the Tomcat servlet container, connecting to an Oracle database. Listing 44.1 shows the code of the class.

```
01: import javax.servlet.*;
02: import javax.servlet.http.*;
03: import java.io.*;
04: import java.util.*;
05: import java.sql.*;
06:
07: public class SalaryServlet extends HttpServlet {
08:
09: Connection connection;
11:
12:   private static final String CONTENT_TYPE = "text/html";
13:
14:   public void init(ServletConfig config) throws ServletException {
15:     super.init(config);
16:
17:     // Database config information
```

Listing 44.1 SalaryServlet *(continued)*

```
18:     String driver = "oracle.jdbc.driver.OracleDriver";
19:     String url = "jdbc:oracle:thin:@joemama:1521:ORACLE";
20:     String username = "scott";
21:     String password = "tiger";
22:
23:     // Establish connection to database
24:     try {
25:       Class.forName(driver);
26:       connection =
27:         DriverManager.getConnection(url, username, password);
28:
29:     } catch(ClassNotFoundException cnfe) {
30:       System.err.println("Error loading driver: " + cnfe);
31:
32:     } catch(SQLException sqle) {
33:           sqle.printStackTrace();
34:
35:     }
36:   }
37:   /**Process the HTTP Post request*/
38:   public void doPost(HttpServletRequest request, ⮐
HttpServletResponse response) throws ServletException, IOException {
39:     response.setContentType(CONTENT_TYPE);
40:
41:     String salary = request.getParameter("salary");
42:
43:     String queryFormat =
44:       "SELECT ename FROM emp WHERE sal > ";
45:
46:     try {
47:
48:     Statement statement = connection.createStatement();
49:
50:     ResultSet results =
51:         statement.executeQuery(queryFormat + salary);
52:
53:     PrintWriter out = response.getWriter();
54:     out.println("<html>");
55:     out.println("<head><title>SalaryServlet</title></head>");
56:     out.println("<body>");
57:     out.println("<table>");
58:     out.println("<tr>");
59:     out.println("<td><b>Employee</b></td></tr>");
60:
61:     while (results.next()) {
62:       out.println("<tr>");
63:       out.println("<td>");
64:       out.println(results.getString(1));
```

Listing 44.1 *(continued)*

```
65:       out.println("</td>");
66:       out.println("</tr>");
67:     }
68:
69:     out.println("</table>");
70:     out.println("</body></html>");
72:     } catch ( SQLException sqle ) {
74:       sqle.printStackTrace();
76:     }
77:
78:   }
79:   /**Clean up resources*/
80:   public void destroy() {
82:     connection.close();
84:   }
85: }
86:
```

Listing 44.1 *(continued)*

Notice that the `init()` method handles all of the database connection issues. Keep in mind that if this were going to be a production servlet, the connection handling and pooling would be delegated to another class (possibly another servlet). The `doPost()` method handles the building and execution of a JDBC statement from the connection, as well as the parsing of the resulting result set into HTML. Once again, best practice would cause this handling and presentation to be delegated elsewhere (usually a JSP), rather than building all of these `out.println()` statements to render the HTML.

This method is the most simplistic, but it clearly lacks efficiency. Essentially, a new `String` is constructed each time the user makes a request with the parameter appended to that `String`. What about precompiling the query so that only the parameter must be passed into the query? JDBC provides a method to do this, called the `PreparedStatement`. Listing 44.2 gives an example of the `doPost()` method rewritten to use a `PreparedStatement`.

```
01: public void doPost
02: (HttpServletRequest request, HttpServletResponse response)
03: throws ServletException, IOException {
04:     response.setContentType(CONTENT_TYPE);
05:
06:     String salary = request.getParameter("salary");
07:
08:     try {
09:
10:     PreparedStatement statement
```

Listing 44.2 Using prepared statements *(continued)*

```
11:       = connection.prepareStatement("SELECT ename FROM emp WHERE
sal > ?");
12:
13:     statement.setFloat(1, Float.parseFloat(salary));
14:
15:     ResultSet results =
16:         statement.executeQuery();
17:
// ... remaining code Identical to Listing 44.1 ...
```

Listing 44.2 *(continued)*

This example shows precompiling the statement and accepting parameters instead of any string. This is helpful for controlling data types and escape sequences. For example, calling `setDate()` or `setString()` handles the appropriate escape characters for the specific database being used, and it also prevents mismatching types—like specifying a `String` for a long field.

However, while this is a common practice among many developers, the `PreparedStatement` is not reused. This means that the statement is recompiled each time; thus, no advantage is seen in the precompiling, aside from data typing. So, the `PreparedStatement` should persist between invocations to receive the most effective and efficient results. Listing 44.3 is an example of the servlet rewritten to reuse the `PreparedStatement`.

```
01: import javax.servlet.*;
02: import javax.servlet.http.*;
03: import java.io.*;
04: import java.util.*;
05: import java.sql.*;
06:
07: public class SalaryServlet extends HttpServlet {
08:
09:   Connection connection; // Shouldn't be member variable under
normal conditions.
10:   PreparedStatement statement;
11:
12:   private static final String CONTENT_TYPE = "text/html";
13:
14:   public void init(ServletConfig config) throws ServletException {
15:     super.init(config);
16:
17:     // Database config information
18:     String driver = "oracle.jdbc.driver.OracleDriver";
19:     String url = "jdbc:oracle:thin:@joemama:1521:ORACLE";
```

Listing 44.3 Best use of prepared statements

```
20:     String username = "scott";
21:     String password = "tiger";
22:
23:     // Establish connection to database
24:     try {
25:       Class.forName(driver);
26:       connection =
27:         DriverManager.getConnection(url, username, password);
28:
29:         PreparedStatement statement
30:             = connection.prepareStatement("SELECT ename FROM emp
WHERE sal > ?");
31:
32:     } catch(ClassNotFoundException cnfe) {
33:       System.err.println("Error loading driver: " + cnfe);
34:
35:     } catch(SQLException sqle) {
36:           sqle.printStackTrace();
37:
38:     }
39:   }
40:   /**Process the HTTP Post request*/
41:   public void doPost(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {
42:     response.setContentType(CONTENT_TYPE);
43:
44:     String salary = request.getParameter("salary");
45:
46:     try {
47:
48:     statement.setFloat(1, Float.parseFloat(salary));
49:
50:     ResultSet results =
51:         statement.executeQuery();
52:
// ... remaining code Identical to listing 44.1 ...
85: }
86:
```

Listing 44.3 *(continued)*

Notice in this example that the parameter is set and the `PreparedStatement` is executed. This allows the reuse of the `PreparedStatement` between requests to the servlet. This improves the responsiveness, since it does not require the repetitive and time-consuming step of recompiling the statement on every invocation of the servlet.

It seems logical that there would be a performance improvement in reusing rather than recompiling statements. Testing was conducted to determine how much better the reuse option was. The three previously mentioned scenarios were tested. All used

an Oracle database over a local area network. A test class was built to connect to the database and sequentially call a method representing each scenario, using the same connection for each. The same query was executed 100 times using each methodology. Here is the output of the tester class:

```
Executing prepared statement 100 times took 0.791 seconds.
Executing raw query 100 times took 1.472 seconds.
Executing repeated prepared statements 100 times took 1.622 seconds.
```

This class was executed several times with very similar results. Notice two things: First, the reused prepared statement was approximately 40 to 50 percent faster than the raw query statement, and second, the repeated compiling of prepared statements actually gave about a 10 percent performance hit.

This example shows a classic pitfall in the use of JDBC. When building desktop applications or when in the development phase of Web and enterprise applications, many developers fail to notice the performance hit that comes from not effectively using the `PreparedStatement`. A 50 percent hit in response time can get very significant in Web applications when loads can go up tremendously without much notice.

Item 45: Take a Dip in the Resource Pool

Most of the time, an "interactive Web site" is really a controlled chaos of Web applications that gradually evolve into a common purpose. Most organizations come up with an idea that "we can put these applications on the Web," and they usually do. Depending on the skill level of the development organization, various technologies are used to "Web-enable" their application.

This phenomenon is not limited to the spread of Web application platforms available—PHP, Cold Fusion, ASP, and so on. This occurs within Java also. The power of servlets, the ease of JSP, and the reuse of JSP tag libraries have caused a lot of developers to use a wide variety of techniques to Web-enable applications in Java.

Here are a few common ways that Web developers handle database connections. The first is to create a simple class to abstract the DB from the rest of the code, as Listing 45.1 demonstrates.

```
01: package org.javapitfalls;
02:
03: import java.io.*;
04: import java.net.*;
05: import java.sql.*;
06: import java.util.*;
07:
08: public class DbAbstract implements java.io.Serializable {
09:
10:   private Connection connection;
```

Listing 45.1 DBAbstract.java

```
11:   private Statement statement;
12:
13:   public DbAbstract ()
14:       throws ClassNotFoundException, SQLException
15:   {
16:
17:     String driverName = "";
18:     String className = "";
19:     String user = "";
20:     String pass = "";
21:
22:  Properties resource = new Properties();
23:  // Obtain a resource bundle that will allow use to get the
24:  // appropriate configurations for the server.
25:  try
26:  {
27:    URL url = 
this.getClass().getClassLoader().getResource("db.properties");
28:      resource.load( new FileInputStream(url.getFile()) );
29:      // Get properties
30:      driverName = resource.getProperty("driverName");
31:      className = resource.getProperty("className");
32:      user = resource.getProperty("user");
33:      pass = resource.getProperty("pass");
34:      System.out.println("Using parameters from the db.properties 
file for Database.");
35:     } catch (Exception e) {
36:         System.out.println("ERROR: Couldn't load db.properties." 
+ e.toString());
37:     }
38:
39:     Class.forName(className);
40:     connection = DriverManager.getConnection(driverName,user, pass);
41:
42:     connection.setAutoCommit(false);
43:
44:     statement = connection.createStatement();
45:   }
46:
47:   public void executeUpdate(String sqlCommand)
48:     throws SQLException
49:   {
50:     statement.executeUpdate(sqlCommand);
51:   }
52:
53:   public ResultSet executeQuery(String sqlCommand)
54:     throws SQLException
55:   {
56:     return statement.executeQuery(sqlCommand);
```

Listing 45.1 *(continued)*

```
57:    }
58:
59:    public void commit() throws SQLException
60:    {
61:      connection.commit();
62:    }
63:
64:    public void rollback() throws SQLException
65:    {
66:      connection.rollback();
67:    }
68:
69:    protected void finalize() throws SQLException
70:    {
71:      statement.close();
72:      connection.close();
73:    }
74: }
75:
```

Listing 45.1 *(continued)*

This example works very well. Taking it one step further, we find the development of servlets to pool database connections or broker database connections. I have seen a number of innovative approaches to creating these types of servlets, including HTTP interfaces to get state information, manipulate the interface, or control lifecycle events. Because I don't want to take anything away from these approaches or distract from my main point, I've just provided a simple example of one of these in Listing 45.2.

```
01: package org.java.pitfalls;
02:
03: import javax.servlet.*;
04: import javax.servlet.http.*;
05: import java.io.*;
06: import java.util.*;
07: import java.sql.*;
08: import java.text.*;
09: import com.javaexchange.dbConnectionBroker.*;
10:
11: public class CPoolServlet extends HttpServlet
12:                                implements SingleThreadModel {
13:
14:
15:     ServletContext      context = null;
16:     DbConnectionBroker  broker = null;
```

Listing 45.2 CPoolServlet.java

```
17:     String              logDir = "";
18:
19:     public void init(ServletConfig config) {
20:
21:        try {
22:
23:          startBroker();
24:
25:           // Set it up so that other servlets can get a handle to ⮐
it.
26:           context = config.getServletContext();
27:           context.setAttribute("pooler", broker);
28:
29:       }catch (IOException e) {
30:           System.out.println(
31:             "CPoolServlet: Problems with database connection: " + ⮐
e);
32:
33:        }catch (Exception e) {
34:           System.out.println(
35:             "CPoolServlet: General pooler error: " + e);
36:        }
37:     }
38:
39:     public void doPost(HttpServletRequest request,             ⮐
HttpServletResponse response)
40:               throws IOException {
41:        doGet(request, response);
42:     }
43:
44:     public void doGet(HttpServletRequest request,              ⮐
HttpServletResponse response)
45:               throws IOException {
46:
47:         response.setContentType("text/html");
48:          ServletOutputStream  out = response.getOutputStream();
49:
50:         StringBuffer html = new StringBuffer();
51:
52:         html.append("");
53:         html.append("<html><title>CPoolServlet</title><body>");
54:
55:         html.append("<center><h2>CPoolServlet                   ⮐
Information</h2></center>");
56:         html.append("<p><center>Connection Pool is available     ⮐
under the pooler attribute.</center>");
57:
58:         html.append("</body></html>");
59:
```

Listing 45.2 *(continued)*

```
60:         out.println(html.toString());
61:     }
62:
63:     private void startBroker() throws Exception {
64:        try {
65:           broker = new DbConnectionBroker(
66:                    "com.sybase.jdbc2.jdbc.SybDriver",
67:                    "jdbc:sybase:Tds:localhost:2638",
68:                    "Test",
69:                    "Test",
70:                    5,
71:                    25,
72:                    "CPool.log",
73:                    1.0);
74:        }catch (Exception e) {
75:           throw e;
76:        }
77:     }
78:
79:     public void destroy() {
80:        broker.destroy();
81:     }
82: }
83:
```

Listing 45.2 *(continued)*

In this example, the servlet creates a connection broker object, in this case, a popular one from the Java Exchange. It places the attribute within the servlet context so that other components (servlets, JSPs) within the container can use it. This shows that there is an ability to create objects and make them available to the rest of the Web container. Why can't the container create these objects and make them available through the same JNDI lookups? It can. In fact, J2EE 1.3 containers are required to support resource pools, which are called *resource managers*.

Another interesting thing about the specification is that application developers are required to get their database connections through the javax.sql.DataSource connection factory. That means that to be compliant with the specification, developers ought to use this facility.

As an example, let's look at how you would do this using Tomcat. In this case, we are using an open-source connection pool factory provided by the Jakarta Commons project called DBCP (http://jakarta.apache.org/commons/dbcp.html). Assuming you have placed your JDBC driver and the connection pool JARs (if needed) in the right place (%CATALINA_HOME%/common/lib for Tomcat), the first step is to place a resource reference in your Web application's web.xml file—as demonstrated in Listing 45.3.

```
01: <resource-ref>
03:   <description>
05:     Reference to a factory for java.sql.Connection instances.
The actual implementation is configured in the container.
07:   </description>
09:   <res-ref-name>
11:     jdbc/myDB
13:   </res-ref-name>
15:   <res-type>
17:     javax.sql.DataSource
19:   </res-type>
21:   <res-auth>
23:     Container
25:   </res-auth>
27: </resource-ref>
```

Listing 45.3 Excerpt from web.xml

As noted in the above declaration, you then need to configure the container to handle the actual implementation of the reference. In Tomcat, this is done in the server.xml file. Here is an example of what should be added to the file in order to create the resource pool.

Note this fits into the `Context` element, or in the `DefaultContext` element. More information on these can be read from the sample server.xml and other documents that the Jakarta project maintains on Tomcat. Listing 45.4 shows how to modify the server.xml in Tomcat to create the Connection Pool resource.

```
01: <Context ...>
03:   ...
04:
05: <Resource name="jdbc/myDB" auth="Container"
06:     type="javax.sql.DataSource"
07:  description="The PKO Portal DB"/>
08:  <ResourceParams name="jdbc/pkoDB">
09:        <parameter>
10:         <name>factory</name>
11: <value>org.apache.commons.dbcp.BasicDataSourceFactory</value>
12:        </parameter>
13:        <parameter>
14:         <name>driverClassName</name>
15:          <value>com.sybase.jdbc2.jdbc.SybDriver</value>
16:          </parameter>
17:         <parameter>
```

Listing 45.4 Excerpt from server.xml *(continued)*

```
18:            <name>maxActive</name>
19:            <value>3</value>
20:          </parameter>
21:          <parameter>
22:             <name>maxIdle</name>
23:           <value>3</value>
24:          </parameter>
25:          <parameter>
26:           <name>username</name>
27:           <value>user</value>
28:          </parameter>
29:          <parameter>
30:           <name>password</name>
31:           <value>pwd</value>
32:          </parameter>
33:          <parameter>
34:          <name>maxWait</name>
35:           <value>-1</value>
36:          </parameter>
37:          <parameter>
38:          <name>url</name>
39:           <value>jdbc:sybase:Tds:localhost:2638</value>
40:          </parameter>
41:    </ResourceParams>
```

Listing 45.4 *(continued)*

So, now the JDBC DataSource resource pool is available to the applications. The last step would be to actually use it. Listing 45.5 is an example of how to use the resource pool.

```
01: Context initCtx = new InitialContext();
03: Context envCtx = (Context) initCtx.lookup("java:comp/env");
05: DataSource ds = (DataSource) envCtx.lookup("jdbc/myDB");
07: Connection conn = ds.getConnection();
09: <<< Execute specific JDBC code here >>>
11: conn.close();
```

Listing 45.5 Using the resource pool

There are many different ways to create and manage resource pools. Like a lot of things in Java, better solutions evolve with time, and developers often are not aware of improvements offered in the platform. This is a common example where developers who have created their own mechanisms may not be aware of the improvements.

If you are working within a container, you should use the resource pooling capability provided by the container. Often it is invalid not to use the container resource pool, which can cause component developers to have problems with the portability of their code. Also, this allows for one convenient place for the container configuration to provide this service to all of its applications. It can be configured or changed without having to change every application that runs in the container.

The J2EE Connector Architecture is increasing in popularity for integrating back-end systems. This architecture provides for resource managers, which provide similar access to heterogeneous systems like Enteprise Resource Planning (ERP) and Human Resources Management Systems (HRMS) systems.

Item 46: JDO and Data Persistence

The foundation of any enterprise application is the data layer. This also determines how data will be persisted on your system. To me, deciding on a proper data persistence mechanism is like passing a very large kidney stone. Perhaps that is an overstatement, but it's an arduous process that requires an exhaustive understanding of your system and the technologies that will allow you to apply the best solution to your data storage/retrieval activities.

A proper persistence strategy typically consists of storing information in byte streams through serialization, hierarchically arranged data in XML files, Enterprise Java Beans, relational/object databases, and now Java Data Objects (JDO). The problem is that once you make a commitment to one technology, there's always a lot of second-guessing because of unforeseeable implementation hardships and performance issues.

Relational databases and EJBs can be effective tools for performing data persistence activities, but Java Data Objects offer a powerful new alternative that should be considered in allowing object-based access to your back-end data stores. JDO's strength lies in its ability to allow developers to use both Java and SQL constructs to manage data manipulation and access, and in its binary portability across data stores.

There are always design and implementation trade-offs in software development, especially when it comes to data persistence and retrieval. Although serialization is fairly simple to implement and does a decent job of preserving relationships between Java objects, it is nontransactional and can be a performance hog. JDBC is great for handling transactions, but it can require extensive field management and data mapping activities and can be tedious when you are implementing several SQL language constructs. JDO is fairly new and is not fully developed, but it promises to make developers more productive by allowing business logic manipulation through the use of Java interfaces and classes without having to explicitly understand SQL operations.

To gain a better appreciation of the strengths of JDO and how its transactional capabilities should be considered along with traditional methodologies, you need to understand data persistence in its most common form: serialization. Serialization is a simple technique used to transfer and store objects in byte streams and, most recently, in XML, but this is performed in a nontransactional manner. The code below demonstrates a simple manner that data can be serialized to the system disk space and read back. The `ObjectInputStream` class on line 122 deserializes data that was created previously using `ObjectOutputStream`.

```
001: package serialization;
002:
003: import java.util.*;
004: import java.text.*;
005: import java.util.Date;
006: import java.io.*;
007: import java.net.*;
008: import java.beans.*;
009: import java.util.logging.*;
010:
011: public class serializeBean {
012:
013:  private String firstname;
014:  private String lastname;
015:  private String department[];
016:  private String ssn;
017:  private String comments;
018:  private String filename;
019:  private String filename2;
020:
021:  //private String firstname2;
022:  //private String lastname2;
023:  //private String ssn2;
024:  //private String comments2;
025:
026:  private Hashtable errors;
027:
028:  private static Logger logger = 
Logger.getLogger(serializeBean.class.getName());
029:
030:  public boolean validate() {
031:     boolean errorsFound=false;
032:
033:     logger.info("[validate:] lastname, firstname, ssn, comments= 
" + lastname + ", " + firstname + ", " + ssn + ", " + comments);
034:
035:     if (firstname.equals("")) {
036:        errors.put("firstname","Please enter a valid Firstname");
037:        errorsFound=true;
038:     }
039:     if (lastname.equals("")) {
040:        errors.put("lastname","Please enter a valid Lastname");
041:          errorsFound=true;
042:     }
043:     if (ssn.equals("")) {
044:        errors.put("ssn","Please enter a valid Social Security #");
045:        errorsFound=true;
046:     }
047:     if (comments.equals("")) {
048:        errors.put("comments","Please enter a valid comment");
```

Listing 46.1 serializeBean.java

```
049:       errorsFound=true;
050:    }
051:
052:    return errorsFound;
053: }
054:
055: public String getErrorMsg(String s) {
056:    String errorMsg =(String)errors.get(s.trim());
057:   return (errorMsg == null) ? "":errorMsg;
058: }
059:
060: public serializeBean() {
061:
062:    try
063:    {
064:       URL url = null;
065:       Properties props = new Properties();
066:
067:       url =
this.getClass().getClassLoader().getResource
("serialization.properties");
068:    props.load( new FileInputStream(url.getFile()) );
069:     // Get properties
070:       filename = props.getProperty("filename");
071:       setFilename(filename);
072:       logger.info("[serializeBean:(constructor)] Filename is:
" + getFilename());
073:
074:       BufferedInputStream in = new BufferedInputStream(new
FileInputStream(filename));
075:       XMLDecoder decoder = new XMLDecoder(in);
076:       Object f = decoder.readObject(); // firstName
077:       Object l = decoder.readObject(); // lastName
078:       Object s = decoder.readObject(); // SSN
079:       Object c = decoder.readObject(); // comments
080:
081:       lastname = (String)l;
082:       firstname = (String)f;
083:       ssn = (String)s;
084:       comments = (String)c;
085:
086:       setLastname(lastname);
087:       setFirstname(firstname);
088:       setSsn(ssn);
089:       setComments(comments);
090:
091:       decoder.close();
092:
093:    }
094:    catch(Exception e) {
```

Listing 46.1 *(continued)*

```
095:
096:        System.out.println("ERROR: " + e.toString());
097:        System.out.println("Using default values.");
098:
099:        // use default values
100:        firstname="";
101:        lastname="";
102:        // department[];
103:        ssn="";
104:        comments="";
105:
106: }
```

Listing 46.1 *(continued)*

The serialization process described in lines 109 to 210 can be a problem with different versions of class libraries or Java Runtime Environments. The latest Merlin release has addressed this with the introduction of XML serialization. Since the data is saved in XML files, it is possible to serialize data, manually modify the serialized file by hand, and have it deserialized without losing a step. Previously, this would not have been possible because the data was saved in binary byte streams.

```
108:    /*
109:    try
110:    {
111:      URL url = null;
112:      Properties props = new Properties();
113:      url = this.getClass().getClassLoader().
114:            getResource("serialization.properties");
115:      props.load( new FileInputStream(url.getFile()) );
116:      // Get properties
117:      filename2 = props.getProperty("filename2");
118:      setFilename2(filename2);
119:      logger.info("Filename2 is: " + getFilename2());
120:      FileInputStream inputFile =
121:         new FileInputStream(getFilename2());
122:      ObjectInputStream inputStream =
123:         new ObjectInputStream(inputFile);
124:      // Read data
125:      lastname2 = (String)inputStream.readObject();
126:      firstname2 = (String)inputStream.readObject();
127:      ssn2 = (String)inputStream.readObject();
128:      comments2 = (String)inputStream.readObject();
129:
130:     inputStream.close();
131:
132:    }
133:    catch(Exception e) {
134:
135:     System.out.println("ERROR: " + e.toString());
```

Listing 46.1 *(continued)*

```
136:    System.out.println("Using default values.");
137:
138:    // use default values
139:    firstname2="";
140:    lastname2="";
141:    ssn2="";
142:    comments2="";
143:
144:   }
145:    */
146:    errors = new Hashtable();
147:    logger.info("lastname, firstname, ssn, comments= " +
148:      lastname + ", " + firstname + ", " + ssn + ", " + comments);
149: // logger.info("lastname2, firstname2, ssn2, comments2= " +
150: // lastname2 + ", " + firstname2 + ", " + ssn2 + ", " + comments2);
151:  }
152:  public void write()
153:  {
154:   try
155:   {
156:     logger.info("Filename is: " + getFilename());
157:
158:     System.out.println("Firstname= " + getFirstname());
159:     System.out.println("Lastname= " + getLastname());
160:     System.out.println("SSN= " + getSsn());
161:     System.out.println("Comments= " + getComments());
162:
```

Listing 46.1 *(continued)*

The code below demonstrates a new serialization mechanism that saves the states of internal methods in an XML format.

```
163:     // try XMLEncode
164:     BufferedOutputStream out =
165:      new BufferedOutputStream(new FileOutputStream(getFilename()));
166:     XMLEncoder encoder = new XMLEncoder(out);
167:     encoder.writeObject(getFirstname());
168:     encoder.writeObject(getLastname());
169:     encoder.writeObject(getSsn());
170:     encoder.writeObject(getComments());
171:     encoder.close();
172:
173:   }
174:   catch(IOException ioe) {
175:     System.err.println("IOERROR: " + ioe);
176:   }
177:   catch(Exception e) {
178:     System.err.println("ERROR: " + e);
179:   }
180:
```

Listing 46.1 *(continued)*

```
181:   /*
182:   try
183:   {
184:     logger.info("Filename is: " + getFilename2());
185:     // avoid XMLEncode
186:     FileOutputStream outputFile2 =
187:       new FileOutputStream(getFilename2());
188:     ObjectOutputStream outputStream2 =
189:       new ObjectOutputStream(outputFile2);
190:     outputStream2.writeObject(getFirstname());
191:     outputStream2.writeObject(getLastname());
192:     outputStream2.writeObject(getSsn());
193:     outputStream2.writeObject(getComments());
194:
195:     // close stream
196:     outputStream2.flush();
197:     outputStream2.close();
198:
199:   }
200:   catch(IOException ioe) {
201:     System.err.println("IOERROR: " + ioe);
202:   }
203:   catch(Exception e) {
204:     System.err.println("ERROR: " + e);
205:   }
206:   */
207:
208:    logger.info("[serializeBean:write] writing data.");
209:  }
```

Listing 46.1 *(continued)*

If you've used Remote Method Invocation in your software implementations, you've come across serialization. With RMI implementations, objects are compressed into byte streams when they are marshaled across a network. Performance becomes an issue during the deserialization of objects because it involves the conversion of byte array elements into object and data types. This process involves the use of reflection to discover properties of an object.

Some shortcomings with serialization are that once an object is written to, additional writes are ignored because the object reference is maintained, unless the `reset()` method is invoked. Certainly, performance issues have always been associated with serialization because of disk read (reflection)/write operations, but depending on your situation, serialization can be a perfectly appropriate persistence solution in your enterprise deployments. However, if your application requires concurrency in database access and queries, then Java Data Objects can be a solution for you.

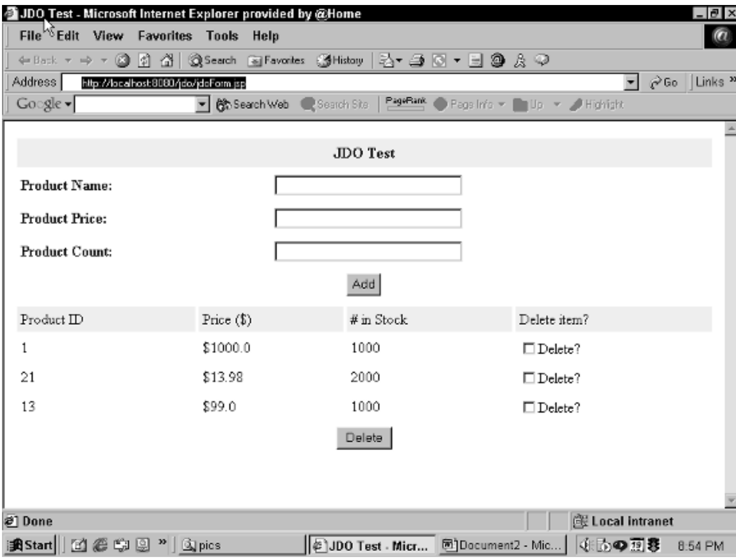


Figure 46.1 JDO Test form.

Now that you have a better understanding of serialization, I think you'll have a better appreciation of JDO and the great promise it offers in abstracting away the inane details of the SQL language implementation in Java applications.

A simple inventory program, shown in Listing 46.2, was crafted with a recent open-source JDO offering from the Apache Software Foundation (ASF) called ObjectRelational Bridge (OJB). OJB is a fairly ambitious project by the good folks at the ASF, and although their JDO implementation is not fully compliant with the JDO Draft (JSR-012), ASF promises to move in that direction. OJB uses a `PersistentBroker` API as its persistence kernel and has its JDO implementation built on top of it and XML data mappings that bind to your database elements.

What makes OJB JDO worthy of examination is its ability for transparent persistence of database elements in the form of Java objects. JDO abstracts away the complexities of SQL transactions and allows developers to query instances of data stores with very fine granularity and without having to understand the intimate details about the database structure itself.

With OJB JDO, caches can be transactional, which means that each cache area can maintain a different transactional view of the data store instances. This is an important concept that separates it from the serialization process described above.

Mentioned earlier, the following code is a simple inventory Web application that inserts, deletes, and displays inventory items using the OJB JDO libraries. The `jdoForm` is a simple JavaServer Page that accepts user requests and interfaces with a back-end MySQL database.

```
001:
002: <%@page import="org.apache.ojb.tutorial4.*" %>
003: <%@page import="org.apache.ojb.jdo.PersistenceManagerImpl" %>
004: <%@page import=
005:  "org.apache.ojb.jdo.PersistenceManagerFactoryImpl" %>
006: <%@page import="javax.jdo.PersistenceManager" %>
007: <%@page import="javax.jdo.Transaction" %>
008:
009: <%@page import="org.odmg.*" %>
010:
011: <%@page import="javax.jdo.Query" %>
012: <%@page import="java.util.Collection" %>
013:
014: <%@page import="javax.jdo.PersistenceManager" %>
015: <%@page import="javax.jdo.PersistenceManagerFactory" %>
016: <%@page import="java.io.BufferedReader" %>
017: <%@page import="java.io.InputStreamReader" %>
018: <%@page import="java.util.Vector" %>
019: <jsp:useBean id="pm"
020:  class="org.javapitfalls.item46.jdoPersistenceMgr"
021:  scope="application"/>
022: <jsp:useBean id="validate"
023:  class="org.javapitfalls.item46.jdoBean" scope="request">
024: <jsp:setProperty name="validate" property="*"/>
025: </jsp:useBean>
026: <head>
027:  <title>JDO Test</title>
028: </head>
029:
030: <%
031: String addItem=request.getParameter("Add");
032: String deleteItem=request.getParameter("Delete");
033:
034: if (addItem != null)
035: {
036:   String pName = request.getParameter("productName");
037:   String pPrice = request.getParameter("productPrice");
038:   String pCount = request.getParameter("productCount");
039:
040:   if (!validate.validate()) {
041:
042:     Product newProduct = new Product();
043:
044:     newProduct.setName(pName);
045:     newProduct.setPrice(Double.parseDouble(pPrice));
046:     newProduct.setStock(Integer.parseInt(pCount));
047:
```

Listing 46.2 jdoForm.jsp

The `Transaction` interface on line 49 provides the operations necessary to perform database transactions—specifically, accessing, creating, and modifying persistent objects and their fields. Prior to performing any database operations, a thread must explicitly create a transaction object or associate itself with an existing transaction object, and that transaction must be open (through a call to `begin`). All operations that follow the thread are performed under the transaction of the thread.

The operational states of a transaction are either open or closed. An open transaction means that the application has called `begin()` but has not invoked `commit()` or `abort()`. Once an application calls the `commit()` or `abort()` methods, the transaction is considered closed. The `isOpen()` method can be used to determine the state of a transaction. As objects are accessed by an application, read locks are implicitly used to allow proper access to data and write locks are used when objects are modified to ensure that only one transaction is modifying data at one time.

The `makePersistent(...)` operation on line 52 is executed on an open transaction so that the transient object can be made available for the commit method to make it durable. If the transaction is made available and the operation aborts, then the `makePersistent` operation is nullified and the target object is marked as transient.

```
048:      // now perform persistence operations
049:      Transaction tx = null;
050:      // open transaction
051:      PersistenceManager pmInstance = pm.getManagerInstance();
052:      pmInstance.makePersistent(newProduct);
053:      tx = pmInstance.currentTransaction();
054:      tx.begin();
055:
056:      // commit transaction
057:      tx.commit();
058:
059:      System.out.println("successful submission...");
060:
061:   } else {
062:
063:      System.out.println("Validation error.");
064:
065:   }
066: }
067: else if (deleteItem != null)
068: {
069:    String[] deleteArray = request.getParameterValues("deleteItems");
070:
071:    if ((deleteArray != null) && (deleteArray.length > 0))
072:    {
073:      for (int i=0; i < deleteArray.length; i++)
074:      {
075:        System.out.println("deleting: " + deleteArray[i]);
076:
```

Listing 46.2 *(continued)*

```
077:       Product test = new Product();
078:       test.setId(Integer.parseInt(deleteArray[i]));
079:
080:       PersistenceManager pmInstance = pm.getManagerInstance();
081:
082:       try
083:       {
084:         Product toBeDeleted = 
(Product)pmInstance.getObjectById(test, false);
085:         Transaction tx = null;
086:         tx = pmInstance.currentTransaction();
087:         tx.begin();
088:         pmInstance.deletePersistent(toBeDeleted);
089:         tx.commit();
090:       }
091:       catch (Throwable t)
092:       {
093:         // rollback in case of errors
094:         pmInstance.currentTransaction().rollback();
095:         t.printStackTrace();
096:       }
097:   }
098:  }
099: }
100:
101: %>
102:
103: <html>
104: <body>
105:
106: <form action="jdoForm.jsp" method=post>
107: <center>
108:
109: <table border="0" cellpadding="4" cellspacing="4" width="100%">
110:  <tr valign="top" bgcolor="#eeeeee">
111:    <td colspan="3">
112:    <table cellspacing="0" border="0" width="100%">
113:     <tr>
114:      <td align="center" valign="top">
115:       <table cellspacing="0" cellpadding="1" border="0" 
width="100%" align="center">
116:        <tr>
117:         <td valign="top" align="center">
118:           <b>JDO Test</b>
119:         </td>
120:        </tr>
121:       </table>
122:      </td>
123:     </tr>
```

Listing 46.2 *(continued)*

```
124:     </table>
125:     </td>
126:   </tr>
127:   <tr bgcolor="" >
128:     <td align="left"><b>Product Name:</b> <b></b>  
129:     <font size=2 color=red>
130:     <%=validate.getErrorMsg("productName")%></font></td>
131:     <td align="left">
132:      <input type="text" name="productName"
133:       value="<%=validate.getProductName()%>" size="30"></td>
134:   </tr>
135:   <tr bgcolor="" >
136:     <td align="left"> <b>Product Price:</b> <b></b>  
137:     <font size=2 color=red>
138:     <%=validate.getErrorMsg("productPrice")%></font></td>
139:     <td align="left">
140:      <input type="text" name="productPrice"
141:       value="<%=validate.getProductPrice()%>" size="30"></td>
142:   </tr>
143:   <tr bgcolor="" >
144:     <td align="left"><b>Product Count:</b> <b></b>  
145:     <font size=2 color=red>
146:     <%=validate.getErrorMsg("productCount")%></font></td>
147:     <td align="left">
148:      <input type="text" name="productCount"
149:       value="<%=validate.getProductCount()%>" size="30"></td>
150:   </tr>
151:   <tr bgcolor="" >
152:     <td align="center" colspan="2">
153:      <input type="Submit" name="Add" value="Add">
154:     </td>
155:   </tr>
156:
157: </table>
158:
159: </center>
160:
161: <table border="0" cellpadding="4" cellspacing="4" width="100%">
162:    <tr bgcolor="#eeeeee"><td>Product ID</td><td>Price ($)</td><td>
163:    # in Stock</td><td>Delete item?</td></tr>
164: <%
```

Listing 46.2 *(continued)*

The following code performs the rendering of the inventory data to the user display. A query is performed using the product table mapping. If data items are found in the product table, a collection handle is obtained and iterated through in the JSP display. A screenshot is provided in Listing 46.1 for this application.

```
165: // List all items
166: Query query = pm.getManagerInstance().newQuery(Product.class);
167: try
168: {
169:     // ask the broker to retrieve the Extent collection
170:
171:     Collection allProducts = (Collection)query.execute();
172:     // now iterate over the result to print each product
173:     java.util.Iterator iter = allProducts.iterator();
174:     while (iter.hasNext())
175:     {
176:       Product a = (Product) iter.next();
177: %>
178:   <tr>
179:    <td><%= a.getId() %></td>
180:    <td>$<%= a.getPrice() %></td>
181:    <td><%= a.getStock() %></td>
182:    <td><input type="checkbox" name="deleteItems" value="<%=
183:            a.getId() %>">Delete?</td>
184:   </tr>
185: <%
186:    }
187: }
188: catch (Throwable t)
189: {
190:   t.printStackTrace();
191: }
192: %>
193:   <tr bgcolor="" >
194:    <td align="center" colspan="4">
195:     <input type="Submit" name="Delete" value="Delete">
196:    </td>
197:   </tr>
198:
199: </table>
200:
201: </form>
202:
203: </body>
204: </html>
205:
```

Listing 46.3 Inventory results

The jdoPersistenceMgr application which follows uses the JDO `PersistenceManager` class on line 22 as its primary interface between the Web application jdoForm.jsp and the back-end MySQL database. The `PersistenceManager` provides both query and transaction management. It uses the resource adapter to access the data store, specifically the OJB libraries.

```
01: package org.javapitfalls.item46;
02:
03: import org.apache.ojb.tutorial4.*;
04:
05: import org.apache.ojb.jdo.PersistenceManagerImpl;
06: import org.apache.ojb.jdo.PersistenceManagerFactoryImpl;
07: import javax.jdo.PersistenceManager;
08: import javax.jdo.Transaction;
09:
10: import javax.jdo.Query;
11: import java.util.Collection;
12:
13: import javax.jdo.PersistenceManager;
14: import javax.jdo.PersistenceManagerFactory;
15: import java.io.BufferedReader;
16: import java.io.InputStreamReader;
17: import java.util.Vector;
18:
19: public class jdoPersistenceMgr implements java.io.Serializable {
20:
21:   PersistenceManagerFactory factory;
22:   PersistenceManager manager;
23:
24:   public jdoPersistenceMgr()
25:   {
26:     manager = null;
27:     try
28:     {
29:       factory = PersistenceManagerFactoryImpl.getInstance();
30:       manager = factory.getPersistenceManager();
31:
32:     }
33:     catch (Throwable t)
34:     {
35:       System.out.println("ERROR: " + t.getMessage());
36:       t.printStackTrace();
37:     }
38:   }
39:
40:   public PersistenceManager getManagerInstance()
41:   {
42:     return manager;
43:   }
44:
45: }
```

Listing 46.4 do.PersistenceManager.java

There are many discussions in the development community about JDO and the perception that it suffers from its association with Object Data Modeling Group (ODMG), which has not garnered widespread community support because of proprietary language extensions and nonstandard meta data support.

Personally, I feel that this misperception is badly placed and will be displaced once the development community implements new reference implementations by Sun and the ASF. JDO is still a young standard, but as products like OJB mature, developers will gain a better appreciation of JDO's ability to propagate consistent persistence behavior across implementations, its use of XML meta data representation for mapping data, and its ability to use mature Java Collection class libraries to manipulate data. These capabilities could make your persistence questions easier to tackle on your system's data layer.

JDO is relevant because it minimizes database transaction implementations by developers and it supports reuse, particularly with Java components. OJB JDO facilitates cache and object management through object models and avoids relational modeling and EJB/CMP intricacies that can be specific to an application.

Item 47: Where's the WSDL? Pitfalls of Using JAXR with UDDI

The Java API for XML Registries (JAXR) provides a Java API for accessing different kinds of registries. Built on the OASIS ebXML Registry Information Model (RIM), mappings between ebXML interfaces and JAXR interfaces are direct and seem intuitive. When JAXR is used with UDDI (Universal Description and Discovery Integration) registries, however, there is sometimes a bit of confusion. In our experience, Java developers that are new to JAXR face a learning curve and a few pitfalls that revolve around misunderstanding the relationships between JAXR objects and the UDDI data structures. This item shows examples of these pitfalls and provides guidance on how to avoid them.

To show these possible pitfalls, we will build a skeletal program that includes the basics of JAXR, shown in our Listing 47.1. In this listing, we have an empty `makeCall()` method on lines 64 to 69. After we explain our example setup in the listing, we will implement several versions of `makeCall()` to query the registry.

In using JAXR, you must first set up a connection to a registry server. Listing 47.1 shows the beginning of a program using JAXR to speak to a registry server. In this example, we are using the public Microsoft UDDI registry, passed in to the constructor of `JAXRQueryExample`. In lines 29 to 32, we are able to get a connection to the registry using the `ConnectionFactory` object from the `javax.xml.registry` package. From that connection, we are able to get the `RegistryService` object, which allows us to receive the `BusinessQueryManager` object on line 34.

```
001: package org.javapitfalls.item47;
002:
003: import javax.xml.registry.*;
004: import javax.xml.registry.infomodel.*;
005: import java.net.*;
006: import java.util.*;
```

Listing 47.1 The start of our UDDI query example

```
007:
008: public class JAXRQueryExample
009: {
010:   RegistryService      m_regserv    = null;
011:   BusinessQueryManager m_querymgr   = null;
012:   Connection           m_connection = null;
013:
014:   public JAXRQueryExample(String registryURL) throws JAXRException
015:   {
016:     ConnectionFactory factory     = null;
017:
018:     Properties props = new Properties();
019:     props.setProperty("javax.xml.registry.queryManagerURL",
020:                       registryURL);
021:
022:     try
023:     {
024:       /*
025:        * Create the connection, passing it the
026:        * properties -- in this case, just the URL
027:        */
028:
029:       factory = ConnectionFactory.newInstance();
030:       factory.setProperties(props);
031:
032:       m_connection = factory.createConnection();
033:       m_regserv    = m_connection.getRegistryService();
034:       m_querymgr   = m_regserv.getBusinessQueryManager();
035:     }
036:     catch ( JAXRException e )
037:     {
038:       cleanUp();
039:       //pass it on..
040:       throw e;
041:     }
042:   }
043:
044:   /**
045:    * Close the connection
046:    */
047:   public void cleanUp()
048:   {
049:     if ( m_connection != null )
050:     {
051:       try
052:       {
053:         m_connection.close();
054:       }
```

Listing 47.1 *(continued)*

```
055:       catch ( JAXRException je )
056:       {
057:       }
058:     }
059:   }
060:
061:   /**
062:    * Our main focus in this pitfall
063:    **/
064:   public void makeCall(String query) throws Exception
065:   {
066:     //We will use m_querymgr in this method
067:
068:     System.out.println("makeCall() Not implemented yet!");
069:   }
070:
071:   /** Simple convenience function, since strings from registry
072:    *  are in this format
073:    */
074:   public String convertToString(InternationalString intl)
075:   throws JAXRException
076:   {
077:     String stringVal = null;
078:     if ( intl != null )
079:     {
080:       stringVal = intl.getValue();
081:     }
082:     return(stringVal);
083:   }
084:
085:   public static void main(String[] args)
086:   {
087:     JAXRQueryExample jaxrex = null;
088:     try
089:     {
090:       String uddiReg = "http://uddi.microsoft.com/inquire";
091:       jaxrex = new JAXRQueryExample(uddiReg);
092:
093:       jaxrex.makeCall("truman");
094:
095:       System.out.println("-----------SECOND QUERY--------------");
096:       jaxrex.makeCall("sched");
097:     }
098:     catch ( Exception e )
099:     {
100:       e.printStackTrace();
101:     }
102:     finally
```

Listing 47.1 *(continued)*

```
103:     {
104:       jaxrex.cleanUp();
105:     }
106:   }
107:
108: }
109:
```

Listing 47.1 *(continued)*

On lines 64 to 69, we have an empty `makeCall()` method that will be our focus for the rest of this pitfall item. In that method, we will make extensive use of the `BusinessQueryManager` interface returned from the `RegistryService` object on line 33. Because this is where some of the confusion comes in, we have listed the available methods of the `javax.xml.registry.BusinessQueryManager` interface in Table 47.1.

Table 47.1 The BusinessQueryManager Interface

METHOD	DESCRIPTION
`BulkResponse findAssociations (Collection findQualifiers, String sourceObjectId, String targetObjectId, Collection assocTypes)`	Finds `Association` objects that match parameters of this call.
`BulkResponse findCallerAssociations(Collection findQualifiers, Boolean confirmedByCaller, Boolean confirmedByOtherParty, Collection associationTypes)`	Finds all `Association` objects owned by the caller that match all of the criteria specified by the parameters of this call.
`ClassificationScheme findClassificationSchemeByName( Collection findQualifiers, String namePattern )`	Finds a `ClassificationScheme` by name based on the specified find qualifiers and name pattern.
`BulkResponse findClassificationSchemes(Collection findQualifiers, Collection namePatterns, Collection classifications, Collection externalLinks)`	Finds all `ClassificationScheme` objects that match all of the criteria specified by the parameters of this call.
`Concept findConceptByPath(java.lang.String path)`	Finds a Concept object by the path specified.

(continued)

Table 47.1 *(continued)*

METHOD	DESCRIPTION
`BulkResponse findConcepts(Collection findQualifiers, Collection namePatterns, Collection classifications, Collection externalIdentifiers, Collection externalLinks)`	Finds all `Concept` objects that match the parameters of this call.
`BulkResponse findOrganizations(Collection findQualifiers, Collection namePatterns, Collection classifications, Collection specifications, Collection externalIdentifiers, Collection externalLinks)`	Finds all `Organization` objects that match the parameters of this call.
`BulkResponse findRegistryPackages(Collection findQualifiers, Collection namePatterns, Collection classifications, Collection externalLinks)`	Finds all `RegistryPackage` objects that match the parameters of the call.
`BulkResponse findServiceBindings(Key serviceKey, Collection findQualifiers, Collection classifications, Collection specifications)`	Finds all `ServiceBinding` objects that match the parameters of this call.
`BulkResponse findServices(Key orgKey, Collection findQualifiers, Collection namePatterns, Collection classifications, Collection specifications)`	Finds all `Service` objects that match the parameters of this call.

As you can see from Table 47.1, the `BusinessQueryManager` interface queries the registry for information. In calling these methods, many searches can be constrained by collections of search qualifiers, patterns, classifications, external links, and specifications. Many of the methods return a `BulkResponse` object that contains a `Collection` of objects. Depending on the method call, the objects contained in the collection can be associations between registry instances (`Association`), taxonomies used to describe the classifications of registry objects (`ClassificationScheme`), taxonomy elements themselves (`Concept`) present in the registry, organizations in the registry (`Organization`), registry entries logically organized together (`RegistryPackage`), services that are available (`Service`), and bindings to interfaces of the service (`ServiceBinding`).

All of the objects in the collection contained by the `BulkResponse` objects are interfaces in the `javax.xml.registry.infomodel` package and are all subinterfaces to the `RegistryObject` class. Figure 47.1 shows the methods of the `RegistryObject` interface, as well as the classes that realize that interface. Because the `RegistryObject` interface contains a `getExternalLinks()` method, every class that implements this interface may have a named URI to content residing outside the registry—such as a Web Service Description Language (WSDL) document in a UDDI registry. As we show potential pitfalls in making JAXR searches, you will find that it is important to have an understanding of where objects in a registry reside.

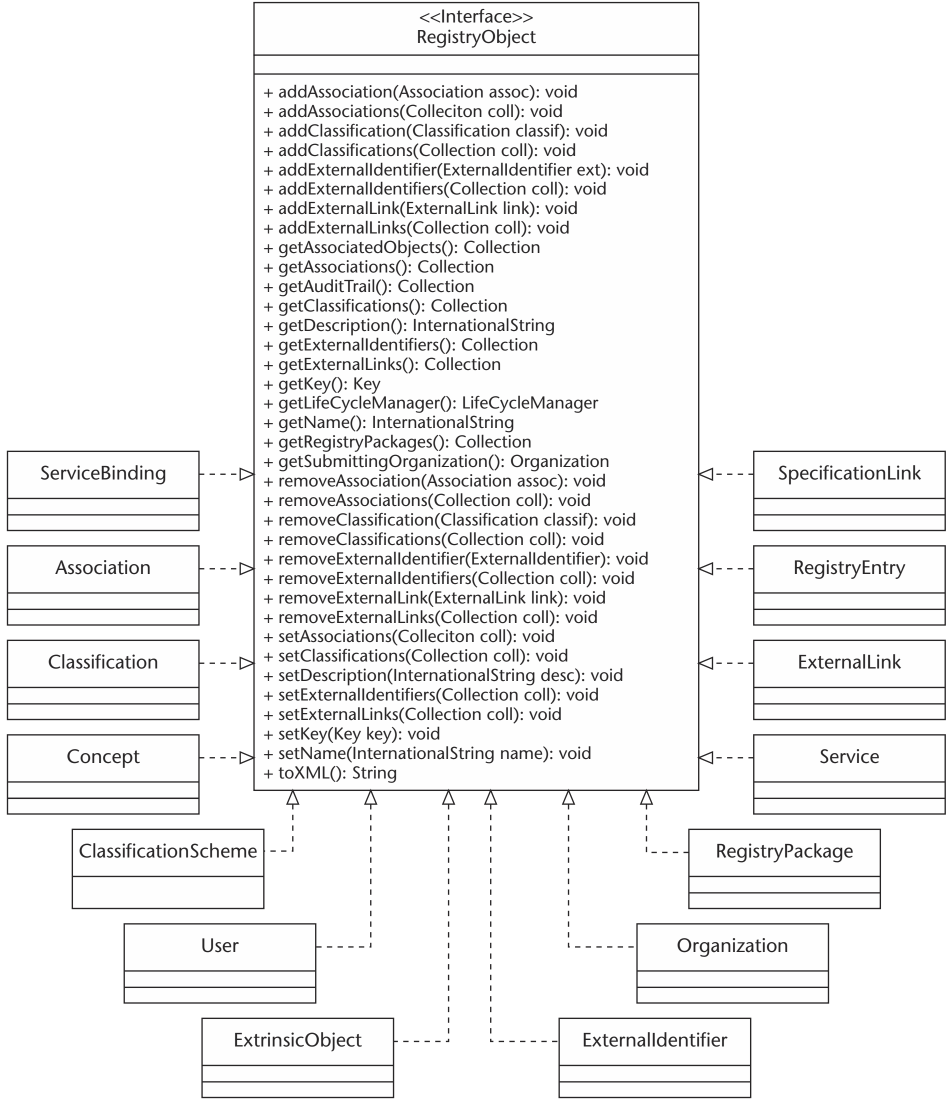


Figure 47.1 The `RegistryObject` interface.

Where's the WSDL?

We're about to embark on a quest for a Web service's WSDL found in a UDDI registry. For our example, we would like to constrain the search by searching for a string found in an organization name in the registry. From there, we would like to print out all available Web services—and more importantly, their WSDL—so that we can dynamically call their Web services. To do this, we will modify our original code from Listing 47.1 and will replace that code's empty `makeCall()` method. For convenience, we will list the following code listings starting with line 64—this will make it easier when you download the code for the examples from our Web site.

Listing 47.2 shows the `makeCall()` method where we are attempting to get the WSDL document from a search of the registry. Our search constraint, the query string, is set up to be a pattern on lines 74 and 75. At the same time, we qualify our search on line 74 by setting our responses to be sorted alphabetically by name. Our example then queries the BusinessQueryManager's `findOrganizations()` method with these parameters. Because a collection of `Organization` objects will be returned from the collection in the `BulkResponse` object, and because organizations contain services that can be retrieved by the organization's `getServices()` method, we will be able to get the services. Because the `Service` object inherits the `getExternalLinks()` method from `RegistryObject`, the programmer in this example assumes that we should be able to get the WSDL document with that method on the service. Just to make certain, the programmer calls that method from the service's `ServiceBinding` objects, which was returned from `getServiceBindings()` on the service on line 123.

```
064:   public void makeCall(String query) throws Exception
065:   {
066:     if ( m_querymgr == null )
067:     {
068:       throw new Exception("No query manager!!!!! Exiting ⮐
makeCall()");
069:     }
070:     else
071:     {
072:
073:       Collection findQualifiers = new ArrayList();
074:       findQualifiers.add(FindQualifier.SORT_BY_NAME_DESC);
075:       Collection namePatterns = new ArrayList();
076:       namePatterns.add("%" + query + "%");
077:
078:       BulkResponse response =
079:       m_querymgr.findOrganizations(findQualifiers,
080:                                    namePatterns,
081:                                    null, null, null, null);
082:
083:       Iterator orgIterator = response.getCollection().iterator();
084:
```

Listing 47.2 Our first attempt for WSDL, in makeCall( )

```
085:       System.out.println("Made an organizational query for '" +
086:                          query + "'...");
087:
088:       int orgcnt = 0;
089:       while ( orgIterator.hasNext() )
090:       {
091:         orgcnt ++;
092:         //Let's get every organization that matches the query!
093:         Organization org = (Organization) orgIterator.next();
094:         System.out.println(orgcnt + ") Organization Name: " +
095:                            convertToString(org.getName()));
096:         System.out.println("   Organization Desc: " +
097:                            convertToString(org.getDescription()));
098:
099:         //Now get the services provided by the organization
100:         Iterator svcIter = org.getServices().iterator();
101:         while ( svcIter.hasNext() )
102:         {
103:           Service svc = (Service)svcIter.next();
104:           System.out.println("   Service Name: " +
105:                              convertToString(svc.getName()));
106:
107:           //External links are associated with every registry
object,
108:           //so maybe it's in the external link of service..
109:           Iterator extlinkit = svc.getExternalLinks().iterator();
110:           if ( !extlinkit.hasNext() )
111:           {
112:             System.out.println("   ? - " +
113:                       "No WSDL document at
svc.getExternalLinks()..");
114:           }
115:           while ( extlinkit.hasNext() )
116:           {
117:             ExternalLink extlink = (ExternalLink)extlinkit.next();
118:             System.out.println("WSDL Document: " +
119:                                extlink.getExternalURI());
120:           }
121:
122:           //Let's get the service binding object
123:           Iterator bindit = svc.getServiceBindings().iterator();
124:           while ( bindit.hasNext() )
125:           {
126:             ServiceBinding sb = (ServiceBinding)bindit.next();
127:             System.out.println("   Service Binding Name: " +
128:                                convertToString(sb.getName()));
129:             System.out.println("   Service Binding Desc: " +
130:                                convertToString(sb.getDescription()));
```

Listing 47.2 *(continued)*

```
131:             System.out.println("   Service Binding Access URI:\n" +
132:                                "    " + sb.getAccessURI());
133:
134:             //Maybe WSDL is on the external link here..
135:             Iterator extlinkit2 = sb.getExternalLinks().iterator();
136:             if ( !extlinkit2.hasNext() )
137:             {
138:               System.out.println("   ? - " +
139:                  "No WSDL document at \n" +
140:                 "  svc.getServiceBindings().getExternalLinks()..");
141:             }
142:             while ( extlinkit2.hasNext() )
143:             {
144:               ExternalLink extlink =
(ExternalLink)extlinkit2.next();
145:               System.out.println("WSDL Document: " +
146:                                   extlink.getExternalURI());
147:             }
148:
149:
150:           }
151:
152:
153:         }
154:
155:       }
156:     }
157:   }
```

Listing 47.2 *(continued)*

Our example in Listing 47.2 shows a search of the UDDI registry, and printing organization, service, and service binding information for the result of our search. Listing 47.3 shows the output of our program. There are two calls to `makeCall()`: a query including the string `'truman'` and a query including the string `'sched'`. Both queries return two results. As you can see by the bolded content in Listing 47.3, the programmer got it wrong.

```
01: Made an organizational query for 'truman'...
02: 1) Organization Name: TrumanTruck.com
03:    Organization Desc: Restoration of Truman Schermerhorn's Truck
04:    Service Name: TrumanTruck Page
05:    ? - No WSDL document at svc.getExternalLinks()..
06:    Service Binding Name: null
```

Listing 47.3 Output of our first attempt

```
07:    Service Binding Desc: hyperlink
08:    Service Binding Access URI:
09:     http://www.trumantruck.com
10:    ? - No WSDL document at
11:     svc.getServiceBindings().getExternalLinks()..
12: 2) Organization Name: Harry S. Truman Scholarship Foundation
13:    Organization Desc:
14:    Service Name: Web home page
15:    ? - No WSDL document at svc.getExternalLinks()..
16:    Service Binding Name: null
17:    Service Binding Desc: hyperlink
18:    Service Binding Access URI:
19:     http://www.truman.gov/welcome.htm
20:    ? - No WSDL document at
21:     svc.getServiceBindings().getExternalLinks()..
22: -----------SECOND QUERY--------------
23: Made an organizational query for 'sched'...
24: 1) Organization Name: LFC Scheduling
25:    Organization Desc:
26:    Service Name: Classroom Scheduling
27:    ? - No WSDL document at svc.getExternalLinks()..
28:    Service Binding Name: null
29:    Service Binding Desc:
30:    Service Binding Access URI:
31:     http://www.contest.eraserver.net/Scheduling/Scheduler.asmx
32:    ? - No WSDL document at
33:     svc.getServiceBindings().getExternalLinks()..
34: 2) Organization Name: Interactive Scheduler
35:    Organization Desc:
36:    Service Name: Interactive Schedule
37:    ? - No WSDL document at svc.getExternalLinks()..
38:    Service Binding Name: null
39:    Service Binding Desc:
40:    Service Binding Access URI:
41:                                                                    ⮐
http://www.contest.eraserver.net/InteractiveScheduler/service1.asmx
42:    ? - No WSDL document at
43:     svc.getServiceBindings().getExternalLinks()..
44:
```

Listing 47.3 *(continued)*

The problem in that example demonstrates the programmer's misunderstanding of where the WSDL document lives in the JAXR object hierarchy. This is a common problem with developers new to JAXR and UDDI. Luckily, Appendix D of the JAXR specification (found at http://java.sun.com/xml/jaxr/) shows the mappings of JAXR to UDDI. In Figure 47.2, we have provided a graphical depiction of where the WSDL document is located. To get the WSDL document for a Web service, you need to somehow

traverse the structures to get to the registry object classified as a "wsdlspec", and get that object's ExternalLink. Because our organizational query returns a collection of organizations, we have shown a logical "road map" from the Organization object. For the sake of eliminating confusion, we have left out potential paths to ExternalLink and SpecificationLink that are available from every object (since they all realize the RegistryObject interface).

TIP If you are used to UDDI terminology, there are different terms for each of these interfaces. For example, a JAXR Organization is known in UDDI as a "businessEntity." A JAXR Service is known as a UDDI "businessService." A UDDI "bindingTemplate" is a JAXR ServiceBinding. There are many more of these mappings. For more information, download the JAXR specification at http://java.sun.com/jaxr/.

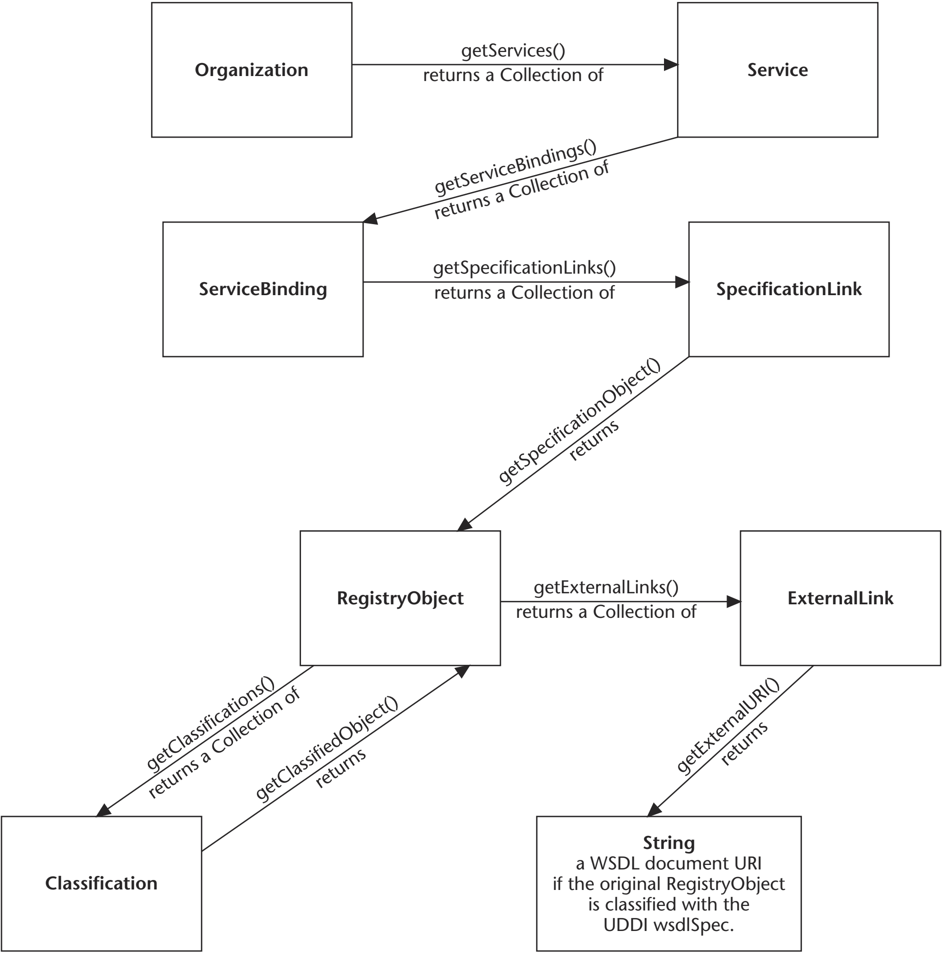


Figure 47.2 Navigating through JAXR objects to get WSDL .

Now that we have provided the "traversal road map" in Figure 47.2, we will demonstrate another potential programming pitfall. In Listing 47.4, the programmer traverses the objects as shown from the figure. On line 83, we get our collection of organizations from our `BulkResponse` object. On line 100, we get a collection of services from each organization. On line 109, we get a collection of service bindings from each service. On lines 120 and 121, we get a collection of specification links from each service binding. On line 127, we get a registry object from the specification link with the `getSpecificationObject()` method. Finally, on lines 128 to 134, we get the external link from the registry object, and we call the `getExternalURI()` method on the specification link.

```
064:   public void makeCall(String query) throws Exception
065:   {
066:     if ( m_querymgr == null )
067:     {
068:       throw new Exception("No query manager!!!!! Exiting ⮐
makeCall()");
069:     }
070:     else
071:     {
072:
073:       Collection findQualifiers = new ArrayList();
074:       findQualifiers.add(FindQualifier.SORT_BY_NAME_DESC);
075:       Collection namePatterns = new ArrayList();
076:       namePatterns.add("%" + query + "%");
077:
078:       BulkResponse response =
079:       m_querymgr.findOrganizations(findQualifiers,
080:                                    namePatterns,
081:                                    null, null, null, null);
082:
083:       Iterator orgIterator = response.getCollection().iterator();
084:
085:       System.out.println("Made an organizational query for '" +
086:                          query + "'...");
087:
088:       int orgcnt = 0;
089:       while ( orgIterator.hasNext() )
090:       {
091:         orgcnt ++;
092:         //Let's get every organization that matches the query!
093:         Organization org = (Organization) orgIterator.next();
094:         System.out.println(orgcnt + ") Organization Name: " +
095:                            convertToString(org.getName()));
096:         System.out.println("   Organization Desc: " +
097:                            convertToString(org.getDescription()));
098:
```

Listing 47.4 Our second attempt for WSDL, in makeCall() *(continued)*

```
099:         //Now get the services provided by the organization
100:         Iterator svcIter = org.getServices().iterator();
101:         while ( svcIter.hasNext() )
102:         {
103:           Service svc = (Service)svcIter.next();
104:           System.out.println("   Service Name: " +
105:                                convertToString(svc.getName()));
106:
107:
108:           //Let's get the service binding object
109:           Iterator bindit = svc.getServiceBindings().iterator();
110:           while ( bindit.hasNext() )
111:           {
112:             ServiceBinding sb = (ServiceBinding)bindit.next();
113:             System.out.println("   Service Binding Name: " +
114:                                convertToString(sb.getName()));
115:             System.out.println("   Service Binding Desc: " +
116:                                convertToString(sb.getDescription()));
117:             System.out.println("   Service Binding Access URI:\n" +
118:                                "    " + sb.getAccessURI());
119:
120:             Iterator speclinkit =
121:             sb.getSpecificationLinks().iterator();
122:             while ( speclinkit.hasNext() )
123:             {
124:               SpecificationLink slink =
125:               (SpecificationLink)speclinkit.next();
126:
127:               RegistryObject ro = slink.getSpecificationObject();
128:               Iterator extlinkit =
ro.getExternalLinks().iterator();
129:               while ( extlinkit.hasNext() )
130:               {
131:                 ExternalLink extlink =
(ExternalLink)extlinkit.next();
132:                 System.out.println("   WSDL: " +
133:                                    extlink.getExternalURI());
134:               }
135:             }
136:           }
137:         }
138:       }
139:     }
140:   }
```

Listing 47.4 *(continued)*

Unfortunately, the programmer was wrong again. As you can see in the output of the program in Listing 47.5, the first response from the 'truman' query returned no WSDL document. The second response from the 'truman' query returned an HTML document on line 17. Strangely, the first and second responses from the 'sched' query did return the WSDL document. What went wrong?

```
01:
02: Made an organizational query for 'truman'...
03: 1) Organization Name: TrumanTruck.com
04:    Organization Desc: Restoration of Truman Schermerhorn's Truck
05:    Service Name: TrumanTruck Page
06:    Service Binding Name: null
07:    Service Binding Desc: hyperlink
08:    Service Binding Access URI:
09:     http://www.trumantruck.com
10: 2) Organization Name: Harry S. Truman Scholarship Foundation
11:    Organization Desc:
12:    Service Name: Web home page
13:    Service Binding Name: null
14:    Service Binding Desc: hyperlink
15:    Service Binding Access URI:
16:     http://www.truman.gov/welcome.htm
17:    WSDL: http://www.uddi.org/specification.html
18: -----------SECOND QUERY--------------
19: Made an organizational query for 'sched'...
20: 1) Organization Name: LFC Scheduling
21:    Organization Desc:
22:    Service Name: Classroom Scheduling
23:    Service Binding Name: null
24:    Service Binding Desc:
25:    Service Binding Access URI:
26:     http://www.contest.eraserver.net/Scheduling/Scheduler.asmx
27:    WSDL: ↩
http://www.contest.eraserver.net/Scheduling/Scheduler.asmx?wsdl
28: 2) Organization Name: Interactive Scheduler
29:    Organization Desc:
30:    Service Name: Interactive Schedule
31:    Service Binding Name: null
32:    Service Binding Desc:
33:    Service Binding Access URI:
34: ↩
http://www.contest.eraserver.net/InteractiveScheduler/service1.asmx
35:    WSDL: ↩
http://www.contest.eraserver.net/InteractiveScheduler/ ↩
InteractiveScheduler.wsdl
```

Listing 47.5 Output of our second attempt

The problem from our second attempt to find WSDL documents revolves around the concept of classifications. Because both registry objects for our first query were not classified as the UDDI "wsdlspec" type, they do not have WSDL documents. Unfortunately, this is a common programming mistake. If the programmer had assumed that these were both Web services registered with a WSDL URL, the program would have mixed results—the program may work for one query, but not for another. In the case of the false WSDL document output on line 17, if the program had tried to dynamically call the Web service using JAX-RPC, the program would have failed.

The answer to our dilemma lies in our call from the Classification object to the RegistryObject in Figure 47.2. If you call getClassifications() from the returned RegistryObject, and if the classification of that object is the "wsdlspec" classification, then you can call getClassifiedObject() to then get the WSDL-classified object, and then retrieve the external link. Listing 47.6 does just that in lines 123 to 150.

```
064: public void makeCall(String query) throws Exception
065: {
066:   if ( m_querymgr == null )
067:   {
068:     throw new Exception("No query manager!!!!! Exiting ⮐
makeCall()");
069:   }
070:   else
071:   {
072:
073:     Collection findQualifiers = new ArrayList();
074:     findQualifiers.add(FindQualifier.SORT_BY_NAME_DESC);
075:     Collection namePatterns = new ArrayList();
076:     namePatterns.add("%" + query + "%");
077:
078:     BulkResponse response =
079:     m_querymgr.findOrganizations(findQualifiers,
080:                                  namePatterns,
081:                                  null, null, null, null);
082:
083:     Iterator orgIterator = response.getCollection().iterator();
084:
085:     System.out.println("Made an organizational query for '" +
086:                        query + "'...");
087:
088:     int orgcnt = 0;
089:     while ( orgIterator.hasNext() )
090:     {
091:       String orgname  = null;
092:       String orgdesc  = null;
```

Listing 47.6 The solution to our dilemma

```
093:       String svcname  = null;
094:
095:       //Let's get every organization that matches the query!
096:       Organization org = (Organization) orgIterator.next();
097:
098:       orgname = convertToString(org.getName());
099:       orgdesc = convertToString(org.getDescription());
100:
101:       //Now get the services provided by the organization
102:       Iterator svcIter = org.getServices().iterator();
103:       while ( svcIter.hasNext() )
104:       {
105:         Service svc = (Service)svcIter.next();
106:         svcname = convertToString(svc.getName());
107:
108:         //Let's get the service binding object from service
109:         Iterator bindit = svc.getServiceBindings().iterator();
110:         while ( bindit.hasNext() )
111:         {
112:           ServiceBinding sb = (ServiceBinding)bindit.next();
113:
114:           Iterator speclinkit =
115:           sb.getSpecificationLinks().iterator();
116:           while ( speclinkit.hasNext() )
117:           {
118:             SpecificationLink slink =
119:             (SpecificationLink)speclinkit.next();
120:
121:             RegistryObject ro = slink.getSpecificationObject();
122:
123:             //Now, let's see the classification object..
124:             Iterator classit = ro.getClassifications().iterator();
125:             while ( classit.hasNext() )
126:             {
127:               Classification classif =
128:               (Classification)classit.next();
129:               if ( classif.getValue().equalsIgnoreCase("wsdlspec") )
130:               {
131:                 orgcnt++;
132:                 System.out.println(orgcnt +
133:                              ") Organization Name: " + orgname);
134:                 System.out.println(
135:                              "   Organization Desc: " + orgdesc);
136:                 System.out.println(
137:                              "   Service Name: " + svcname);
138:
139:                 RegistryObject ro2 = classif.getClassifiedObject();
```

Listing 47.6 *(continued)*

```
140:
141:                    Iterator extlinkit =
142:                    ro2.getExternalLinks().iterator();
143:                    while ( extlinkit.hasNext() )
144:                    {
145:                      ExternalLink extlink =
146:                      (ExternalLink)extlinkit.next();
147:
148:                      System.out.println("   WSDL: " +
149:                                         extlink.getExternalURI());
150:                    }
151:
152:                  }
153:                }
154:              }
155:            }
156:          }
157:        }
158:      }
159: }
```

Listing 47.6 *(continued)*

The result of our example is shown in Listing 47.7. In our output, our program only prints the listing of Web services that have WSDL documents.

```
01: Made an organizational query for 'trumantruck'...
02: -----------SECOND QUERY--------------
03: Made an organizational query for 'sched'...
04: 1) Organization Name: LFC Scheduling
05:    Organization Desc:
06:    Service Name: Classroom Scheduling
07:    WSDL:
http://www.contest.eraserver.net/Scheduling/Scheduler.asmx?wsdl
08: 2) Organization Name: Interactive Scheduler
09:    Organization Desc:
10:    Service Name: Interactive Schedule
11:    WSDL:
http://www.contest.eraserver.net/InteractiveScheduler/
InteractiveScheduler.wsdl
```

Listing 47.7 The output of our solution

It is important to note that we could get to the service much easier. If we knew the name of the service, for example, and we wanted to go directly to the RegistryObject that has the "wsdlspec" classification, we could do a concept query. The RegistryObject, shown in Figure 47.2, on page 408, is always a Concept object when using UDDI registries (even though the getSpecificationObject() from the SpecificationLink interface returns the RegistryObject interface to be more flexible for other registries).

To demonstrate this, we will show another example of the makeCall() method in Listing 47.8. We will call the findConcepts() method on the BusinessQueryManager object. To constrain the search, we will use the same namePatterns query pattern that we used in the previous examples, but we will add a classification constraint on lines 83 to 100. In doing so, the objects that are returned will be Concept objects that have WSDL documents and that match the query pattern passed in as a parameter.

```
064:   public void makeCall(String query) throws Exception
065:   {
066:     if ( m_querymgr == null )
067:     {
068:       throw new Exception("No query manager!!!!! Exiting ⮐
makeCall()");
069:     }
070:     else
071:     {
072:
073:       Collection findQualifiers = new ArrayList();
074:       findQualifiers.add(FindQualifier.SORT_BY_NAME_DESC);
075:       Collection namePatterns = new ArrayList();
076:       namePatterns.add("%" + query + "%");
077:
078:       /*
079:        * Find the classification scheme defined by
080:        *  the UDDI spec
081:        */
082:
083:       String schemeName = "uddi-org:types";
084:       ClassificationScheme uddiOrgTypes =
085:       m_querymgr.findClassificationSchemeByName(null,
086:                                                  schemeName);
087:       /*
088:        * Create a classification, specifying the scheme
089:        *  and the taxonomy name and value defined for
090:        *  WSDL documents by the UDDI spec
091:        */
092:       BusinessLifeCycleManager blm =
093:       m_regserv.getBusinessLifeCycleManager();
```

Listing 47.8 Good example of querying by concept *(continued)*

```
094:
095:        Classification wsdlSpecClass =
096:        blm.createClassification(uddiOrgTypes,
097:                                 "wsdlSpec", "wsdlSpec");
098:
099:        Collection classifications = new ArrayList();
100:        classifications.add(wsdlSpecClass);
101:
102:        // Find concepts
103:        BulkResponse response =
104:        m_querymgr.findConcepts(null, namePatterns,
105:                                classifications, null, null);
106:
107:        System.out.println("Made an wsdlSpec concept query for \n'" +
108:                     "services matching '" +query + "'");
109:
110:        // Display information about the concepts found
111:        int itnum = 0;
112:        Iterator iter = response.getCollection().iterator();
113:        if ( !iter.hasNext())
114:        {
115:          System.out.println("   No matching items!");
116:        }
117:        while ( iter.hasNext() )
118:        {
119:          itnum++;
120:          Concept concept = (Concept) iter.next();
121:          System.out.println(itnum + ")   Name: " +
122:                         convertToString(concept.getName()));
123:          System.out.println("     Description: " +
124:                         convertToString(concept.getDescription()));
125:
126:          Iterator linkit = concept.getExternalLinks().iterator();
127:          if ( linkit.hasNext() )
128:          {
129:            ExternalLink link =
130:            (ExternalLink) linkit.next();
131:            System.out.println("    WSDL: '" +
132:                               link.getExternalURI() + "'");
133:          }
134:
135:        }
136:      }
137:    }
```

Listing 47.8 *(continued)*

The result of our program is shown in Listing 47.9. On our concept query for services with the string `'sched'` in them with WSDL documents, we had four results.

```
01: Made an wsdlSpec concept query for
02: 'services matching 'truman'
03:    No matching items!
04: -----------SECOND QUERY--------------
05: Made an wsdlSpec concept query for
06: 'services matching 'sched'
07: 1)   Name: Continental-com:Schedule-v1
08:      Description: Flight schedule
09:      WSDL: 
'http://webservices.continental.com/schedule/schedule.asmx?WSDL'
10: 2)   Name: Interactive Scheduler
11:      Description: A Web Service that provides a method to schedule 
                      meetings into someone else's calendar.
12:     WSDL: 
'http://www.contest.eraserver.net/InteractiveScheduler/
InteractiveScheduler.wsdl'
13: 3)   Name: Lake Forest College-com:SchedulingInterface-v1
14:      Description: Scheduling Web Service for Institutions-
Scheduling Classes to appropriate rooms
15:     WSDL: 
'http://www.contest.eraserver.net/Scheduling/Scheduler.asmx?wsdl'
16: 4)   Name: Metric-com:Aeroflot Flights Schedule
17:      Description: Web service deliver on-line flights schedule 
information
18:     WSDL: 'http://webservices.aeroflot.ru/flightSearch.wsdl'
```

Listing 47.9 Output of querying by concept

In this pitfall, we demonstrated problems that developers encounter when using the JAXR API with UDDI registries. We showed two examples of potential pitfalls while traversing the data structures of the registry and provided solutions for these problems. Because of the difficulty that some programmers have with JAXR and UDDI, reading the JAXR specification is highly recommended.

Item 48: Performance Pitfalls in JAX-RPC Application Clients

The Java API for XML-based Remote Procedure Calls (JAX-RPC) allows us to continue to think like Java developers when we develop, deploy, and communicate with RPC-based Web services. Although JAX-RPC relies on underlying protocols (HTTP and SOAP), the API hides this complexity from the application developer. Using basic programming techniques that enterprise developers are accustomed to, you can create a Web service easily. Building a client that communicates with the Web service is also

easy—proxy stubs for the Web service can be compiled prior to runtime, they can be dynamically generated at runtime, or the Dynamic Invocation Interface (DII) can be used to discover a Web service's API on-the-fly.

In this pitfall item, we use different techniques in building clients for a simple Web service. We run a timing experiment on each of the techniques and give recommendations for building clients using JAX-RPC. As a result of reading this pitfall item, you will understand the performance implications of using each technique—and hopefully use this to your advantage in your projects.

Example Web Service

For this pitfall item, we used Sun's Java Web Services Developer Pack (WSDP) and created a simple Web service called "SimpleTest." The Web service has one method called `doStuff()`, and the interface used to develop this Web service is shown in Listing 48.1.

```
001: package org.javapitfalls.item48;
002:
003: import java.rmi.Remote;
004: import java.rmi.RemoteException;
005:
006: public interface SimpleTestIF extends Remote
007: {
008:    public String doStuff(String s) throws RemoteException;
009: }
```

Listing 48.1 Interface to our simple Web service.

The Web Service Description Language (WSDL) that was automatically generated from the tools available with the developer's pack is shown in Listing 48.2. Because this was automatically generated and deployed with the developer tools that generated this from our Java interface, our implementation class, and deployment descriptors, we weren't forced to write it by hand. As JAX-RPC defines Web services as collections of remote interfaces and methods, WSDL defines Web services as a collection of ports and operations. The WSDL provided in Listing 48.2 is for your reference, as we develop our Web service clients later in the pitfall examples.

```
001: <?xml version="1.0" encoding="UTF-8" ?>
002: <definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
003:      xmlns:tns="http://org.javapitfalls.item48/wsdl/SimpleTest"
```

Listing 48.2 WSDL for a simple Web service

```
005:      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
004:      xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
006:      name="SimpleTest"
007:      targetNamespace="http://org.javapitfalls.item48/wsdl/SimpleTest">
008:    <types />
009:    <message name="SimpleTestIF_doStuff">
010:        <part name="String_1" type="xsd:string" />
011:    </message>
012:    <message name="SimpleTestIF_doStuffResponse">
013:        <part name="result" type="xsd:string" />
014:    </message>
015:    <portType name="SimpleTestIF">
016:        <operation name="doStuff" parameterOrder="String_1">
017:            <input message="tns:SimpleTestIF_doStuff" />
018:            <output message="tns:SimpleTestIF_doStuffResponse" />
019:        </operation>
020:    </portType>
021:    <binding name="SimpleTestIFBinding" type="tns:SimpleTestIF">
022:        <operation name="doStuff">
023:          <input>
024:            <soap:body
025:             encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
026:             use="encoded"
027:            namespace="http://org.javapitfalls.item48/wsdl/SimpleTest"
028:            />
029:          </input>
030:          <output>
031:            <soap:body
032:             encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
033:             use="encoded"
034:            namespace="http://org.javapitfalls.item48/wsdl/SimpleTest"
035:             />
036:            </output>
037:            <soap:operation soapAction="" />
038:        </operation>
039:       <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
040:                    style="rpc" />
041:    </binding>
042:    <service name="SimpleTest">
043:     <port name="SimpleTestIFPort" binding="tns:SimpleTestIFBinding">
044:      <soap:address xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
045:       location="http://localhost:8080/simpletest-jaxrpc/simpletest" />
046:     </port>
047:    </service>
048: </definitions>
```

Listing 48.2 *(continued)*

Next, we will get to the meat of this pitfall: writing different clients that will call the doStuff() method on the "SimpleTest" Web service. In the next sections, we show different approaches to building JAX-RPC clients.

A Simple Client That Uses Precompiled Stub Classes

The first, and easiest, way to call an RPC-style Web service is by using precompiled stubs. To create these stubs, the Java Web Services Developer Pack contains a tool called "wscompile." As a result, a client can communicate with the Web service interface using the java.xml.rpc.Stub interface. The wscompile tool is run against a configuration file listing details about the Web services (the URL of the WSDL, the package name, etc). When the wscompile tool runs successfully, it processes the WSDL for our Web service and generates proxy stubs so that our client can invoke methods on our SimpleTestIF interface at runtime.

Listing 48.3 shows a client that uses precompiled stubs. Lines 37 to 40 show the static createProxy() method that returns the stub that is cast to the SimpleTestIF interface in line 16. As a result, you do not have to know anything about SOAP or WSDL. Instead, you write code like you're using RMI. Note that in lines 27 and 28, we are printing out the invocation setup time. This will be used later in this pitfall item to compare pre-invocation times with our other techniques.

```
001: package org.javapitfalls.item48;
002:
003: import javax.xml.rpc.*;
004: import javax.xml.namespace.*;
005:
006: public class NoDynamicStuffClient
007: {
008:    public static void main(String[] args)
009:    {
010:       try
011:        {
012:            long initial, afterproxy, preInvokeTime, invokeTime;
013:
014:            initial = System.currentTimeMillis();
015:
016:            SimpleTestIF simpletest = (SimpleTestIF)createProxy();
017:
018:            afterproxy = System.currentTimeMillis();
019:            preInvokeTime = afterproxy - initial;
020:
021:            //Now, invoke our method
022:
023:            String response =
024:             simpletest.doStuff("Hi there from NoDynamicStuffClient!");
025:            //Print out stats
026:
```

Listing 48.3 A simple client using precompiled stubs

```
027:            System.out.println("Invocation setup took "
028:                    + preInvokeTime + " milliseconds.");
029:
030:        }
031:        catch ( Exception ex )
032:        {
033:            ex.printStackTrace();
034:        }
035:    }
036:
037:    private static Stub createProxy()
038:    {
039:        return(Stub)(new SimpleTest_Impl().getSimpleTestIFPort());
040:    }
041:}
```

Listing 48.3 *(continued)*

A Client That Uses Dynamic Proxies for Access

JAX-RPC includes the concept of using dynamic proxies—a second way for clients to access Web services. A *dynamic proxy class* is a class that implements a list of interfaces specified at runtime, and using this technique, does not require pregeneration of the proxy class. Listing 48.4 shows an example of building such a proxy. In line 30, we create a new instance of `ServiceFactory`. We then specify the service in lines 32 and 33 by passing the URL for our WSDL in the example, as well as the `javax.xml.namespace.QName`, which represents the value of a qualified name as specified in the XML Schema specification. By calling the `getPort()` method on our `javax.xml.rpc.Service` class on lines 35 to 37, we have generated a proxy class that is cast to our original interface class from Listing 48.1.

```
001: package org.javapitfalls.item48;
002:
003: import java.net.URL;
004: import javax.xml.rpc.Service;
005: import javax.xml.rpc.JAXRPCException;
006: import javax.xml.namespace.QName;
007: import javax.xml.rpc.ServiceFactory;
008:
009: public class DynamicProxyClient
010: {
011:
```

Listing 48.4 A simple client using dynamic proxies *(continued)*

```
012:    public static void main(String[] args)
013:    {
014:        try
015:        {
016:
017:            long initial, afterproxy, preInvokeTime, invokeTime;
018:
019:            initial = System.currentTimeMillis();
020:
021:            String UrlString =
022:             "http://localhost:8080/simpletest-
jaxrpc/simpletest?WSDL";
023:            String nameSpaceUri =
024:                 "http://org.javapitfalls.item48/wsdl/SimpleTest";
025:            String serviceName = "SimpleTest";
026:            String portName = "SimpleTestIFPort";
027:
028:            URL WsdlUrl = new URL(UrlString);
029:
030:            ServiceFactory serviceFactory =
031:                 ServiceFactory.newInstance();
032:            Service simpleService =
033:                     serviceFactory.createService(WsdlUrl,
034:                              new QName(nameSpaceUri, serviceName));
035:            SimpleTestIF myProxy = (SimpleTestIF)
simpleService.getPort(
036:                                  new QName(nameSpaceUri, portName),
037:                       org.javapitfalls.item48.SimpleTestIF.class);
038:
039:            afterproxy = System.currentTimeMillis();
040:            preInvokeTime = afterproxy - initial;
041:
042:            String response = myProxy.doStuff(
043:                                "Hello from Dynamic Proxy..");
044:
045:            //Print out stats
046:            System.out.println("Invocation setup took "
047:                                + preInvokeTime + " milliseconds.");
048:
049:        }
050:        catch ( Exception ex )
051:        {
052:            ex.printStackTrace();
053:        }
054:    }
055: }
```

Listing 48.4 *(continued)*

It is important to realize that this example requires a copy of the compiled Java interface class that we created (in order to cast it on line 35), but it does not require any precompilation steps. Where our precompilation process for our first example involved downloading the WSDL and creating the stubs before runtime, the dynamic proxy method does everything at runtime. This convenience will come at a cost. On line 46 of Listing 48.4, we print out our invocation setup time.

Two Clients Using the Dynamic Invocation Interface (DII)

A client can call a Web service using the Dynamic Invocation Interface (DII). The `javax.xml.rpc.Call` interface provides support for the dynamic invocation of an operation on a target service endpoint. In this section, we demonstrate two examples. In the first example, shown in Listing 48.5, we know where our Web service is, and we know the methods of our Web service. We will simply create our `Call` object and invoke the `doStuff()` method.

In lines 28 to 30 of Listing 48.5, we create our `Service` object. In line 35, we create the `Call` object by passing the port name (which is the qualified name for `"SimpleTestIF"` set up on line 32). In lines 36 to 55, we create our call by setting properties, setting our return types, and setting up our parameter information. Finally, on line 62, we perform the invocation.

```
001: package org.javapitfalls.item48;
002:
003: import javax.xml.rpc.*;
004: import javax.xml.namespace.*;
005:
006: public class DIIClient
007: {
008:
009:
010:    public static void main(String[] args)
011:    {
012:        String endpoint =
013:            "http://localhost:8080/simpletest-jaxrpc/simpletest";
014:        String servicenamespace =
015:               "http://org.javapitfalls.item48/wsdl/SimpleTest";
016:        String encodingStyleProperty =
017:               "javax.xml.rpc.encodingstyle.namespace.uri";
018:
019:        try
020:        {
021:
022:            long initial, afterproxy, preInvokeTime, invokeTime;
023:
```

Listing 48.5 A simple client using DII hard-coded calls *(continued)*

```
024:            initial = System.currentTimeMillis();
025:
026:            //Set up the service, giving it the service name
027:            //and the port (interface)
028:            ServiceFactory factory = ServiceFactory.newInstance();
029:            Service service = factory.createService(
030:                                    new QName("SimpleTest")
031:                                  );
032:            QName port = new QName("SimpleTestIF");
033:
034:            //Set up the call & the endpoint..
035:            Call call = service.createCall(port);
036:            call.setTargetEndpointAddress(endpoint);
037:
038:            //Set up the Call properties...
039:            call.setProperty(Call.SOAPACTION_USE_PROPERTY,
040:                                 new Boolean(true)
041:                                 );
042:            call.setProperty(Call.SOAPACTION_URI_PROPERTY, "");
043:            call.setProperty(encodingStyleProperty,
044:                    "http://schemas.xmlsoap.org/soap/encoding/");
045:
046:            QName qnametype =
047:               new QName("http://www.w3.org/2001/XMLSchema","string");
048:            call.setReturnType(qnametype);
049:
050:            //Set up the operation name & parameter...
051:            call.setOperationName(
052:                          new QName(servicenamespace, "doStuff"));
053:
054:           call.addParameter("String_1", qnametype, ParameterMode.IN);
055:            String[] params = { "Hello from DII Client!"};
056:
057:            afterproxy = System.currentTimeMillis();
058:            preInvokeTime = afterproxy - initial;
059:            System.out.println("Invocation setup took " +
060:                                   preInvokeTime + " milliseconds.");
061:
062:            String response = (String)call.invoke(params);
063:         }
064:        catch ( Exception ex )
065:        {
066:            ex.printStackTrace();
067:        }
068:    }
069:}
```

Listing 48.5 *(continued)*

Our example in Listing 48.5 requires no precompilation or WSDL lookups. Because we knew the information in advance, we could hard-code it into our application. On line 59, we made sure to print out the milliseconds for invocation setup.

Our final client code example is another example of using DII. DII is quite powerful, because you can dynamically determine your interfaces at runtime, looking at the WSDL, and discover information about the operations that a Web service supports. Listing 48.6 is such an example. Like the last example, using DII is convenient because it requires no precompilation of code or local Java interfaces at compile time. Everything with DII occurs at runtime.

In this example, we ask the service for the first method, and we simply invoke the method. In our example, we know the method takes a string as a parameter. Because we pass the URL of the WSDL in lines 31 to 33 of Listing 48.6, our `Service` object will have the ability to download the WSDL and get call information for the Web service. In line 38, we ask the Web service for the names of available calls. On line 48, we request the first call available, which we know is the method we would like to invoke. Finally, we set up the parameter on line 52, and we invoke the method on line 63.

```
001: package org.javapitfalls.item48;
002:
003: import java.net.*;
004: import javax.xml.rpc.*;
005: import javax.xml.namespace.*;
006:
007: public class DIILookupClient
008: {
009:
010:
011:    public static void main(String[] args)
012:    {
013:        try
014:        {
015:          long initial, afterproxy, preInvokeTime, invokeTime;
016:
017:          initial = System.currentTimeMillis();
018:
019:          String UrlString =
020:           "http://localhost:8080/simpletest-jaxrpc/simpletest?WSDL";
021:          String nameSpaceUri =
022:             "http://org.javapitfalls.item48/wsdl/SimpleTest";
023:          String serviceName = "SimpleTest";
024:          String portName = "SimpleTestIFPort";
025:          String qnamePort = "SimpleTestIF";
026:
027:          URL wsdlUrl = new URL(UrlString);
```

Listing 48.6 A simple client using DII with lookups *(continued)*

```
028:
029:          ServiceFactory serviceFactory =
030:                 ServiceFactory.newInstance();
031:          Service simpleService =
032:                  serviceFactory.createService(wsdlUrl,
033:                      new QName(nameSpaceUri,serviceName));
034:
035:          QName port = new QName(nameSpaceUri, portName);
036:
037:
038:          Call[] servicecalls = simpleService.getCalls(port);
039:          Call mycall = null;
040:
041:          /*
042:           * We will assume that we know to call the first method &
043:           * pass it a string..
044:           */
045:
046:          if ( servicecalls != null )
047:          {
048:              mycall = servicecalls[0];
049:              String operationName =
050:                  mycall.getOperationName().toString();
051:
052:              String[] params = { "Hello from DII Client!"};
053:
054:
055:              afterproxy = System.currentTimeMillis();
056:              preInvokeTime = afterproxy - initial;
057:
058:              System.out.println("Invocation setup took "
059:                                  + preInvokeTime + " milliseconds.");
060:              System.out.println("About to call the "
061:                                   + operationName + " operation..");
062:
063:              String response = (String)mycall.invoke(params);
064:              System.out.println("Received '"
065:                                   + response + "' as a response..");
066:
067:
068:          }
069:          else
070:
071:              System.out.println("Problem with DII command..");
```

Listing 48.6 *(continued)*

```
072:           }
073:
074:         }
075:         catch ( Exception ex )
076:         {
077:             ex.printStackTrace();
077:         }
078:     }
079: }
```

Listing 48.6 *(continued)*

Everything in this example used the capabilities of DII, getting call information at runtime. Many more methods on the `javax.xml.rpc.Call` interface could be used, but for the purpose of simplicity, this example doesn't use them all. For more information about the JAX-RPC API and specification, go to Sun's JAX-RPC Web page at http://java.sun.com/xml/jaxrpc/.

Performance Results

For our experiment, we deployed our Web service and used the code from Listings 48.3, 48.4, 48.5, and 48.6. Our experiment was run with the Web service and the client running on a 1300-MHz Pentium IV PC with 1 GB RAM, using the Windows 2000 operating system. Each "run" was done by running each client, in random order, one after another. The client VM was Java HotSpot Client VM (build 1.4.0-b92, mixed mode), and the tools and Web services were running with the Java Web Services Developer's Pack, version 1.0_01. Table 48.1 shows the results.

As you can see, there is an obvious trade-off between flexibility at runtime and performance. The best performing clients (with respect to invocation setup) were those in Listings 48.3 and 48.5. Our hard-coded DII example in Listing 48.5 matched the performance of our precompiled stubs example. This makes sense, because no WSDL lookup was necessary in the hard-coded example.

The worst performance for call setup revolved around the clients where calls were determined at runtime, or where stubs were generated at runtime. The DII with call lookups example in Listing 48.6 and the dynamic proxy example in Listing 48.4 were approximately 2.5 times slower in performance for pre-invocation setup. Both of those clients downloaded the WSDL. The dynamic proxy example was slightly slower because of the generation of the stubs on the fly, but they were quite similar in speed.

Table 48.1 Table of Results: Pre-invocation Times

	LISTING 48.3 (PRECOMPILED STUBS)	LISTING 48.4 (DYNAMIC PROXY GENERATION)	LISTING 48.5 (DII HARD-CODED)	LISTING 48.6 (DII - WITH CALL LOOKUPS)
Run 1	590 ms	1683 ms	591 ms	1642 ms
Run 2	601 ms	1683 ms	581 ms	1653 ms
Run 3	591 ms	1672 ms	591 ms	1643 ms
Run 4	591 ms	1672 ms	581 ms	1633 ms
Run 5	591 ms	1783 ms	581 ms	1643 ms
Run 6	601 ms	1663 ms	591 ms	1653 ms
Run 7	591 ms	1692 ms	601 ms	1642 ms
Run 8	591 ms	1713 ms	601 ms	1663 ms
Run 9	591 ms	1672 ms	590 ms	1662 ms
Run 10	591 ms	1682 ms	641 ms	1662 ms
Run 11	641 ms	1662 ms	581 ms	1632 ms
Run 12	591 ms	1683 ms	581 ms	1642 ms
Run 13	591 ms	1722 ms	591 ms	1642 ms
Run 14	581 ms	1692 ms	631 ms	1642 ms
Run 15	591 ms	1682 ms	591 ms	1692 ms
Average Time	594.9333333 ms	1690.4 ms	594.9333333 ms	1649.733333 ms

Conclusion

Our simple example has shown that there is obviously a trade-off between the dynamic features of JAX-RPC and performance. Imagine if we needed to write a client application for working with multiple Web services, or a complex Web service with pages and pages of WSDL. The performance implications of doing everything dynamically—just because those techniques are available—would be awful. Just because you *can* do things dynamically doesn't necessarily mean you *should* do things dynamically with JAX-RPC. If there is a reason, do it. When in doubt, use precompiled stubs—your code will look prettier—and if your Web service's interface changes, you will simply need to recompile your client.

Item 49: Get Your Beans Off My Filesystem!

It is easy to overlook best practices for J2EE because some habits of using the Java 2 Standard Edition die hard. When designing Enterprise JavaBeans solutions, it is sometimes easy to forget what type of programming should *not* be done. We know, for example, that spawning threads within beans is a big no-no, because we understand that the EJB container handles all threading issues. Oftentimes, however, we overlook the pitfalls that await us when we use the *java.io* classes to access files in the filesystem.

To demonstrate this issue, we provide a very simple example of a session bean reading properties from the filesystem. In our example, a session bean is used to calculate the amount of tax that shoppers must pay when they proceed to check out. In doing so, the two-character abbreviation for the shoppers' state is passed to the bean, so the bean can calculate the approximate sales tax. Because sales tax varies from state to state, the bean reads from a local properties file with different sales tax percentages:

```
Salestax.AB=.04
Salestax.VA=.045
```

For brevity (and to do a simple demonstration of this pitfall), we are only listing two states. In our example, we assume that if a state is not in this properties file, the state has no sales tax. Listing 49.1 shows the code for our `BadTaxCalculatorBean` that reads the properties file. A client calling this session bean passes the purchase amount (`double cost`) and the two-letter state abbreviation (`String state`), and the tax on the purchase amount is returned. We load the properties file in our session bean's `calculateTax()` method, and we calculate the sales tax and return it as a `double`. In lines 18 to 31, we load the properties file using the `java.io.FileInputStream` class and calculate the tax:

```
001: package org.javapitfalls.item49;
002:
003: import java.rmi.RemoteException;
004: import javax.ejb.SessionBean;
005: import javax.ejb.SessionContext;
006: import java.util.Properties;
007: import java.io.*;
008: import javax.ejb.EJBException;
009:
010:
011: public class BadTaxCalculatorBean implements SessionBean
012: {
013:     public double calculateTax(double cost, String state)
014:    {
015:        Properties p = new Properties();
016:        double tax = 0;
```

Listing 49.1 Bean reading properties file for values *(continued)*

```
017:
018:       try
019:       {
020:          p.load(new FileInputStream("C://salestax.properties"));
021:         tax = Double.parseDouble(p.getProperty("Salestax." + state)) *
022:               cost;
023:
024:       }
025:       catch ( IOException e )
026:       {
027:          e.printStackTrace();
028:
029:          throw new EJBException("Can't open the properties file! "
030:                                     + e.toString());
031:       }
032:
033:       String taxString = p.getProperty("Salestax." + state);
034:
035:       if ( taxString != null && !taxString.equals("") )
036:       {
037:             tax = Double.parseDouble(taxString) * cost;
038:       }
039:
040:       return (tax);
041:      }
042:
043:    public void ejbCreate() {}
044:    public void ejbPostCreate() {}
045:    public void ejbRemove()
046:    public void ejbActivate()
047:    public void ejbPassivate()
048:    public void setSessionContext(SessionContext sc) {}
049: }
```

Listing 49.1 *(continued)*

What could go wrong in Listing 49.1? First of all, loading a properties file every time the `calculateTax()` method is called is bad practice. This is a given. More importantly, because the container is responsible for management of this session bean, who knows how many `BadTaxCalculator` beans may be actually instantiated during heavy loads on the server? When an EJB uses the `java.io` package to access files on the filesystem, bad things could happen, ranging from very poor performance to running out of file descriptors and bringing down the server.[5]

[5] Van Rooijen, Leander. "Programming Restrictions in EJB Development: Building Scalable and Robust Enterprise Applications." *Java Developers Journal*. Volume 7, Issue 7, July 2002.

For this reason, the EJB specification lists programming restrictions related to the java.io classes. In the "Runtime Environment" chapter of the specifications (EJB specifications 1.1, 2.0, and 2.1), the restriction is listed plainly: "An enterprise bean must not use the `java.io` package to access files and directories in the filesystem. The filesystem APIs are not well-suited for business components to access data. Business components should use a resource manager API, such as the JDBC API, to store data."[6] Simply put, because a filesystem is not transactional, and because there is no resource manager involved in `java.io` operations, you need to keep your beans off the filesystem! This presents us with a challenge: If a bean compiles and deploys into our EJB container without any problems, and it seems to work well when we test it, fatal errors may end up diagnosing the problem.

For our sales tax example, how should we rewrite it? If we store all the sales tax information in a database, this would eliminate the possible problems that we could encounter when using the `java.io` package. Unfortunately, using a database in this example may be overkill because sales tax usually doesn't change very often, and we could pay a performance penalty for hitting the database every time. In our bad example, we used properties files because they are convenient for configuring our Java applications. Luckily, we can do something similar—with our deployment descriptor.

To customize your beans at runtime, you can use the environment properties in your deployment descriptor. For data that doesn't change often, and for an alternative to using a database for storing information, environment properties work well. Listing 49.2 shows our deployment descriptor for our new bean, in a file called ejb-jar.xml. Each entry, designated by the XML tag `<env-entry>`, contains a `<description>` tag, a name designated by `<env-entry-name>`, a type designated by `<env-entry-type>`, and a value designated by `<env-entry-value>`. As you can see from our listing of this deployment descriptor, we converted the original properties file to this format.

```
001: <?xml version="1.0" encoding="UTF-8"?>
002: <ejb-jar>
003:  <description>GoodTaxCalculatorBean</description>
004:  <display-name>GoodTaxCalculatorBean</display-name>
005:  <enterprise-beans>
006:   <session>
007:    <ejb-name>GoodTaxCalculator</ejb-name>
008:    <home>org.javapitalls.item49.GoodTaxCalculatorHome</home>
009:    <remote>org.javapitfalls.item49.GoodTaxCalculator</remote>
010:    <ejb-class>
011:            org.javapitfalls.item49.GoodTaxCalculatorBean
012:    </ejb-class>
013:    <session-type>Stateless</session-type>
014:    <transaction-type>Bean</transaction-type>
015:    <env-entry>
016:       <description>Alabama Sales Tax</description>
```

Listing 49.2 Setting environment entries in ejb-jar.xml *(continued)*

[6] Enterprise JavaBeans Specification 2.1, Chapter 25; Enterprise JavaBeans Specification 2.0, Chapter 20; Enterprise JavaBeans Specifications 1.1, Chapter 18.

```
017:       <env-entry-name>Salestax.AB</env-entry-name>
018:       <env-entry-type>java.lang.String</env-entry-type>
019:       <env-entry-value>.04</env-entry-value>
020:     </env-entry>
021:     <env-entry>
022:       <description>Virginia Sales Tax</description>
023:       <env-entry-name>Salestax.VA</env-entry-name>
024:       <env-entry-type>java.lang.String</env-entry-type>
025:       <env-entry-value>.04</env-entry-value>
026:     </env-entry>
027:   </session>
028:  </enterprise-beans>
029: </ejb-jar>
```

Listing 49.2 *(continued)*

To access these properties, the bean must do a JNDI lookup. Listing 49.3 demonstrates this approach. On lines 18 and 19, the session bean gets the initial context and uses it to look up the environment entries. These entries are always found under JNDI in `java:comp/env`. Finally, the lookup for the sales tax for a certain state is done by looking up the value of the current state's sales tax on line 20.

```
001: package org.javapitfalls.item49;
002:
003: import java.rmi.RemoteException;
004: import javax.ejb.EJBException;
005: import javax.ejb.SessionBean;
006: import javax.ejb.SessionContext;
007: import javax.naming.*;
008:
009: public class GoodTaxCalculatorBean implements SessionBean
010: {
011:    public double calculateTax(double cost, String state)
012:    {
013:        double tax = 0;
014:
015:        try
016:        {
017:
018:            Context ctx = new InitialContext();
019:            Context env = (Context)ctx.lookup("java:comp/env");
020:           String taxString = (String)env.lookup("Salestax." + state);
021:
022:            if ( taxString != null && !taxString.equals("") )
023:            {
```

Listing 49.3 Using environment entries in our bean

```
024:                tax = Double.parseDouble(taxString) * cost;
025:            }
026:
027:        }
028:        catch ( NamingException ne )
029:        {
030:            ne.printStackTrace();
031:            // Instead of throwing an EJBException, let's just assume
032:            // there is no tax!
033:        }
034:
035:        return(tax);
036:    }
037:
038:    public void ejbCreate() {}
039:    public void ejbPostCreate() {}
040:    public void ejbRemove() {}
041:    public void ejbActivate() {}
042:    public void ejbPassivate() {}
043:    public void setSessionContext(SessionContext sc) {}
044:
045: }
```

Listing 49.3 *(continued)*

In this pitfall, we discussed the potential problems our beans may encounter if they use the `java.io` package to access files and directories on the filesystem. We demonstrated a potentially flawed session bean that loaded a properties file with the `java.io.FileInputStream` class. We listed the programming restriction from the EJB specification and discussed what could go wrong. Finally, we modified our example and provided an alternative to beans loading properties files by using environmental entries in the deployment descriptor. As you are looking through your code base, you may want to look for this potential problem. If you see any code where beans are loading properties files, it will be an easy fix to switch to environment entries.

Item 50: When Transactions Go Awry, or Consistent State in Stateful Session EJBs

Many pitfalls are based on erroneous assumptions—assumptions that, at least on the surface, appear perfectly reasonable. As EJB developers become more educated, one of the first areas they branch out into is transactions. In one way or another, one of their EJBs becomes part of a transaction, and the developer feels confident that his or her code is stable and works correctly because, after all, when a transaction rolls back, the data is rolled back to a consistent state. Herein lies the pitfall; the developer *assumed* that the state of his or her EJB was rolled back when the transaction failed.

As you will recall, stateful session EJBs are those EJB components that maintain state between method calls. We can easily imagine such a bean in an example of a police call center. When you call the police operator, he or she remembers the context of your call. In such an example, each question to the call center represents a method invocation on the "call center" EJB. The assumption that EJB developers made was that when the transaction rolled back, the state of all the data within their EJB was restored to its pretransaction state. On the surface, it seems perfectly reasonable.

The problem with stateful session beans and transactions comes from the interaction of bean state and transaction state. The pitfall with session beans comes about as a result of assuming that the bean state is reset to correspond to the pretransaction state when a transaction fails.

Imagine a stateful session bean that represents a bank teller. A client speaks to the bank teller to make a withdrawal. The teller notes that a withdrawal has been requested (transaction start), attempts to perform the withdrawal (method during transaction), and then, depending on whether the teller's cash drawer has enough cash, either commits the transaction (hands cash to the client) or rolls back the transaction ("sorry, I don't have enough cash on hand"). A problem comes about if the client's passbook has already been updated to show the withdrawal. The client's passbook shows the withdrawal, but no cash was given (an inconsistency between the transaction state and the client state).

To solve the problem of inconsistent state within an EJB, we can implement the `SessionSynchronization` interface on our stateful session EJBs. The `SessionSynchronization` interface requires you to implement three methods: `beforeCompletion()`, `afterCompletion()`, and `afterBegin()`. Figure 50.1 shows the lifecycle of stateful session EJBs. Understanding the lifecycle will clarify exactly how we can use these methods to solve our state problem.

NOTE Note that the session synchronization interface is designed to be used with container-managed transactions. Obviously, if you are managing your own transaction state, you would know where transaction boundaries exist and you could store and reset the EJB state yourself.

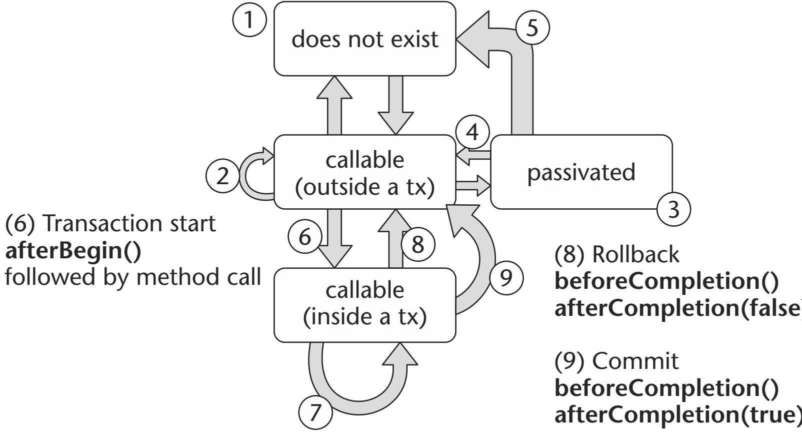


Figure 50.1 Stateful session EJB lifecycle.

Examining how a bean moves through its various states during its lifecycle will help us understand how transactions affect bean state and how we can solve the transaction state problem.

1. Initially, an EJB is in the *does not exist* state. In the *does not exist* state, a stateful bean has not been called for (often referred to as *pooled*) but is potentially ready for quick creation and use. Various application servers handle this state in different ways. Many, including BEA WebLogic Server, precreate beans so that they are ready for immediate use.
2. A bean moves into the *callable* state when a client calls the home interface `create()` method. In the *callable (outside a transaction*) state, a bean has been created and is ready for use. Beans in the callable state are method ready and stay in the method state until they are destroyed, or *passivated*.
3. Beans move to the *passivated* state because they have been inactive for a long period of time. Passivated beans are normally stored in some sort of high-speed backing store outside RAM to conserve system memory.
4. A bean can move from the *passivated* state back to callable as a result of a client calling a method on the bean. Under normal circumstances, *passivated* beans are restored without any developer work or interaction. However, the EJB specification provides methods for notifying a bean that it has been *passivated* and then restored.
5. If a bean is inactive long enough, it may be destroyed by the bean container. Further method calls on such a bean will result in a client exception. Many application servers will reset the state of such an object and return it to a ready-for-use object pool.
6. A bean moves to the *callable (inside a transaction)* state the first time a method is called after a transaction is started. The session synchronization interface method `afterBegin()` is called at this point, but before the method call is evoked, to signal the bean that a transaction has been started. We will use `afterBegin()` to signal we should store our state for reset later.
7. After a transaction has been started, all bean method calls become part of that transaction until either a commit or a rollback occurs. As long as the transaction is active, we remain in this state.
8. If a transaction is rolled back, the bean moves to the *callable (outside a transaction)* state. The `beforeCompletion()` method is called, followed by `afterCompletion(false)` being called. The bean could then revert its state to a known point before the transaction began.
9. When a transaction is committed, the bean moves from back to the *callable (outside a transaction)* state. However, unlike item 8 above, the transaction was committed and the `afterCompletion(true)` method is called. Since the session state is consistent with the transaction state, no action need be taken.

We can solve the transaction/bean state problem by a combination of the `SessionSynchronization` interface and careful store and reset logic within our application. The `SessionSynchronization` interface requires that we implement three specific methods:

- `public void afterBegin()`. Represented by transition 6 in the EJB lifecycle diagram. Here is where we would implement any logic to store current state. In the EJB lifecyle, `afterBegin()` is called after a transaction has begun but before any methods that would be part of the transaction. Transaction context can be obtained and manipulated within the `afterBegin()` method.
- `public void beforeCompletion()`. Represented by transitions 8 and 9 in the EJB lifecycle diagram. `beforeCompletion()` is called when a transaction is committed but before the actual commit operation takes place. Any work done within `beforeCompletion()` is within the context of the transaction.
- `public void afterCompletion(boolean committed)`. Represented by transaction 8 (failed) and 9 (committed) in the EJB lifecycle diagram. `afterCompletion()` is called after the transaction has been committed or rolled back. The `committed` flag can be used to determine if the transaction was successfully committed. Here is where we would implement any required logic to store or update bean state. An important note: `afterCompletion()` is *not* called inside the context of a transaction.

NOTE A complete description of Stateless Session EJB is beyond the scope of this text. Only the implementation of stateless session EJBs is presented in this section. For complete coverage of EJB development, see *Mastering Enterprise JavaBeans, Second Edition* by Ed Roman (Wiley, 2002) or *Developing Java Enterprise Applications* by Stephen Asbury and Scott R. Weiner (Wiley, 2001).

In Listing 50.1 we can take advantage of transaction boundaries by specifying the session synchronization interface, as shown on line 7, and implementing the required methods. To solve our problem, we must save our current state, have all methods operate within transaction boundaries, and then be able to determine if our transaction succeeded or failed and restore state appropriately.

The `afterBegin()` method, shown on lines 28 to 32, called before the first method invocation inside a transaction, allows us to save our current state. All methods, which are transaction-aware, are then within transaction boundaries. Line 34 shows the before completion method, which is unused in our example but could be used for final cleanup, to reset transaction state (to rollback only for example). Once the transaction has been committed, we need to either reset from our prior state (transaction failed) or simply continue on (transaction succeeded). Lines 39 to 52 show the implementation of the `afterCompletion()` method and allow us to correctly reset our bean state when a transaction either commits or fails. The `boolean committed` flag tells us whether the transaction committed (we can discard our previous state) or failed (we need to restore our state).

```
01: package org.javapitfalls.item50;
02:
03: import javax.ejb.*;
04: import java.util.*;
05:
06: public class TellerBean implements SessionBean,
07:                                     SessionSynchronization
08: {
09:
10:     private SessionContext sc;
11:     private double balance;
12:     private double priorBalance;
13:
14:     public TellerBean() { balance = 0.0;}
15:     public void setSessionContext(SessionContext sc)
16:     {
17:       this.sc=sc;
18:     }
19:
20:     public void ejbCreate() { }
21:
22:     public void ejbRemove() { }
23:
24:     public void ejbPassivate() { }
25:
26:     public void ejbActivate() { }
27:
28:     public void afterBegin()
29:     {
30:       System.out.println("TellerBean::afterBegin");
31:       priorBalance = balance;
32:     }
33:
34:     public void beforeCompletion()
35:     {
36:       System.out.println("TellerBean::beforeCompletion");
37:     }
38:
39:     public void afterCompletion(boolean committed)
40:     {
```

Listing 50.1 TellerBean.java, using session synchronization *(continued)*

```
41:         System.out.println("TellerBean::afterCompletion ("
42:         + committed + ")");
43:         if (committed)
44:         {
45:           priorBalance=balance; // prior and current state match
46:         }
47:         else
48:         {
49:           balance=priorBalance; // restore state
50:         }
51:
52:       }
53:
54:       public double getBalance()
55:       {
56:         System.out.println("getBalance" + balance);
57:         return balance;
58:       }
59:
60:       public double Withdraw(double amount )
61:       {
62:         System.out.println("Withdraw" + amount);
63:         balance = balance - amount;
64:         return balance;
65:       }
66:
67:       public double Deposit(double amount )
68:       {
69:         System.out.println("Deposit" + amount);
70:         balance = balance + amount;
71:         return balance;
72:       }
73: }
```

Listing 50.1 *(continued)*

The Memento Pattern

Our previous scenario showed an example where we only needed to restore a simple set of variables. However, often in real life, we need to restore a complex set of data where setting and restoring variables can be a tedious and error-prone operation. To solve the restore problem, we can use a variant of the value object pattern known as the *Memento pattern*.

The Memento pattern is a rather simple pattern whose purpose is to store the state of an object. When using a memento, an originator, such as an EJB or other object that requires state, creates an instance of a memento and passes it to a caretaker, which manages the object's state on behalf of the originator. Typically, mementos hide the contents of the object from all but the originator of that object. In our case, the originator and caretaker are the same class; however, in a more complex system, we could easily envision an object monitor managing mementoes for us. When the originator needs to restore state, the caretaker provides the memento back to the originator, and the object can then restore.

The Memento pattern is actually a rather simple one to implement in Java. Using inner classes, as shown on lines 04 to 16 of Listing 50.2, we create a memento that contains two methods, a constructor and a restore method, which allows us to reset state using the contents of the moment.

In Listing 50.2, we use the memento by creating two instances representing current state and prior state, as shown in lines 19 to 21.

The value of the memento pattern is most clear when we need to contain a large amount of state. The actual object's state is contained wholly within the memento and not within the object itself. We access the object state throughout a set of state variables, `currentState` and `priorState`. Current state is used whenever we perform normal object operations such as getting and setting data. We store object state for later use into the `priorState` object whenever necessary. In the example, a simple method that changes the object's state, shown on lines 24 to 28, saves the current object's state contents in the `priorState` memento using the `setFrom` method. Likewise, line 31 shows how we might restore our state as required.

Our memento example is a rather simple one using predefined variables to contain an object's state. It's easy to imagine a more generic memento implementation that uses reflection to examine an object's content and save and restore state automatically without requiring object-specific code.

```
01: class MementoExample {
02:  Memento currentState = null;
03:  Memento priorState = null;
04:  private class Memento {
05:   int imaInt;
06:   double imaDouble;
07:   Memento(int i, double d) {
08:    imaInt = i;
09:    imaDouble = d;
10:   }
11:   void setFrom(Memento source) {
12:    imaInt=source.imaInt;
13:    imaDouble=source.imaDouble;
14:   }
15:
```

Listing 50.2 MementoExample.java *(continued)*

```
16:  }
17:
18:  public MementoExample() {
19:   currentState = new Memento(1,20.0);
20:   priorState = new Memento(0,0.0);
21:   priorState.setFrom(currentState);
22:   }
23:
24:  public void changeState(int i, double d) {
25:    priorState.setFrom(currentState);
26:    currentState.imaInt = i;
27:    currentState.imaDouble = d;
28:  }
29:
30:  public void restoreState() {
31:   currentState.setFrom(priorState);
32:  }
33:
34:  public void printState() {
35:   System.out.println("State:");
36:   System.out.println("    imaInt = " +
37:        currentState.imaInt);
38:   System.out.println("    imaDouble = " +
39:        currentState.imaDouble);
40:  }
41:  public static void main(String[] arg) {
42:   MementoExample mt = new MementoExample();
43:   System.out.println("Original state to 1,20.0");
44:   mt.printState();
45:   System.out.println("Changing state to 3, 7");
46:   mt.changeState(3,7.00);
47:   mt.printState();
48:   System.out.println("Restoring state");
49:   mt.restoreState();
50:   mt.printState();
51:
52:  }
53:
54: } // end MementoExample
```

Listing 50.2 *(continued)*

The `StateTellerBean` interface, shown in Listing 50.3, is a remote interface that extends the `EJBObject` class so that all the methods in it will be exposed to remote clients.

```
01: package org.javapitfalls.item50;
02:
03: import javax.ejb.EJBObject;
04: import java.rmi.RemoteException;
05:
06: public interface StateTellerBean extends EJBObject {
07:
08:   public double getBalance() throws RemoteException;
09:
10:   public double Withdraw(double amount ) throws RemoteException;
11:
13:   public double Deposit(double amount ) throws RemoteException;
14:
15: }
```

Listing 50.3 StateTellerBean.java

The `StateTellerBeanHome` interface, shown in Listing 50.4, extends the `EJBHome` class so that remote clients can access the application. Our application does not use this, but it is provided so that other applications can use it if they want to.

```
01: package org.javapitfalls.item50;
02:
03: import java.rmi.RemoteException;
04: import javax.ejb.CreateException;
05: import javax.ejb.EJBHome;
06:
07: public interface StateTellerBeanHome extends EJBHome {
08:
09:   StateTellerBean create() throws RemoteException, CreateException;
10:
11: }
12:
```

Listing 50.4 StateTellerBeanHome.java

Listed below is the output if the Memento pattern application is run. Notice the transitions between states and the restoration of the original values. This is the result of the Memento pattern implementation.

```
01: prompt> java org.javapitfalls.item50.MementoExample
02:
03:
04: Original state 1,20.0
05: State:
06:     imaInt = 1
07:     imaDouble = 20.0
08: Changing state 3, 7
09: State:
10:     imaInt = 3
11:     imaDouble = 7.0
12: Restoring state
13: State
14:     imaInt = 1
15:     imaDouble = 20.0
16:
```

As shown in the rather simple Memento application, the memento operation remembers the context of your call, which allows an application to roll back the state of all the data within an EJB.

Index

SYMBOLS AND NUMERICS

@ (at sign), 264
{ } braces, 262
[ ] brackets, 94
. (dot), 94
4 KB buffer size, 21
16-bit unicode characters, 18

A

abort() method, 393
AbstractCollection class, 162
abstracting details, bad example of, 39–40
AbstractList class, 162
AbstractSelectableChannel class, 18
AbstractSet class, 162
Abstract Windowing Toolkit (AWT), 122, 352
acceptChanges() method, 33
accept() method, 33, 152
AccessControl, 39–40
AccessException class, 41–42
Action event handler, 167
actionPerformed() method, 177–178
addMode() method, 173, 194
addOperation() method, 173
addType() method, 173
afterBegin() method, 434–436
afterCompletion() method, 434–435
allocateDirect() method, 26
AllTests.java example, 100
Another Neat Tool (ANT)
  BugRat tracking tool and, 92
  Checkstyle tool and, 94
  command-line build operations, 94
  CruiseControl and, 98
  database creation, destruction and population, 96–98
  Help targets, 91
  JavaNCSS utility and, 93
  JDepend utility and, 93
  overview, 88–89
  running JUnit tests using, 90
Apache Software Foundation (ASF), 391
applets, problems with, 227–231
application programming interfaces (APIs)
  collision of, 53–57
  Java Regular Expressions, 266
applications
  Java Web Start, 231–232
  servlet-based, 223
  XSLTransformFilter, 249
ArrayList class, 162
ASF (Apache Software Foundation), 391
assertions
  AssertionExample.java, 61
  enabling, 64–65
  errors, 60, 62–63
  how to use, 59
  invariants, 60
  postconditions, 60, 63–64
  preconditions, 60
  unreachable conditions, 63
at sign (@), 264
attachment() method, 31
attach() method, 31
authenticateHelper class, 239

authenticate() method, 40, 42–44
authentication, 236, 238
AWT (Abstract Windowing Toolkit), 122, 352
AxisConstants class, 352

B

BadDomLookup.java example, 67
BadExecJavac.java example, 5–6
BadLayeredPane.java example, 147–148
BadLoggerExample, 45–49
BadNavigationUtils.java example, 114–115
BadRightMouseButton.java example, 81–83
BadSetSize.java example, 123
BadStringTokenizer.java example, 141
BadURLPost.java example, 131–132
BadVoterServlet.java example, 292–293
BadWinRedirect.java example, 12
BasicService interface, 233
batch scripts, 98
BeanContextServicesSupport class, 162
BeanContextSupport class, 162
beans. *See* session beans
BEA Web Logic Server, 317
beforeCompletion() method, 434–435
begin() method, 393
BmpWriter3.java, 23–25
boolean isNew() method, 221
BorderLayout manager, 124
braces ({ }), 262
brackets ([ ]), 94
buffers
  BufferedReader class, 182
  ByteBuffer class, 26
  defined, 18
  DirectBuffer class, 26
BugRat tracking tool, 92
BulkResponse objects, 403
BusinessQueryManager interface, 401–402
ByteArrayOutputStream class, 182
ByteBuffer class, 26

C

cache.debug property, 203
cache.path property, 203
cache.unlimited property, 203
caching
  database/server architecture, 207
  generate.Tests.java example, 205–207
  JCACHE specification, 207
  OSCache tags, 201–203
CallableStatement interface, 364
Callback object, 31
canonical file copy operation, 20–21
catch() method, 295
Cewolf tag library, 348
CGI (Common Gateway Interface), 127, 210
Channel class, 17–18
character class operator precedence, 266
Charset class, 18
Checkstyle tool, 94
checksubmitcount() method, 315
ClassCastExceptions, 112
classes
  AbstractCollection, 162
  AbstractList, 162
  AbstractSelectableChannel, 18
  AbstractSet, 162
  AccessException, 41–42
  ArrayList, 162
  authenticateHelper, 239
  AxisConstants, 352
  BeanContextServicesSupport, 162
  BeanContextSupport, 162
  BufferedReader, 182
  ByteArrayOutputStream, 182
  ByteBuffer, 26
  Channel, 18
  Charset, 18
  CommandListener, 175
  ConsoleHandlers, 51–52
  ControlServlet, 237
  DatagramChannel, 18
  DDConnectionBroker, 281, 285
  DetailsDialog, 171
  DirContext, 241
  DirectBuffer, 26
  DocumentBuilderFactory, 127
  DomUtil, 71
  EJBObject, 440
  File, 151–154
  FileChannel, 17, 21, 25, 152
  FileDescriptor, 152
  FileFilter, 152
  FileHandler, 48, 52
  FileInputStream, 17, 152
  FileLock, 153
  FileOutputStream, 17, 152
  FilePermission, 152
  FirstCallSingleton, 118
  GatheringByteChannel, 17
  HashSet, 162
  HTTPConnection, 137
  HTTPResponse, 137
  HttpSession, 221

HttpURLConnection, 133
InitialDirContext, 243
Iterator, 157–158, 161–162
java.lang.Throwable, 41
java.sql.DatabaseMetaData, 353
JButton, 165
JComboBox, 165
JEditorPane, 298
JFrame, 149, 151
JLayeredPane, 146–150
JList, 165
JPanel, 151
JRootPane, 148
JScrollPane, 298
JTextArea, 171
JTextField, 298
LDAPException, 41
LinkedHashSet, 162
LinkedList, 162
Matcher, 266
NIO packages, 18–19
org.w3c.dom, 53
org.xml.sax, 53
OutputStream
  Runtime.exe() method, 15
  writeInt() method, 23
OutputStreamWrite, 137
Pattern, 266
Pipe.SinkChannel, 18
Pipe.SourceChannel, 18
PixelGrabber, 25
PrintWriter, 273, 294
Process
  exitValue() method, 4
  getInputStream() method, 11
  waitFor() method, 5
ProtocolException, 133
RandomAccessFile, 17, 152
RegistryObject, 403
Runtime, 4
ScatteringByteChannel, 17
SelectableChannel, 18
SelectionKey, 31, 33
Selector, 18
ServerSocket, 17, 30
ServerSocketChannel, 18, 30
ServletOutputStream, 294
Socket, 17
SocketChannel, 17–18
StreamGobbler, 11
StringBuffer, 172
StringTokenizer, 4, 140, 142, 146
SwingUtilities, 88
TreeSet, 162
URLConnection, 127, 134
URLConnectionFactory, 127
Vector, 162
XMLOutputter, 80
XSLTFilter, 252
XulRunner, 146
See also interfaces
class loading, 111–112
ClassNotFoundExceptions, 112
CLASSPATHS, 108–111
ClipboardService interface, 233
cloneNode() method, 58–59
clustering, 335
code. *See* listings
commandAction() method, 177
CommandListener class, 175
commands
  dir
    exec() method, 8
    GoodWindowsExec.java, 11
  java classname, 108
  java -r jarname, 108
  SQL, 96
  See also methods
commit() method, 393
committed flags, 436
Common Gateway Interface (CGI), 127, 210
compile-time errors, 251
componentization, 335
concurrent result sets, multiple, problems with, 353, 355, 357–359
Concurrent Versioning System (CVS), 92
conditional constructs, 266
Connection Pool resource, 383
connections
  DDConnectionBroker class, 281, 285
  freeConnection() method, 285
  getConnection() method, 285, 288
  m_broker.freeConnection() method, 281
  m_broker.getConnection() method, 281
  pooling, 282
  within servlets, design flaws, 278–279, 281, 285, 288
ConsoleHandlers class, 51–52
constructs, 266
ControllerServlet class, 237
convert() method, 341
cookies, 210, 221
-cp switch, 109
create() method, 341, 366–367, 369
createNewFile() method, 152
createProxy() method, 420

createURL() method, 194
CruiseControl tool, 98
currentState variable, 439
CVS (Concurrent Versioning System), 92

D

DatabaseAccessControl, 40
databases, creation, destruction and population, 96–98
data corruption, time line of, 275
DatagramChannel class, 18
date formats, 264
dbQueryBean.java example, 105
dbQueryCase.java example, 105–107
DDConnectionBroker class, 281, 286
DefaultContext element, 383
delete() method, 152, 155, 157
depends tag, 89
deploying Java applications, 109–110
Design by Contract utility, 59–60
The Design of Everyday Things (Donald Norman), 127
destroy() method, 175
DetailsDialog class, 171
DII (Dynamic Invocation Interface), 423
dir command
  exec() method, 8
  GoodWindowsExec.java, 11
DirContext class, 242
DirectBuffer class, 26
directives
  forward, 321
  include, 256, 258
  See also tags
Display.getdisplay() method, 176
Display.setCurrent() method, 177
doAmazonSearch() method, 180, 191–192
.doc extensions, 11
DocumentBuilderFactory class, 127
Document.cloneNode() method, 59
DocumentLS interface, 77
Document Object Model (DOM)
  DOMBuilder, 78
  DOMimplementation, 78
  DOM tree, searching, 66–70
  DOMWriter, 78
  JDOM (Java Document Object Model), 79
  level 3 specifications, 76–77
  lifecycle of, 73
  Modified state, 74
  New state, 74
  overview, 170
  Persisted state, 74
  W3C Load specification, 80
doFilter() method, 245, 252
doGet() method, 222, 272, 292
DomEditor utility, 74
DomUtil class, 71
doPost() method, 208–209, 222, 279, 375
doStuff() method, 423
dot (.), 94
DownloadService interface, 233–234
Dynamic Invocation Interface (DII), 423
dynamic proxy class, 421

E

EJB Design Patterns (Floyd Marinescu), 340, 364
EJBObject class, 440
embedded code constructs, 266
embedded comment syntax, 266
employeeFormBean.java example, 101–102
employeeFormBeanTestCase.java example, 102–103
enabling assertions, 64–65
Endorsed Standards Override Mechanism (ESOM), 59
Enterprise JavaBeans (EJBs)
  overview, 291
  primary keys for, generating
    application server-specific approach, 363
    client control approach, 360–362
    database autogeneration approach, 363–364
    networked Singleton approach, 363
    simple scenario, 359–360
    Singleton approach, 362
Enterprise Resource Planning (ERP), 385
entity beans, 366
Enumeration interface, 157–158, 161–162
environment variables, CLASSPATHS and, 109
errors
  assertions, 60, 62–63
  compile-time, 251
  file not found, 8
  NoClassDefFoundError, 352
  not a valid Win32 application, 16
  OutofMemoryError, 188
ESOM (Endorsed Standards Override Mechanism), 59
example code. *See* listings
exec() method
  BadWinRedirect.java, 12
  dir command, 8
  method prototypes, list of, 4
  Runtime class, 4
executable JAR files, 108–112
exitValue() method, 4–5
ExternalLink path, 408

F

failover, 335
FileChannel class, 17, 21, 25, 152
File class, 151–154
FileDescriptor class, 152
FileDialog.setFilenameFilter() method, 152
FileFilter class, 152
FileHandler class, 48, 52
FileInputStream class, 17, 152
File.list() method, 152
FileLock class, 153
FilenameFilter class, 152
file not found error code, 8
FileOpenService interface, 233
FileOutputStream class, 17, 152
FilePermission class, 152
File.renameTo() method, 151, 155
files
  reading from servlets, 302–306
  references, URLs as, 297–300, 302
FileSaveService interface, 233
FilterConfig object, 251
findAssociations() method, 401
findByPrimaryKey() method, 366
findCallerAssociations() method, 401
findClassificationSchemeByName() method, 401
findClassificationSchemes() method, 401
findConceptByPath() method, 401
findConcepts() method, 402, 415
findNodeWithContent() method, 170
findOrganizations() method, 402, 404
findRegistryPackages() method, 402
findServiceBindings() method, 402
findServices() method, 402
FirstCall.getInstance() method, 119
FirstCallSingleton class, 118
flip() method, 26
forward directive, 259, 321
4 KB buffer size, 21
freeConnection() method, 286
Front Control pattern, 239, 245

G

GatheringByteChannel class, 17
getActionCmd() method, 177
getAttribute() method, 318
getBytes() method, 137
getChannel() method, 25
getClassifications() method, 412
getClassifiedObject() method, 412
getConnection() method, 286, 288
getElementsByTagName() method, 180
getExternalLinks() method, 403
getExternalURI() method, 409
getFirstChildElement() method, 71
getGeneratedKeys() method, 364
getGUI() method, 120
getInitParameter() method, 253
getInitParameterNames() method, 253
getInputStream() method, 11, 130, 134, 182
getInstance() method, 117, 119–120, 285
getLabel() method, 177
getLevel() method, 48
getMinimumSize() method, 124
getNamedDispatcher() method, 247, 255, 324
getName() method, 180
getNextKey() method, 362–363
getNodeName() method, 180
getNodeValue() method, 180
getOutputStream() method, 130, 133–134
getPollOfTheDay() method, 292
getPort() method, 421
getPreferredSize() method, 124
getProductNames() method, 180
getRealPath() method, 304
getResourceAsStream() method, 306
getResource() method, 306
getRuntime() method, 4
getSelectedItem() method, 180
getServiceBindings() method, 404
getServices() method, 404
getSession() method, 222, 308
getSpecificationObject() method, 409, 415
getString() method, 180
getText() method, 116, 180
getValue() method, 222
getValueNames() method, 221
getWriter() method, 295
global variables, Singletons as, 120–121
goGet() method, 222
GoodDomLookup.java example, 69–71
GoodFileRename.java example, 155–156
GoodLayeredPane.java example, 150
GoodNavigationUtils.java example, 116
GoodRightMouseButton.java example, 85–86
GoodSetSize.java example, 124–125
GoodStringTokenizer.java example, 142–144
GoodURLPost.java example, 135–136
GoodVoter.Servlet.java example, 296–297
GoodWindowsExec.java example, 9–11
GoodWinRedirect.java example, 13–15
GridLayout manager, 124

H

HashMap class, 162
HashSet class, 116, 162
Hashtable class, 162
Help targets (ANT), 91
hiding implementation details, 39–43
HRMS (Human Resources Management Systems), 385
HTTPClientPost.java example, 137–140
HTTPConnection class, 137
httpGet() method, 169, 174, 180, 195
HTTPResponse class, 137
HttpSession class, 221
HttpURLConnection class, 133
Human Resources Management Systems (HRMS), 385
Hypertext Transfer Protocol (HTTP), 126

I

IANA charset registry, 18
IDEs (integrated development environments), 99
IllegalArgumentException value, 31
IllegalStateException value, 247
IllegalThreadSafeException value, 5–6
ImageAnnotationServer.java, 27–29
image obsession, 348–352
implementation details, hiding, 39–43
import tag, 258
include directive, 256, 258
InitialDirContext class, 243
init() method, 239, 273–274
init-params element, 253
input/output (I/O), non-blocking server, 26–30
instance variables, in servlets, 269, 272–274, 278
instantiation, lazy instantiation, 118, 187
Integer object, 149
integrated development environments (IDEs), 99
interfaces
  BasicService, 233
  BusinessQueryManager, 401–402
  CallableStatement, 364
  ClipboardService, 233
  DocumentLS, 77
  DownloadService, 233–234
  Enumeration, 157–158, 161–162
  FileOpenService, 233
  FileSaveService, 233
  MouseListener, 80
  MouseMotionListener, 80
  MouseWheelListener, 80
  NIO packages, 18–19
  ParseErrorEvent, 77
  PersistenceService, 233
  PrintService, 233
  RequestDispatcher, 247
  SessionSynchronization, 434–435
  SimpleTestIF, 420
  StateTellerBean, 440–441
  Transaction, 393
  See also classes
int getMaxInactiveInterval() method, 221
invalidate() method, 225–227
invariants, 60
I/O (input/output), non-blocking server, 26–30
isAcceptable() method, 33
isDirectory() method, 152
isFile() method, 152
isNew() method, 222, 226–227, 318
isOpen() method, 393
isPopupTrigger() method, 84, 88
isUserInRole() method, 238
Iterator class, 157–158, 161–162

J

Java API for XML-based Remote Procedure Call (JAX-RPC)
  DII calls, 423–426
  dynamic proxies, 421–423
  overview, 328, 417–418
  stub classes, 420
  web services, 418
Java API for XML Processing (JAXP), 68
Java API for XML Registries (JAXR), 328
Java applications, deploying, 109–110
java classname command, 108
Java Community Process (JCP) program, 116, 155
Java Data Objects (JDO), 385
Javadoc tags, 94
Java Document Object Model (JDOM), 79
Java Extension Mechanism, 110–111
Java Hotspot Client VM, 423–424
java -jar jarname command, 108
java.lang.Throwable class, 41
JavaNCSS utility, 93
Java Network Launching Protocol (JNLP), 229–230
Java Regular Expressions API, 266
JavaServer Pages (JSPs)
  design errors
    compiled code, 210–213
    model 1 architecture, 214
    model 2 architecture, 220
    request/response paradigm, 208
    state, maintaining, 209–210

overview, 201
reuse and content delivery, 255–259
java.sql.DatabaseMetaData class, 353
Java Standard Template Library (JSTL), 258
Java 2 Enterprise Edition (J2EE) architecture considerations, 329–333
Java 2 Micro Edition (J2ME), 162
java.util.logging package
BadLoggerExample, 45–47
getLevel() method, 48
GoodLoggerExample.java, 52
logger-handler relationship, 51
logging granularity, levels of, 45
overview, 44
setLevel() method, 51–52
Java Virtual Machine (JVM), 108–109
Java Web Start application, 231–232
jaxp.jar package, 111
JAXP (Java API for XML Processing), 68
JaxpSave.java, 74–75
JAXR (Java API for XML Registries), 328
JAX-RPC. *See* Java API for XML-based Remote Procedure Call
JButton class, 165
JCACHE specification, 207
JComboBox class, 165
JConfig library, 7
JCP (Java Community Process) program, 116, 155
JDBC DataSource resource pool, 384
JDepend utility, 93
JDO (Java Data Objects), 385
JDOM (Java Document Object Model), 79
JdomSave.java example, 79
JEditorPane class, 298
JFrame class, 149, 151
JFreeChart package, 348
JLayeredPane class
BadLayeredPane.java example, 147–148
GoodLayeredPane.java example, 150
overview, 146
JList class, 165
JNLP (Java Network Launching Protocol), 229–230
JPanel class, 151
JRootPane class, 148
JScrollPane class, 298
JSPs. *See* JavaServer Pages
JSTL (Java Standard Template Library), 258
JTextArea class, 171
JTextField class, 298
J2EE (Java 2 Enterprise Edition) architecture considerations, 329–333
J2ME (Java 2 Micro Edition), 162
JUnit tests
AllTests.java example, 100
ANT (Another Neat Tool) and, 90
dbQueryBean.java example, 105
dbQueryTestCase.java example, 105–107
employeeFormBean.java example, 101–102
employeeFormBeanTestCase.java example, 102–103
JVM (Java Virtual Machine), 108–109

L

lastModified() method, 152
lazy instantiation, 118, 187
LDAPAccessControl, 40
LDAPException class, 41
least common denominator (LCD), 80
levels of granularity, 45
level 3 specifications (DOM), 76–77
lifecycle_build.xml example, 89
Lightweight Directory Access Protocol (LDAP)
directories, 235–239
overview, 39
LinkedHashSet class, 162
LinkedList class, 162
listCategories() method, 114–115
listings
abstracting details, bad example of, 39–40
acceptConnections() method, 33
AssertionExample.java, 61
assertions, postconditions, 63–64
authenticate() method, 42–44
BadExecJavac.java, 5–6
BadWinRedirect.java, 12
BmpWriter3.java, 23–25
caching, generateTests.java, 205–207
concurrent results sets, 354–358
connections, in servlets, 279–285
DII hard-coded calls, 423–424
DOM tree, searching
BadDomLookup.java, 67
GoodDomLookup.java, 69–71
XpathLookup.java, 72–73
doPost(), 208–209
dynamic proxies, 421–422
environment variables, 109
FileHandler class, XML-formatted output from, 50
files
BadFileRename.java, 153–154
GoodFileRename.java, 155–156
GoodWindowsExec.java, 9–10
GoodWinRedirect.java, 13–14
ImageAnnotationServer.java, 27–29
JaxpSave.java, 74–75

listings (*continued*)
JdomSave.java, 79
JLayeredPane class
BadLayeredPane.java, 147–148
GoodLayeredPane.java, 150
JUnit tests
AllTests.java, 100
dbQueryBean.java, 105
dbQueryCase.java, 105–107
employeeFormBean.java, 101–102
employeeFormBeanTestCase.java, 102–103
lifecycle_build.xml, 89
Little Endian byte operations, 22
LocalEJBServlet, 345
logging, BadLoggerExample, 45–49
makeCall() method, 404–406
MediocreExecJavac.java, 7
Memento pattern, 439–441
mouse button portability
BadRightMouseButton.java, 81–83
GoodRightMouseButton.java, 85–86
navigation
BadNavigationUtils.java, 114–115
GoodNavigationUtils.java, 116
Navigation.xml, 113
optional packages, versioning and sealing, 112
preferences, storing user, 37
prepared statements, 376–377
printMetaDataStuff(), 358–359
properties, preferences file, 35–36
querying, by concept, 415–417
RemoteEJBServlet, 342–344
ScheduleSwitcher.java, 53–56
servlets
instance variables in, 269–271, 276–277
output mechanisms in, 292–293, 296–297
reading files from, 303, 305–306
session synchronization, 437–438
setSize() method, 123–125
Singletons
global variables as, 120–121
instantiated at class loading time, 118
instantiated at first call, 118–119
SlowFileCopy.java, 20
SOAPHandler object, 347
SOAPHome object, 345–346
stack tracing, 351–352
strings
BadStringTokenizer.java, 141
GoodStringTokenizer.java, 142–144
TokenCollectionTester.java, 145
submissions, multiple
handling, 318–321
preventing, 314–315
TestExec.java, 15–16
UDDI query example, 398–401
URLs, posting
BadURLPost.java, 131–132, 134
echocgi.c, 128–130
GoodURLPost.java, 135–136
HTTPClientPost.java, 137–139
validateBean.java, 262–263
Web Application Deployment Descriptor, 308–312
XercesSave.java, 77–78
XML schedule file, 56
Little Endian byte operations, 21–26
load balancing, 335
LocalEJBServlet example, 345
loggers. *See* java.util.logging package
log() method, 60
long getCreationTime() method, 221
long getLastAccessedTime() method, 221
loops, iterating to zero, 190

M

main() method, 47, 121
makeCall() method, 398, 401, 404–406
makePersistent() method, 393
MalformedURLException, 302
Marinescu, Floyd (*EJB Design Patterns*), 340, 364
Matcher class, 266
m_broker.freeConnection() method, 281
m_broker.getConnection() method, 281
MediocreExecJavac.java, 7
Memento pattern, 438–442
message-drive beans, 366
methods
abort(), 393
accept(), 33, 152
actionPerformed(), 177–178
addMode(), 173, 194
addOperation(), 173
addType(), 173
afterBegin(), 434–436
afterCompletion(), 434–435
allocateDirect(), 26
attach(), 31
attachment(), 31
authenticate(), 40, 42–44
beforeCompletion(), 434–435
begin(), 393
boolean isNew(), 221

catch(), 295
checksubmitcount(), 315
cloneNode(), 58–59
commandAction(), 177
commit(), 393
convert(), 341
create(), 341, 366, 368–370
createNewFile(), 152
createProxy(), 420
createURL(), 194
delete(), 152, 155, 157
destroy(), 175
Display.getdisplay(), 176
Display.setCurrent(), 177
doAmazonSearch(), 180, 191–192
Document.cloneNode(), 59
doFilter(), 245, 252
doGet(), 222, 272, 276, 292
doPost(), 208–209, 222, 279, 281, 375
doStuff(), 423
exec()
  BadWinRedirect.java, 12
  dir command, 8
  method prototypes, list of, 4
  Runtime class, 4
exitValue(), 4–5
FileDialog.setFilenameFilter(), 152
File.list(), 152
File.renameTo(), 151, 155
findAssociations(), 401
findByPrimaryKey(), 366
findCallerAssociations(), 401
findClassificationSchemeByName(), 401
findClassificationSchemes(), 401
findConceptByPath(), 401
findConcepts(), 402, 415
findNodeWithContent(), 170
findOrganizations(), 402, 404
findRegistryPackages(), 402
findServiceBindings(), 402
findServices(), 402
FirstCall.getInstance(), 119
flip(), 26
freeConnection(), 285
getActionCmd(), 177
getAttribute(), 318
getBytes(), 137
getChannel(), 25
getClassifications(), 412
getClassifiedObject(), 412
getConnection(), 285, 288
getElementsByTagName(), 180
getExternalLinks(), 403
getExternalURI(), 409
getFirstChildElement(), 71
getGeneratedKeys(), 364
getGUI(), 120
getInitParameter(), 253
getInitParameterNames(), 253
getInputStream(), 11, 130, 134, 182
getInstance(), 117, 119–120, 285
getLabel(), 177
getLevel(), 48
getMinimumSize(), 124
getName(), 180
getNamedDispatcher(), 247, 255, 324
getNextKey(), 362–363
getNodeName(), 180
getNodeValue(), 180
getOutputStream(), 130, 133–134
getPollOfTheDay(), 292
getPort(), 421
getPreferredSize(), 124
getProductNames(), 180
getRealPath(), 304
getResource(), 306
getResourceAsStream(), 306
getRuntime(), 4
getSelectedItem(), 180
getServiceBindings(), 404
getServices(), 404
getSession(), 222, 308
getSpecificationObject(), 409, 415
getString(), 180
getText(), 116, 180
getValue(), 222
getValueNames(), 221
getWriter(), 295
httpGet(), 169, 174, 180, 195
init(), 239, 273–275
int getMaxInactiveInterval(), 221
invalidate(), 225–227
isAcceptable(), 33
isDirectory(), 152
isFile(), 152
isNew(), 222, 226–227, 318
isOpen(), 393
isPopupTrigger(), 84, 88
isUserInRole(), 238
lastModified(), 152
listCategories(), 114–115
log(), 60
long getCreationTime(), 221

methods *(continued)*
- long getLastAccessedTime(), 221
- main(), 47, 121
- makeCall(), 398, 401, 404
- makePersistent(), 393
- m_broker.freeConnection(), 282
- m_broker.getConnection(), 282
- mouseClicked(), 80
- mouseDragged(), 80
- mouseEntered(), 80
- mouseExited(), 80
- mouseMoved(), 80
- mousePressed(), 80, 88
- mouseReleased(), 80, 88
- mouseWheelMoved(), 80
- Object getValue(), 221
- onMessage(), 365
- openInputStream(), 182
- order(), 26
- out.println(), 210
- parse(), 79
- pauseApp(), 175
- print(), 294–295
- printLeague(), 63
- println(), 83, 86, 294–295
- printMetaDataStuff(), 354–358
- Process.getOutputStream(), 17
- putInt(), 26
- putInUserData(), 272–273
- putShort(), 26
- putValue(), 221, 224, 227
- removeBlankNodes(), 170
- renameTo(), 152, 155
- repaint(), 124
- replaceAll(), 265
- replaceString(), 173
- reset(), 390
- Runtime.exec(), 6, 8, 15
- search(), 366, 368–370
- select(), 31, 33
- selectNodes(), 117
- selectSingleNode(), 73
- setAttribute(), 318
- setDate(), 376
- setFrom(), 439
- setGUI(), 120
- setInternalValues(), 126
- setLayout(), 124
- setLevel(), 51–52
- setPage(), 298
- setRequestProperty(), 133
- setSize(), 122–125
- setString(), 376
- setUseParentHandlers(), 51–52
- showDetails(), 190
- showSwitchedDays(), 56
- split(), 266
- startApp(), 175, 177
- static allocate(), 26
- String.equals(), 167
- stringExists(), 173
- String getId(), 221
- System.out.println(), 60, 102
- Throwable.printStackTrace(), 41
- tokenize(), 142, 144
- transferTo(), 21
- values(), 162
- Vector.remove(), 159
- void invalidate(), 221
- void putValue(), 221
- void setMaxInactiveInterval(), 222
- waitFor(), 5
- write(), 295
- writeInt(), 23
- writeNode(), 79
- *See also* commands

m_instance variable, 117
Model-View-Controller (MVC), 244
Modified state (DOM), 74
Momento Pattern, 438
mouse button portability
- BadRightMouseButton.java example, 81–83
- event interfaces, list of, 80
- GoodRightMouseButton.java example, 85–86

mouseClicked() method, 80
mouseDragged() method, 80
mouseEntered() method, 80
mouseExited() method, 80
MouseListener interface, 80
MouseMotionListener interface, 80
mouseMoved() method, 80
mousePressed() method, 80, 88
mouseReleased() method, 80, 88
MouseWheelListener interface, 80
mouseWheelMoved() method, 80
m_out variable, 273–274
m_sc variable, 276
multiplexing, 26–27
m_useridparam variable, 274
MVC (Model-View-Controller), 244

N

namespace collision, 34
navigation
- BadNavigationUtils.java example, 114–115
- GoodNavigationUtils.java example, 116
- Navigation.xml example, 113

network problems, elimination design strategies
bad client design, 337
EJB design considerations, 340–341
general design considerations, 336–339
overview, 335–336
simple scenario, 336
New state (DOM), 74
NIO packages
abstractions in, 17–18
canonical file copy example, 20–21
classes and interfaces in, list of, 18–19
Little Endian byte operations, 21–26
managing connections using, 26
Selector utility, 26–27
`NoClassDefFoundError`, 352
non-blocking server I/O, 26–30
Norman, Donald *(The Design of Everyday Things)*, 127
not a valid Win32 application error code, 16
`NullPointerException` value, 275
`numTickets` parameter, 322

O

Object Data Modeling Group (ODMG), 397
`Object getValue()` method, 221
Object Management Group (OMG), 59
Object Relational Bridge (OJB), 391
object-relational mapping (ORM) tool, 332
`onClick` attribute, 315
`onMessage()` method, 365
`OP_ACCEPT` operation, 3, 31
`OP_CONNECT` operation, 31
`openInputStream()` method, 182
operator precedence, 267
`OP_READ` operation, 31
optional packages, 110–112
`OP_WRITE` operation, 31
`order()` method, 26
`org.w3c.dom` class, 53
`org.xml.sax` class, 53
ORM (object-relational mapping) tool, 332
OSCache tags, 201–203
`os.name` system property, 11
`OutofMemoryError`, 188
`out.println()` method, 210
`OutputStream` class
`Runtime.exe()` method, 15
`writeInt()` method, 23
`OutputStreamWriter` class, 137

P

`pageNumber` variable, 247
`ParseErrorEvent` interface, 77
`parse()` method, 79
passing parameters, 209
`Pattern` class, 266
`pauseApp()` method, 175
`PersistenceService` interface, 233
Persisted state (DOM), 74
personal digital assistants (PDAs), 162
personalization, 235
`Pipe.SinkChannel` class, 18
`Pipe.SourceChannel` class, 18
`PixelGrabber` class, 25
possessive quantifiers, 266
postconditions, 60, 63–64
preconditions, 60
preferences
file example, 35
user, storing, 37–38
in Windows Registry, 38
pre-invocation times, 428
prepared statements, 375–378
preprocessing operations, 266
`printLeague()` method, 63
`println()` method, 83, 86, 294–295
`printMetaDataStuff()` method, 354–358
`print()` method, 294–295
`PrintService` interface, 233
`PrintWriter` class, 274, 294
`priorState` variable, 439
`Process` class
`exitValue()` method, 4
`getInputStream()` method, 11
`waitFor()` method, 5
`Process.getOutputStream()` method, 17
properties
`cache.debug`, 203
`cache.path`, 203
`cache.unlimited`, 203
namespace collision, avoiding, 34
preferences, user, storing, 37–38
user, storing, 36
Properties API, 38
`ProtocolException` class, 133
publications
The Design of Everyday Things (Donald Norman), 127
EJB Design Patterns (Floyd Marinescu), 340, 364
`public Process exec` commands, 4
`putInt()` method, 26
`putInUserData()` method, 272–273
`putShort()` method, 26
`putValue()` method, 221, 224, 227

Q

query builders, 353
querying, by concept, 415–417

R

RandomAccessFile class, 17, 152
Reactor pattern, 30–31
Registry Information Model (RIM), 398
RegistryObject class, 403
Relational Database Management Systems (RDBMS), 235
RemoteEJBServlet example, 342–343, 345
Remote Method Invocation (RMI), 335, 390
Remote Procedure Call (RPC), 126
removeBlankNodes() method, 170
renameTo() method, 152, 155
repaint() method, 124
replaceAll() method, 265
replaceString() method, 173
Representational State Transfer (REST), 163
RequestDispatcher interface, 247
reset() method, 390
resource managers
  defined, 382
  resource manager connection factory, 370
result sets, multiple concurrent, problems with, 353, 355, 357–359
RIM (Registry Information Model), 398
RMI (Remote Method Invocation), 335
Root Logger, 51
RPC (Remote Procedure Call), 126
Runtime class, 4
Runtime.exec() method, 6, 8, 15

S

scalability, 334
ScatteringByteChannel class, 17
ScheduleSwitcher.java example, 53–56
scripts
  batch, 98
  ThreadGroup, 201, 204
sealed packages, 111–112
search() method, 366–367, 369
security, optional packages, 112
SelectableChannel class, 18
SelectionKey class, 31, 33
select() method, 31, 33
selectNodes() method, 117
Selector class, 18
Selector utility (NIO packages), 26–27
selectSingleNode() method, 73
ServerSocketChannel class, 18, 30
ServerSocket class, 17, 30
servlet-based applications, 223
ServletContext object, 276, 304
ServletOutputStream class, 294
servlets
  connections within, 278–279, 281, 285, 288
  files from, reading, 302–306
  instance variables in, 269, 272–274, 278
  output mechanisms in, 291–295
  URL rewriting, 321
servlet sessions, 316–317
session beans
  entity beans, 366
  message-driven, 365–366
  passivated state, 435
  stateful, 367–369, 434
  stateless, 369–372
  transaction state problem, 435
sessions
  defined, 316
  getSession() method, 308
  in-use stage, 318
  multiple-submit problem, 317
  servlet sessions, 316
SessionSynchronization interface, 434–435
session variables, 210
setAttribute() method, 318
setDate() method, 376
setFrom() method, 439
setGUI() method, 120
setInternalValues() method, 126
setLayout() method, 124
setLevel() method, 51–52
setPage() method, 298
setRequestProperty() method, 133
setSize() method, 122–125
setString() method, 376
setUseParentHandlers() method, 51–52
showDetails() method, 190
showSwitchedDays() method, 56
SignOnVideoBean session bean, 360–361
Simple Object Access Protocol (SOAP), 126
SimpleTestIF interface, 420
Singleton pitfalls, avoiding, 117–121
16-bit unicode characters, 18
SlowFileCopy.java, 20–21
SOAPHandler object, 341, 347
SOAPHome object, 341, 345–346
SocketChannel class, 17–18
Socket class, 17
SpecificationLink path, 408
split() method, 266
SQL commands, 96
stack tracing, 351–352
startApp() method, 175, 177
stateful session beans, 368, 434
stateless session beans, 368–372

state parameter, 204
StateTellerBean interface, 440–441
static allocate() method, 26
StreamGobbler class, 11
StringBuffer class, 172
string comparisons, avoiding, 185
String.equals() method, 167
stringExists() method, 173
String getId() method, 221
StringTokenizer class, 4, 140, 142, 146
Struts framework, 220
stub classes, 420
submissions
  checksubmitcount() method, 315
  multiple
    handling, 316–320
    preventing, 313–315
  processing simple, 313
SwingUtilities class, 88
System.out.println() method, 60, 102

T

tags
  Cewold tag library, 348
  depends, 89
  forward, 259
  import, 258
  Javadoc, 94
  See also directives
telephone number formats, 262–263
temporary objects, avoiding, 186
TestExec.java, 15–16
theIndex object, 367
ThreadGroup scripts, 201, 204
Throwable.printStackTrace() method, 41
TokenCollectionTester.java example, 145–146
tokenize() method, 142, 144
topicId parameter, 204
Transaction interface, 393
transactions, defined, 334–335
transferTo() method, 21
TreeSet class, 162

U

UDP (User Datagram Protocol), 26–27
Uniform Resource Identifiers (URIs)
  as file references, 297–300, 302
  overview, 163
Universal Description and Discovery Integration (UDDI), 398
Universally Unique Identifier (UUID), 364
UnknownHostException value, 302
URLConnection class, 127, 134
URLConnectionFactory class, 127
URLStreamHandler class, 127
User Datagram Protocol (UDP), 26–27
user properties, storing, 36

V

validateBean.java example, 262–263
values() method, 162
Vector class, 162
Vector.remove() method, 159
versioning optional packages, 112
VideoUser entity bean, 360–361, 363
Virtual Machine (VM), 4
void invalidate() method, 221
void putValue() method, 221
void removeValue() method, 222
void setMaxInactiveInterval() method, 222
VoterApp object, 291–292

W

waitFor() method, 5
Web Application Deployment Descriptor, 307–308
Web Archive (WAR) files, 95
Web controller architecture, 245
Web Service Description Language (WSDL), 403
Win32 error code. *See* errors
Windows Registry, preferences stored in, 38
writeInt() method, 23
write() method, 295
writeNode() method, 79
Write Once, Run Anywhere (WORA), 155
WSDL (Web Service Description Language), 403
W3C Load specification, 80

X

xalan.jar package, 111
xerces.jar package, 111
XMLOutputter class, 80
XML schedule file example, 56
XpathLookup.java example, 72–73
XSLTFilter class, 252
XSLTransformFilter application, 249
XulRunner class, 146