





Component Development for the Java™ Platform

Stuart Dabbs Halloway

◆◆Addison-Wesley

Boston • San Francisco • New York • Toronto • Montreal
London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book and Addison-Wesley, Inc. was aware of a trademark claim, the designations have been printed with initial capital letters or all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers discounts on this book when ordered in quantity for special sales. For more information, please contact:

Pearson Education Corporate Sales Division
201 W. 103rd Street
Indianapolis, IN 46290
(800) 428-5331
corpsales@pearsoned.com

Visit AW on the Web: www.aw.com/cseng/

Library of Congress Cataloging-in-Publication Data

Halloway, Stuart Dabbs.

Component development for the Java platform / Stuart Dabbs Halloway.
p. cm. — (DevelopMentor series)

Includes bibliographical references and index.

ISBN 0-201-75306-5

1. Java (Computer programming language) 2. System design. I. Title. II. Series.

QA76.73 J38 H346 2002
005.13'3—dc21

200105379

Copyright © 2002 Pearson Education, Inc.

All rights reserved. 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, or otherwise, without the prior consent of the publisher. Printed in the United States of America. Published simultaneously in Canada.

For information on obtaining permission for use of material from this work, please submit a written request to:

Pearson Education, Inc.
Rights and Contracts Department
75 Arlington Street, Suite 300
Boston, MA 02116
Fax: (617) 848-7047

0-201-75306-5
Text printed on recycled paper
1 2 3 4 5 6 7 8 9 10—MA—0504030201
First printing, December 2001

To Joey





Contents

Foreword	xiii
Preface	xv
1 From Objects to Components	1
2 The Class Loader Architecture	11
2.1 Assembling an Application	11
2.2 Goals of the Class Loader Architecture	14
2.2.1 <i>Transparency</i>	15
2.2.2 <i>Extensibility</i>	15
2.2.3 <i>Capability</i>	16
2.2.4 <i>Configurability</i>	16
2.2.5 <i>Handling Name and Version Conflicts</i>	16
2.2.6 <i>Security</i>	17
2.3 Explicit and Implicit Class Loading	17
2.3.1 <i>Explicit Loading with URLClassLoader</i>	18
2.3.2 <i>Implicit Class Loading</i>	19
2.3.3 <i>Reference Type versus Referenced Class</i>	20
2.3.4 <i>ClassLoader.loadClass versus Class.forName</i>	21
2.3.5 <i>Loading Nonclass Resources</i>	22
2.4 The Class Loader Rules	23
2.4.1 <i>The Consistency Rule</i>	23
2.4.2 <i>The Delegation Rule</i>	24
2.4.3 <i>The Visibility Rule</i>	25
2.4.4 <i>Delegations as Namespaces</i>	27
2.4.5 <i>Static Fields Are Not Singletons</i>	28
2.4.6 <i>Implicit Loading Hides Most Details</i>	29
2.5 Hot Deployment	29
2.5.1 <i>Using Hot Deployment</i>	33
2.6 Unloading Classes	35
2.6.1 <i>Making Sure Classes Are Collectable</i>	35



2.7	Bootclasspath, Extensions Path, and Classpath	36
2.7.1	<i>The Classpath</i>	37
2.7.2	<i>The Extensions Path</i>	39
2.7.3	<i>The Bootclasspath</i>	41
2.8	Debugging Class Loading	43
2.8.1	<i>Instrumenting an Application</i>	44
2.8.2	<i>Using <code>-verbose:class</code></i>	45
2.8.3	<i>Instrumenting the Core API</i>	46
2.9	Inversion and the Context Class Loader	49
2.10	Onward	55
2.11	Resources	56
3	Type Information and Reflection	57
3.1	The Binary Class Format	58
3.1.1	<i>Binary Compatibility</i>	58
3.1.2	<i>Binary Class Metadata</i>	63
3.1.3	<i>From Binary Classes to Reflection</i>	66
3.2	Reflection	66
3.2.1	<i>Reflecting on Fields</i>	68
3.2.2	<i>The Difference between <code>get</code> and <code>getDeclared</code></i>	68
3.2.3	<i>Type Errors Occur at Runtime</i>	70
3.2.4	<i>Reflecting on Methods</i>	71
3.3	Reflective Invocation	72
3.3.1	<i>A Reflective Launcher</i>	73
3.3.2	<i>Wrapping Primitive Types</i>	74
3.3.3	<i>Bypassing Language Access Rules</i>	76
3.3.4	<i>Exceptions Caused by Reflective Invocation</i>	81
3.4	Dynamic Proxies	83
3.4.1	<i>Delegation instead of Implementation Inheritance</i>	83
3.4.2	<i>Dynamic Proxies Make Delegation Generic</i>	84
3.4.3	<i>Implementing <code>InvocationHandler</code></i>	85
3.4.4	<i>Implementing a Forwarding Handler</i>	86
3.4.5	<i>The <code>InvocationHandler</code> as Generic Service</i>	87
3.4.6	<i>Handling Exceptions in an <code>InvocationHandler</code></i>	89
3.4.7	<i>Either Client or Server Can Install a Proxy</i>	90
3.4.8	<i>Advantages of Dynamic Proxies</i>	91

3.5	Reflection Performance	92
3.6	Package Reflection	94
3.6.1	Setting Package Metadata	95
3.6.2	Accessing Package Metadata	96
3.6.3	Sealing Packages	97
3.6.4	Weaknesses of the Versioning Mechanism	97
3.7	Custom Metadata	98
3.8	Onward	103
3.9	Resources	103
4	Serialization	105
4.1	Serialization and Metadata	105
4.2	Serialization Basics	106
4.2.1	Serialization Skips Some Fields	109
4.2.2	Serialization and Class Constructors	110
4.3	Using readObject and writeObject	111
4.4	Matching Streams with Classes	113
4.4.1	The serialVersionUID	114
4.4.2	Overriding the Default SUID	115
4.4.3	Compatible and Incompatible Changes	117
4.5	Explicitly Managing Serializable Fields	119
4.5.1	ObjectInputStream.GetField Caveats	120
4.5.2	Writer Makes Right	121
4.5.3	Overriding Class Metadata	122
4.5.4	Performance Problems	123
4.5.5	Custom Class Descriptors	124
4.6	Abandoning Metadata	124
4.6.1	Writing Custom Data after defaultWriteObject	124
4.6.2	Externalizable	125
4.6.3	Using writeObject to Write Raw Data Only: Bad Idea	128
4.7	Object Graphs	130
4.7.1	Pruning Graphs with Transient	131
4.7.2	Preserving Identity	131
4.7.3	Encouraging the Garbage Collector with reset	132
4.8	Object Replacement	133
4.8.1	Stream-Controlled Replacement	134
4.8.2	Class-Controlled Replacement	137
4.8.3	Ordering Rules for Replacement	139
4.8.4	Taking Control of Graph Ordering	145

4.9	Finding Class Code	147
4.9.1	<i>Annotation in RMI</i>	148
4.9.2	<i>RMI MarshalledObjects</i>	150
4.10	Onward	150
4.11	Resources	151
5	Customizing Class Loading	153
5.1	Java 2 Security	155
5.1.1	<i>The Role of Class Loaders</i>	157
5.2	Custom Class Loaders	159
5.2.1	<i>Pre-Java 2 Custom Class Loaders</i>	159
5.2.2	<i>Class Loading since SDK 1.2</i>	160
5.2.3	<i>A Transforming Class Loader</i>	162
5.3	Protocol Handlers	168
5.3.1	<i>Implementing a Handler</i>	169
5.3.2	<i>Installing a Custom Handler</i>	171
5.3.3	<i>Choosing between Loaders and Handlers</i>	174
5.4	Getting Past Security to the Loader You Need	175
5.5	Reading Custom Metadata	177
5.5.1	<i>Example: Version Attributes</i>	178
5.5.2	<i>Serializable Classes as Attributes</i>	179
5.5.3	<i>Reading Attributes during Class Loading</i>	183
5.5.4	<i>Debugging Support</i>	188
5.6	Onward	189
5.7	Resources	190
6	Interop 1: JNI	191
6.1	Why Interoperate?	191
6.2	The Dangers of Native Code	193
6.3	Finding and Loading Native Code	194
6.3.1	<i>Name Mappings</i>	195
6.3.2	<i>Type Mappings</i>	195
6.3.3	<i>Overloaded Names</i>	198
6.3.4	<i>Loading Native Libraries</i>	199
6.3.5	<i>Class Loaders and JNI</i>	202
6.3.6	<i>Common Errors Loading Native Libraries</i>	205
6.3.7	<i>Troubleshooting Native Loading</i>	207
6.4	Calling Java from C++	208
6.4.1	<i>Minimizing Round Trips</i>	211
6.4.2	<i>Performance Comparisons</i>	214
6.4.3	<i>Differences between JNI and Reflective Invocation</i>	214

6.5	Error Handling in JNI	217
6.5.1	<i>Failures in Native Code</i>	217
6.5.2	<i>Handling C++ Exceptions</i>	218
6.5.3	<i>Handling Java Exceptions from Native Code</i>	219
6.5.4	<i>Throwing Java Exceptions from Native Code</i>	222
6.6	Resource Management	223
6.6.1	<i>Interacting with the Garbage Collector</i>	224
6.6.2	<i>Managing Native Resources</i>	231
6.6.3	<i>Managing Arrays</i>	233
6.6.4	<i>Managing Strings</i>	239
6.7	Onward	240
6.8	Resources	241
7	Generative Programming	243
7.1	Why Generate Code?	243
7.1.1	<i>Object-Oriented Approaches to Modeling Variabilities</i>	244
7.1.2	<i>Thinking in Terms of Bind Time</i>	246
7.1.3	<i>Separating Specification from Bind Time</i>	247
7.1.4	<i>Choosing a Specification Language</i>	249
7.1.5	<i>Reuse Requires More Than One Use</i>	249
7.1.6	<i>A Little Domain Analysis Is a Dangerous Thing</i>	250
7.2	Why Generate Code with Java?	250
7.2.1	<i>Type Information Acts as a Free Specification Document</i>	250
7.2.2	<i>Class Loading Supports Flexible Binding Modes</i>	251
7.2.3	<i>Java Source Is Easy to Generate</i>	251
7.2.4	<i>Java Binary Classes Are Easy to Generate</i>	252
7.2.5	<i>Code Generation Boosts Performance</i>	252
7.2.6	<i>Levels of Commitment to Code Generation</i>	252
7.3	A Taxonomy of Bind Times and Modes	253
7.4	Code Generation in RMI	255
7.5	Code Generation in JSP	257
7.6	Code Generation in EJB	260
7.6.1	<i>The Deployment Descriptor</i>	263
7.6.2	<i>Alternate Implementations</i>	265
7.7	Generating Strongly Typed Collections	267
7.7.1	<i>Code Generation Language versus Target Language</i>	270
7.8	Generating Custom Serialization Code	271
7.9	Onward	276
7.10	Resources	279

8	Onward	281
8.1	Where We Are	281
8.2	Where We Are Going	282
8.3	Resources	283
A	Interop 2: Bridging Java and Win32/COM	285
	Bibliography	319
	Index	323



Foreword

Several years ago, Stu abandoned the world of COM for what he had hoped would be greener pastures. While many of his colleagues felt he had lost his senses, Stu ignored our skepticism and walked away from COM completely. This was especially difficult given the fact that his employer had a tremendous investment in COM and had achieved relatively little traction in the Java world at the time.

Based on this book, I feel the move was beneficial both to Stu and to those who will be influenced by this book.

Stu's view on the Java platform is quite novel. This book portrays the Java Virtual Machine (JVM) as a substrate for component software. Are there languages and compilers that generate these components? Sure, but that isn't the focus of this book. Does the JVM perform a variety of services such as garbage collection and JIT compilation? Absolutely, but again, that isn't the focus of this book either. Rather, Stu focuses the reader on the role the JVM plays in software integration.

I am especially happy to see the book's emphasis on the class loader architecture. After spending over eight years working with COM and now two years with its successor, the Common Language Runtime (CLR), I believe that the key to understanding any component technology is to first look at how component code is discovered, initialized, and scoped during execution. In the JVM, the class loader is responsible for all of these tasks, and Stu gives that devil more than its due.

The JVM (and the Java platform as a whole) has a serious competitor now that Microsoft has more or less subsumed most Java technology into its .NET initiative, most specifically the CLR. It will be interesting to see how Sun adapts





to the challenge. In looking at the JVM and CLR side-by-side, the JVM exemplifies the “less is more” philosophy, which I believe is its greatest strength. Hopefully, Sun will remain true to this basic design principle as the pressures of platform warfare pull them in the direction of adding feature upon feature for market positioning rather than aesthetic reasons.

— Don Box,
September 2001
Manhattan Beach, California



Preface

This book is about developing components using the Java platform. In this book, the term *component* has a very specific meaning. A component is an independent unit of production and deployment that is combined with other components to assemble an application.

To elaborate on this definition, consider the difference between objects and components. An object represents an entity in the problem domain, while a component is an atomic¹ piece of the installed solution. The object and component perspectives are complementary, and good designs take account of both.

Modern development platforms such as Java provide the infrastructure that developers need to create classes and components. To support object-oriented programming, Java provides encapsulation, inheritance, and polymorphism. To support components, Java provides loaders and rich type information. This book assumes that you already understand object-oriented programming in Java, and it explains how to use Java's component infrastructure effectively.

Loaders are responsible for locating, bringing into memory, and connecting components at runtime. Using Java's loaders, you can

- Deploy components at fine granularity.
- Load components dynamically as needed.
- Load components from other machines on the network.
- Locate components from custom repositories.
- Create mobile code agents that live across multiple virtual machines.
- Import the services of non-Java components.

1. Atomic here means "indivisible," not necessarily "stands alone." Most components will have dependencies on other components.

Loaders manage the binary boundaries between components. In a world of distributed applications and multiple component suppliers, loaders locate and connect compatible components.

Type information describes the capabilities of some unit of code. In some development environments type information is present only in source code. In Java, type information is not merely a source code artifact; it is also an intrinsic part of a compiled class and is available at runtime through a programmatic interface. Because Java type information is never “compiled away,” loaders use it to verify linkages between classes at runtime. In application programming, you can use type information to

- Serialize the state of Java objects so that they can be recreated on another virtual machine.
- Create dynamic proxies at runtime, to provide generic services that can decorate any interface.
- Translate data into alternate representations to interoperate with non-Java components.
- Convert method calls into network messages.
- Convert between Java and XML, the new lingua franca of enterprise systems.
- Annotate components with application-specific metadata.

Type information automates many tasks that might otherwise be coded by hand, and it helps to make components forward compatible to platforms of the future.

Who Should Read This Book

You should read this book if you want to design, develop, or deploy substantial applications in Java. Taking a full-lifecycle view of a Java application requires that you consider not just objects, but components. This book is about the core features of Java as a component platform: class loaders, reflection, serialization, and interoperation with other platforms. You should already know the basics of Java syntax and have some experience in object-oriented programming with Java.

This book is not specifically about high-level Java technologies, such as Remote Method Invocation (RMI), Enterprise JavaBeans (EJB), JINI, Java Server Pages (JSP), servlets, or JavaBeans, but understanding the topics in this book is critical to using those technologies effectively. If you learn how to use the component services described here, you will understand how these high-level technologies are built, which is the key to employing them effectively.

Security is also an important aspect of component development and deployment. It is too complex a topic to handle fairly here, and it deserves its own book-length treatment. (See [Gon99] for coverage of security on the Java platform.)

Organization of the Book

The chapters of this book fall into three sections. Chapter 1 introduces components. Chapters 2 through 6 explain loaders and type information on the Java platform. Chapter 7 shows more advanced uses of these services.

Chapter 1 introduces component-oriented programming. Component relationships must be established not only at compile time, but also at deployment and runtime. This chapter asks the key questions of component programming and relates them to the Java platform services discussed in subsequent chapters. Though the other chapters might be read out of order, you should definitely read this chapter first.

Chapter 2 shows how to use and troubleshoot class loaders. Class loaders control the loading of code and create namespace boundaries between code in the same process. With class loaders you can load code dynamically at runtime, even from other machines. Class loader namespaces permit multiple versions of the same class in a single Java virtual machine. You can use class loaders to reload changed classes without ever shutting down the virtual machine. You will see how to use class loaders, how the class loader delegation model creates namespaces, and how to troubleshoot class loading bugs. You will also learn to effectively control the bootclasspath, extensions path, and classpath.

Chapter 3 introduces Java type information. Java preserves type information in the binary class format. This means that even after you compile your

Java programs, you still have access to field names, field types, and method signatures. You can access type information at runtime via reflection, and you can use type information to build generic services that add capability to any object. You will see how to use dynamic invocation, dynamic proxies, package reflection, and custom attributes. Chapter 3 also includes a discussion of reflection performance.

Chapter 4 shows how Java serialization uses reflection. Serialization is a perfect example of a generic service. Without any advance knowledge of a class's layout, serialization can ship both code and state from one virtual machine to another across time or space. You will see how the serialization format embeds its own style of type information and how you can customize that representation. You will also see how to extend default serialization, replace it entirely with custom externalization code, or tune it to handle multiple versions of a class as code evolves. You will then learn how to validate objects being deserialized into your application and how to annotate serialized objects with instructions for finding the correct class loader.

Chapter 5 returns to class loaders and shows you how to implement your own. While the standard class loaders are dominant in most applications, custom class loaders allow you to transform class code as classes are loaded. These transformations could include decryption, adding instrumentation for performance monitoring, or even building new classes on-the-fly at runtime. You will see how to tie your custom class loaders into Java's security architecture, how to write a custom class loader, and how to write protocol handlers that can customize not just how you load classes, but also how you load any other type of resource.

Chapter 6 presents the Java Native Interface (JNI) as a basic means of controlling the boundary between Java code and components written in other environments. JNI provides a set of low-level tools for exposing Java objects to platform native code and native code to Java objects. You will learn to use the JNI application programming interface (API) to translate between Java and native programming styles—which differ markedly in their approach to class loading, type information, resource management, error handling, and array storage.

Understanding the deficiencies of JNI sets the stage for Appendix A, which describes a higher-level approach.

Chapter 7 discusses using Java metadata to automate the creation of source code or bytecode. Generated code is a high-performance strategy for reuse because you generate only the exact code paths that you will need at runtime. The chapter first presents JSP and EJB as examples of existing applications that auto-generate code, and then it introduces some ideas for code generation in your own programs.

Appendix A returns to interoperation. By building on the code generation techniques from Chapter 7, Appendix A shows you how to build an interoperation layer between Java and another component platform: Win32/COM. This chapter uses the open source Jawin library as an example, to show you how to generate Java stubs for Win32 objects, and vice versa.

Sample Code, Website, Feedback...

Unless specifically noted otherwise, all the sample code in this book is open source. You can download sample code from the book's website at <http://staff.develop.com/halloway/compsvcs.html>.

Unless otherwise noted, the code in this book is compiled and tested against the Java 2 Software Development Kit (SDK) version 1.3. Most of the code in the book will work identically under SDK versions 1.2, 1.3, and 1.4. Where this is not the case, the text will include a specific reference to the appropriate SDK version.

The author welcomes your comments, corrections, and feedback. Please send email to stu@develop.com.

Acknowledgments

First and foremost, thanks to my wife Joanna. You challenged me to think better, and then actually put up with being around me when I took the challenge. Thanks also to my parents, Ronald and Olive Dabbs, for raising me in an environment that enabled me to find the richly satisfying life I lead today.



Thanks to everyone at DevelopMentor for creating such a fantastic play environment. Thanks to Don Box and Mike Abercrombie for starting it all, and for bringing together such a talented team. Thanks to Brian Maso, whose Intensive Java course materials were the seed of many ideas in this book. Thanks to Simon Horrell, Kevin Jones, and Ted Neward for running an excellent Guerrilla Java class, and for many lengthy conversations on the minutiae of the Java platform.

Thanks to the DevelopMentor folk and other friends who volunteered to review drafts of this book. In addition to Brian, Simon, Kevin, and Ted, these also include Ian Griffiths, Tim Ewald, and Jason Masterman. Thanks to Eric Johnson for reviewing the entire manuscript. Special thanks to Justin Gehtland and Chris Sells, who also reviewed the entire manuscript, despite the fact that their day jobs keep them tied to the other component platform.

Thanks to the excellent group of reviewers provided by Addison-Wesley: Carl Burnham, Joshua Engel, Eric Freeman, Peter Hagggar, Howard Lee Harkness, Norman Hensley, Tim Lindholm, and Paul McLachlan. I don't know you all personally, and in some cases do not even have your names, but your contributions to the book were invaluable. Few problems could escape the notice of such an elite group. For any inconsistencies and errors that remain, the fault is mine.

Thanks to Mike Hendrickson and Julie Dinicola, my editors at Addison-Wesley. Thanks also to all the other wonderful people at Addison-Wesley who helped make this book happen: Tyrrell Albaugh, John Fuller, Giaconda Mateu, Patrick Peterson, Tracy Russ, Mary Cotillo, Stephane Thomas, and Ross Venables.

Thanks to the staff of Neo-China restaurant in Durham, North Carolina, for providing a substantial fraction of my caloric intake while I was writing this book.

