

THE EXPERT'S VOICE®

Subrahmanyam Allamaraju and
Cedric Buest with
John Davies, Tyler Jewel, Rod Johnson, Andrew Longshaw,
Ramesh Nagappan, Dr P.G. Sarang, Alex Toussaint,
Sameer Tyagi, Gary Watson, Mark Wilcox, Alan Williamson,
Daniel O'Connor



Professional Java Server Programming

J2EE 1.3 Edition

- The J2EE container architecture and runtime services
- Web component development with Servlets 2.3 and JavaServer Pages 1.2
- Business logic components with EJB 2.0, including container-managed persistence, EJB QL, and message-driven beans
- Underlying J2EE technologies for distributed development – RMI, JDBC and JNDI
- Introduction to Web Services covering SOAP, SWA, WSDL, and UDDI

a!
Apress®

Table of Contents

| | |
|---|--------------|
| <u>Introduction</u> | 1 |
| <u>The J2EE 1.3 Edition</u> | 1 |
| <u>What's Changed in this Edition of the Book?</u> | 2 |
| <u>Who is this Book For?</u> | 2 |
| <u>What's Covered in this Book</u> | 2 |
| <u>What You Need to Use this Book</u> | 3 |
| <u>Conventions</u> | 4 |
| <u>Customer Support</u> | 4 |
| <u>How to Download the Sample Code for the Book</u> | 5 |
| <u>Errata</u> | 5 |
| <u>E-mail Support</u> | 5 |
| <u>Chapter 1: The J2EE Platform</u> | 9 |
| <u>Programming for the Enterprise</u> | 10 |
| <u>The Enterprise Today</u> | 10 |
| <u>Is Java the Answer?</u> | 12 |
| <u>Platform Independence</u> | 13 |
| <u>Managed Objects</u> | 13 |
| <u>Reusability</u> | 13 |
| <u>Modularity</u> | 13 |
| <u>Enterprise Architecture Styles</u> | 14 |
| <u>Two-Tier Architecture</u> | 14 |
| <u>Three-Tier Architecture</u> | 14 |
| <u>n-Tier Architecture</u> | 15 |
| <u>Enterprise Architecture</u> | 16 |
| <u>The J2EE Platform</u> | 18 |
| <u>The J2EE Runtime</u> | 18 |
| <u>The J2EE APIs</u> | 19 |
| <u>J2EE Architecture – Containers</u> | 21 |
| <u>Container Architecture</u> | 23 |
| <u>Component Contracts</u> | 24 |
| <u>Container Service APIs</u> | 25 |
| <u>Declarative Services</u> | 26 |
| <u>Other Container Services</u> | 27 |

Table of Contents

| | |
|--|-----------|
| J2EE Technologies | 28 |
| Component Technologies | 28 |
| Web Components | 29 |
| Enterprise JavaBean Components | 29 |
| XML | 30 |
| Service Technologies | 31 |
| JDBC | 31 |
| Java Transaction API and Service | 31 |
| JNDI | 31 |
| JMS | 32 |
| JavaMail | 32 |
| The Java Connector Architecture | 32 |
| JAAS | 32 |
| Communication Technologies | 32 |
| Internet Protocols | 32 |
| Remote Object Protocols | 33 |
| Developing J2EE Applications | 33 |
| Application Development and Deployment Roles | 34 |
| Application Component Development | 35 |
| Composition of Application Components into Modules | 35 |
| Composition of Modules into Applications | 35 |
| Application Deployment | 36 |
| Summary | 37 |
| Chapter 2: Directory Services and JNDI | 39 |
| Naming and Directory Services | 39 |
| Naming Services | 40 |
| Directory Services | 41 |
| LDAP | 42 |
| LDAP Data | 43 |
| Introducing JNDI | 44 |
| The Tradeoffs | 47 |
| Why Use JNDI When We Have LDAP? | 47 |
| Using JNDI | 48 |
| Installing JNDI | 49 |
| JNDI Service Providers | 49 |
| How to Obtain JNDI Service Providers | 50 |
| Developing Your Own Service Provider | 50 |
| Java and LDAP | 50 |
| Access Control | 51 |
| Authentication | 51 |
| Authorization | 52 |
| White Pages Services | 52 |
| Distributed Computing Directory | 52 |
| Application Configuration | 53 |
| LDAP Operations | 53 |
| Standard LDAP Operations | 54 |
| Connecting to the LDAP Server with JNDI | 54 |
| Binding | 55 |

Table of Contents

| | |
|--|-----------|
| Simple, SSL/TLS, and SASL Security | 55 |
| LDAP v2 and LDAP v3 Authentication | 57 |
| Searching an LDAP Server | 57 |
| Example LDAP Filters | 57 |
| Determining LDAP Scope | 57 |
| LDAP_SCOPE_SUBTREE | 58 |
| LDAP_SCOPE_ONELEVEL | 58 |
| LDAP_SCOPE_BASE | 59 |
| Performing a JNDI Search | 59 |
| How the Search Program Works | 61 |
| Authenticated Searching | 62 |
| Restricting the Attributes Displayed | 64 |
| Adding Entries | 66 |
| Modifying an Entry | 72 |
| Delete an Entry | 74 |
| Storing and Retrieving Java Objects in LDAP | 75 |
| Traditional LDAP | 76 |
| Serialized Java | 77 |
| Java References | 77 |
| Returning to JNDI without LDAP | 77 |
| Example DNS Application | 78 |
| Summary | 80 |
| Chapter 3: Distributed Computing Using RMI | 83 |
| The RMI Architecture | 84 |
| The Stub and Skeleton Layer | 85 |
| Stubs | 86 |
| Skeletons | 86 |
| The Remote Reference Layer | 86 |
| The Transport Layer | 87 |
| Locating Remote Objects | 88 |
| Policy Files | 90 |
| RMI Exceptions | 91 |
| Developing Applications with RMI | 92 |
| Defining the Remote Interface | 93 |
| Implementing the Remote Interface | 93 |
| Writing the Client that Uses the Remote Objects | 95 |
| Generating Stubs and Skeletons | 96 |
| Registering the Object | 96 |
| Running the Client and Server | 97 |
| The RMISecurityManager | 98 |
| Parameter Passing in RMI | 98 |
| Primitive Parameters | 98 |
| Object Parameters | 99 |
| Remote Parameters | 99 |

Table of Contents

| | |
|--|------------|
| The Distributed Garbage Collector | 100 |
| Dynamically Loading Classes | 104 |
| Remote Callbacks | 108 |
| Object Activation | 111 |
| The Activation Group | 113 |
| ActivationID | 114 |
| Activation Descriptor | 115 |
| Making Objects Activatable | 116 |
| Step 1: Create the Remote Interface | 117 |
| Step 2: Create the Object Implementation | 117 |
| Step 3: Register the Object with the System | 117 |
| Alternative to Extending the Activatable Class | 119 |
| Starting Multiple JVMs other than with mid | 120 |
| Deactivation | 123 |
| Custom Sockets and SSL | 123 |
| RMI, Firewalls, and HTTP | 136 |
| HTTP Tunneling | 136 |
| HTTP-to-Port | 136 |
| HTTP-to-CGI | 137 |
| The SOCKS Protocol | 138 |
| Downloaded Socket Factories | 138 |
| RMI Over IIOP | 138 |
| Interoperability with CORBA | 139 |
| Writing Programs with RMI-IIOP | 140 |
| On the Server | 140 |
| In the Client | 141 |
| RMI-IIOP and Java IDL | 144 |
| The IDL File | 144 |
| The Server Implementation | 145 |
| The Client Implementation | 146 |
| RMI-IIOP and J2EE | 148 |
| Tuning RMI Applications | 149 |
| Summary | 154 |
| Chapter 4: Database Programming with JDBC | 157 |
| Database Drivers | 159 |
| Type 1 – JDBC-ODBC Bridge | 159 |
| Type 2 – Part Java, Part Native Driver | 160 |
| Type 3 – Intermediate Database Access Server | 161 |
| Type 4 – Pure Java Drivers | 161 |
| Getting Started | 162 |
| The java.sql Package | 162 |
| Connection Management | 163 |
| Database Access | 163 |

Table of Contents

| | |
|---|------------|
| Data Types | 164 |
| Database Metadata | 165 |
| Exceptions and Warnings | 165 |
| Loading a Database Driver and Opening Connections | 166 |
| JDBC URLs | 166 |
| DriverManager | 167 |
| Driver | 170 |
| Establishing a Connection | 170 |
| Creating and Executing SQL Statements | 174 |
| An Example: Movie Catalog | 175 |
| Querying the Database | 180 |
| Prepared Statements | 183 |
| Mapping SQL Types to Java | 186 |
| Transaction Support | 188 |
| Savepoints | 190 |
| Scrollable and Updateable Resultsets | 191 |
| Scrollable ResultSets | 192 |
| Updateable Resultsets | 197 |
| Batch Updates | 198 |
| The javax.sql Package | 199 |
| JDBC Data Sources | 200 |
| The javax.sql.DataSource Interface | 201 |
| The getConnection() Method | 201 |
| The getLoginTimeout() Method | 201 |
| The setLoginTimeout() Method | 202 |
| The getLogWriter() Method | 202 |
| The setLogWriter() Method | 202 |
| JNDI and Data Sources | 202 |
| Creating a Data Source | 203 |
| Retrieving a DataSource Object | 205 |
| Key Features | 205 |
| The Movie Catalog Revisited | 205 |
| Connection Pooling | 207 |
| Traditional Connection Pooling | 208 |
| Connection Pooling with the javax.sql Package | 210 |
| The javax.sql.ConnectionPoolDataSource Interface | 211 |
| The javax.sql.PooledConnection Interface | 211 |
| The javax.sql.ConnectionEventListener Interface | 212 |
| The javax.sql.ConnectionEvent Class | 212 |
| Connection Pooling Implementation | 213 |
| Distributed Transactions | 213 |
| What is a Transaction? | 214 |
| Brief Background | 214 |
| Transaction Processing – Concepts | 215 |
| Transaction Demarcation | 215 |
| Transaction Context and Propagation | 216 |
| Resource Enlistment | 216 |
| Two-Phase Commit | 216 |
| Building Blocks of Transaction Processing Systems | 217 |
| Application Components | 217 |
| Resource Managers | 218 |
| Transaction Manager | 218 |

Table of Contents

| | |
|--|---------------------|
| JDBC Distributed Transactions | 219 |
| The javax.sql.XADataSource Interface | 219 |
| The javax.sql.XAConnection Interface | 219 |
| The javax.transaction.UserTransaction Interface | 220 |
| Steps for Implementing Distributed Transactions | 221 |
| Configuration | 221 |
| Beginning a Transaction | 221 |
| Database Operations | 222 |
| Ending a Transaction | 223 |
| Special Precautions | 223 |
| RowSet Objects | 223 |
| The javax.sql.RowSet Interface | 225 |
| Properties | 225 |
| Events | 225 |
| Command Execution and Results | 226 |
| Types of RowSet Objects | 226 |
| The CachedRowSet Implementation | 227 |
| The JDBC Rowset Implementation | 228 |
| The Web Rowset Implementation | 229 |
| Summary | 230 |
| Chapter 5: Introduction to Web Containers | 233 |
| The HTTP Protocol | 234 |
| HTTP Request Methods | 234 |
| The GET Request Method | 235 |
| The POST Request Method | 235 |
| The HEAD Request Method | 235 |
| The HTTP Response | 235 |
| Web Containers and Web Applications | 236 |
| Java Servlets | 237 |
| JavaServer Pages | 239 |
| Deployment Descriptors | 241 |
| Structure of Web Applications | 241 |
| Types of Web Containers | 242 |
| A Simple Web Application | 242 |
| Prepare the Web Container | 243 |
| Create the HTML File | 243 |
| Create a Servlet | 243 |
| Constructing the Web Application | 244 |
| Deploying the Web Application | 248 |
| Running the Web Application | 250 |
| How the Application Works | 251 |
| The Greeting Servlet | 252 |
| The Deployment Descriptor | 254 |
| Summary | 255 |

| | |
|---|------------|
| Chapter 6: Servlet Programming | 259 |
| Overview of the Java Servlet API | 260 |
| Servlet Implementation | 264 |
| The Servlet Interface | 264 |
| The init() Method | 265 |
| The service() Method | 265 |
| The destroy() Method | 265 |
| The getServletConfig() Method | 266 |
| The getServletInfo() Method | 266 |
| The GenericServlet Class | 266 |
| The SingleThreadModel Interface | 267 |
| The HttpServlet Class | 268 |
| The service() Methods | 268 |
| The doXXX() Methods | 269 |
| The getLastModified() Method | 269 |
| Servlet Configuration | 270 |
| The ServletConfig Interface | 270 |
| The getInitParameter() Method | 270 |
| The getInitParameterNames() Method | 271 |
| The getServletContext() Method | 271 |
| The getServletName() Method | 271 |
| Obtaining a Reference to ServletConfig | 271 |
| During Servlet Initialization | 271 |
| Using the getServletConfig() Method | 271 |
| Servlet Exceptions | 272 |
| The ServletException Class | 272 |
| The UnavailableException Class | 272 |
| The Servlet Lifecycle | 273 |
| The Servlet Lifecycle – FreakServlet | 275 |
| Instantiation | 282 |
| Initialization | 282 |
| Service | 283 |
| Destroy | 285 |
| Requests and Responses | 286 |
| The ServletRequest Interface | 286 |
| Methods for Request Parameters | 287 |
| Methods for Request Attributes | 287 |
| Methods for Input | 288 |
| The ServletRequestWrapper Class | 289 |
| The HttpServletRequest Interface | 289 |
| Methods for Request Path and URL | 291 |
| Methods for HTTP Headers | 292 |
| The getMethod() Method | 292 |
| The HttpServletRequestWrapper Class | 292 |
| The ServletResponse Interface | 292 |
| Methods for Content Type and Length | 293 |
| Methods for Output | 293 |
| Methods for Buffered Output | 294 |

Table of Contents

| | |
|--|------------|
| The ServletResponseWrapper Class | 294 |
| The HttpServletResponse Interface | 295 |
| Methods for Error Handling | 295 |
| The <code>sendRedirect()</code> Method | 296 |
| The HttpServletResponseWrapper Class | 296 |
| Role of Wrapper Classes | 296 |
| Servlet Programming – Tech Support Application | 297 |
| Setting up the HTML Page | 298 |
| Prepare the Database | 299 |
| Writing the Servlet | 301 |
| Compile the Source | 303 |
| Construct the Web Application | 303 |
| Configuring the DataSource | 308 |
| Deploying the Web Application | 309 |
| Tech Support in Action | 312 |
| Summary | 315 |
| Chapter 7: Servlet Sessions, Context, and Collaboration | 317 |
| Statelessness and Sessions | 318 |
| Approaches to Session Tracking | 320 |
| URL Rewriting | 321 |
| Hidden Form Fields | 322 |
| Cookies | 322 |
| Session Tracking with the Java Servlet API | 324 |
| Session Creation and Tracking | 325 |
| The <code>HttpSession</code> Interface | 326 |
| Methods for Session Lifetime | 326 |
| Demonstrating Session Lifecycle with Cookies | 328 |
| Session Lifecycle without Cookies | 332 |
| Methods for Managing State | 333 |
| Demonstrating State Management | 334 |
| Session Lifecycle Event Handling | 337 |
| The <code> HttpSessionListener </code> Interface | 337 |
| The <code> HttpSessionActivationListener </code> Interface | 338 |
| The <code> HttpSessionEvent </code> Class | 339 |
| Session Attribute Event Handling | 339 |
| The <code> HttpSessionBindingListener </code> Interface | 339 |
| The <code> HttpSessionAttributeListener </code> Interface | 340 |
| The <code> HttpSessionBindingEvent </code> Class | 340 |
| A Simple Shopping Cart using Sessions | 341 |
| The Catalog Servlet | 341 |
| The ShoppingCart Servlet | 343 |
| Servlet Context | 346 |
| The <code>ServletContext</code> Interface | 346 |
| ServletContext Lifecycle Event Handling | 349 |
| The <code> ServletContextListener </code> Interface | 349 |
| <code>ServletContextAttributeListener</code> | 349 |

Table of Contents

| | |
|---|------------|
| <u>A Chat Application using Context and Sessions</u> | 350 |
| The ChatRoom Class | 352 |
| The ChatEntry Class | 353 |
| The Administration Servlet | 353 |
| Servlets for Chatting | 357 |
| Chat Setup | 364 |
| Servlet Collaboration | 367 |
| Servlet Chaining | 367 |
| Request Dispatching | 368 |
| RequestDispatcher Interface | 368 |
| Obtaining a RequestDispatcher Object | 369 |
| Tech Support Revisited | 369 |
| The techsupp.html Page | 370 |
| TechSupportServlet | 372 |
| The register.html Page | 374 |
| RegisterCustomerServlet | 375 |
| ResponseServlet | 376 |
| BannerServlet | 377 |
| Tech Support Setup and Deployment | 378 |
| Using RequestDispatchers for Collaboration | 379 |
| Summary | 381 |
| Chapter 8: Filters for Web Applications | 383 |
| What is a Filter? | 384 |
| A Sample Filter | 386 |
| Deployment | 389 |
| The Filter API | 390 |
| The Filter Interface | 391 |
| The FilterConfig Interface | 392 |
| The FilterChain Interface | 393 |
| Deployment Descriptor for Filters | 393 |
| The Chat Application with Filters | 395 |
| Message Logging | 396 |
| Message Moderation | 398 |
| Deployment Descriptor | 403 |
| Deployment | 405 |
| Chat with Logging and Moderation | 406 |
| Summary | 407 |
| Chapter 9: Web Deployment, Authentication, and Packaging | 409 |
| Web Application Structure | 409 |
| Directory Structure | 410 |
| Web Archive Files | 412 |
| Deploying a WAR File | 413 |
| When should WAR Files be Used? | 414 |

Table of Contents

| | |
|--|------------|
| Mapping Requests to Applications and Servlets | 414 |
| Securing Web Applications | 419 |
| Programmatic Security | 422 |
| Form-Based Authentication | 422 |
| Deployment Configuration | 425 |
| Context Initialization Parameters | 426 |
| Servlet Initialization Parameters | 427 |
| Loading Servlets on Startup | 427 |
| Session Timeout | 428 |
| MIME Mappings | 428 |
| Welcome Files | 429 |
| Error Pages | 430 |
| Sending Errors | 430 |
| Throwing ServletException | 431 |
| Handling HTTP Errors and Exceptions | 432 |
| Distributable Applications | 433 |
| Summary | 436 |
| Chapter 10: JSP Basics and Architecture | 439 |
| The JSP 1.2 Specification | 439 |
| Introducing JSP | 440 |
| The Nuts and Bolts | 444 |
| JSP Directives | 445 |
| The page Directive | 445 |
| The include Directive | 447 |
| The taglib Directive | 450 |
| Scripting Elements | 450 |
| Declarations | 450 |
| Scriptlets | 452 |
| Expressions | 454 |
| Comments | 455 |
| Standard Actions | 455 |
| <jsp:useBean> | 456 |
| <jsp:setProperty> | 458 |
| <jsp:getProperty> | 460 |
| <jsp:param> | 463 |
| <jsp:include> | 463 |
| <jsp:forward> | 468 |
| <jsp:plugin> | 470 |
| Implicit Objects | 473 |
| The Request Object | 473 |
| The Response Object | 473 |
| The PageContext Object | 473 |
| The Session Object | 473 |
| The Application Object | 474 |
| The Out Object | 474 |
| The Config Object | 474 |
| The Page Object | 474 |

| | |
|--|------------|
| <u>Scope</u> | 474 |
| <u>Page Scope</u> | 474 |
| <u>Request Scope</u> | 475 |
| <u>Session Scope</u> | 475 |
| <u>Application Scope</u> | 475 |
| <u>JSP Pages as XML Documents</u> | 475 |
| <u>Directives</u> | 476 |
| <u>Scripting Elements</u> | 476 |
| <u>Actions</u> | 477 |
| <u>Example Page</u> | 477 |
| JSP Technical Support | 477 |
| <u>Application Design</u> | 479 |
| <u>The Welcome Page</u> | 480 |
| <u>The Request-Processing JSP</u> | 481 |
| <u>The JDBCHelper class</u> | 481 |
| <u>The TechSupportBean</u> | 483 |
| <u>The Registration Form</u> | 485 |
| <u>The Registration JSP</u> | 485 |
| <u>The Response and Banner JSP pages</u> | 486 |
| <u>The Error Page</u> | 487 |
| <u>Deploying the Application</u> | 487 |
| JSP Design Strategies | 488 |
| <u>Page-Centric or Client-Server Designs</u> | 489 |
| <u>Page-View</u> | 490 |
| <u>Page-View with Bean</u> | 491 |
| <u>The Front Controller Pattern</u> | 491 |
| <u>Implementing a Front Controller Architecture</u> | 492 |
| <u>Controller Servlet</u> | 493 |
| <u>Request Handlers</u> | 493 |
| <u>Page Beans</u> | 494 |
| <u>JSP Views</u> | 494 |
| <u>Implementation</u> | 494 |
| <u>Benefits</u> | 503 |
| <u>Using a Generic Controller Framework</u> | 504 |
| <u>Summary</u> | 504 |
| Chapter 11: JSP Tag Extensions | 507 |
| Tag Extensions | 507 |
| <u>JSP 1.2 Tag Extension Enhancements</u> | 509 |
| A Simple Tag | 510 |
| Anatomy of a Tag Extension | 514 |
| <u>Tag Handlers</u> | 515 |
| <u>The javax.servlet.jsp.tagext.Tag Interface</u> | 515 |
| <u>The javax.servlet.jsp.tagext.IterationTag Interface</u> | 518 |
| <u>The javax.servlet.jsp.tagext.BodyTag Interface</u> | 518 |
| <u>The javax.servlet.jsp.tagext.BodyContent Class</u> | 520 |
| <u>Convenience Classes</u> | 521 |
| <u>Objects Available to Tag Handlers</u> | 522 |
| <u>The Simple Example Revisited</u> | 522 |

Table of Contents

| | |
|--|------------|
| Tag Library Descriptors | 523 |
| Using Tag Extensions in JSP Pages | 526 |
| Deploying and Packaging Tag Libraries | 527 |
| Mapping in web.xml | 527 |
| Packaged Tag Library JAR | 528 |
| Default Mapping | 528 |
| Mapping Combinations | 529 |
| Writing Tag Extensions | 529 |
| Processing Attributes | 529 |
| Body Content | 533 |
| Tags Introducing Scripting Variables | 534 |
| Changes to the Tag Handler | 535 |
| Scripting Variables Example | 535 |
| Programmatic Definition of Scripting Variables | 538 |
| Iteration and Manipulation of Body Content | 540 |
| Repeated Evaluation with the IterationTag Interface | 540 |
| Body Tags that Filter their Content | 544 |
| Tag Nesting | 546 |
| Validating the Use of Tag Extensions in JSP Pages | 550 |
| Handling Errors | 551 |
| TryCatchFinally – A Danger to Good Design? | 557 |
| Application Lifecycle Events | 557 |
| Tag Extension Idioms | 562 |
| Summary | 563 |
| Chapter 12: Writing JSP Applications with Tag Libraries | 567 |
| Benefits of Using Custom Tag Libraries | 568 |
| Examples of Existing Tag Libraries | 569 |
| Introducing the JSP Standard Tag Library (JSPTL) | 569 |
| Obtaining the JSPTL | 570 |
| What does the JSPTL Cover? | 570 |
| Getting Started with the JSPTL | 571 |
| Integrating the JSPTL into Your JSP Pages | 572 |
| An Iteration Example | 572 |
| Getting it Running | 573 |
| The JSPTL Tags | 576 |
| Some JSPTL Design Considerations | 576 |
| Some Basic Tags | 577 |
| Control Flow Tags | 578 |
| Iteration Tags | 580 |
| Iteration Status | 586 |
| Iteration Tag Extensibility | 586 |
| Conditional Tags | 587 |
| A Registration and Authentication Application | 590 |
| The Welcome Page | 591 |

Table of Contents

| | |
|---|----------------|
| The Registration Controller Servlet | 592 |
| <u>The Registration Form Page</u> | 596 |
| Tags for HTML Form Handling | 599 |
| Thanks for Registering! | 603 |
| Internationalizing our JSP Code | 605 |
| The Registration Error Page | 610 |
| The Login Form Page | 611 |
| Viewing your User Profile | 612 |
| <u>Some Fundamental Tag Ideas in JSP</u> | 615 |
| The Login Error Page | 618 |
| Viewing the User's Favorite Web Site | 619 |
| Server-Side Includes (SSI) | 619 |
| Some Issues with Server-Side Includes | 620 |
| A Server-Side Include Tag | 620 |
| Administration: The Display-All-Users Page | 623 |
| Administration: An XML-based Web Service | 626 |
| Deploying the Registration Application | 629 |
| What Lies Ahead: The Standard Tag Library | 631 |
| Summary | 632 |
| Chapter 13: JavaMail | 635 |
| Mail Protocols | 636 |
| SMTP | 636 |
| POP3 | 637 |
| IMAP | 637 |
| MIME | 638 |
| JavaMail Overview | 638 |
| Installation and Configuration | 639 |
| Quick, Send Me an e-Mail! | 639 |
| JavaMail API | 641 |
| <u>javax.mail.Session</u> | 641 |
| <u>javax.mail.Authenticator</u> | 642 |
| <u>javax.mail.Message</u> | 644 |
| <u>javax.mail.internet.MimeMessage</u> | 645 |
| <u>javax.mail.Part</u> | 648 |
| <u>Message Flags</u> | 654 |
| <u>javax.mail.Address</u> | 655 |
| <u>javax.mail.internet.InternetAddress</u> | 656 |
| <u>javax.mail.internet.NewsAddress</u> | 656 |
| <u>javax.mail.Store</u> | 657 |
| Accessing Folders | 658 |
| <u>javax.mail.URLName</u> | 660 |
| <u>javax.mail.Folder</u> | 661 |
| Opening and Closing Folders | 661 |
| <u>Listing Messages</u> | 662 |
| Copying and Moving Messages | 664 |
| Searching Messages | 664 |
| <u>javax.mail.Transport</u> | 666 |

Table of Contents

| | |
|--|------------|
| Working with Mail | 667 |
| Sending Mail | 667 |
| Sending Attachments | 669 |
| Reading Mail | 672 |
| Deleting Mail | 676 |
| Receiving Attachments | 677 |
| Saving and Loading Mail | 678 |
| JavaMail Resources | 683 |
| Summary | 683 |
| Chapter 14: EJB Architecture and Design | 685 |
| What are EJBs? | 686 |
| Enterprise JavaBeans vs. JavaBeans | 687 |
| Varieties of Beans | 688 |
| Why use EJBs? | 689 |
| The EJB Container and its Services | 690 |
| Persistence | 691 |
| Declarative Transactions | 692 |
| Declarative Security | 692 |
| Error Handling | 692 |
| Component Framework for Business Logic | 692 |
| Scalability | 693 |
| Portability | 693 |
| Manageability | 693 |
| How the Container Provides Services | 693 |
| Contracts | 694 |
| Services | 695 |
| Interposition | 696 |
| Working with EJBs | 697 |
| The Client Developer's View | 698 |
| The Bean Provider's View | 700 |
| What an EJB Cannot Do | 711 |
| Threads or the Threading API | 712 |
| AWT | 713 |
| Act as a Network Server | 713 |
| Write to Static Fields | 713 |
| The java.io Package | 713 |
| Load a Native Library | 713 |
| Use 'this' as an Argument or Return Value | 713 |
| Loopback Calls | 714 |
| EJB Components on the Web | 714 |
| Client-Tier Access to EJBs | 717 |
| Design of the EJB Tier | 718 |
| UML Use Cases | 718 |
| Analysis Objects | 718 |
| Analysis vs. Implementation | 720 |
| The Role of State in the Interface Object | 722 |

| | |
|---|------------|
| An Example of EJB Design | 722 |
| Create a Product | 724 |
| Place an Order | 724 |
| Cancel an Order | 725 |
| Select an Order for Manufacture | 725 |
| Build a Product | 725 |
| Ship an Order | 725 |
| List Overdue Orders | 725 |
| Summary | 731 |
| Chapter 15: Session Beans and Business Logic | 733 |
| Session Beans and State | 734 |
| Representing Business Logic | 734 |
| Session Beans as a Façade | 735 |
| The Difficult Problem of Conversational State | 737 |
| Session Beans and Persistent Storage | 739 |
| The Financial Aid Calculator Bean | 739 |
| The Stateless Financial Need Calculator Bean | 740 |
| The Financial Need Calculator Remote Interface | 740 |
| The Financial Need Client | 741 |
| The Financial Need Calculator Home Interface | 742 |
| The Financial Need Calculator Implementation Class | 742 |
| The Deployment Descriptor | 746 |
| The Stateful Financial Need Calculator Bean | 748 |
| The Stateful Financial Need Remote Interface | 748 |
| The Stateful Financial Need Calculator Home Interface | 748 |
| The Stateful Financial Next Calculator Implementation Class | 749 |
| The Client for the Stateful Financial Need Calculator | 752 |
| The Deployment Descriptors | 753 |
| Combining Stateful and Stateless Beans | 755 |
| The New Stateful Need Calculator Interfaces and Client | 755 |
| The New Stateful Need Calculator Implementation Class | 756 |
| The Deployment Descriptors | 758 |
| Implementing Our Manufacturing Application | 760 |
| Clients and the Business Logic Interfaces | 762 |
| The ManageOrders EJB | 763 |
| The Manufacture EJB | 765 |
| The Application Clients | 766 |
| Stateless Session Bean Implementation | 775 |
| Business Methods | 775 |
| Implementation Helper Methods | 781 |
| Lifecycle and Framework Methods | 783 |
| Stateful Session Bean Implementation | 783 |
| Summary | 790 |
| Chapter 16: Entity Beans and Persistence | 793 |
| Why not use Session Beans? | 794 |
| Using a Stateful Session bean | 794 |

Table of Contents

| | |
|---|---------------------|
| Using a Stateless Session Bean | 797 |
| Benefits of Entity Beans | 799 |
| Container- vs. Bean-Managed Persistence | 799 |
| New Features Introduced in EJB 2.0 for CMP | 801 |
| Abstract Accessors | 802 |
| Local Interfaces | 803 |
| Relationships | 803 |
| EJB Query Language | 804 |
| The SportBean Laboratory | 807 |
| Primary Keys | 807 |
| The C.R.U.D. Callbacks | 809 |
| Create | 810 |
| Read | 814 |
| Update | 817 |
| Delete | 818 |
| BMP Callbacks vs. CMP Callbacks | 819 |
| The Deployment Descriptor | 819 |
| Persistence in the Deployment Descriptors | 820 |
| Caching | 822 |
| Finder Methods | 824 |
| Implementing Finder Methods | 825 |
| Activation and Passivation | 828 |
| The Complete Lifecycle | 828 |
| Reentrancy | 829 |
| Completing the Sports Team Example | 830 |
| Configuring WebLogic 6.1 for our EJB's | 835 |
| Relationships | 837 |
| Creating Local Interfaces | 837 |
| Defining the Relationships | 837 |
| Completing our Manufacturing Application | 838 |
| The Order Bean | 839 |
| The Product Bean | 844 |
| The Complete Deployment Descriptor | 848 |
| Running the Manufacturing Application | 855 |
| Summary | 858 |
| Chapter 17: EJB Container Services | 861 |
| Transactions | 862 |
| Transactions Without a Container | 864 |
| Declarative Semantics for Transactions | 871 |
| Specifying Transactional Attributes | 872 |
| Choosing Transaction Attributes | 873 |
| User-Controlled Transactions | 875 |
| Isolation Levels | 877 |
| Long Transactions | 879 |
| Optimistic Locking | 880 |
| Pessimistic Locking | 880 |
| Two-Phase Commit | 881 |
| Security | 881 |

Table of Contents

| | |
|---|------------|
| <u>Specifying the Security Requirements</u> | 883 |
| Security Roles | 883 |
| Method Permissions | 884 |
| <u>Programmatic Access Control</u> | 886 |
| Security and Application Design | 888 |
| Exceptions | 889 |
| <u>Application Exceptions</u> | 889 |
| Predefined Application Exceptions | 892 |
| System Exceptions | 893 |
| Communication | 895 |
| Communication between Heterogeneous Servers | 895 |
| In-VM Method Calls | 897 |
| Summary | 897 |
| Chapter 18: Development and Deployment Roles | 901 |
| The Enterprise Bean Provider | 902 |
| The Application Assembler | 916 |
| The Deployer | 927 |
| The System Administrator | 936 |
| Container/Application-Server Vendor | 938 |
| A Web Interface for the Manufacturing App | 939 |
| Troubleshooting Tips | 963 |
| Summary | 964 |
| Chapter 19: JMS and Message-Driven Beans | 967 |
| A Brief History of Messaging | 968 |
| The Java Message Service | 969 |
| Point-to-Point | 970 |
| Publish/Subscribe | 971 |
| The JMS Architecture | 972 |
| Point-to-Point Queue Example | 973 |
| Producing a Message | 974 |
| <u>Consuming a Message Synchronously</u> | 977 |
| <u>Consuming a Message Asynchronously</u> | 979 |
| Publish/Subscribe Topic Example | 982 |
| The JMS API | 989 |
| Using Transactions with JMS | 993 |
| Three Industrial-Strength JMS Implementations | 994 |
| SwiftMQ | 994 |
| SpiritWave | 995 |
| SonicMQ | 995 |

Table of Contents

| | |
|--|-------------|
| Message-Driven Beans | 995 |
| Message-Driven Bean Lifecycle | 997 |
| Transactions in Message-Driven Beans | 997 |
| Bean-Managed Transactions | 997 |
| Container-Managed Transactions | 997 |
| A Message-Driven Bean Example | 998 |
| Deploying the Message-Driven Bean | 998 |
| Summary | 1006 |
| Chapter 20 : The J2EE Connector Architecture | 1009 |
| EIS Integration and the Role of JCA | 1010 |
| J2EE Connector Architecture and its Elements | 1011 |
| Comparing JCA with JDBC | 1013 |
| The Resource-adapter and its Contracts | 1014 |
| Application Contracts | 1014 |
| System-Level Contracts | 1014 |
| Packaging and Deploying a Resource-adapter | 1016 |
| Packaging a Resource-adapter | 1017 |
| The Resource-adapter Deployment Descriptor (ra.xml) | 1019 |
| Roles and Responsibilities in Deployment | 1021 |
| Deployment Options | 1022 |
| Deploying a Resource-adapter as a J2EE Application | 1022 |
| Black-Box Resource-adapters | 1023 |
| Using a Black-Box Adapter – The DemoConnector Example | 1023 |
| Selecting a Black-box Adapter | 1024 |
| Deploying the Black-box Resource-adapter | 1024 |
| Testing the Black-box Resource-adapter | 1030 |
| The Common Client Interface (CCI) | 1039 |
| CCI Interfaces and Classes | 1040 |
| Connection with a Managed-Application (J2EE) | 1042 |
| Connection with a Non-Managed Application (Two-tier) | 1043 |
| Using a CCI Black-box Adapter | 1045 |
| Developing a Session Bean with CCI | 1045 |
| Deploying the CCI Black-box Resource-adapter | 1050 |
| Deploying and Testing the CCI Application | 1054 |
| Benefits of the J2EE Connector Architecture | 1055 |
| Enterprise Application Integration (EAI) with JCA | 1055 |
| Web-Enabled Enterprise Portals with JCA | 1055 |
| Business-to-Business Integration with JCA | 1056 |
| Missing Elements in JCA 1.0 | 1056 |
| Summary | 1057 |
| Chapter 21: Design Considerations for J2EE Applications | 1059 |
| The World is Still Changing | 1060 |

Table of Contents

| | |
|---|-------------|
| Architecture and Design | 1061 |
| Architectural Styles | 1061 |
| Design Context | 1063 |
| The Business Requirements | 1064 |
| The Existing System | 1064 |
| The Desired System | 1065 |
| The Business Context | 1066 |
| Elaborating the Requirements | 1067 |
| Building the Model | 1067 |
| Exploring the Model | 1068 |
| Elaborating the Context | 1068 |
| Fitting the Terrain | 1068 |
| Distributed Design | 1069 |
| Choosing and Refining an Architecture | 1071 |
| Architecture for the Purchase Order System | 1071 |
| Iteration and Feedback | 1072 |
| Applying Patterns | 1073 |
| What are Patterns? | 1073 |
| J2EE Patterns | 1074 |
| Front Controller Pattern | 1075 |
| Composite View Pattern | 1076 |
| Session Façade Pattern | 1078 |
| Service Locator Pattern | 1079 |
| Value Object Pattern | 1080 |
| Business Delegate Pattern | 1082 |
| Data Access Object Pattern | 1082 |
| View Helper Pattern | 1083 |
| Dispatcher View Pattern | 1084 |
| Service to Worker Pattern | 1085 |
| Value List Handler Pattern (Page-by-Page Iterator) | 1085 |
| Fast Lane Reader Pattern | 1087 |
| Start at the Beginning | 1088 |
| Displaying Product Data to the User | 1088 |
| Abstracting Data Access | 1089 |
| Separating Functionality from Presentation | 1089 |
| Partitioning User Interaction, Presentation, and Data | 1090 |
| Evolution of the System | 1092 |
| Adding the Middle-Tier | 1094 |
| Going Shopping | 1096 |
| Encapsulating the Ordering Workflow | 1096 |
| Stateful vs. Stateless Models | 1098 |
| Submitting the Order | 1099 |
| Choosing an Asynchronous Mechanism | 1100 |
| Scalability and Availability through Asynchronicity | 1101 |
| Issues with Asynchronous Systems | 1102 |
| Approving the Order | 1102 |
| Beyond the Purchase Order System | 1104 |
| EJB Interface Design | 1104 |
| Distributed Events | 1106 |
| Designing for Databases | 1106 |

Table of Contents

| | |
|---|-----------------|
| Lessons Learned | 1107 |
| Separate Concerns Where Possible | 1107 |
| Minimize Network Traffic | 1107 |
| Use Abstraction to Aid Flexibility | 1108 |
| Use Common Patterns | 1108 |
| Reduce Coupling with Asynchronous Mechanisms | 1108 |
| Plan Transactions | 1109 |
| Summary | 1109 |
| Chapter 22: J2EE and Web Services | 1111 |
| What are Web Services? | 1112 |
| Smart Services | 1113 |
| Web Service Technologies | 1113 |
| SOAP | 1113 |
| Interoperability | 1115 |
| Implementations | 1115 |
| SOAP Messages with Attachments (SwA) | 1115 |
| <u>WSDL</u> | <u>1116</u> |
| <u>WSDL Document</u> | <u>1116</u> |
| UDDI | 1120 |
| Implementation | 1120 |
| ebXML | 1121 |
| J2EE Technologies for Web Services | 1121 |
| JAXP | 1122 |
| JAXB | 1122 |
| XML Schema | 1123 |
| XML Data Binding | 1123 |
| <u>JAXM</u> | <u>1125</u> |
| <u>JAX-RPC</u> | <u>1126</u> |
| JAXR | 1127 |
| Architecture | 1127 |
| Design Goals | 1128 |
| Developing Web Services | 1128 |
| Web Service Architecture | 1128 |
| Locating Web Services | 1129 |
| The Web Service Interface | 1129 |
| Implementing the Business Logic | 1130 |
| Integrating Other Resources | 1130 |
| Returning Results to Clients | 1130 |
| A Simple Web Service | 1130 |
| Developing the StockQuote Java File | 1131 |
| Creating the JAR | 1132 |
| Generating WSDL Files | 1132 |
| The StockQuote_Service.wsdl File | 1134 |
| <u>The StockQuote_Service-interface.wsdl File</u> | <u>1135</u> |
| <u>Generating the Proxy Class</u> | <u>1137</u> |
| Registering the Service with the Web Server | 1137 |
| Developing a Client | 1139 |

Table of Contents

| | |
|---|-------------|
| Making Services Smarter | 1140 |
| Shared Context | 1141 |
| Smart Web Services | 1141 |
| The Sun ONE Initiative | 1142 |
| Functional Architecture Overview | 1142 |
| Smart Web Services Architecture | 1143 |
| Smart Delivery | 1144 |
| Smart Management | 1144 |
| Smart Process | 1144 |
| Smart Policy | 1144 |
| Vendor Support for Web Services | 1144 |
| Simple Web Service | 1144 |
| Complex Web Service | 1145 |
| Summary | 1145 |
| Chapter 23: Choosing a J2EE Implementation | 1147 |
| Application Servers | 1147 |
| Implementing the J2EE Specifications | 1150 |
| Competition in the Application Server Market | 1151 |
| J2EE Implementations | 1152 |
| Value-Added Features | 1154 |
| Evaluation Parameters | 1158 |
| Development Community | 1161 |
| Summary | 1162 |
| Chapter 24: J2EE Packaging and Deployment | 1165 |
| J2EE Packaging Overview | 1166 |
| What can be Packaged? | 1166 |
| Packaging Roles | 1168 |
| The Limitations of Packaging | 1170 |
| Understanding Class Loading Schemes | 1170 |
| The Pre-EJB 2.0 Option | 1171 |
| The Post-EJB 2.0 Option | 1173 |
| An Ambiguity in the J2EE Specification | 1173 |
| Configuring J2EE Packages | 1174 |
| The Enterprise Application Development Process | 1174 |
| The Structure of a J2EE Package | 1175 |
| Working with the EAR Deployment Descriptor | 1177 |
| Issues with the Ordering of Modules | 1182 |
| Issues with Dependency Packages | 1183 |
| Solutions | 1183 |
| Dependency Example | 1185 |
| The Impact of Dependency Libraries | 1188 |
| Summary | 1189 |
| Index | 1191 |

Copyrighted image

Copyrighted material

Introduction

Welcome to the third time out for *Professional Java Server Programming*. Unlike the change from the first to second editions, the differences to the J2EE 1.3 version are relatively minor. The most basic changes is that all the chapters on servlets, JSP, and EJB have been updated to reflect the changes to the relevant specifications, for example EJB 2.0. In addition, new chapters have been included for Features new to J2EE such as the Connector Architecture, and some chapters which were less relevant to core J2EE development have been removed.

The J2EE 1.3 Edition

The latest release of JSR-58, more commonly known as the **Java 2 Platform, Enterprise Edition (J2EE)**, represents the evolution of Sun Microsystem's server-side development platform into a more mature and sophisticated specification. Beyond the inclusion of some new sub-specifications such JAAS (the Java Authentication and Authorization Service) and the Connector Architecture, the actual container-centric nature of the J2EE specification has not changed significantly.

The more noticeable changes in this release relate to the modifications made to the sub-specifications, notably Servlet, JavaServer Pages (JSP) and Enterprise JavaBeans (EJB). Servlets gain events and filtering; JSP gain a new XML syntax and enhancements to the custom tag mechanisms; and EJB has some significant changes to its container-managed persistence model.

The current API versions in J2EE 1.3 stand at:

- Servlets 2.3
- JavaServer Pages 1.2
- Enterprise JavaBeans 2.0
- JDBC 2.0 Extension
- Java Message Service 1.0

- ❑ Java Transaction API 1.0
- ❑ JavaMail 1.2
- ❑ Java Activation Framework 1.0
- ❑ Java API for XML Processing 1.1
- ❑ Java Connector Architecture 1.0
- ❑ Java Authentication and Authorization Service 1.0

Over the course of this book we'll be looking at all of them.

What's Changed in this Edition of the Book?

You will find that a certain amount of the material in this J2EE 1.3 edition is very similar to that of its predecessor, the J2EE edition. This is because rather than significantly revamp what was essentially good, solid material, we've elected to simply update where relevant to remain consistent with the latest specs as identified by J2EE. In some cases, this has meant the inclusion of an additional chapter, but in others you'll find only small changes.

Having said that, there are some significant differences at the chapter level between this and the previous edition. The J2EE 1.3 edition contains chapters on the Java Connector Architecture, Web Services, Choosing an J2EE Implementation, and J2EE Packaging.

What's gone are the more extraneous chapters that were not core to learning about J2EE application development, such as Internationalization.

Who is this Book For?

This book is aimed at professional Java programmers who although they may not have much practical experience of, are at least familiar with, the fundamental concepts of network and web programming. It also assumes familiarity with the Java language and the core APIs – through reading Beginning Java 2, or some other tutorial book that covers similar ground. All the concepts that relate to server-side Java programming, however, will be covered assuming no prior knowledge.

Having said that, some familiarity with the basic server-side Java technologies is recommended as this book covers a large area of ground very quickly and does not claim to be exhaustive in all areas.

Copyrighted image

What's Covered in this Book

In this book, we discuss three things:

- ❑ The rules in the technology's specifications that developers must follow to write enterprise components
- ❑ The benefits and limits of the typical real-world vendor implementations of the J2EE specification
- ❑ The resulting practical aspects of real-word design using the J2EE technologies

The book has the following basic structure:

- We'll start with a look at the latest demands placed on a Java enterprise developer and how Java (more particularly J2EE) rises to meet these challenges. You'll also get your first real taste of the J2EE container architecture.
- After we're up to speed on the J2EE architecture we'll start by looking at some of the fundamental technologies in enterprise development: RMI, JDBC, and JNDI.
- Then we'll get back into J2EE more explicitly by looking at how to develop web components using Java servlets.
- Once we understand the servlet technology we'll look at how JavaServer Pages takes it and extends to provide a more flexible means of creating dynamic web content.
- We'll then take a step further into the enterprise by looking at the sophisticated component technology of Enterprise JavaBeans.
- Finally, we'll look at some larger J2EE issues such as design considerations and how to package your J2EE applications.

What You Need to Use this Book

Most of the code in this book was tested with the Java 2 Platform, Standard Edition SDK (JDK 1.3) and the Java 2 Platform, Enterprise Edition SDK 1.3 Reference Implementation. However, for some of the chapters, either the reference implementation is not sufficient or you need some additional software:

Web Container

In order to run the web components used in this book you will need a web container that supports the Servlet 2.3 and the JSP 1.2 specifications. We used the Reference Implementation, which uses the Jakarta Tomcat engine under the hood. You may need the latest build of Tomcat (available from <http://jakarta.apache.org/tomcat>) in order to run some of the JSP tag library examples.

EJB Container

For the EJB chapters you will also need an EJB container supporting version 2.0 of the EJB specification. We used BEA's WebLogic Server 6.1 – <http://www.bea.com/>.

Databases

Several of the chapters also require access to a database. For these chapters we used:

- Cloudscape (an in-process version comes with the J2EE RI), <http://www.cloudscape.com/>.

Additional Software

Finally, there are a few additional pieces of software that a couple of chapters also require:

- Sun's JNDI SDK, which is included with JDK 1.3
- Java Secure Sockets Extension (JSSE), 1.0.1, <http://java.sun.com/products/jsse/>
- LDAP server – Netscape's iPlanet Directory Server version 4.11, <http://www.iplanet.com/>

- SMTP and/or POP3 service
- The JSP Standard Tag Library, <http://jakarta.apache.org/taglibs>
- The IBM Web Services Toolkit, <http://www.alphaworks.ibm.com/tech/webservicestoolkit>

The code in the book will work on a single machine, provided it is networked (that is, it can see <http://localhost> through the local browser).

The complete source code from the book is available for download from:

<http://www.apress.com/>

Conventions

To help you get the most from the text and keep track of what's happening, we've used a number of conventions throughout the book.

For instance:

Copyrighted image

While the background style is used for asides to the current discussion.

As for styles in the text:

- When we introduce them, we **highlight** important words.
- We show keyboard strokes like this: *Ctrl-A*
- We show filenames and code within the text like so: `doGet()`
- Text on user interfaces and URLs are shown as: **Menu**

We present code in three different ways. Definitions of methods and properties are shown as follows:

```
protected void doGet(HttpServletRequest req, HttpServletResponse resp)
                      throws ServletException, IOException
```

Example code is shown:

```
In our code examples, the code foreground style shows new, important,
pertinent code
while code background shows code that's less important in the present context,
or has been seen before.
```

Customer Support

We always value hearing from our readers, and we want to know what you think about this book: what you liked, what you didn't like, and what you think we can do better next time. You can send us your comments, either by returning the reply card in the back of the book, or by e-mail to support@apress.com. Please be sure to mention the book title in your message.

How to Download the Sample Code for the Book

When you visit the Apress site, <http://www.apress.com/>, simply locate the title through our Search facility or by using one of the title lists. Click on Download in the Code column, or on Download Code on the book's detail page.

The files that are available for download from our site have been archived using WinZip. When you have saved the attachments to a folder on your hard-drive, you need to extract the files using a de-compression program such as WinZip or PKUnzip. When you extract the files, the code is usually extracted into chapter folders. When you start the extraction process, ensure your software (WinZip, PKUnzip, etc.) is set to use folder names.

Errata

We've made every effort to make sure that there are no errors in the text or in the code. However, no one is perfect and mistakes do occur. If you find an error in one of our books, like a spelling mistake or a faulty piece of code, we would be very grateful for feedback. By sending in errata you may save another reader hours of frustration, and of course, you will be helping us provide even higher quality information. Simply e-mail the information to support@apress.com; your information will be checked and if correct, posted to the errata page for that title, or used in subsequent editions of the book.

To find errata on the web site, go to <http://www.apress.com/>, and simply locate the title through our Advanced Search or title list. Click on the Book Errata link, which is below the cover graphic on the book's detail page.

E-mail Support

If you wish to directly query a problem in the book with an expert who knows the book in detail then e-mail support@apress.com, with the title of the book and the last four numbers of the ISBN in the subject field of the e-mail. A typical e-mail should include the following things:

- The title of the book, last four digits of the ISBN, and page number** of the problem in the Subject field.
- Your name, contact information, and the problem** in the body of the message.

Introduction

Copyrighted image

Copyrighted material

The J2EE Platform

Today, Java is one of the most mature and commonly used programming languages for building enterprise software. The evolution of Java from a means of developing applets to be run in browsers, to a programming model capable of driving today's enterprise applications has been remarkable. Over the years, Java has evolved into three different platform editions, each addressing a distinct set of programming needs:

- **The Java 2 Platform, Standard Edition (J2SE):**
This is the most commonly used of these Java platforms, consisting of a run-time environment and a set of APIs for building a wide variety of applications ranging from applets, through standalone applications that run on various platforms, to client applications for various enterprise applications.
- **The Java 2 Platform, Enterprise Edition (J2EE):**
J2EE is a platform for building server-side applications. As you will see throughout this book, server-side applications have additional requirements during the development phase. J2EE provides the infrastructure for meeting these needs.
- **The Java 2 Platform, Micro Edition (J2ME):**
The latest addition to the Java family, this edition enables the building of Java applications for "micro-devices" (devices with limited display and memory support, such as mobile phones, PDAs, etc.).

Of these three, J2SE is the most commonly used form of Java technology, and forms the basis for both J2EE and J2ME. J2SE is what is usually referred to as the **Java Development Kit (JDK)**. The current version of J2SE is 1.3, while the next version of J2SE, 1.4 is available as a beta.

This book is a journey through J2EE and in particular the 1.3 version of the specification. Since J2EE 1.3 requires J2SE 1.3, we will also be discussing some of the J2SE 1.3 features as appropriate.

In this book, we will start by introducing the programming needs for enterprise application development, the various technologies that are available for building such applications, and how to build such applications using those technologies. As we progress from chapter to chapter, we'll come across a wide variety of technologies, some of which are more complex than others, each fulfilling a specific set of technical needs.

This book will introduce you to almost the entire range of J2EE technologies. Despite the complexity of some of these areas, this book attempts to introduce each of these topics from fundamentals. In this way, you can study these topics in whichever order you prefer.

Programming for the Enterprise

J2EE has now been around for around four years. Over these years, it has replaced several proprietary and non-standard technologies as the preferred choice for building e-commerce and other web-based enterprise applications. Considering the multitude and complexity of technical infrastructure requirements for e-commerce applications, you will find out in this chapter how J2EE suits such requirements by providing a comprehensive infrastructure. Today, J2EE is the one of the two available alternatives for building e-commerce applications – the other alternative being Microsoft's Windows and .NET-based technologies.

From its inception, Java has triggered new programming models and technologies in different domains – ranging from devices, through telephony applications, to the enterprise. At the same time, Java has acted as a catalyst in making certain technology domains take more robust and secure shapes. Java's enterprise computing platform, the **Java 2 Platform, Enterprise Edition (J2EE)**, is one such domain.

In the past, debates in the media, as well as in technical circles, used to question whether Java is a programming language or a platform upon to build and host applications. J2EE is in fact one of the most successful attempts by Sun and its associates in making Java credible as a *platform* for distributed enterprise computing.

But what is J2EE? Why is it relevant? Why should you choose this technology for building enterprise-level applications – from client-server to Internet to mobile? This chapter gives you a balanced view of J2EE, and assists you in answering these questions. The rest of this book takes a more in-depth look at the individual technologies and demonstrates how to successfully build and manage such enterprise applications.

In this introductory chapter, we'll focus on:

- The J2EE technical architecture
- What makes J2EE credible as a platform
- What are the challenges it addresses
- What technologies constitute the J2EE platform

First, however, let us start with the challenges of developing applications for today's enterprises.

The Enterprise Today

This book is about building enterprise applications, but what exactly do we mean by an enterprise?

An enterprise means a business organization, and enterprise applications are those software applications that facilitate various activities in an enterprise.

Enterprise applications can be those that cater to end-users via the Internet, partners via the Internet or private networks, various business units within the enterprise via various kinds of user interfaces, etc. In essence, enterprise applications are those that let an enterprise manage its business activities. Examples of such activities include resource planning, product inventories and catalogues, processing invoices, fulfillment of goods or services rendered, etc. Building applications for the enterprise has always been challenging. Some of the factors that contribute to this challenge and complexity are:

□ Diversity of information needs

In an enterprise, information is created and consumed by various users in a number of different forms, depending on specific needs. It is very common to find that each business activity may process the same information in a different form.

□ Complexity of business processes

Most of the enterprise business processes involve complex information capture, processing, and sharing. Very often, you will encounter complex logic to capture and process information. This leads to complex technical and architectural requirements for building enterprise applications.

□ Diversity of applications

Due to the complex nature of enterprise business processes, it is common to find that an enterprise consists of a large number of applications each built at various times to fulfill different needs of various business processes. This commonly leads to the presence of applications built using different architectures and technologies. One of the challenges that enterprises face today is a need to make such applications talk to each other so that business processes can be implemented seamlessly.

These factors are very common and enterprises incur a tremendous costs to build and manage applications that face these challenges.

Over the last few years, these challenges have taken more monstrous shapes. Through the Internet and the recent growth of e-commerce, an enterprise's information assets have now become more valuable. This shift to an information economy is forcing many businesses to re-think even their most basic business practices. In order to maintain a competitive edge, the adoption of new technologies to quickly meet the needs of the day has become a key factor in an enterprise's ability to best exploit its information assets. More importantly, adapting these new technologies to work in tandem with existing legacy systems has become one of the foremost requirements of the enterprise.

One place these shifts in business practices have been felt most keenly is at the *application development* level. Over the last few years, the funding and the time allocated to application development has been shrinking, while demands for building complex business processes have increased. However, these are both hurdles that developers can overcome, but there are also the following requirements to be met:

□ Programming Productivity

Direct adoption of new technologies is insufficient unless they are properly utilized to their full potential and appropriately integrated with other relevant technologies. Thus, the ability to develop and then deploy applications as effectively and as quickly as possible is important. Achieving this can be complicated by the sheer variety of technologies and standards that have been developed over the years, requiring highly developed skill sets, acquiring and keeping up with which is a problem in itself. Moreover, the rapid pace of change in standards themselves makes it harder to ensure efficient meshing of technologies.

- **Reliability and Availability**

In today's Internet economy, downtime can be fatal to a business. The ability to get your web-based operations up and running, and to keep them running, is critical to success. As if that wasn't enough, you must also be able to guarantee the reliability of your business transactions so that they will be processed completely and accurately.

- **Security**

The Internet has not only exponentially increased the number of potential users but also the value of a company's information, so the security of that information has become a prime concern. What's more, as technologies become more advanced, applications more sophisticated, and enterprises more complex, implementing an effective security model becomes increasingly difficult.

- **Scalability**

The ability for the application to grow to meet new demand, both in its operation and in its user base, is vital. This is especially true when we consider that an application's potential user base may be millions of individual users, via the Internet. To scale effectively requires not only the ability to handle a large increase in the number of clients but also effective use of system resources.

- **Integration**

Although information has grown to be a key business asset, much of this exists as data in old and outdated information systems. In order to maximize the usefulness of this information, applications need to be able to integrate with the existing information system – not necessarily an easy task as current technologies have often advanced far ahead of some of these legacy systems. The ability to combine old and new technologies is key to the success of developing for today's enterprises.

Are we looking for silver bullets to address these challenges? Despite the promises by evangelists and vendors alike, it is unwise to expect (or rely on) one solution to completely avoid or overcome these challenges. What enterprises seek is an enabling technology and infrastructure to simplify some of the complex technical issues. As you progress through this book, you will come across these issues.

None of these problem domains are especially new to enterprise developers. But solving them in a comprehensive and economical manner is still crucial. You may be aware that there have been several technologies to address one or more of the above demands. However, what has been missing is a comprehensive platform with a rich infrastructure and numerous architectural possibilities which also promotes a rapid development environment.

The purpose of J2EE is to simplify some of the technical complexities, and specify the following for building enterprise applications:

- **A programming model**, consisting of a set of application programming interfaces (APIs) and approaches to building applications
- **An application infrastructure**, to support enterprise applications built using the APIs

Is Java the Answer?

So far we have discussed our system architecture from an implementation-agnostic perspective. In fact, there exist many potential paths that you can take to actually implement your enterprise. There are essentially two major paths – one being driven by Microsoft, with its new .NET suite, and the second by Sun and other vendors such as BEA, IBM, Oracle, etc. Given these choices, why should Java make such a great choice? Let's take a look at some of its advantages.

Platform Independence

With an enterprise's information spread in disparate formats, across many different platforms and applications it is important to adopt a programming language that can work equally well throughout the enterprise without having to resort to awkward, inefficient translation mechanisms. A unifying programming model also reduces the difficulties encountered from integrating many of the different technologies that grow up specific to certain platforms and applications.

Managed Objects

J2EE provides a managed environment for components, and J2EE applications are **container-centric**. Both these notions are very critical to building server-side applications. By being managed, J2EE components utilize the infrastructure provided by J2EE servers without the programmer being aware of it. J2EE applications are also **declarative**, a mechanism using which you can modify and control the behavior of applications without changing code. At first glance, these features may appear cumbersome to implement. However, when you consider large-scale applications with several hundreds of components interacting to execute complex business processes, these features make both building and maintaining such applications less complex. Along with being platform-independent, this is one of the most important aspects of J2EE.

Reusability

Code reuse is the holy grail of all programming. Segregating an application's business requirements into component parts is one way to achieve reuse; using object-orientation to encapsulate shared functionality is another. Java uses both. Java is an object-oriented language and, as such, provides mechanisms for reuse. However, unlike objects, distributed components require a more complex infrastructure for their construction and management. Basic object-oriented concepts do not provide such a framework, but the Enterprise Edition of Java provides a significantly stringent architecture for the reuse of components. Business components (called Enterprise JavaBeans) developed in J2EE are coarse-grained (although the EJB 2.0 specification in J2EE 1.3 does allow a fine-grained approach), and reflective. By keeping these components coarsely-grained, it is possible to build complex business functionality in a loosely-coupled manner. Secondly, since the components are reflective, meaning that it is possible to identify certain meta data about the components, applications can be built by composing such components. Both these features encourage code reuse at a high granularity.

Modularity

When developing a complete server-side application, programs can get large and complex in a hurry. It is always best to break down an application into discreet modules that are each responsible for a specific task. This makes our applications much easier to maintain and understand. For example, Java servlets, JavaServer Pages, and Enterprise JavaBeans each provide a way to modularize our application, breaking our applications down into different tiers and individual tasks.

Despite the **Enterprise** features in its favor, it was not until early 2000 that Java introduced a unifying model for applications on the enterprise scale (not that it lacked the capabilities, but rather was in the past). Sun recognized this shortcoming and released the **Java 2 Platform, J2EE**.

Copyrighted image

We will spend the rest of the chapter, and indeed the rest of the book, looking at what J2EE brings to server-side development with Java.

Enterprise Architecture Styles

Before we delve into the J2EE architecture, it is necessary to consider the architectural styles of contemporary distributed applications – 2-tier, 3-tier, and n-tier architectures. The purpose of this section is mainly to enable the reader to recognize the scope of and place for these patterns. We should bear in mind, however, that in reality our applications may require more complex patterns – or sometimes a combination of them.

Although these architectural styles are quite common in today's enterprises, it is worth noting that these styles emerged due to the advent of cheaper hardware platforms for clients and servers, and networking technologies. Prior to this, what powered enterprises were mainframes with all the computing (from user-interface rendering to high-volume transaction processing) centralized. Subsequently, the client-server technology is what fuelled a widespread automation of enterprise.

Typical client-server systems are based on the 2-tiered architecture, whereby there is a clear separation between the data and the presentation/business logic. Such systems are generally data-driven with the server most of the times being a database server, and the client being a graphical user interface to operate on data. While this approach allows us to share data across the enterprise, it does have many drawbacks.

Two-Tier Architecture

In a traditional 2-tiered application, the processing load is given to the client PC while the server simply acts as a traffic controller between the application and the data. As a result, not only does the application performance suffer due to the limited resources of the PC, but the network traffic tends to increase as well. When the entire application is processed on a PC, the application is forced to make multiple requests for data before even presenting anything to the user. These multiple database requests can heavily tax the network:

Copyrighted image

Another typical problem with a 2-tiered approach is that of maintenance. Even the smallest of changes to an application might involve a complete rollout to the entire user base. Even if it's possible to automate the process, you are still faced with updating every single client installation. What's more, some users may not be ready for a full rollout and may ignore the changes while another group may insist on making the changes immediately. This can result in different client installations using different versions of the application.

Three-Tier Architecture

To address these issues, the software community developed the notion of a 3-tier architecture. An application is broken up into three separate logical layers, each with a well-defined set of interfaces. The first tier is referred to as the **presentation layer** and typically consists of a graphical user interface of some kind. The middle tier, or **business layer**, consists of the application or business logic, and the third tier – the **data layer** – contains the data that is needed for the application.

The middle tier (application logic) is basically the code that the user calls upon (through the presentation layer) to retrieve the desired data. The presentation layer then receives the data and formats it for display. This separation of application logic from the user interface adds enormous flexibility to the design of the application. Multiple user interfaces can be built and deployed without ever changing the application logic, provided the application logic presents a clearly defined interface to the presentation layer. As you will see later in this chapter, and the rest of this book, J2EE provides several abstractions to meet the needs of each of these tiers. For instance, EJBs provide mechanisms to abstract both data access and business logic. Similarly, servlets and JavaServer Pages allow you abstract the presentation layer and its interaction with the business layer.

The third tier contains the data that is needed for the application. This data can consist of any source of information, including an enterprise database such as Oracle or Sybase, a set of XML documents, or even a directory service like an LDAP server. In addition to the traditional relational database storage mechanism, there are many different sources of enterprise data that your applications can access:

Copyrighted image

However, we're not quite finished with our subdivision of the application. We can take the segregation one step further to create an n-tier architecture.

n-Tier Architecture

As the title suggests, there is no hard-and-fast way to define the application layers for an n-tier system. In fact, this kind of system can support a number of different configurations. In an n-tier architecture the application logic is logically divided by function, rather than physically.

n-tier architecture breaks down like this:

- **A user interface** that handles the user's interaction with the application – this can be a web browser running through a firewall, a heavier desktop application, or even a wireless device.
- **Presentation logic** that defines what the user interface displays and how a user's requests are handled. Depending on what user interfaces are supported, you may need to have slightly different versions of the presentation logic to handle the client appropriately.

- **Business logic** that models the application's business rules, often through interaction with the application's data.
- **Infrastructure services** that provide additional functionality required by the application components, such as messaging, transactional support.
- The **data layer** where the enterprise's data resides.

Copyrighted image

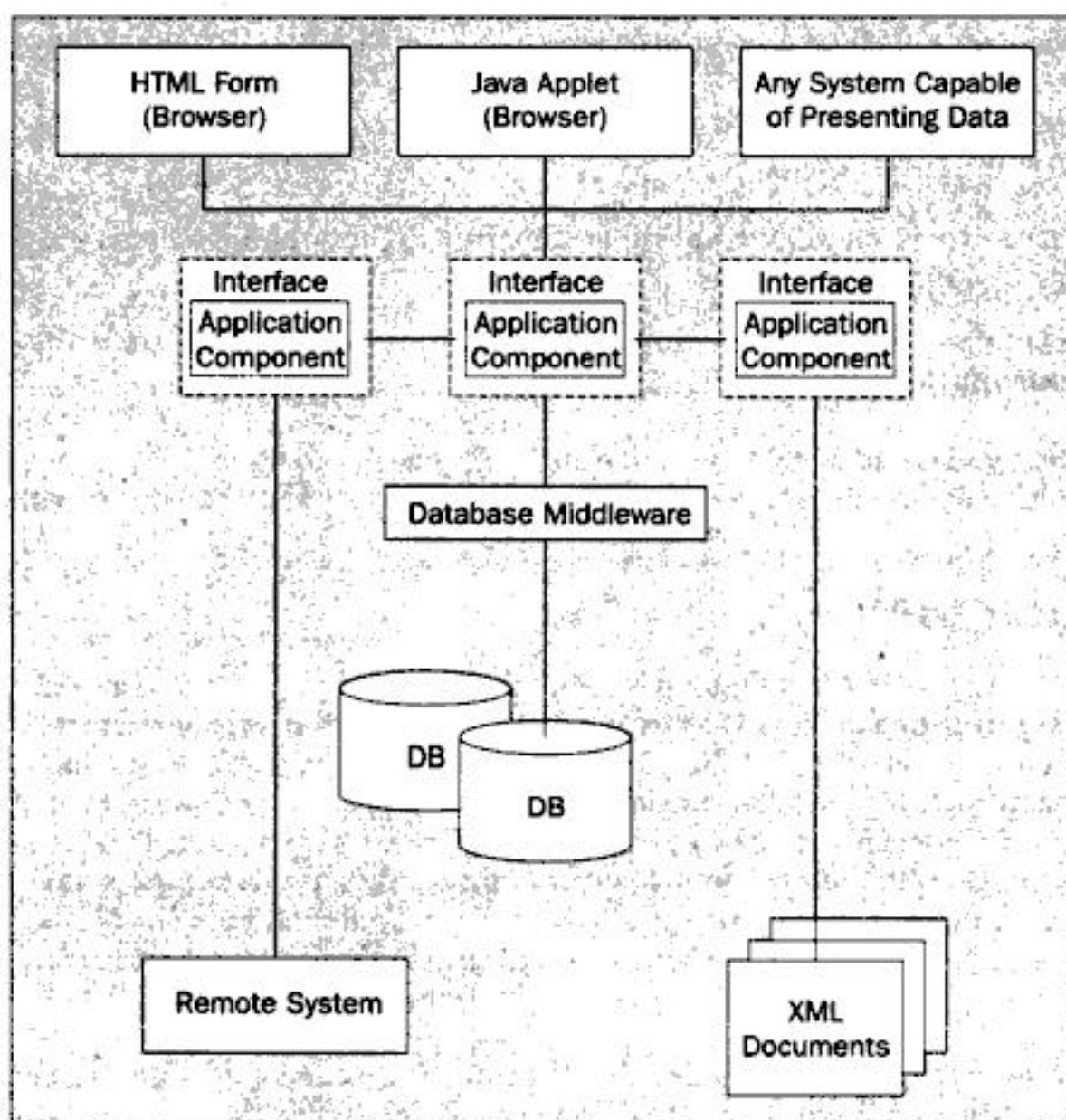
*Applications based on this architecture are essentially employing the **Model-View-Controller (MVC)** pattern. What this ultimately means is that the data (the model) is separated from how the information is presented (the view). In between this is the application/business logic (the controller) that controls the flow of the information. Thus, an application is designed based on these three functional components (model, view, and controller) interacting with each other.*

Enterprise Architecture

So far we have really been concentrating on a single application's architecture – we are in danger of considering these applications as "stovepipes". In other words, we might end up with many different applications – possibly even with different architectures – all of which don't communicate with one another. In an enterprise, we are looking to create a cohesive architecture incorporating different applications.

Rather than a change in architecture – enterprise architecture is basically just n-tier – we need a change in perception. To turn an n-tier system into an enterprise system, we simply extend the middle tier by allowing for multiple **application objects** rather than just a single application. These application objects must each have an interface that allows them to work together with each other.

An interface can be thought of as a contract. Each object states through its interface that it will accept certain parameters, perform certain operations, and return a specific set of results. Application objects communicate with each other using their interfaces, as shown in the following diagram:



With enterprise architecture, we can have multiple applications using a common set of components across an organization. This promotes the standardization of business practices by creating a single set of business functions for the entire organization to access. If a business rule changes, then changes have to be made to the relevant "business object" and, if necessary, the associated interface and subsequently any object that accesses the interface. Note, however, that certain functionalities in existing legacy systems can't be removed and a "wrapper" (or interface) specific to this functionality has to be developed in the middle tier.

It is important to note that when designing an object and its interface, it is a good idea to make the interface as generic as possible to avoid changes later on. Since other objects communicate with the interface and not the object itself, changes to the object, and not the interface, are relatively simple and quick.

In all these patterns, notice that user interfaces are what are shown at the top of each diagram. From these diagrams, you may get the impression that user interfaces are what always drive systems. Although this is the most commonly visible form of system interaction, as you consider intra- and inter-business application integration, and cross-enterprise business processes, you'll find applications interacting with other applications. Nonetheless, most of the technologies that we shall discuss, and the designs that you will find this book, apply to such systems as well.

The J2EE Platform

As you'll see in the rest of this book, J2EE is one of the best solutions that we've had so far for meeting the demands of today's enterprise. J2EE specifies both the infrastructure for managing your applications, and the service APIs for building your applications.

The J2EE platform is essentially a distributed application-server environment – a Java environment that provides the following:

- A set of Java extension APIs to build applications. These APIs define a programming model for J2EE applications.
- A run-time infrastructure for hosting and managing applications. This is the server runtime in which your applications reside.

The applications that you could develop with the above may be programs to drive web pages, or components to implement complex database transactions, or even Java applets, all distributed across the network.

The J2EE Runtime

While J2EE bundles together APIs that have been in existence in one form or another for quite some time, perhaps its most significant aspect is the abstraction of the **run-time infrastructure**. Note that the J2EE specification does not specify how a J2EE runtime should or could be built. Instead, J2EE specifies roles and interfaces for applications, and the runtime onto which applications could be deployed. This results in a clear demarcation between applications and the run-time infrastructure. This demarcation allows the runtime to abstract most of the infrastructure services that enterprise developers have traditionally attempted to build on their own. As a result, J2EE application developers could just focus on the application logic and related services, while leveraging the runtime for all infrastructure-related services. However, what are the infrastructure requirements of enterprise applications?

It is interesting to note that application development in a fast-paced environment, such as the Internet, typically leads to designs that are short-lived.

On the other hand, when considering the long-term design aspects, we may have difficulty finding a compromise between short-term and long-term demands. There are technologies that lend themselves to rapid development, and then there are technologies that let you build applications taking care of such long-term concerns as reusability, cost of maintenance, etc. Often the two do not mix, but J2EE is flexible enough to allow us to build applications that include both. This is because it lets you build each layer of your application loosely coupled to all other layers. Each layer can therefore evolve to meet respective evolutionary needs. Thus, by conforming to a system designed on interfaces, the implementation is highly extendable.

Apart from specifying a set of standard APIs, the J2EE architecture also provides a uniform means of accessing platform-level services via its run-time environment. Such services include distributed transactions, security, messaging, and so on. Before we see details of J2EE's approach, let's take a look at traditional distributed computing.

Until the advent of J2EE, distributed computing was, in general, considered as client-server programming. We would write a server application implementing an interface, a client application to connect to the server, and then start both the server and the client! Although this process seems so simple, in practice there are several critical stumbling blocks in this process, depending on the technology used.

For instance, consider a CORBA object request broker for building distributed applications. The typical procedure for building the server-side CORBA objects would start from specifying an interface, using the Interface Definition Language (IDL), for each object. Subsequent steps include compiling this IDL to generate "stubs" and "skeletons" for the language of your choice, implementing the object based on the skeleton, then writing the client application, preparing your environment, etc. Here the stub is the class that represents the CORBA object on the client side. The skeleton provides the method with which the server-side logic is implemented. This procedure in itself is not complicated and could easily be automated. As you will learn in this book, EJBs too require some of these steps and are conceptually similar to CORBA objects.

Now consider that our servers, as well as clients, require access to services such as distributed transactions, messaging, etc. To utilize such services, we'd be required to add a significant amount of plumbing code to our applications. More often than not, we'd be required to set up and configure different middleware solutions, and make API calls to the vendor-specific APIs to access the services. Apart from services such as relational database access, most of these services are either proprietary or non-standard. The result is that our applications will be more complex, time consuming, and expensive to develop, manage, and maintain.

Apart from having to manage all these different APIs, there is another critical demand on server-side applications. On the serverside, resources are scarce. For example, we cannot afford to create the same number of objects that we can typically afford to create in client-side applications. Other server-side resources that require special attention include threads, database connections, security, transactions, etc. Custom-building an infrastructure that deals with these resources has always been a challenge. This task is almost impossible in the Internet economy. Would you care to build a connection pool, or an object cache, or an "elegant" object layer for database access, when your development lifecycle is only three months?

Since these server-side requirements are common across a wide variety of applications, it is more appropriate to consider a platform that has built-in solutions. This lets us separate these infrastructure-level concerns from the more direct concern of translating your application requirements to software that works. The J2EE runtime addresses such concerns. Essentially, we leave it to the J2EE server vendor to implement the specific features for us in a standards-compliant manner.

As mentioned above, the J2EE does not specify the nature and structure of the runtime. Instead, it introduces what is called a **container**, and via the J2EE APIs, specifies a contract between containers and applications. You'll see more details of this contract at appropriate places later in this book.

Before looking into more details of J2EE containers, let's have a brief look at the J2EE APIs.

The J2EE APIs

Distributed applications require access to a set of enterprise services. Typical services include transaction processing, database access, messaging, multithreading, etc. The J2EE architecture unifies access to such services in its enterprise service APIs. However, instead of having to access these services through proprietary or non-standard interfaces, application programs in J2EE can access these APIs via the container.

A typical commercial J2EE platform (or J2EE application server) includes one or more containers, and access to the enterprise APIs is specified by the J2EE.

Note that J2EE application servers need not implement these services themselves; containers are only required to provide access to each service implementation via a J2EE API. For example, a J2EE implementation might delegate the Java Message Service API calls to a commercial message-oriented middleware solution. Yet another implementation might include a message-oriented middleware solution within a container.

The specification of the J2EE 1.3 platform includes a set of Java standard extensions that each J2EE platform must support:

- **JDBC 2.0**
More specifically, the JDBC 2.0 Optional Package. This API improves the standard JDBC 2.0 API by adding more efficient means of obtaining connections, connection pooling, distributed transactions, etc. Note that the next version of JDBC, the JDBC 3.0 API combines the extension API with the usual JDBC API. Future versions of J2EE may include JDBC 3.0.
- **Enterprise JavaBeans (EJB) 2.0**
This specifies a component framework for multi-tier distributed applications. This provides a standard means of defining server-side components, and specifies a rich run-time infrastructure for hosting components on the serverside.
- **Java Servlets 2.3**
The Java Servlet API provides object-oriented abstractions for building dynamic web applications.
- **JavaServer Pages (JSP) 1.2**
This extension further enhances J2EE web applications by providing for template-driven web application development.
- **Java Message Service (JMS) 1.0**
JMS provides a Java API for message queuing, and publish and subscribe types of message-oriented middleware services.
- **Java Transaction API (JTA) 1.0**
This API is for implementing distributed transactional applications.
- **JavaMail 1.2**
This API provides a platform-independent and protocol-independent framework to build Java-based e-mail applications.
- **JavaBeans Activation Framework (JAF) 1.0** This API is required for the JavaMail API. The JavaMail API uses JAF to determine the contents of a MIME (Multipurpose Internet Mail Extension) message and determine what appropriate operations can be done on different parts of a mail message.
- **Java API for XML Parsing (JAXP) 1.1**
This API provides abstractions for XML parsers and transformation APIs. JAXP helps to isolate specific XML parsers, or XML Document Object Model (DOM) implementations, or XSLT transformation APIs from J2EE application code.
- **The Java Connector Architecture (JCA) 1.0**
This API has recently been included in J2EE, and provides a means to integrate J2EE application components to legacy information systems.
- **Java Authentication and Authorization Service (JAAS) 1.0**
This API provides authentication and authorization mechanisms to J2EE applications.

Most of these API are specifications, independent of implementation. That is, one should be able to access services provided by these API's in a standard way, irrespective of how they are implemented.

The J2EE-specific APIs mentioned above are in addition to the following J2SE APIs:

- **Java Interface Definition Language (IDL) API**
This API allows J2EE application components to invoke CORBA objects via IIOP (see below).
- **JDBC Core API**
This API provides the basic database programming facilities.

- **RMI-IIOP API**

This provides an implementation of the usual **Java Remote Method Invocation (RMI)** API over the **Internet Inter-ORB Protocol (IIOP)**. This bridges the gap between RMI and CORBA applications. This is the standardized communication protocol to be used between J2EE containers.

- **JNDI API**

The **Java Naming and Directory Interface (JNDI)** API standardizes access to different types of naming and directory services available today. This API is designed to be independent of any specific naming or directory service implementation. J2SE also specifies a JNDI service provider interface (SPI), for naming and directory service providers to implement.

We will take a more detailed look at these APIs later in the chapter.

J2EE Architecture – Containers

As discussed in the previous section, a typical commercial J2EE platform includes one or more **containers**. But what exactly is a container?

A J2EE container is a runtime to manage application components developed according to the API specifications, and to provide access to the J2EE APIs.

Beyond the identity associated with the runtime, J2EE does not specify any identity for containers. This gives a great amount of flexibility to achieve a variety of features within the container runtime.

The following figure shows the architecture of J2EE in terms of its containers and APIs:

Copyrighted image

This architecture shows four containers:

- A **web container** for hosting Java servlets and JSP pages
- An **EJB container** for hosting Enterprise JavaBean components
- An **applet container** for hosting Java applets
- An **application client** container for hosting standard Java applications

In this book, our focus is limited to the web and EJB containers only.

Each of these containers provides a run-time environment for the respective components. J2EE components are also called managed objects, as these objects are created and managed within the container runtime.

In the above figure, the vertical blocks at the bottom of each container represent the J2EE APIs. Apart from access to these infrastructure-level APIs, each container also implements the respective container-specific API (Java Servlet API for the web container, and the EJB API for the EJB container).

The stacks of rectangles (servlets, JSP pages, and EJBs) in this figure are the programs that you develop and host in these containers. In the J2EE parlance, these programs are called **application components**.

Note that the above figure does not include the J2SE APIs listed above, that is, the Java IDL, JDBC core, RMI-IIOP, and JNDI.

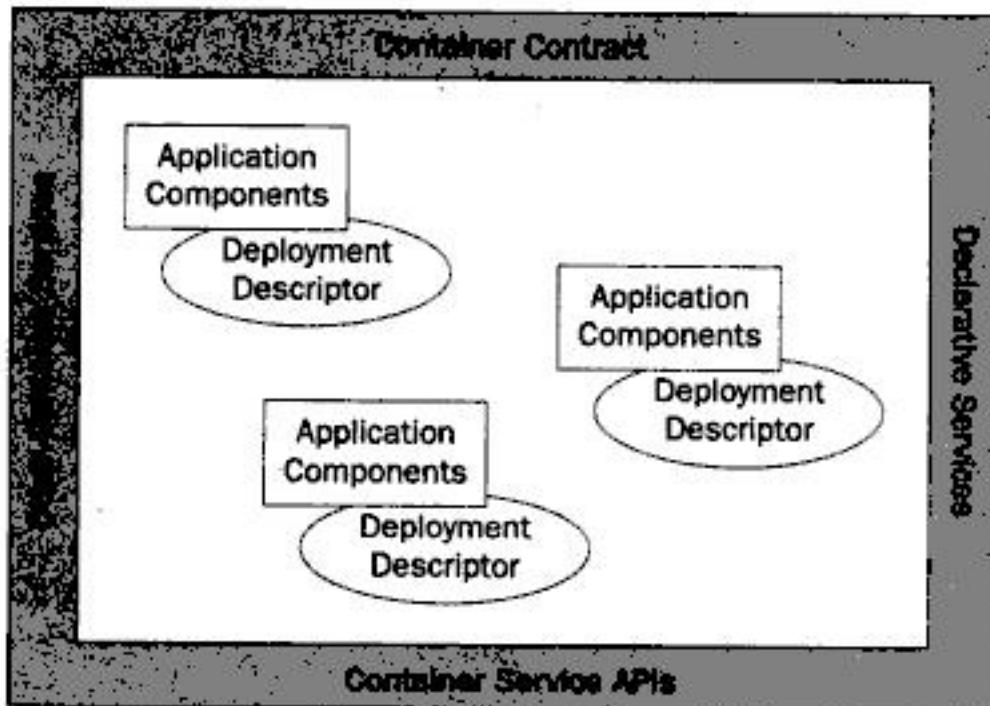
In this architecture, there are primarily two types of clients:

- **Web clients** normally run in web browsers.
For these clients, the user interface is generated on the serverside as HTML or XML, and is downloaded and then rendered by the browsers. These clients use HTTP to communicate with web containers. Application components in web containers include Java servlets and JSP pages. These components implement the functionality required for web clients. Web containers are responsible for accepting requests from web clients, and generating responses with the help of the application components.
- **EJB clients** are applications that access EJB components in EJB containers.
There are three possible types of EJB clients. The first category is application clients. Application clients are standalone applications accessing the EJB components using the RMI-IIOP protocol. The second category of application clients are components in the web container. That is, Java servlets and JSP pages can also access the EJB components via the RMI-IIOP protocol in the same way as the application clients. The final category is other EJBs running within the EJB container. These communicate via standard Java method calls through a local interface (this is new in J2EE 1.3).

In either case, clients access application components via the respective container. Web clients access JSP pages and Java servlets via the web container, and EJB clients access the EJB components via the EJB container.

Container Architecture

Having seen the core constituents of J2EE, let's now return to our architecture discussion and study the architecture of a J2EE container:



In this architecture, as developers we are required to provide the following:

- ❑ **Application components**
As discussed in the previous section, application components include servlets, JSP pages, EJBs, etc. In J2EE, application components can be packaged into archive files.
 - ❑ **Deployment descriptors**
A deployment descriptor is an XML file that describes the application components. It also includes additional information required by containers for effectively managing application components.

The rest of this figure forms the container. The architecture of a container can be divided into four parts:

- ❑ **Component contract**
A set of APIs specified by the container, that your application components are required to extend or implement.
 - ❑ **Container service APIs**
Additional services provided by the container, which are commonly required for all applications in the container.
 - ❑ **Declarative services**
Services that the container interposes on your applications, based on the deployment description provided for each application component, such as security, transactions etc.
 - ❑ **Other container services**
Other runtime services, related to component lifecycle, resource pooling, garbage collection, etc.

Let's now discuss each of the above in detail.

Component Contracts

As we mentioned earlier, the basic purpose of the container in the J2EE architecture is to provide a runtime for application components. That is, instances of the application components are created and invoked within the JVM of the container. This makes the container responsible for managing the lifecycle of application components. However, if they are to be manageable within the container runtime, they are required to abide by certain **contracts** specified by the container. In other words, applications should be developed according to a defined framework of operation.

To better understand this aspect, consider a Java applet. Typically, an applet is downloaded by the browser and instantiated and initialized in the browser's JVM. That is, the applet lives in the runtime provided by the JVM of the browser.

However, in order for the container to be able to create, initialize, and invoke methods on application components, the application components are required to implement or extend certain Java interfaces or classes. For example, considering the applet example again, a Java applet is required to extend the `java.applet.Applet` class specified by the JDK. The JVM of the browser expects your applets to extend this class. This enables the browser's JVM to call the `init()`, `start()`, `stop()`, and `destroy()` methods on your applets. These methods control the lifecycle of an applet – unless your applet extends the `java.applet.Applet` class, the browser JVM has no means of calling these methods to control its lifecycle.

In J2EE, all application components are instantiated and initialized in the JVM of the container. In addition, since J2EE application components are always remote to the client, clients cannot directly call methods on these components – in fact, clients make requests to the applications server and it is the container that actually invokes the methods. Since the container process is the only entry point into the application components, all application components are required to follow the contract specified by the container. In J2EE, this contract is in the form of interfaces and classes that your classes implement or extend, with additional rules that the component definition should follow.

There is one important implication of such a component contract. All J2EE application components are **managed**. The management includes location, instantiation, pooling, initialization, service invocation, and removal of components from service. These aspects of component lifecycle are the responsibility of the container, and the application has no direct control.

Let's now look at the various contracts specified by J2EE.

In the case of web containers, web application components are required to follow the Java Servlet and JSP APIs. In this model, all Java servlets are required to extend the `javax.servlet.http.HttpServlet` class, and to implement certain methods of this class such as `doGet()`, `doPost()`, and so on. Similarly, when compiled, classes corresponding to JSP pages extend the `javax.servlet.jsp.HttpJspPage` class.

In the case of EJB containers, session and entity enterprise beans are required to have `javax.ejb.EJBHome` and `javax.ejb.EJBObject` interfaces specified, while implementing either a `javax.ejb.SessionBean` or `javax.ejb.EntityBean` interface. Similarly, message-driven beans must implement both `javax.ejb.MessageDrivenBean` and `javax.jmx.MessageListener` interfaces. As you'll see in Chapter 14, the component specifications and implementations are also required to follow certain rules. Apart from the interfaces, all such rules form part of the component contract for EJB components.

Container Service APIs

As discussed earlier, the J2EE platform defines a set of Java standard extensions that each J2EE platform must support. J2EE containers provide a service-level abstraction of the APIs. As a result, you can access the service APIs such as JDBC, JTS, JNDI, JMS, etc., within the container, as though the underlying container is implementing them.

Copyrighted image

In J2EE, application components can access these APIs via appropriate objects created and published in the JNDI service or implementation. For example, when we want to use JMS within our application components, we would configure our J2EE platform to create JMS connection factories, message queues and topics, and publish these objects in the JNDI service. Our applications can then look in the JNDI, obtain references to these objects, and invoke methods. In this process, it does not matter if the J2EE platform has a JMS implementation built in or if it is using a third-party messaging middleware solution.

Let's consider Java applets again. For example, if we want to play an audio file from an applet, we would call the `play()` method defined in the super-class (`java.applet.Applet`), with a URL pointing to the audio file. In this case, the functionality for playing audio files is implemented by the super-class. This is one of the approaches for code reuse. A better alternative to this approach is to delegate this functionality to a common component. Such a component need not be a part of the same inheritance hierarchy. All that is required is a reference to an object implementing this functionality. Once our application is able to get a reference to such an object, we can delegate the "play" functionality to that object. A significant advantage of this approach is that it allows other objects from more than one inheritance hierarchy to access the "play" functionality.

In the case of distributed applications this gets complicated, as the services are remote. Just as an analogy, consider the same audio component running as a separate server! How do we get hold of a reference to such a component? The solution is to let the J2EE platform create the audio component, and publish its name in a naming service (that is, JNDI) available to your application. This provides a simplified method of access to the service APIs. This also allows us to plug in different implementations of these services without disturbing the applications using the services. The following schematic illustrates this possibility:

Copyrighted image

No
pro
su
rec

by the
ost
also

All J2EE service APIs use the above approach for providing services to applications.

The following are the key implications of this approach:

- As a single standard that can sit on top of a variety of existing database systems, transaction processing systems, naming and directory services, etc., the service APIs eliminate the inherent heterogeneity involved in bringing these technologies together in our applications.
- In J2EE, these services are also integrated tightly with the programming model. In later chapters, we'll see how to access these APIs seamlessly from our application components and clients.
- The J2EE platform also specifies a uniform mechanism for accessing these services.

Declarative Services

One of the important features of the J2EE architecture is its ability to dynamically interpose services for application components. This is based on declarations specified outside our application components. The J2EE architecture provides simple means of specifying such declarations. These are called **deployment descriptors**.

Copyrighted image

For example, a set of EJB components can be described together in a single deployment descriptor file. Similarly, in the case of web containers, each web application is required to have a deployment descriptor specified.

Depending on the type of the component, certain types of services (such as transactions, security, etc.) can be specified in the deployment descriptor. The purpose of this approach is to minimize the application programming required in order to make use of such services.

The standard method of invoking services is via **explicit invocation**. For example, to implement transactions for database access, we can programmatically start a transaction before accessing the database, and commit or rollback the transaction once the business methods are completed. In the case of **declarative invocation**, our application components need not explicitly start and stop transactions. Instead, we can specify in the deployment descriptor that our business methods should be invoked within a new transaction. Based on this information, the container can automatically start a transaction whenever the business methods in your applications are invoked.

Copyrighted image

How does this approach work? J2EE containers are, by nature, distributed and the application components are remote to clients. Accordingly, requests to application components and responses back from application components occur across process, platform, and network boundaries.

In addition, since application components are maintained in the container runtime, the container process is responsible for receiving the requests, and delegating them to appropriate application components. For instance, in the case of the EJB container, the container receives all client requests and delegates them to appropriate EJB objects deployed on the container. Similarly, in the case of web containers, the web container receives HTTP requests and delegates them to servlets and JSP pages. The following diagram depicts a simplified view of this invocation. In this figure, the remote interface is what the clients use to communicate with the EJB on the container. Note that the container process handles all requests and responses to and from the application components:

Copyrighted image

This approach gives the container an ability to **interpose** a new service before transferring a request to the application component. In the case of declarative transactions, the container can start the transaction before delegating the incoming request to the business method implication, and end the transaction as soon as the method returns.

What's the advantage of this approach? We can interpose new services without changing the application component. More specifically, this facility allows us to postpone decisions about such services to the run-time, instead of the design time. In other words, the container can selectively enhance our components based on the deployment descriptor.

For EJB containers, the declarative services include transactions and security. Both these can be specified in deployment descriptors. In the case of web containers, we can specify the required security roles for accessing components within web applications.

Other Container Services

The following are some of the runtime services that containers provide:

- **Lifecycle management of application components**
This involves creating new instances of application components and pooling or destroying them when the instances are no longer required.
- **Resource pooling**
Containers can optionally implement resource pooling such as object pooling and connection pooling.
- **Populating the JNDI namespace based on the deployment names associated with EJB components**
This information is typically supplied at deployment time. We'll discuss more about deployment in the next section.
- **Populating the JNDI namespace with objects necessary for utilizing container service APIs**
Some of the objects include data source objects for database access, queue and topic connection factories for obtaining connections to JMS, and user transaction objects for programmatically controlling transactions.

□ Clustering

In J2EE, containers can be distributable. A distributable container consists of a number of JVMs running on one or more host machines. In this setup, application components can be deployed on a number of JVMs. Depending on the type of load-balancing strategy and the type of the component, the container can distribute the load of incoming requests to one of these JVMs. Clustering is essential for enhancing scalability and availability of applications.

Now that we have covered the architecture of J2EE applications, let's take a closer look at some of the various technologies included in the J2EE platform.

J2EE Technologies

Having discussed all the architecture of the J2EE platform, we now want to cover the collection of technologies that provide the mechanics we need to build large, distributed enterprise applications. This large collection, of quite disparate technologies, can be divided according to use:

□ The component technologies

These technologies are used to hold the most important part of the application – the business logic. There are three types of components: JSP pages, servlets, and Enterprise JavaBeans; we will look at each of these in a moment.

□ The service technologies

These technologies provide the application's components with supported services to function efficiently.

□ The communication technologies

These technologies, which are mostly transparent to the application programmer, provide the mechanisms for communication among different parts of the application, whether they are local or remote.

Let's now examine how the J2EE APIs and associated technologies can be categorized.

Component Technologies

With any application, the most important element is modeling the necessary business logic through the use of components, or application level reusable units. Earlier in the chapter, we described a container as hosting the runtime for application components, so although the container may be able to supply many of the services and much of the communication infrastructure, it is ultimately the responsibility of the developer to create the application components. However, these components will be dependent upon their container for many services, such as lifecycle management, threading, and security. This allows us to concentrate on providing the requisite business functionality without getting into details of low-level (container-level) semantics.

The J2EE platform provides three technologies for developing components.

One thing that should be made clear is that the J2EE platform does not specify that an application need make use of all three types of component technologies – in many cases using EJBs may well be overkill.

Web Components

These can be categorized as any component that responds to an HTTP request. A further distinction that can be drawn is based on the hosting container for the application components. As we saw earlier in the chapter, the two basic server-side containers are the web container and the EJB container.

Servlets

Servlets are server-side programs that allow application logic to be embedded in the HTTP request-response process. Servlets provide a means to extend the functionality of the web server to enable dynamic content in HTML, XML, or other web languages. With the release of J2EE 1.3, the Servlets specification has reached version 2.3.

We'll be working with servlets in Chapters 5 to 9.

JavaServer Pages

JavaServer Pages (JSP) provide a way to embed components in a page, and to have them do their work to generate the page that is eventually sent to the client. A JSP page can contain HTML, Java code, and JavaBean components. JSP pages are in fact an extension of the servlet programming model. When a user requests a JSP page, the web container compiles the JSP page into a servlet. The web container then invokes the servlet and returns the resulting content to the web browser. Once the servlet has been compiled from the JSP page, the web container can simply return the servlet without having to recompile each time. Thus, JSP pages provide a powerful and dynamic page assembly mechanism that benefits from the many advantages of the Java platform.

Compared to servlets, which are pure Java code, JSP pages are merely text-based documents until the web container compiles them into the corresponding servlets. This allows a clearer separation of application logic from presentation logic. This, in turn, allows application developers to concentrate on business matters and web designers to concentrate on presentation. With the current release of J2EE, the JSP specification reached version 1.2. Chapters 10 to 12 discuss JSP pages in detail.

Copyrighted image

Enterprise JavaBean Components

Enterprise JavaBeans (EJB) 2.0 is a distributed component model for developing secure, scalable, transactional, and multi-user components. To put it simply, EJBs are (ideally) reusable software units containing business logic. Just as JSP pages allow the separation of application and presentation logic, EJBs allow separation of application logic from system-level services thus allowing the developer to concentrate on the business domain issues and not system programming. These enterprise bean business objects take three basic forms – again, it is not necessary to implement them all – **session beans**, **entity beans**, and **message-driven beans**. We'll see EJBs in action in Chapters 14 to 19.

Session Beans

Session beans themselves come in two types. A **stateful session bean** is a transient object used to represent a client's interaction with the system – it performs the client's requests in the application, accessing a database etc., and when the client's operations are complete it is destroyed (that is, it exists for the length of the client session) – one example of this is an application client sending a series of requests to an application to perform a business process. In such cases, a stateful session bean can track the state of the interaction between the client and the application. The alternative, a **stateless session bean**, maintains no state between client requests. Generally, this type of session bean is used to implement a specific service that does not require client state, for instance, a simple database update.

Entity Beans

An **entity bean** on the other hand is a persistent object that models the data held within the data store, that is, it is an object wrapper for the data. For instance, data about a purchase order can be modeled using a PurchaseOrder entity bean presenting an aggregated view of all purchase order related data. Compared to session beans that can be used by any client, entity beans can be accessed concurrently by many clients but must maintain a unique identity through a primary key. In fact, under the J2EE container architecture we can elect whether to have the persistent state of the entity bean managed for us by the container or whether to implement this "by hand" in the bean itself.

Message-Driven Beans

Message-driven beans are a special class of EJBs that are not meant for direct client invocation. The purpose of message-driven beans is to process messages received via JMS. Message-driven beans complement the asynchronous nature of JMS by providing a means of processing messages within the EJB container. When an application client or an application sends a message via JMS, the container invokes the appropriate message-driven bean to process the message.

XML

As mentioned in our overview earlier in the chapter, J2EE 1.3 includes the JAXP API. This serves to abstract XML parsing, and XML translation APIs. In addition, there are several other XML-related Java APIs are being developed currently. For a summary and further details, refer to <http://java.sun.com/xml>.

XML (Extensible Markup Language) influences how we view, process, transport, and manage data. XML is a self-describing language using which you can send data as well as its meta data. In today's enterprise systems, XML is used to represent business data both within and across applications.

One of the strengths of XML is the sharing of such "vocabularies", all of which use the same basic syntax, parsers, and other tools. Shared XML vocabularies provide more easily searchable documents and databases, and a way to exchange information between many different organizations and computer applications.

XML can be far more than just a description mechanism though:

- **Transforming XML**

Transformation allow a programmer to map an XML document in one form into another form based on a set of rules. XML transformations are used to translate between similar XML vocabularies as well as translating XML documents into other text-based file formats like comma-delimited values. This is the standard representation of data across an enterprise.

- **XML and Databases**

Although the XML data model is inherently hierarchical whereas databases are essentially relational – creating some mapping difficulties – it does provide a mechanism of integrating existing data into new systems. Many database vendors are now adding native support for XML into their engines in recognition that programmers need ways to interface XML and databases.

- **Server-to-Server Communication**

Complex enterprise applications often utilize differing server software running and distributed across many computing technologies. XML provides a layer of abstraction in order to integrate these dissimilar systems. XML can be obtained from one server, manipulated, and then passed to another server in such a way that it can understand the request.

Although we won't be explicitly covering XML in this book, we will encounter it when we create various J2EE deployment descriptors. For more information on using XML with Java, please refer to *Professional Java XML* and *Java XML Programmers Reference*, both by Wrox Publishing.

Service Technologies

As we have discussed, some of the J2EE services for application components are managed by the containers themselves, thus allowing the developer to concentrate on the business logic. However, there will be times when developers find it necessary to programmatically invoke some services themselves using some of the various service technologies.

JDBC

Although all data access should be accessible through the single standard API of the Connector architecture in the future, database connectivity is probably one of the key services that developers implement in their application component.

The JDBC API (un-officially standing for Java Database Connectivity) provides the developer with the ability to connect to relational database systems. As we will see in Chapter 4, it allows the transactional querying, retrieval, and manipulation of data from a JDBC-compliant database. For the moment, it is worth noting that J2EE adds an extension to the core JDBC API (which comes with J2SE) to give advanced features, such as connection pooling and distributed transactions.

Java Transaction API and Service

The Java Transaction API (JTA) is a means for working with transactions and especially distributed transactions independent of the transaction manager's implementation (the Java Transaction Service (JTS)). Under the J2EE platform, distributed transactions are generally considered to be container-controlled so we, as developers, shouldn't have to be too concerned with transactions across your components – having said that though, the J2EE transaction model is still somewhat limited, so at times it may be necessary to do the hard work yourself.

JNDI

The role of the Java Naming and Directory Interface (JNDI) API in the J2EE platform is twofold:

- Firstly, it provides the means to perform standard operations on a directory service resource such as LDAP, Novell Directory Services, or Netscape Directory Services
- Secondly, a J2EE application utilizes JNDI to look up interfaces used to create, among other things, EJBs, and JDBC connections

In the next chapter, we'll see how to use JNDI to access a directory service resource.

JMS

In the enterprise environment, the various distributed components may not always be in constant contact with each other. Therefore, there needs to be some mechanism of sending data asynchronously. The **Java Message Service (JMS)** provides just such functionality to send and receive messages through the use of message-oriented middleware (MOM).

JavaMail

JavaMail is an API that can be used to abstract facilities for sending and receiving e-mail. JavaMail supports the most widely used Internet mail protocols such as IMAP4, POP3, and SMTP, but compared to JMS it is slower and less reliable.

The Java Connector Architecture

The **Java Connector Architecture (JCA)**, introduced in version 1.3, is a standardized means by which J2EE applications can access a variety of legacy applications, typically Enterprise Resource Planning systems such as SAP R/3 and PeopleSoft. The Connector specification defines a simple architecture in which vendors of J2EE application servers and legacy systems collaborate to produce 'plug-and-play' components that allow you access the legacy system without having to know anything too specific about how to work with it.

JAAS

The **Java Authentication and Authorization Service** provides a means to grant permissions based on who is executing the code. JAAS utilizes a pluggable architecture of authentication modules so that you can drop in modules based on different authentication implementations, such as Kerberos or PKI (Public Key Infrastructure).

Communication Technologies

The final technology grouping is those technologies that provide the means for the various components and services within a J2EE application to communicate with each other – a distributed application would be pretty ineffectual if these technologies didn't provide the 'glue' to hold it all together.

Internet Protocols

As we are talking about n-tier applications in this book, our client will very often be a browser potentially situated anywhere in the world. A client's requests and the server's responses are communicated over three main protocols.

HTTP

HTTP or Hypertext Transfer Protocol is a generic, stateless, application-level protocol, which has many uses beyond simply hypertext capabilities. It works on a request/response basis. A client sends a request to the server in the form of a request method, URI (Uniform Resource Identifier), and protocol version, followed by a MIME-like message containing request modifiers, client information, and possible body content over a connection with a server. The server in turn responds with a status line followed by a MIME-like message containing server information, entity meta-information, and possible entity-body content.

TCP/IP

TCP (Transmission Control Protocol) over **IP (Internet Protocol)** is actually two separate protocols, which are typically combined into a single entity. IP is the protocol that takes care of making sure that data is received by both endpoints in communication over the Internet. When we type the address of a web site into our browser, IP is what ensures that our requests and the fulfillment of those requests make it to the proper destinations. For efficiency, the data being sent back and forth between a client and a web server is broken into several pieces, or packets. These packets do not all have to take the same route when they are sent between the client and the web server. TCP is the protocol that keeps track of the packets and makes sure they are assembled in the same order as they were dispatched and are error-free. Therefore, TCP and IP work together to move data around on the Internet. For this reason, we will almost always see these two protocols combined into TCP/IP.

SSL

Secure Sockets Layer (SSL) uses cryptography to encrypt the flow of information between the client and server. This also provides a means for both parties to authenticate each other. Secure HTTP (HTTPS) is usually distinguished from regular unencrypted HTTP by being served on a different port number (443, by default).

Remote Object Protocols

In applications where the components are often distributed across many tiers and servers, some mechanism for using the components remotely is required, preferably leaving the client unaware that the component is not local to itself.

RMI and RMI-IIOP

Remote Method Invocation (RMI) is one of the primary mechanisms in distributed object applications. It allows us to use interfaces to define remote objects. We can then call methods on these remote objects as if they were local. The exact wire-level transportation mechanism is implementation-specific. For example, Sun uses the Java Remote Method Protocol (JRMP) on top of TCP/IP, but other implementations, such as BEA WebLogic for instance, have their own protocol.

RMI-IIOP is an extension of RMI but over IIOP (Inter-ORB Protocol), which allows us to define a remote interface to any remote object that can be implemented in any language that supports OMG mapping and ORB. In Chapter 3, we'll take an in-depth look at how to write distributed applications with RMI.

JavaIDL

Through the use of **JavaIDL**, a Java client can invoke method calls on CORBA objects. These CORBA objects need not be written in Java but merely implement an IDL defined interface.

Developing J2EE Applications

The J2EE specification specifies the following steps in the application development and deployment process:

1. Application component development

During this step we model the business rules in the form of application components.

2. Composition of application components into modules

In this step, the application components are packaged into modules. This phase also involves providing deployment descriptors for each module.

3. Composition of modules into applications

This step integrates multiple modules into J2EE applications. This requires assembling one or more modules into a J2EE application, and supplying it with the descriptor files.

4. Application deployment

In the final step the packaged application is actually deployed and installed on the J2EE platform application server(s).

The J2EE platform specifies multiple levels for packaging, customization, and installation. Packaging is the process of composing applications out of application components. J2EE specifies a three-level packaging scheme for composing components into applications. These are: – application components, modules, and applications.

What is the purpose of these stages for application development and deployment? In J2EE, application architecture starts with decomposing the application into modules, and modules into application components. This is a top-down process towards building fine-grained building blocks. Once the development of these building blocks is achieved, we need to construct higher-level constructs from these fine-grained blocks. The notion of application packaging and deployment in multiple levels attempts to achieve this. In this process, we compose fine-grained application components into modules, and then modules into applications.

Application Development and Deployment Roles

Before we look at the development and deployment process in a bit more detail, let's take a brief sidetrack to examine who should be doing what. The J2EE specification, as well as defining a process, also defines a number of roles in development of J2EE applications:

- **J2EE Product Provider**

The J2EE Product Provider provides the base J2EE platform upon which we develop our applications. This will be our relevant server vendor who implements the container architecture and the J2EE APIs are defined by the J2EE specification.

- **Application Component Provider**

This is essentially the application developer who creates the application functionality, although it is possible to sub-divide the role into specific areas of expertise, for example, web developer, EJB developer, etc.

- **Application Assembler**

As we will see shortly, the application assembler takes the application components and packages them together through a series of modules and descriptor files so they can be deployed to the production servers.

- **Deployer**

The deployer takes the packaged application, installs, and configures it for the particular operating environment on which the application will be running.

- **System Administrator**

The System Administrator is responsible for maintaining and administering the application once it has been deployed.

- **Tool Provider:**

The Tool Provider provides tools that can be of use in the development and deployment of application components.

Note that this classification of roles may or may not suit your specific application, and this classification should only be considered as a typical set-up to provide some guidance.

Application Component Development

In J2EE, the development process starts with designing and developing application components. We'll be spending most of the book looking at this so there's no need to go any further here.

Composition of Application Components into Modules

A module is used to package one or more related application components of the same type. Apart from the application components, each module also includes a deployment descriptor describing the structure of the module. There are three types of modules in J2EE:

- **Web Modules**

A web module is a deployable unit consisting of Java Servlets, JSP pages, JSP tag libraries, library JAR files, HTML/XML documents, and other public resources such as images, applet class files, etc. A web module is packaged into a **Web ARchive file**, also called a **WAR** file. A WAR file is similar to a JAR file, except that a WAR file contains a **WEB-INF** directory with the deployment description contained in a **web.xml** file. We'll see more details of web modules and their packaging in later chapters.

- **EJB Modules**

An EJB module is a deployable unit consisting of EJBs, associated library JAR files, and resources. EJB modules are packaged into JAR files, with a deployment descriptor (**ejb-jar.xml**) in the **META-INF** directory of the JAR file.

- **Java Modules**

A Java module is a group of Java client classes packaged into JAR files. The deployment descriptor for a Java module is an **application-client.xml** file.

Composition of Modules into Applications

The highest level of packaging is in the form of applications. Each J2EE application is an independent unit of code within a virtual sandbox of its own. Typically, J2EE containers load each application using a different class loader such that each application is isolated from others. The purpose of such isolation is to allow multiple J2EE applications to coexist within a J2EE container.

A J2EE application consists of one or more modules composed into an **Enterprise Archive (EAR) file**. An EAR file is similar to a JAR file, except that it contains an **application.xml** file (located in the **META-INF** directory) describing the application:

Copyrighted image

What is the role of the multiple descriptor files in the above figure? Fine-grained application components can be customized while integrating them into modules. This necessitates the use of a deployment descriptor in a module. However, not all information related to final deployment onto a J2EE platform will be available at the time of assembling modules. In addition, we need a means of specifying which modules make up the application. The `application.xml` file lets you achieve this.

The advantage of this structure is that it allows reuse at different levels. Application components can be reused across multiple web modules. For example, web components related to user login can be packaged into different web modules responsible for online ordering and customer service. Similarly, modules can be reused across multiple applications, so that an EJB module for shopping cart management need not be restricted to a single application, but can be packaged into multiple online commerce applications.

Without such a reusable means of packaging, the goal of component reuse will only be partially met. By defining the above structure, the J2EE allows more refined reuse of application components.

Application Deployment

Finally, application deployment is the process of installing and customizing packaged modules onto a J2EE platform. This process involves two steps:

- Preparing the application for installing onto a J2EE application server. This involves copying the EAR files onto the application server, generating additional implementation classes with the help of the container, and finally installing the application onto the server.
- Configuring the application with application server-specific information. An example is creating data sources and connection factories, as the actual creation of these objects is application server-specific.

Summary

You should now have a better idea of the topology of the J2EE landscape. In particular, we have seen how J2EE extends the traditional multi-tier architecture of distributed computing to include the concept of containers. We discussed the constituents of this container architecture, with relation to:

- The component contract
- Container services
- Declarative services
- Other run-time facilities

The following are the key points of this discussion:

- J2EE is a container-centric architecture. A container is more than a simple runtime: it provides several levels of abstraction. In later chapters, we will examine how these abstractions simplify application development.
- J2EE recognizes the need for composing components into modules, and modules into applications. This is an attempt to standardize reuse of application components and modules.
- J2EE represents a very intuitive approach to building applications. While the design process is top-down, the deployment process is bottom-up and is a composition process, composing modules from components and applications from components.

Having finished our introductory look at the J2EE perspective of server-side Java programming, we will start our more detailed journey with a closer look at service lookup using JNDI.

Copyrighted image

Copyrighted material

2

Directory Services and JNDI

The **Java Naming and Directory Interfaces (JNDI)** are designed to simplify access to the directory infrastructure used in the development of advanced network applications. **Directories** are a special type of databases that provide quick access to their data stores. Databases are usually thought of as the relational data storage model, as in Oracle, Microsoft SQL Server, etc. By contrast, a directory database stores information in a read-optimized, hierarchical form.

Traditionally, we have to use different APIs to access different directory services, such as Sun's Network Information Service (NIS) or the **Lightweight Directory Access Protocol (LDAP)**. However, JNDI supplies a standard API to access any type of directory. JNDI also allows us to store and retrieve Java objects on the network, such as from a J2EE-compliant application server, like BEA WebLogic.

In this chapter, we'll be answering the following questions:

- What is a directory service?
- What separates JNDI from a traditional directory API?
- What is LDAP, and how do we use it to work with a directory service?

Furthermore, by looking at some example programs, we'll demonstrate first hand the practical applications of using JNDI to manage directory information.

Naming and Directory Services

"The network is the computer" is a phrase that Sun Microsystems has used to sell a lot of hardware and software. It is also the central tenet of much successful Java programming. One of the reasons why Java has become so popular is because it has made network programming much easier than other languages, such as C. This is because Java is much more than just a language specification. Java has networking built into its core API, whereas languages such C require an external library to reproduce the same functionality. Built-in networking support helps make Java the success it is because the number of computers connected using the standard Internet protocols has grown at an exceptional rate.

Devices are often placed on a network to help make them easier to manage, which saves time, which in turn saves money. For example, in a multi-building complex a networked environmental system can report to a central server. This server can display the status of all of the air conditioners and thermostats in the various buildings to the environmental engineers. From this central source, the engineers can know when a unit is overheating, or when to schedule maintenance. If the environmental system were not networked, the members of the team would have to check each device individually, which would be a time-consuming task when dealing with dozens of buildings.

Of course, even if we have networked devices and a great development language like Java, it can be difficult to combine them effectively. Whenever a device is added to the network, configuration information must be set. This configuration information includes properties like an IP address, a domain name, the abilities of each networked device, and user privileges for each device. This information is usually available to the device itself, but not always to the outside world.

For example, when we set up a new workstation on the network, we often need to configure it to use a specific network printer. Discovering the capabilities of the printers that are available is often not possible, because the printers do not make this information available. By using a directory service and Java, the printer's capabilities and configuration can be built into the network. Devices can then query the printer to discover if, for example, it can print in color.

The designers of the J2EE environment and its APIs understood these problems. This is why the Java Naming and Directory Interfaces (JNDI) API was made a fundamental part of J2EE. You will find that virtually all large J2EE applications use JNDI at some point.

To understand JNDI, we need a basic understanding of what a naming and a directory service are. We also need to understand why it is important that JNDI provides a standardized interface to these services.

Naming Services

A naming service is a service that enables the creation of a standard name for a given set of data.

On the Internet, each host has a **Fully Qualified Domain Name (FQDN)** such as www.apress.com or www.coe.unt.edu. An FQDN is constructed from a hostname, zero or more sub-domain names, and a domain name. For example:

| FQDN | Hostname | Sub-Domain Name | Domain Name |
|--|----------|-----------------|-------------|
| www.coe.unt.edu | www | coe | unt.edu |
| www.apress.com | www | - | apress.com |
| www.java.apress.com | www | java | apress.com |

Each FQDN is unique, so there is only one www.apress.com and only one www.coe.unt.edu allowed. By using sub-domains and domain names, systems that share the same hostname are still considered separate entities. For example, www.java.apress.com is different from www.apress.com. In this case, the sub-domain java.apress.com is actually below the root domain of apress.com providing for its own unique namespace.

Directory Services

Copyrighted image

The hierarchical information model that directory services employ is in contrast to the relational information model employed by databases, such as Oracle and Microsoft SQL Server, which are built to handle transactions; an operation only succeeds if a collection of smaller operations is successfully completed. A directory service is also usually designed so that it can be distributed and replicated.

A directory service will always have a naming service, but a naming service does not always have a directory service. A simple example of a directory service is a telephone book. A telephone book allows a user to look up the telephone number of a person quickly – provided the user knows the name of the person they are looking for.

Staying with this example for a moment, we can emphasize the benefit of networked directory services. Namely, a directory service is only as useful as the freshness of its information and the efficiency of performing a lookup. Static resources, such as physical copies of phone books, TV Guides, etc., become obsolete soon after being published, and they usually only offer one means of access (for example, via "surname" in the case of a phone book). An electronic directory service, on the other hand, is clearly more dynamic and enables much more flexible searching. The information that you receive is likely to be as fresh as when you queried it.

There are many directory (and 'pseudo-directory') services in use on networks today. One commonly used directory service is the **Domain Naming Service (DNS)** used on the Internet. DNS takes an FQDN and returns an IP address. This service is vital to the Internet as successful communication between two computers relies on each system knowing the IP address of the other.

IP addresses currently consist of 32 bit numbers, for example 198.137.240.92, while a hostname takes the form of www.apress.com. Computers are happy dealing with numbers, but most humans are better at remembering names – it is a lot easier to remember www.apress.com than it is to remember 198.137.240.92. The DNS system is an example of a highly specialized directory service. When we enter a hostname in a browser, the computer gets the server's IP address via DNS. We'll take a more direct look at the practical side of DNS with an example later in the chapter.

DNS actually provides us with a bit more information than just IP addresses. For example, the names of the machines that are responsible for routing mail, but it is still highly specialized in what it does.

Organizations often use one or more of the following general-purpose directory services:

- Novell Directory Services (NDS)
- Network Information Services (NIS/NIS+)
- Active Directory Services (ADS)
- Windows NT Domains

Although the NT Domains service is not a true directory service, many people try to use it as one.

Each of these directory services provides more information than the simple name-to-IP mapping we get from DNS. Each directory service also allows information to be stored about users (usernames and passwords), user groups (for access control), and computers (ethernet and IP addresses). NDS and ADS allow more functions – such as the location of network printers, and software – than either NIS or NT Domains.

As there are so many directory services (each with a proprietary protocol) and we have so many systems on our networks, some larger problems have arisen. Essentially, they boil down to two issues:

- Keeping track of users
- Keeping track of network resources such as computers and printers.

A particularly difficult scenario would be where users will need access to:

- A Novell system (which uses NDS) for file and print sharing
- An account on a Windows NT box for running Microsoft Office
- An account on an UNIX box (using NIS) to use e-mail and publish web pages

Unfortunately for network managers, none of these directory services can easily interact with each other, which makes it very difficult for the traditional application developer to interact with several different directory services. As a result users often end up with different usernames and passwords on each system. If a user leaves, it can be difficult to ensure that all their accounts are removed. Security holes occur in the form of 'orphan' accounts, which are accounts that belong to users who are no longer with an organization, yet still have open access to the systems.

A solution to these problems is to use JNDI, which provides a standard API to a variety of directory services. However, because not everyone uses Java, there needs to be another way to make it easier to communicate directory information between systems. The way to do this is to use LDAP.

LDAP

The **Lightweight Directory Access Protocol (LDAP)** was developed in the early 1990s as a standard directory protocol. LDAP is now probably the most popular directory protocol and, because JNDI can access LDAP, we'll spend some time learning how to harness LDAP to improve Java applications with JNDI.

LDAP can trace its roots back to the X.500 protocol (also known as the 'Heavy' Directory Access Protocol), which was originally based on the OSI networking protocols (an early 'competitor' to the Internet protocols).

You will often interact with a server that has been specifically built for LDAP, such as the **iPlanet Directory Server**. However, LDAP can be a frontend to any type of data store. Because of this, most popular directory services now have an LDAP front-end of some type including NIS, NDS, Active Directory, and even Windows NT Domains.

Copyrighted image

LDAP Data

Data in LDAP is organized in a hierarchical tree, called a **Directory Information Tree (DIT)**. Each 'leaf' in the DIT is called an entry and the first entry in a DIT is called the root entry.

An entry comprise a **Distinguished Name (DN)** and any number of attribute-value pairs. The DN is the name of an entry and must be unique, like the unique key of a relational database table. A DN shows how the entry is related to the rest of the DIT, just as the full path name of a file shows how it is related to the rest of the files on a system.

The path to a file on a system reads left to right when reading from root to file. A DN reads right to left when reading from root to entry, for example:

```
uid=scarter, ou=People, o=airius.com
```

The leftmost part of a DN is called a **Relative Distinguished Name (RDN)** and is made up of an attribute-value pair that is in the entry. The RDN in this example would be `uid=scarter`.

LDAP attributes often use mnemonics as their names. These are some of the more common LDAP attributes:

| LDAP Attribute | Definition |
|------------------------|--------------------|
| <code>cn</code> | Common name |
| <code>sn</code> | Surname |
| <code>givenname</code> | First name |
| <code>uid</code> | User ID |
| <code>dn</code> | Distinguished Name |
| <code>mail</code> | E-mail address |

Attributes can have one or more values. For example, a user can have more than one e-mail address so they would need more than one value for their `mail` attribute. Attribute values can be text or binary data and are referred to in name-value pairs.

There is also a special attribute called `objectclass`. The `objectclass` attribute of an entry specifies what attributes are required and what attributes are allowed in a particular entry. Like objects in Java, object classes in LDAP can be extended. When an object class is extended for a particular entry, the object class keeps the existing attributes, but new attributes can be specified.

Here is an example LDAP entry (which we'll see more of later in this chapter) represented in the **LDAP Data Interchange Format (LDIF)**, which is the most common way to show LDAP data in a human-readable format:

Copyrighted image

Copyrighted image

Attributes also have matching rules. These rules tell the server how it should decide whether a particular entry is a 'match' or not for a given query. The possible matching rules are:

| Matching Rule | Meaning |
|-------------------------------|---|
| DN | Attribute is in the form of a Distinguished Name |
| Case-Insensitive String (CIS) | Attribute can match if the value of the query equals the attribute's value, regardless of case |
| Case-Sensitive String (CSS) | Attribute can match if the value of the query equals the attribute's value including the case |
| Telephone | Is the same as CIS except that characters like "-" and "(" are ignored when determining the match |
| Integer | Attribute match is determined using only numbers |
| Binary | Attribute matches if the value of the query and the value of the attribute are the same binary values (for example, searching a LDAP database for a particular photo) |

The definition of attributes, attribute matching rules, and the relationship between object classes and attributes are defined in the server's schema. A server contains a pre-defined schema but, as long as the server supports the LDAP v3 protocol as defined in RFC 2251 (<http://www.ietf.org/rfc/rfc2251.txt>), the schema can be extended to include new attributes and object classes.

LDAP servers have other benefits. They support referrals – pointers to other LDAP directories where data resides – so a single LDAP server could search millions of entries from just one client request and LDAP data can be replicated to improve reliability and speed. LDAP also has a very strong security model using ACLs to protect data inside the server and supporting the **Secure Socket Layers (SSL)**, **Transport Layer Security (TLS)**, and **Simple Authentication and Security Layer (SASL)** protocols.

LDAP has growing momentum as a central directory service for network systems. For more information about LDAP you should see Implementing LDAP by Mark Wilcox, from Wrox Press, ISBN 1-861002-21-1.

Introducing JNDI

While LDAP is growing in popularity and in usage, it is still a long way from being ubiquitous. Other directory services such as NIS are still in widespread use and are likely to remain so for some time yet. Another issue is that enterprise applications often need to support existing distributed computing standards such as the **Common Object Request Broker Architecture (CORBA)**, which is heavily used in many large organizations to allow different types of applications to interact with each other.

CORBA is a language- and platform-independent architecture that enables distributed application programming, where an application on one machine can access a function of a different application located on a different machine as if it was calling an internal function. CORBA uses the COSNaming (CORBA Object Service) service to define the location of available objects.

To overcome these difficulties, a standard Java API has been created to interact with naming and directory services. This API is analogous to how developers use JDBC to interact with all sorts of databases.

JNDI is very important for the long-term development of Java, particularly the Enterprise JavaBeans (EJB) initiative. The significance of EJBs to J2EE is reflected in the fact that a large proportion of this book is dedicated to them. Starting with Chapter 16, we'll see first-hand the importance of JNDI to the EJB functionality. A key component of the EJB technology is the ability to store and retrieve Java objects on the network. A directory service is going to be the primary data store for stable Java objects, which are retrieved from the network more often than they are stored. This is because when objects are loaded from the network, we want to be able to locate them quickly and a directory service enables very fast lookup and retrieval of data. This data can be anything, including a particular user's record or binary data like a serialized Java object.

The following diagrams will help to illustrate the relationship between directory services, JNDI, and LDAP. The first diagram shows the relationship between a client and a variety of directory services. Each directory service requires its own API, which adds complexity and "code bloat" to our client application:

Copyrighted image

This can be simplified by using JNDI. There are still multiple servers and multiple APIs underneath, but to the application developer, it appears to be a single API:

Copyrighted image

So, as application developers, we now have an easier time developing network applications because we can concentrate on a single API. However, we still face potential problems because not all JNDI service providers are created equal. **Service providers** in JNDI terminology are the drivers that allow interaction with different directory services. Each directory service is clearly different – they name their entries differently and have different capabilities – so developers still build applications that are larger and more prone to failure.

More importantly, it can be difficult to integrate with systems that can understand LDAP but are unable to use Java.

Finally, we show how JNDI and LDAP can work together, providing the most elegant solution:

Copyrighted image

Here we use JNDI to communicate with various LDAP servers. The developer only has to worry about one particular protocol (LDAP) and one API (JNDI). Of course, we are relying on vendors to provide LDAP interfaces to their proprietary protocols. However, this is not a problem as for these popular directory services, there are additional products that allow us to communicate with them via LDAP.

The Tradeoffs

While JNDI provides a common API for programming to different directory services, there are tradeoffs that have to be made:

- One such tradeoff is that there is a lot of overhead. For example, every `DirContext` object (the objects that JNDI uses to interact with a directory service) must support a number of methods, most of which you will never use. When you implement a `DirContext` object, it is simple to leave them empty so that they do not cause problems with your application, but it does add extra weight in terms of memory and compiling time during program execution. We'll see an example of this later in this chapter.
- In addition, so that JNDI can keep a common interface, it does not always support all of the functions of the directory service. A good example of this is the LDAP `compare` command. This enables us to compare a particular value of an attribute in a single entry in the LDAP server. The LDAP server must return a result code instead of a full search result set, thus making the `compare` operation lightweight and quick. However, JNDI does not support a true LDAP `compare`, because LDAP is the only directory service that supports such an operation. Instead, JNDI performs a simple search that physically retrieves the entry into memory from the server and performs the `compare` operation in the application. The result appears the same to the application but we no longer have the benefits of a real LDAP `compare`.
- Finally, there is no way to gain direct access to the directory server connection, which means that it's very difficult to implement connection pooling or to check for connection timeouts in an application.

For most applications, these drawbacks do not present a serious problem. JNDI provides a common API to a variety of directory services, which is included as a standard part of Java, in the same way that JDBC provides a standard API to relational databases. In some cases, JNDI may be the only available API for a particular directory service like DNS or a Java application server like BEA WebLogic.

However, in some cases, applications do need low-level access to the directory service and if the application does not need any of the services that JNDI provides (like storage and retrieval of Java objects), we may want to turn to a low-level Java API. For example, in LDAP there is the Netscape Directory SDK for Java, which is actually more mature than the JNDI LDAP API. It is an open source SDK, available at <http://www.mozilla.org/directory/>.

JNDI may also not be the best solution when an application needs to manipulate files on the local file system. Unless Java objects like `DataSource` objects for JDBC programming are to be loaded, it is better in this case to use the standard classes available in the `java.io` package.

Why Use JNDI When We Have LDAP?

So, why do we need JNDI if we can communicate to systems via LDAP? Similarly, why do we need LDAP when we have JNDI? To answer these questions, let's look at how we could use JNDI or LDAP without the other.

LDAP Without JNDI

LDAP is great way to converge access to directory data through a single protocol. This is useful not just in Java programming but to other types of applications as well.

For example, LDAP is a great way to provide a standards-based network address book (most e-mail packages now have the ability to query an LDAP server for e-mail address lookups) or to easily keep track of devices on a network.

JNDI provides the ability to directly talk to different network directory services when we do not need LDAP. For example, you may need to determine if someone has access to a particular application based on their membership in a particular NIS group or you might need to retrieve an MX record in a DNS server. JNDI makes it easier for the developer in these cases by providing a consistent API instead of having to learn how to do the same thing using different sets of APIs.

JNDI Without LDAP

While you could provide a similar service to LDAP for your Java applications through JNDI, this would only be useful to your Java applications and you would likely have to repeat it in every Java application you write that needs this type of service.

LDAP is an open standard maintained by the Internet Engineering Task Force (IETF). This means that it is accessible from a variety of different clients and vendors. This is one of the great things about the Internet and why the Internet-based protocols have replaced proprietary protocols as the primary means of network based communications.

What About XML?

When XML was first unveiled there was some discussion as to whether it would replace all previous technologies including Java and LDAP. Of course, this didn't happen. Instead XML has become the de facto standard for persisting data, in particular when you need to exchange persistent data between applications written in different application languages (for example between Visual Basic and Java).

XML also has a role in LDAP and JNDI. There is a particular XML specification for LDAP called the **Directory Services Markup Language (DSML)** that is being developed primarily to distribute directory information via an XML-RPC-based protocol like SOAP. It consists of a markup language for representing directory services in XML. For more details, take a look at <http://www.dsml.org/>.

DSML is designed to replace the standard way of exchanging LDAP data outside of the LDAP protocol. The current standard is the LDAP Data Interchange Format (LDIF) and that's what we'll use to display our examples in this chapter.

Using JNDI

Now that we have a basic understanding of directory services, JNDI, and LDAP, it is time to get our hands dirty by doing some work.

To use JNDI as described in this chapter we'll need the following:

- JDK 1.3 from Sun, which includes JNDI as standard, or the JNDI Software Development Kit (SDK) 1.2.2 (available from <http://java.sun.com/jndi/>).
- An LDAP v3-compliant directory server. Netscape's Directory server, available from <http://www.ldap.com/downloads/download/index.html>, is used in this chapter. All the examples you will see in this chapter are using the www.ldap.com sample data that comes with this directory server.

While JNDI ships with JDK 1.3 and higher, you might still need to get the `ldap.jar` file from the JNDI 1.2.2 to get all of the functionality, including the ability to use LDAP controls and to store/retrieve RMI objects in an LDAP server.

Installing JNDI

Here are the general steps to follow to get things working on a typical system:

- Obtain connection information from your LDAP server administrator or install an LDAP v3-compliant server. I have used a variety of LDAP servers for various developments, including openLDAP (<http://www.openldap.org>) on Red Hat Linux 6.2, Eudora's free LDAP server for Windows (<http://www.eudora.com>), and The iPlanet 4 & 5 Directory servers (<http://www.iplanet.com>) on both Red Hat Linux 6.2 and Windows 2000.
- Install JDK 1.3, which contains JNDI and the basic LDAP provider. Unless you're using advanced LDAP features such as controls, you will not need to install anything else except the JDK 1.3. If you're exploring advanced LDAP features, read the JNDI page at <http://java.sun.com/jndi/> for more information.

Now we are ready to use JNDI and LDAP.

JNDI Service Providers

You cannot use JNDI without using a service provider so we need understand what they are.

Copyrighted image

For a service provider to be available for use in JNDI it must implement the `Context` interface. Most of the service providers you will use will likely implement the `DirectoryContext` interface, which extends the `Context` interface to enable directory services.

What this means is that we only have to learn JNDI to know the API calls to connect to a naming or directory service, while the service provider worries about the ugly details like the actual network protocol and encoding/decoding values.

Unfortunately, service providers are not a panacea for using directory services. We must know something about the underlying directory service so that we can correctly name our entries and build the correct search queries. Unlike relational databases, directory services do not share a common query language such as SQL.

In JNDI 1.2, service providers can support a concept of **federation** where a service provider can pass an operation to another service provider if it does not understand the naming or operation scheme. For example if we wanted to find out the MX record (this contains the preferred destination e-mail server for a domain) for a particular domain from DNS, but our initial service provider was LDAP, it would not be able to answer the request. If federation was enabled it would be able to pass the request to the next service provider on the federation list, and, if it was a DNS service provider, it would be able to handle the request for us.

Ideally, this should be transparent to the application programmer. However, since this is a new feature, it has not been widely tested or implemented.

So to sum it up, a service provider enables our JNDI applications to communicate with a naming/directory service. The rest of the interfaces, classes, and exceptions all revolve around our interaction with a service provider.

How to Obtain JNDI Service Providers

If we download the JNDI SDK, we'll see that, along with the API and documentation, it comes with a number of existing service providers. These include providers for LDAP, NIS, COS, and the RMI registry and file system. Many different vendors also provide service providers for other directory services, or as replacements for the default providers that Sun ships. For example, Novell has a service provider for NDS, while both IBM and Netscape have written alternative service providers for LDAP.

It is easy to switch between service providers. For example to use the default Sun LDAP service provider we would make a call like this:

Copyrighted image

Now to switch to using IBM's LDAP service provider you would simply replace the `com.sun.jndi.ldap.LdapCtxFactory` with the full package name of the IBM LDAP service provider like this:

```
// Specify which class to use for our JNDI provider  
env.put(Context.INITIAL_CONTEXT_FACTORY,  
        "com.ibm.jndi.LDAPCtxFactory");
```

There is a list of existing service providers at <http://java.sun.com/products/jndi/serviceproviders.html>

Developing Your Own Service Provider

You may also need to implement your own service provider if you need to use a directory service (such as Windows NT domains or Banyan Vines) that does not already have an existing service provider.

In the JNDI SDK Sun provides an example of how to write a service provider. This JNDI tutorial is available at: <http://java.sun.com/products/jndi/tutorial/index.html>. Sun also provides a PDF document that describes the process of writing a service provider at <ftp://ftp.javasoft.com/docs/jndi/jndispi.pdf>. Netscape's LDAP service provider is available as an opensource project at <http://www.mozilla.org/directory/>.

Java and LDAP

Before we begin to actively look at how to use LDAP in Java, we should take a brief look at what LDAP is commonly used for in Java applications.

There are three basic applications of LDAP in Java today:

- Access Control
- White Pages Services
- Distributed Computing Directory

Let's take a look at these applications in greater detail.

Access Control

All applications dictate which users can access them. This can range from allowing anyone who can click on the application's icon to start it up to allowing only a user who matches a particular retinal scan to do so. Most applications lie in between these two extremes.

Access control can be broken into authentication and authorization.

Authentication

Authentication is determining who the person using a piece of software is. Because of the nature of computing we can never be 100% certain of who the user is, but various authentication mechanisms improve the odds that the user is who they claim to be.

Authentication requires the use of a shared secret, the most common form of which is a password. The user gives a username and a password. The username is used to look up the person's record in a database (this database can be a simple flat file like the UNIX `passwd` file or a sophisticated database like LDAP) and if the password given is the same as that stored in the database then we assume that the user is who they say they are.

Password-based authentication is the simplest but one of the least secure methods; because passwords are written down, they can be sniffed in transit over the networks, or they can be easily cracked when a nefarious person gains unauthorized access to the password database. In today's networked world, the biggest threat to passwords (outside of asking the user personally for their password) is to travel over the network in the clear. Thus, it is best to do things to protect passwords when they travel over the network. Two common mechanisms are **MD5** and **Secure Socket Layers (SSL)**.

Under MD5 the client first creates an MD5 hash of the password before passing it across the network. The server then compares this hash with the one it has stored in its database. While this does prevent someone from guessing the plaintext password and using it to login to other systems, this doesn't prevent someone from stealing the hashed password. This stolen hashed password could be used just like a stolen plaintext password to gain access to the system (known as the "man in the middle" attack). Another problem is that most systems do not store their passwords as MD5 (although certain BSD implementations do) so you might be forced to change everyone's passwords before you can implement this system.

Under SSL the entire network connection is encrypted. This protects not only the password, but also any data transferred between the client and the server. This is why SSL is often used to protect e-commerce applications. It is also useful in LDAP because LDAP servers often contain sensitive data like personal identification numbers, driver license numbers, etc. We do not want this data to travel over the network insecurely anymore than we want passwords to.

SSL also adds another form of authentication altogether and that is the **digital certificate**. Digital certificates use public key encryption to improve the authentication process. Under public key encryption the user has a private key, which only they have access to. All data encrypted by the private key can only be decrypted by the public key and vice versa. The server has access to their public key. Under SSL (which already encrypts all data transfer), the server sends the user some data to encrypt with their private key. The user encrypts the data and sends it back to the server. If the server can decrypt the data with the public key and the data matches, then it's assumed to be the user. The digital certificate standard (X.509) is an extension of the original directory services standard. Thus LDAP is one of the best ways of managing digital certificates. It is worth noting though that this user authentication actually happens *after* the SSL server has been authenticated.

LDAP has the capability to support a wide range of authentication services including passwords, digital certificates, and the Simple Authentication and Security Layer (SASL) protocol.

Authorization

After we have authenticated someone, we need to determine what they are actually allowed to do. For example, we might decide that only members of the **Dwarves** have the right to access the **Snow White** files. We might also add finer granularity than just simply saying someone has access or not. For example, we could say that members of the **Jedi Council** have full rights over items in the **Force** database, while members of the **Jedi Knights** have the ability to edit certain elements in the **Force** database, whereas **padawan** only have the ability to read the **Force** database.

We can use LDAP to develop sophisticated authorization policies.

White Pages Services

White pages services are services that enable someone to look up users based on attributes contained in their entries. For example, you can look up Mark Wilcox's e-mail address, and obtain the telephone number of the Engineering office, the building number of Human Resources, etc. They are called **white pages** because this type of information is similar to the type of information you find in the white pages of US telephone books.

These types of services are the most public of all LDAP operations and are what many, if not most, people use LDAP for. Under Java, we're putting it second because most white pages services are provided through an LDAP client found in an e-mail package like Netscape Messenger or Microsoft Outlook instead of through a Java application.

Java applications normally perform this function either as the back-end for a web page LDAP interface or to provide workflow services in an enterprise Java application. As an example, let's look at a simple situation in which John enters an electronic purchase order request for 100 widgets from Acme Widgets. The purchasing application has a business rule that says after a person enters a PO request it must be sent to the PO authorization person for their group. Therefore, the application looks up in LDAP to find out who the PO authorizer is for John's group in LDAP and sends them an e-mail notification using the authorizer's LDAP e-mail attribute.

Distributed Computing Directory

One of the fastest growing segments of server programming is distributed network programming, where an application uses code that actually resides separately from the running application. The code can be either in a separate JVM (or similar engine like a CORBA server) or on a different physical machine located on the other side of the world.

We often use this type of programming to make it easier to reuse existing legacy code or to improve application performance by offloading heavy processing onto a separate machine.

In Java, there are three distributed architectures available:

- Remote Method Invocation (RMI)
- Common Object Request Broker Architecture (CORBA)
- Enterprise JavaBeans (EJB)

All three provide a registry service that a client application uses to locate the distributed code. Because its specification was developed after JNDI, EJB utilizes JNDI for its registry services.

RMI and CORBA use their own independent registry services. One of the problems with these services is that they do not provide a mechanism to search their registry to discover what objects are available to use.

With JNDI and LDAP you can provide indirect references with the JNDI Reference object to these services. For example, you can have an entry with a name of "Real Time Stock Quote Service" that has an attribute that contains the actual network location to the RMI or CORBA object.

This type of service makes your code easier to understand and gives you more flexibility in your application. For example, if you need to move a particular distributed object to a new server, instead of having to change all of the applications that reference that object, you only have to change its location in LDAP and then all of the client applications will update their locations automatically the next time they reference the object.

LDAP can also store other descriptive attributes in an entry, so you could store better descriptions of your object in the directory to build white pages for objects. People who have actually been a part of projects for their companies to develop centralized code libraries often find that developing the code was the easy part; making the code easy to find is where the difficulties arise. Using LDAP as a class directory is a solution to these problems.

We could add other elements to the description of our stock quote object example. For instance, we could add things such as the development language used, a note about its purpose, and the name of the developer or development team.

Application Configuration

LDAP can also be used to store configuration information about an application. This is particularly helpful where the application is used by the same user but on different machines and it would be helpful to provide the same configuration information regardless of what machine the user used to access it.

Now we will move from talking in hypothetical terms and explaining the basic concepts, to actual practice. In the remainder of this chapter, we will see some real code and how to perform basic LDAP operations with JNDI.

LDAP Operations

Before we can perform any type of operation on an LDAP sever, we must first obtain a reference and a network connection to the LDAP server. We must also specify how we wish to be bound to the server - either anonymously or as an authenticated user. Many Internet-accessible LDAP servers allow some type of anonymous access (generally read-only abilities for attributes like e-mail addresses and telephone numbers), but LDAP also supports very advanced security features via ACLs that are dependent upon who the connection is authenticated as.

For example, an LDAP server may have several layers of rights for any given entry:

- Anonymous users can see an employee's e-mail address and telephone number.
- The employee can see their entire entry, but only modify certain attributes such as telephone number, password, and office room number.
- A user's manager can update an employee's telephone number and office room number, but nothing else. They can also see the employee's entire record.
- A small group, the Directory Administrators, have full rights to the entire server including the ability to add or remove any entry.

Standard LDAP Operations

There are a few standard operations in LDAP. These are:

- Connect to the LDAP server
- Bind to the LDAP server (you can think of this step as authenticating)
- Perform a series of LDAP operations:
 - Search the server
 - Add a new entry
 - Modify an entry
 - Delete an entry
- Disconnect from the LDAP server

We will step through each of these, demonstrating with examples where relevant.

Connecting to the LDAP Server with JNDI

We must first obtain a reference to an object that implements the `DirContext` interface. In most applications, we will use an `InitialDirContext` object that takes a hash table as a parameter. This hash table can contain a number of different references. It should contain a reference to a field with the key `Context.INITIAL_CONTEXT_FACTORY`, with a value of the fully qualified class name of the service provider, the hostname, and the port number to the LDAP server. These additional properties are set using the `Context.PROVIDER_URL` key. The value of this key should be the protocol, hostname, and port number to the LDAP server like this: `ldap://localhost:389`.

We first create a `Hashtable` to store our environmental variables that JNDI will use to connect to the directory service:

```
Hashtable env = new Hashtable();
```

Next, we specify the fully qualified package name of our JNDI provider. We are using the standard Sun LDAP service provider that comes with the JNDI SDK:

```
// Specify which class to use for our JNDI provider  
env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.ldap.LdapCtxFactory");
```

We must specify the hostname and port number to our LDAP server, for example:

```
// Specify host and port to use for directory service  
env.put(Context.PROVIDER_URL, "ldap://localhost:389");
```

Finally, we get a reference to our initial directory context with a call to the `InitialDirContext` constructor, giving it our hash table as its only parameter. A directory context tells JNDI what service provider we will be using, what naming/directory server we will be connecting to, from what location we will be accessing the directory from initially (for example, the search base), and any authentication information:

```
// Get a reference to a directory context  
DirContext ctx = new InitialDirContext(env);
```

Binding

If you use the default values, the connection will be bound as anonymous. Many LDAP servers provide some type of read access to their directory data (for example, for address book applications). Specifically the type of access an application has to the LDAP server is dependent upon the **Access Control Lists (ACLS)** of the LDAP server. The ACLs that apply to an operation are determined by how the application is bound or authenticated.

LDAP allows for an extremely flexible security model. ACLs determine what particular access is available to an entry by an application. Because an entry can have several ACLs defined, it is possible that an entry will have several different views to an application simply by changing the binding (for example, who the application is authenticated as). LDAP also supports Transport Security Layer (TSL also still known as Secure Socket Layer (SSL)) for protecting content 'over the wire' or to improve authentication via client-certificates. Finally, LDAP supports the Simple Authentication and Security Layer (SASL) protocol that enables you to use other authentication/encryption mechanisms such as Kerberos without "breaking" the protocol.

You can specify authentication by specifying the `Context.SECURITY_AUTHENTICATION`, `Context.SECURITY_PRINCIPAL`, and `Context.SECURITY_CREDENTIALS` in the hash table passed to the `InitialDirContext` object.

To specifically bind to the server, we must provide the environment with the method for our authentication (for example, "simple", SSL, or SASL). Then we must specify the DN of the entry we wish to bind as and the entry's password:

Copyrighted image

Simple, SSL/TLS, and SASL Security

The latest LDAP specification, LDAP v3, allows for three types of security:

- Simple
- SSL/TLS
- SASL

Let's look at each type, in turn.

Simple

Simple security means that you will only authenticate to the server using standard plain-text user IDs and passwords without any encryption on the network. This is by far the most common, and least secure of the various authentication methods. It is insecure for two reasons, one of which is that user IDs and passwords are transmitted to the server over a public network, where anyone can steal the user IDs and passwords off the network. Secondly, there is nothing to guarantee that the person who types in the user ID and password is the actual owner of that user ID and password.

SSL/TLS

The **Secure Socket Layer** protocol was developed by Netscape Communications to improve the security of web-based transactions. It has become an official standard called **Transport Layer Security (TLS)**, but is still often referred to as SSL.

SSL allows you to encrypt your entire transaction over the network, making it very hard for anyone to steal the information (such as your user IDs and passwords). Standard SSL does not verify the identity of the person who typed in the user ID and password (however, it does ensure that the machine you are issuing your ID and password to is "the real server"). Most servers that implement SSL also support client-certificates (certificates are text files that are used to vouch for the identity of the server and client) for user authentication. Instead of presenting a user ID and password to the system, you can present a certificate to the server. If the certificate you present matches an allowed certificate, you are granted access.

Certificates are considered more secure because they are hard to fake. However, certificates are typically stored as a file on a local user's machine, which means that if the client machine is compromised, then a certificate can be used just like a stolen user ID and password. When certificates are stored locally, a mechanism must be developed to recover them if the machine they are stored on crashes or is upgraded. Certificates can be stored on smart cards instead of files on a local machine for more security and reliability. The issuing and managing of client certificates is still in the early stages of development.

SASL

The **Simple Authentication and Security Layer (SASL)** is an Internet standard for implementing authentication mechanisms besides simple or SSL.

There are two popular SASL mechanisms. One is the **MD5** that we saw earlier, based on comparison of hashed-passwords. MD5 doesn't encrypt the transaction and it doesn't solve the problem of "who typed in the password". If the MD5-hashed password is stolen while on its way to the server, a hacker could use that to gain access to the system, just as if it was a plain-text password. It does however, make it harder to guess what the password originally was, so if a hacker does steal the hashed password, they won't be able to guess what the password was to try and use it to gain access to other systems.

The second popular SASL mechanism is called **Kerberos**. Kerberos encrypts the transaction, and in a Kerberos-aware network, it is very easy to implement a single-logon environment because of the way Kerberos works. However, managing a Kerberos network is very time- and resource-consuming, so many organizations are yet to implement Kerberos.

You can easily write your own SASL mechanisms, if your LDAP server supports them. Thus, it would be possible to enable biometric authentication, once it becomes available (or if your organization has access to the technology already). Biometric authentication uses part of a person's body such as their thumbprint or iris for their user ID and password. These systems are very secure.

LDAP v2 and LDAP v3 Authentication

In LDAP v2, all clients had to authenticate themselves before performing any operations. In LDAP v3, however, if a client does not authenticate itself before performing an operation, the connection is assumed to be an anonymous authenticated connection.

Searching an LDAP Server

The most used operation on any LDAP server is the search operation. Any advanced LDAP applications use searching as their core functionality. Essentially, all search functions take an LDAP connection handle, the base to start the search from, the scope of the search, and a search filter. A search filter is like a SQL query in that you tell the server the criteria to use to find matching entries. Searches always use an attribute name and a value to look for. Filters can use Boolean logic and wildcards. Some servers, such as the Netscape Directory server, support even more advanced query abilities such as "sounds like".

For the search examples in this section, we will be accessing the arius.com sample data that comes with the Netscape Directory Server.

Example LDAP Filters

Some typical LDAP filters are shown in the following table:

| Description | LDAP Filter |
|--|---|
| Find all users with the last name of Carter | sn = Carter |
| Find all users with last names that start with "Ca" | sn = Ca* |
| Find all object classes of the type GroupofUniqueNames that have "Managers" in their Common Name | (&(cn = * Managers *) (objectclass=groupofuniqueNames)) |

In JNDI, we use the search method of the DirContext interface. This will return back a NamingEnumeration object if the search is successful.

Later in this section, we will show you the various ways you can manipulate the values of this object to get back attributes and values of each returned entry.

Determining LDAP Scope

When you perform a search, you must specify the node (which we refer to as the **base**) of the tree you want to start at, as well as the scope of the search. The scope defines exactly how much of the tree you want to search. There are three levels of scope.

| Scope | Description |
|--------------------|--|
| LDAP_SCOPE_SUBTREE | This scope starts at the base entry and searches everything below it including the base entry. |

Table continued on following page

| Scope | Description |
|---------------------|--|
| LDAP_SCOPE_ONELEVEL | This scope searches the entire tree starting one level below the base entry. It does not include the base entry. |
| LDAP_SCOPE_BASE | This scope searches just the base entry, useful if you want to just get the attributes/values of one entry. |

Next we'll examine these three levels of scope with the aid of some diagrams.

LDAP_SCOPE_SUBTREE

The first figure, below, illustrates a search using **LDAP_SCOPE_SUBTREE**:

Copyrighted image

LDAP_SCOPE_ONELEVEL

Here we demonstrate an example of a search using **LDAP_SCOPE_ONELEVEL**:

Copyrighted image

LDAP_SCOPE_BASE

Finally, we present a search using `LDAP_SCOPE_BASE`:

Copyrighted image

Performing a JNDI Search

We perform a search using the `search()` method of an object that implements the `DirContext` interface (such as the `InitialDirContext` class). The minimum requirement for this is the search base and a filter, although there are other parameters we can use to help manage the results.

We should note that in most LDAP APIs, we must specify the scope as a parameter in the LDAP search method. In JNDI, however, the scope is set in the `SearchControls` object, which is an optional parameter to the `search()` function of a `DirContext` interface (the `InitialDirContext` class provides an implementation of the `search()` function). By default, it is set to `subtree`. The search is actually performed with whatever object you have that implements the `DirContext` interface, such as the `InitialDirContext` class.

For our first example we'll show a very simple search example, where the search filter is "`sn=Carter`". This will return all entries that have a surname (the attribute specified by the mnemonic "`sn`") of Carter. This first example is an anonymous search.

The examples are non-GUI based to make it easier to understand what's going on, but there's nothing to prevent them from being included in a GUI. Remember, all of the source code for the examples described in this chapter can be downloaded from <http://www.apress.com>. Here's what the code looks like:

Copyrighted image

```
public static void main(String args[]) {  
  
    try {  
        // Hashtable for environmental information  
        Hashtable env = new Hashtable();  
  
        // Specify which class to use for our JNDI provider  
        env.put(Context.INITIAL_CONTEXT_FACTORY, INITCTX);  
  
        // Specify host and port to use for directory service  
        env.put(Context.PROVIDER_URL, MY_HOST);  
  
        // Get a reference to a directory context  
        DirContext ctx = new InitialDirContext(env);  
  
        // Specify the scope of the search  
        SearchControls constraints = new SearchControls();  
        constraints.setSearchScope(SearchControls.SUBTREE_SCOPE);  
  
        // Perform the actual search. We give it a searchbase, a filter,  
        // and a the constraints containing the scope of the search.  
        NamingEnumeration results = ctx.search(MY_SEARCHBASE,  
                                              MY_FILTER, constraints);  
  
        // Now step through the search results  
        while (results != null && results.hasMore()) {  
            SearchResult sr = (SearchResult) results.next();  
  
            String dn = sr.getName();  
            System.out.println("Distinguished Name is " + dn);  
  
            Attributes attrs = sr.getAttributes();  
  
            for (NamingEnumeration ne = attrs.getAll(); ne.hasMoreElements();) {  
                Attribute attr = (Attribute) ne.next();  
                String attrID = attr.getID();  
  
                System.out.println(attrID + ":" );  
                for (Enumeration vals = attr.getAll(); vals.hasMoreElements();) {  
                    System.out.println("\t" + vals.nextElement());  
                }  
            }  
            System.out.println("\n");  
        }  
    } catch (Exception e) {  
        e.printStackTrace();  
        System.exit(1);  
    }  
}
```

Copyrighted image

I

We perform a search using the `search()` method of an object that implements the `DirContext` interface (such as the `InitialDirContext` class). The minimum requirement for this is the search base and a filter. There are other parameters we can use to help manage the results. If the search is successful, a `NamingEnumeration` object will be returned.

After we get the initial context (which we set the variable `ctx` to), we next specify the scope of our search. If we do not specify a scope, JNDI will assume a scope of `subtree`, so this next line is actually redundant but is useful to show you how to specify the scope.

Let's now go through the process of how this code works. First, we set the scope to `subtree`:

```
// Specify the scope of the search
SearchControls constraints = new SearchControls();
constraints.setSearchScope(SearchControls.SUBTREE_SCOPE);
```

After we specify the scope, we can perform the actual search like this:

```
// Perform the actual search
NamingEnumeration results = ctx.search(MY_SEARCHBASE, MY_FILTER, constraints);
```

The `NamingEnumeration` class is equivalent to the `SearchResults` class in the Netscape Directory SDK for Java.

Each element in a **NamingEnumeration** object will contain a **SearchResult** object that we can retrieve:

```
 SearchResult sr = (SearchResult) results.next();
```

Then we can get the DN of an entry:

```
 String dn = sr.getName();
```

To get the attributes of an entry you use the **getAttributes()** method of the **SearchResult** class:

```
 Attributes attrs = sr.getAttributes();
```

This will return a concrete object that implements the **Attributes** interface (the **InitialDirContext** class returns a **BasicAttributes** object).

After we have an **Attributes** object (remember this is a collection class), we can then step through the attributes using a **NamingEnumeration** object like this:

```
 for (NamingEnumeration ne = attrs.getAll(); ne.hasMoreElements();) {  
     Attribute attr = (Attribute)ne.next();  
     String attrID = attr.getID();  
  
     System.out.println(attrID + ":" );  
     for (Enumeration vals = attr.getAll(); vals.hasMoreElements();) {  
         System.out.println("\t" + vals.nextElement());  
     }  
 }
```

The **NamingEnumeration** class gives us methods that we can use to step through each attribute that was returned in our search. Each element in the **NamingEnumeration** object will contain an **Attribute** object that represents an attribute and its values.

Each element in the **Attribute** object is an object that has implemented the **Attribute** interface (the **InitialDirContext** class uses **BasicAttribute** objects). The **getID()** method of the **Attribute** interface returns the name of the attribute. The **getAll()** method of the **Attribute** interface will return back a standard Java **Enumeration** object, which we can then access to get the values of the individual attribute.

In every LDAP server, there are certain attributes that are not going to be available to anonymous users because the access controls on the server. There are also certain attributes that may only be available to certain privileged users (pay scale, for example, may only be visible to Human Resources).

Authenticated Searching

The following shows how we can modify the example for an authenticated search:

```
 import java.util.Hashtable;  
 import java.util.Enumeration;  
  
 import javax.naming.*;  
 import javax.naming.directory.*;
```

```
public class JNDISearchAuth {  
  
    public static String INITCTX = "com.sun.jndi.ldap.LdapCtxFactory";  
    public static String MY_HOST = "ldap://localhost:389";  
    public static String MGR_DN = "uid=kvaughan, ou=People, o=airius.com";  
    public static String MGR_PW = "bribery";  
    public static String MY_SEARCHBASE = "o=Airius.com";  
  
    public static String MY_FILTER = "(sn=Carter)";  
  
    public static void main(String args[]) {  
        try {  
            // Hashtable for environmental information  
            Hashtable env = new Hashtable();  
  
            // Specify which class to use for our JNDI provider  
            env.put(Context.INITIAL_CONTEXT_FACTORY, INITCTX);  
  
            // Security Information  
            // authenticates us to the server  
            env.put(Context.PROVIDER_URL, MY_HOST);  
            env.put(Context.SECURITY_AUTHENTICATION, "simple");  
            env.put(Context.SECURITY_PRINCIPAL, MGR_DN);  
            env.put(Context.SECURITY_CREDENTIALS, MGR_PW);  
  
            // Rest of class is unchanged.  
            DirContext ctx = new InitialDirContext(env);  
  
            SearchControls constraints = new SearchControls();  
            constraints.setSearchScope(SearchControls.SUBTREE_SCOPE);  
  
            NamingEnumeration results = ctx.search(MY_SEARCHBASE,  
                                              MY_FILTER, constraints);  
  
            // Now step through the search results  
            while (results != null && results.hasMore()) {  
                SearchResult sr = (SearchResult) results.next();  
  
                String dn = sr.getName();  
                System.out.println("Distinguished Name is " + dn);  
  
                Attributes attrs = sr.getAttributes();  
  
                for (NamingEnumeration ne = attrs.getAll(); ne.hasMoreElements();) {  
                    Attribute attr = (Attribute) ne.next();  
                    String attrID = attr.getID();  
  
                    System.out.println(attrID + ":");  
                    for (Enumeration vals = attr.getAll(); vals.hasMoreElements();) {  
                        System.out.println("\t" + vals.nextElement());  
                    }  
                }  
                System.out.println("\n");  
            }  
        } catch (Exception e) {  
            e.printStackTrace();  
            System.exit(1);  
        }  
    }  
}
```

This second search example is the same as the first, except that we have authenticated ourselves to the server.

So, if you try compiling and running this example, you'll see that it produces exactly the same output as before. Note that by default the LDAP server returns all of the attributes for a search. There may, however, be occasions when we don't want this, because we are only concerned with particular attributes.

Restricting the Attributes Displayed

In our third example, we ask to only be shown the common name (**cn**) and e-mail address (**mail**) attributes:

```
import java.util.Hashtable;
import java.utilEnumeration;

import javax.naming.*;
import javax.naming.directory.*;

public class JNDISearchAttrs {

    // Initial context implementation
    public static String INITCTX = "com.sun.jndi.ldap.LdapCtxFactory";

    public static String MY_HOST = "ldap://localhost:389";
    public static String MY_SEARCHBASE = "o=Airius.com";

    public static String MY_FILTER = "(sn=Carter)";

    //Specify which attributes we are looking for
    public static String MY_ATTRS[] = {"cn", "mail"};

    public static void main(String args[]) {
        try {
            //Hashtable for environmental information
            Hashtable env = new Hashtable();

            //Specify which class to use for our JNDI provider
            env.put(Context.INITIAL_CONTEXT_FACTORY, INITCTX);

            env.put(Context.PROVIDER_URL,MY_HOST);
            //Get a reference to a directory context
            DirContext ctx = new InitialDirContext(env);

            SearchControls constraints = new SearchControls();
            constraints.setSearchScope(SearchControls.SUBTREE_SCOPE);

            NamingEnumeration results =
            ctx.search(MY_SEARCHBASE,MY_FILTER,constraints);

            while (results != null && results.hasMore()) {
                SearchResult sr = (SearchResult) results.next();
                String dn = sr.getName() + ", " + MY_SEARCHBASE;
                System.out.println("Distinguished Name is " + dn);
        }
    }
}
```

Copyrighted image

Copyrighted image

```
        }
    }
} catch(Exception e) {
    e.printStackTrace();
    System.exit(1);
}
}
```

Since we have just specified the common name and mail attributes this time, the resulting restricted output look like this:

Copyrighted image

The difference this code and our earlier example searches is that we now limit the number of attributes we retrieved.

We created a String array that listed the attributes we wanted:

```
public static String MY_ATTRS[] = {"cn", "mail"};
```

To retrieve this set of attributes we use the `getAttributes()` method of the `DirContext` interface, where we provide the DN of a specific entry and the array of attributes:

```
Attributes ar = ctx.getAttributes(dn, MY_ATTRS);
```

This will return an **Attributes** object.

We can then retrieve a particular **Attribute** object from an **Attributes** object:

```
Attribute attr = ar.get("cn");
```

At this point, we should emphasize that, although retrieving a specific set of attributes from an individual entry is very quick, it is not very practical for general searching. In a general LDAP search, the end user is not going to know the existing Distinguished Names of the entries they are looking for. So we will have to search the LDAP server and retrieve a set of entries.

In JNDI (as opposed to the Netscape Directory SDK for Java), this search will return all of the attributes associated with each individual entry. If we then make subsequent call to **getAttributes()** to retrieve a subset of attributes like in the previous example, this will require another call to the LDAP server and get back the subset of attributes. This is inefficient because it requires us to use extra memory for all of the attributes and extra bandwidth for the extra communication. The extra memory is required because our application must hold the data of the LDAP search results for us to process.

To improve performance in our Java applications, we want to reduce the amount of extraneous memory we use because the JVM's garbage collector can be slow to react, or slow your application to a crawl, as it reclaims memory. As garbage collection is improving and memory is cheap, we may be tempted to use this without a thought. However, there is another potential problem area and that is the fact that it does require us to use multiple network connections, where we might only have used one under a lower-level LDAP API.

This isn't to say that JNDI is bad to use, just that we should understand the implications of its use.

We can also use JNDI to add new entries to the server, delete entries, and modify existing entries. We'll take a look at these operations next.

Adding Entries

Using JNDI to add entries to an LDAP server is in fact more difficult than it is with other LDAP SDKs. This is because JNDI's primary goal is to read and write Java objects to the network. A consequence of this is that a programmer must jump through some extra hoops, such as creating a Java class for each type of entry they want to add to the LDAP server. Here we'll look at how to add and modify a simple entry in the LDAP server, but later in the chapter we'll learn how to use the LDAP server as an object store.

To store an entry in a LDAP server using JNDI, we must bind an object to a Distinguished Name (DN). This means that each object we store in the server (whether a simple person entry or a serialized Java class) must have a DN associated with it. Remember a DN is the unique name that each entry in a LDAP server must possess. If we switch to a different directory service (such as NDS) we will still be required to have a unique name for each object. Now if you are bit overwhelmed by this, just remember that it will become second nature over time and that we do this (provide an unique name) each time we save a file to hard disk. No file on the file system can share the same name. If we wish to have two files named **myfile.txt**, we must store them in separate directories in our file system otherwise one version will overwrite the other.

To store even a simple entry in the LDAP server, we must create a class that implements the **DirContext** interface. This interface defines how the object (be it a person or serialized Java class) should be stored into the directory server as well retrieved from the server. For instance, if you have a **Person** object, your class will specify how to build its DN, how to store the available attributes (for example, full name, e-mail address, telephone number, user ID, password, etc.) and provide various mechanisms to handle the retrieved data. The **DirContext** also provides for many more sophisticated data handling and is the basic interface for building a directory service provider.

As with any other LDAP SDK an add operation can only be performed by an authenticated user who has rights to add a new entry into the server. LDAP ACLs can be set up so that users can only add entries into particular parts of the directory tree.

Our next code sample shows a very simple **Person** class that implements the **DirContext** interface. Most of the methods in the interface are not actually implemented (except to throw exceptions) because we do not need them for our very simple example here. Note that this class is derived from the **Drink.java** example found in the JNDI tutorial (<http://java.sun.com/products/jndi/tutorial/>).

The methods that we implement here, **getAttributes()** and the constructor, enable us to store/retrieve the data in a **Person** class as traditional LDAP entries. The rest of the methods that we don't fully implement are primarily used to build full service providers. Instead, we simply throw exceptions stating that we don't support the particular service. New objects must also conform to the LDAP server's schema, or the entries will not be added.

It will be easier to explain how to add an entry with JNDI if we explain the code as we go along. Again, note that the complete source code for this example is available for download from the Apress web site.

First is our class declaration. Note that we state that we will implement the methods for the **DirContext** interface:

```
import java.util.*;
import javax.naming.*;
import javax.naming.directory.*;

public class Person implements DirContext {
    String type;
    Attributes myAttrs;
```

Next we have our constructor. This constructor takes several strings that we will use to build an **inetOrgPerson** object class:

```
public Person(String uid, String givenname, String sn,
              String ou, String mail) {
    type = uid;
```

We will use the **BasicAttributes** class to store our attributes and their values. By specifying **true** in the **BasicAttributes** constructor we are telling it to ignore the case of attribute names when doing attribute name lookups:

```
myAttrs = new BasicAttributes(true);
```

To add a multi-valued attribute we need to create a new **BasicAttribute** object, which requires the name of the attribute in its constructor. We then add the values of the attribute with the **add()** method:

```
Attribute oc = new BasicAttribute("objectclass");
oc.add("inetOrgPerson");
oc.add("organizationalPerson");
oc.add("person");
oc.add("top");
```

```
Attribute ouSet = new BasicAttribute("ou");
ouSet.add("People");
ouSet.add(ou);

String cn = givenname + " " + sn;
```

Finally we add all of our attributes to the **BasicAttributes** object:

```
myAttrs.put(oc);
myAttrs.put(ouSet);
myAttrs.put("uid", uid);
myAttrs.put("cn", cn);
myAttrs.put("sn", sn);
myAttrs.put("givenname", givenname);
myAttrs.put("mail", mail);
}
```

When **getAttributes()** is called it will return our **BasicAttributes** object when requested by a name in the form of a String. It is designed to only return the attributes of a specific entry, but since this class will only hold one entry, it's not going to be called. We're showing one way to implement it if it was:

```
public Attributes getAttributes(String name) throws NamingException {
    if (! name.equals("")) {
        throw new NameNotFoundException();
    }
    return myAttrs;
}
```

This method does the same thing as the first **getAttributes()** but is only called when the name is passed a **Name** object:

```
public Attributes getAttributes(Name name) throws NamingException {
    return getAttributes(name.toString());
}
```

The following method returns only the attributes listed in the String array **ids**. The String name should be a DN:

```
public Attributes getAttributes(String name, String[] ids)
    throws NamingException {
    if (! name.equals("")) {
        throw new NameNotFoundException();
    }

    Attributes answer = new BasicAttributes(true);
    Attribute target;
    for (int i = 0; i < ids.length; i++) {
        target = myAttrs.get(ids[i]);
        if (target != null) {
            answer.put(target);
        }
    }
    return answer;
}
```

This is a similar implementation to those of the other `getAttributes()` methods, except that it takes a `Name` object:

Copyrighted image

The `toString()` method is used for serialization:

Copyrighted image

Finally, the following lines are used to implement methods that a JNDI service provider (such as the `InitialDirContext` class) would use to provide an application with services such as reading entries from the directory or for authenticating to the server:

Copyrighted image

The full source code for this example and all of the examples in this book are available for download from
<http://www.apress.com/>

Next, we demonstrate the `JNDIAdd.java` class, which uses the `Person` class, developed above, to add an entry for Mark Wilcox to the LDAP server (we've seen most of this code already):

```
import java.util.Hashtable;
import java.utilEnumeration;

import javax.naming.*;
import javax.naming.directory.*;

public class JNDIAdd {

    // Initial context implementation
    public static String INITCTX = "com.sun.jndi.ldap.LdapCtxFactory";

    public static String MY_HOST = "ldap://localhost:389";
    public static String MGR_DN = "uid=kvaughan, ou=People, o=airius.com";
    public static String MGR_PW = "bribery";
    public static String MY_SEARCHBASE = "o=Airius.com";

    public static void main(String args[]) {
        try {
            // Hashtable for environmental information
            Hashtable env = new Hashtable();

```

```
// Specify which class to use for our JNDI provider
env.put(Context.INITIAL_CONTEXT_FACTORY, INITCTX);

env.put(Context.PROVIDER_URL,MY_HOST);
env.put(Context.SECURITY_AUTHENTICATION,"simple");
env.put(Context.SECURITY_PRINCIPAL,MGR_DN);
env.put(Context.SECURITY_CREDENTIALS,MGR_PW);

// Get a reference to a directory context
DirContext ctx = new InitialDirContext(env);

}

}

} catch (Exception e) {
e.printStackTrace();
System.exit(1);
}
```

Copyrighted image

First we must create a new Java object that implements the `VirContext` interface such as our `Person` class like this:

```
Person p = new Person("mewilcox", "Mark", "Wilcox",
                      "ou=Accounting", "mewilcox@airius.com");
```

Then we associate a name (specifically the DN of the entry) with this object in our current context with the `bind()` method of the `DirContext` interface like this:

```
ctx.bind("uid=mewilcox,ou=People,o=airius.com", p);
```

The `InitialDirContext` interface will actually perform an LDAP add operation. It will take all of the attributes we have placed into our Java class and then encode them for transfer into a LDAP server.

As we used the `BasicAttribute` class to build our attributes, they will be stored/retrieved as standard LDAP data and not as pure Java objects. This means that if you store your LDAP data this way any other LDAP client regardless if its written in C, Perl or Visual Basic will still be able to access it.

After successfully compiling and running our Person.java and JNDIAdd.java class files, we can see the result by looking in the airius.com directory of the Netscape Directory Server. Here we show our new entry:

Copyrighted image

By
to]

id according

Copyrighted image

Modifying an Entry

Just as soon as you add an entry to an LDAP server, you'll need to modify it. This could be for a variety of reasons including changing a user's password, updating an application's configuration, etc.

Modifications to an entry are made with the `ModificationItem` and `BasicAttribute` classes. When we make a modification, it can be one of ADD, REPLACE, or DELETE. A REPLACE will add an attribute if it does not exist yet.

Copyrighted image

Again, modifications must be performed by an authenticated user and those modifications that can be performed will be determined by the rights the bound entry has on a particular entry. For example, users can generally change their passwords but nothing else, while administrative assistants usually can change telephone numbers and mailing addresses. It usually takes a database administrator to do things like change a user's user ID.

The code shown below demonstrates how we can modify the attributes of the Mark Wilcox entry that we added in the previous example:

```
import java.util.Hashtable;
import java.util.Enumeration;

import javax.naming.*;
import javax.naming.directory.*;

public class JNDIMod {

    // Initial context implementation
    public static String INITCTX = "com.sun.jndi.ldap.LdapCtxFactory";

    public static String MY_HOST = "ldap://localhost:389";
    public static String MGR_DN = "uid=kvaughan, ou=People, o=airius.com";
    public static String MGR_PW = "bribery";
    public static String MY_SEARCHBASE = "o=Airius.com";

    public static void main(String args[]) {

        try {
            // Hashtable for environmental information
            Hashtable env = new Hashtable();

            // Specify which class to use for our JNDI provider
            env.put(Context.INITIAL_CONTEXT_FACTORY, INITCTX);

            env.put(Context.PROVIDER_URL, MY_HOST);
            env.put(Context.SECURITY_AUTHENTICATION, "simple");
            env.put(Context.SECURITY_PRINCIPAL, MGR_DN);
            env.put(Context.SECURITY_CREDENTIALS, MGR_PW);

            // Get a reference to a directory context
            DirContext ctx = new InitialDirContext(env);
        }
    }
}
```

Copyrighted image

```
    e.printStackTrace();
    System.exit(1);
}
}
```

To modify an entry we use the **ModificationItem** class. The **ModificationItem** takes a modification type (for example, ADD, REPLACE, or DELETE) and an **Attribute** object such as **BasicAttribute**. Here is a simple example that lets us add a new attribute, locality (the l attribute), with a new value of "Waco" to the entry:

```
Attribute mod1 = new BasicAttribute("l", "Waco");
mods[1] = new ModificationItem(DirContext.ADD_ATTRIBUTE, mod1);
```

The actual modification is performed by the **DirContext** method, **modifyAttributes()**. Here is an example of this:

```
ctx.modifyAttributes("uid=mewilcox,ou=People,o=airius.com", mods);
```

Again this modifies the entry in the LDAP server using traditional LDAP and not as a Java object so that any other client can still access this data.

This time when we take a look at the Mark Wilcox's entry in the Directory Server, and click on Advanced... for a detailed view, we see that the phone number and locality have indeed been added to the directory:

Copyrighted image

Delete an Entry

Eventually, you will need to remove entries from your LDAP server. This is easily accomplished by calling the `destroySubContext()` method of the `DirContext` interface, with the distinguished name of the entry that needs to be removed. Normally, delete operations are restricted to the LDAP database administrators.

Here is our example modified to delete an entry:

```
import java.util.Hashtable;
import java.utilEnumeration;
import javax.naming.*;
import javax.naming.directory.*;

public class JNDIDelete {
    // Initial context implementation
    public static String INITCTX = "com.sun.jndi.ldap.LdapCtxFactory";

    public static String MY_HOST = "ldap://localhost:389";
    public static String MGR_DN = "cn=Directory Manager";
    public static String MGR_PW = "jessica98";
```

```
public static String MY_SEARCHBASE = "o=Airius.com";  
  
public static String MY_ENTRY = "uid=mewilcox, ou=People, o=airius.com";  
  
public static void main(String args[]) {  
    try {  
        // Hashtable for environmental information  
        Hashtable env = new Hashtable();  
  
        // Specify which class to use for our JNDI provider  
        env.put(Context.INITIAL_CONTEXT_FACTORY, INITCTX);  
  
        env.put(Context.PROVIDER_URL, MY_HOST);  
        env.put(Context.SECURITY_AUTHENTICATION, "simple");  
        env.put(Context.SECURITY_PRINCIPAL, MGR_DN);  
        env.put(Context.SECURITY_CREDENTIALS, MGR_PW);  
  
        // Get a reference to a directory context  
        DirContext ctx = new InitialDirContext(env);  
  
        ctx.destroySubcontext(MY_ENTRY);  
    } catch(Exception e) {  
        e.printStackTrace();  
        System.exit(1);  
    }  
}
```

The only real difference in the code of this example is this line:

```
ctx.destroySubcontext(MY_ENTRY);
```

So, after compiling and running this program, it will remove our entry from the LDAP server.

This was a whirlwind tour through LDAP. If you want more information, there are a number of resources. Wrox Press has a complete book on the subject, *Implementing LDAP* by Mark Wilcox, ISBN 1-261002-21-1, or you can check out the LDAP Guru web site at <http://wwwldapguru.com/>

Storing and Retrieving Java Objects in LDAP

One of JNDI's strongest attributes is its ability to use the network as an object store. What this means is that we can use the network to share Java objects either with distributed versions of an application or disparate applications. For example, we might create a class that calculates the sales tax of an item. This type of class could potentially be used in applications by our sales force to create purchase orders and by our accounting office to do their 'bean counting'.

We can use nearly any of the available JNDI providers to store our classes for us, but one of the most useful ways is by using an LDAP server to do this.

There are several reasons why you would like to use LDAP as your data store:

- Leverage an existing centralized resource
- Leverage existing open standards

- ❑ LDAP is available on the network "out of the box"
- ❑ LDAP is optimized for extremely quick read access
- ❑ LDAP has strong security built in

JNDI allows you to store several types of Java-related objects into the LDAP server:

- ❑ **Standard LDAP directory entries**

This gives the ability to manipulate standard directory data (`inetOrgPerson`, `groupOfUniqueNames`, `objectclasses`, and so on). Standard directory data is smaller in size (and therefore quicker to access and modify) and you can share it between different languages. The ability to stay language-neutral with directory data is of utmost importance in a large enterprise, where several different languages (for example, Java, Perl, C/C++, Visual Basic, Cobol, etc.) maybe used for development.

- ❑ **Serialized Java objects**

This gives the ability to store and retrieve Java objects that have been serialized (the current objects, and all related classes, are stored in a binary format) into the LDAP server. Probably the easiest format to use, but also requires the most bandwidth and storage space. This format is only understandable by Java in support for object persistence. The two most common uses of serialized Java objects are EJBs and JDBC `DataSource` objects.

- ❑ **References**

As we discussed earlier, distributed computing platforms like CORBA and RMI use registries to keep track of the object classes they expose. However, these registries do not facilitate the ability to search on them and force you to hard-code the server location of the object in your application. A Java `Reference` object enables you to store data in the directory server as an indirect reference to the real object's location. For example, you could have a real time stock quote object on an RMI server. Instead of hard coding the location of the RMI registry in your application, you could connect to an LDAP server and locate the object "`cn=Real Time Stock Quote`", which contains the RMI URL to the stock quote object. This reduces the need to recompile your class every time you move the stock quote class to a different registry and it makes your code a bit easier to read.

The option you choose to use to store your objects will depend upon the application you are building and how you need to access the data. We'll look at these three objects in more detail in the next few sections.

Traditional LDAP

Organizations will often want to access the data in the directory service from a variety of clients using a number of different languages. A popular use of a directory service is for user authentication. Obviously storing user authentication data in a format that only Java can use reduces the number of applications that can use the directory for authentication. This in turn raises the cost of doing business because you then need another directory service for providing authentication services to applications that can't access Java objects.

A number of applications won't need authentication but could benefit from a directory service's address book features. For example, your e-mail program can use it to find the e-mail address of co-workers, your marketing department can use it to build mail-merged form letters, while your web developers can use it to make a custom portal for each customer. Each of these applications could be built using Java as the development language. Most importantly, your company can significantly reduce its overhead by maintaining consistent data about its people and clients in a central database.

One of the neat things about storing data in this fashion is that you can treat each entry like an object in Java, but other languages don't have to be object-oriented in order to access the data.

Serialized Java

If, however, we have a growing number of Java applications that need access to a central repository of pre-built Java objects, then we could use Java's serialization to store those objects into the LDAP server. Then when an application needs a particular object (for example a 3D rendering engine) it can retrieve it from the LDAP server, only when it's needed, then release it. Another nice feature of this is that when we update the rendering engine, all of the applications that are using the engine will have access to the update without having to patch the end-applications.

Java References

Finally, if we are in an all-Java environment we might wish to take advantage of Java references. References reduce the storage and bandwidth requirements of storing and retrieving entries because they don't store the entire object in the directory, instead only storing key components that are needed to rebuild the object in a factory class. Often the components stored are a URL that points to the real location of the object.

For example, if we create a standard printer object, we can build a `PrinterFactory` that might take the following parameters:

- Network location
- Color options – monochrome, color
- Specialties – postscript, graphic plotter
- Current status – out of paper, on-line

These parameters can be passed to `PrinterFactory` and will always return a `Printer` object that our application can use to print. It reduces bandwidth and storage because only the above parameters need to be stored. It's easier for an application programmer to deal with because part of the reference is the fully qualified package name of the actual factory, so we don't have to include it with our application, we just need to make sure the JVM can find the package.

If you want to read more about using JNDI and LDAP as a Java object store, you can see the official JNDI site at <http://java.sun.com/jndi/>.

Returning to JNDI without LDAP

While LDAP is an important part of an Internet services network, and many people use JNDI so that they can use LDAP in their Java applications, LDAP is not the only option available. LDAP makes sense when you need to manage demographic information that doesn't change very often but needs to be accessed very quickly.

However, there are times when you need to access data that cannot be stored in LDAP, such as DNS entries or Enterprise JavaBeans.

While the rest of the chapter will concentrate on the **Domain Naming Service (DNS)**, we should note that outside of LDAP, the most likely place we'll use JNDI is with EJBs and other J2EE services from an application server. We'll see some examples later in this book.

The reason why we focus on DNS here as opposed to a full J2EE example is that the examples in this chapter hope to demonstrate how flexible JNDI can be by using simpler examples than would be possible using EJBs.

Example DNS Application

To demonstrate how we might use the same basic JNDI classes but with different service providers, we will present a variation on the JNDISearch application that we used in our LDAP section.

This section requires the DNS JNDI Service provider that is available at the Java Developer's Connection and will be included in JDK 1.4 (it's currently there if you install the JDK 1.4 beta). The relevant files are dns.jar and providerutils.jar, which should both be placed into your classpath on running the program.

LDAP and DNS both have entries. Each entry is access by a unique name (in LDAP it is the Distinguished Name and in DNS it is the canonical name). Both services are built in a hierarchy; remember that is one of the characteristics of a directory service.

DNS has a very well-defined way of building the tree (starting at top-level domains like .com, .us, .net, followed by second level domains like yahoo.com, followed by sub-domains and hosts). LDAP, on the other hand, has no set rules for building its tree structure (that's why you'll encounter DNs built like o=Acme Inc, c=US or o=acme.com or dc=acme, dc=com). Both services have entries that contain attributes with one or more values. LDAP has object classes that can define different types of entries. DNS only contains entries in regards to Internet hosts. LDAP has a strong authentication-based security model – DNS does not.

We have already seen the basics of this application earlier, so we will not spend much time explaining the details. We will focus on the DNS-specific parts.

```
import java.util.Hashtable;
import java.util.Enumeration;

import javax.naming.*;
import javax.naming.directory.*;

public class JNDISearchDNS {

    public static void main(String args[]) {
        try {
            // Hashtable for environmental information
            Hashtable env = new Hashtable();
```

Note that we're changing our provider factory from com.sun.jndi.dns.LDAPCtxFactory to com.sun.jndi.dns.DnsContextFactory. Since we're changing our factory, we need to change our provider URL as well from ldap:// to dns://. The 129.120.210.252 refers the IP address of the DNS server we will be communicating with:

Copyrighted image

The instance variable, dns_attributes, contains the list of attributes we would like returned about a particular host, but there is no guarantee that all of these attributes will be available to us. For example, if the server is not a mail server, it won't have an MX record. Or the DNS administrator may not have the OS information in the HINFO field (below) because of security (if an attacker could determine the host OS for a particular machine, then they could focus their attack using tools to break that particular OS):

```
String dns_attributes[] = {"MX", "A", "HINFO"};
```

Here we get the initial DirContext object and we attempt to retrieve the attributes for the host named `www.unt.edu`:

```
// Get a reference to a directory context
DirContext ctx = new InitialDirContext(env);
Attributes attrs1 = ctx.getAttributes("www.unt.edu", dns_attributes);
```

Next we check to make sure that we got something back from DNS. The reason why we might not have any results is that there might not be any hosts that match our hostname in the DNS server:

Copyrighted image has none of the specified attributes\n");

Now we print out our results, using the same algorithm as earlier:

```
: z++) {
:es[z]);
"
;vals.hasMoreElements()); {
());
}

}
System.out.println("\n");
}
}
} catch(Exception e) {
e.printStackTrace();
System.exit(1);
}
}
```

Copyrighted image

which denotes the IP address of the host and there is an HINFO record, which says that the server is currently running UNIX.

As you can see, switching providers in JNDI is truly painless.

Summary

Directory services are an important part of a networked services environment. Directories provide the ability to associate human understandable (and memorable) names with the information computers need to operate (for example domain names to IP addresses, or common names to Java references). They also allow us to look up e-mail addresses and to associate permissions for secure environments.

LDAP provides us with a standard network protocol for communicating with directory services, while JNDI provides a standard API for Java to communicate with LDAP and specialized directory services that are not accessible via LDAP (such as DNS).

After this brief introduction to directory services, JNDI, and LDAP, you should understand the basics and will hopefully be able to venture on from here.

We will continue our journey through some of the fundamental elements of enterprise programming in the next chapter by taking an in-depth look at distributed computing enabled by RMI.

Copyrighted image

Copyrighted material

3

Distributed Computing Using RMI

Distributed computing has become a common term in today's programming vocabulary. It refers to the application design paradigm in which programs, the data they process, and the actual computations are spread over a network, either to leverage the processing power of multiple computers or due to the inherent nature of an application comprising different modules.

Remote Method Invocation (RMI) allows object-to-object communication between different Java Virtual Machines (JVM). JVMs can be distinct entities located on the same or separate computers – yet one JVM can invoke methods belonging to an object stored in another JVM. This enables applications to call object methods located remotely, sharing resources, and processing load across systems. Methods can even pass objects that a foreign virtual machine has never encountered before, allowing the dynamic loading of new classes as required. This really is quite a powerful feature.

If we want two Java applications executing within different virtual machines to communicate with each other there are a couple of other Java-based approaches that can be taken besides RMI:

- Sockets
- Java Message Service (JMS)

Basic network programming with sockets is flexible and sufficient for communication between programs. However, it requires all the involved parties to communicate via application-level protocols, the design of which is complex and can be error-prone. For example, consider a collaborative application, like a simple chat application for instance: for multiple clients to participate we would first need to design some sort of protocol, and then use sockets to communicate in that protocol with the server. RMI, on the other hand, is a distributed system that internally uses sockets over TCP/IP.

The difference between RMI and JMS is that in RMI the objects stay resident and are bound to the virtual machines (although method arguments and returns as well as stubs travel across the network), whereas in JMS the messages (objects) themselves travel asynchronously across the network from one JVM to another.

The RMI Architecture

Before we look at how the RMI mechanism works, let us define some frequently-used terms.

To quote the specification, "In the Java distributed object model, a remote object is one whose methods can be invoked from another Java virtual machine, potentially on a different host. An object of this type is described by one or more remote interfaces, which are Java interfaces that declare the methods of the remote object. Remote method invocation (RMI) is the action of invoking a method of a remote interface on a remote object."

RMI's purpose is to make objects in separate JVMs look and act like local objects. The JVM that calls the remote object is usually referred to as a client and the JVM that contains the remote object is the server. One of the most important aspects of the RMI design is its intended transparency. Applications do not know whether an object is remote or local. A method invocation on a remote object has the same syntax as a method invocation on a local object, though under the hood there is a lot more going on than meets the eye.

In RMI the term "server" does not refer to a physical server or application but to a single remote object having methods that can be remotely invoked. Similarly, the term "client" does not refer to a client machine but actually refers to the object invoking a remote method on a remote object. The same object can be both a client and a server.

Although obtaining a reference to a remote object is somewhat different from doing so for local objects, once we have the reference, we use the remote object as if it were local. The RMI infrastructure will automatically intercept the method call, find the remote object, and process the request remotely. This location transparency even includes garbage collection.

A remote object is always accessed via its remote interface. In other words the client invokes methods on the object only after casting the reference to the remote interface.

The RMI implementation is essentially built from three abstraction layers:

The Stubs/Skeletons Layer

This layer intercepts method calls made by the client to the interface reference and redirects these calls to a remote object. It is worth remembering that stubs are specific to the client side, whereas skeletons are found on the server side.

The Remote Reference Layer

This layer handles the details relating to interpreting and managing references made by clients to the remote objects. It connects clients to remote objects that are running and exported on a server by a one-to-one connection link. In the Java 2 SDK, this layer was enhanced to support the activation framework (discussed later).

The Transport Layer

This layer is based on TCP/IP connections between machines in a network. It provides basic connectivity, as well as some firewall penetration strategies.

yer. The

Copyrighted image

This layered architecture provides good implementation flexibility without affecting the application architecture. Each of the layers can be enhanced or replaced without affecting the rest of the system. For example, the transport layer implementation can be replaced by a vendor with the User Datagram Protocol (UDP) instead of TCP, without affecting the upper layers.

Next we'll look at some of these layers in more detail.

The Stub and Skeleton Layer

To achieve location transparency, RMI introduces two special kinds of objects known as stubs and skeletons that serve as an interface between an application and the rest of the RMI system. This layer's purpose is to transfer data to the Remote Reference Layer via marshaling and unmarshaling. Marshaling refers to the process of converting the data or object being transferred into a byte stream and unmarshaling is the reverse – converting the stream into an object or data. This conversion is achieved via object serialization.

The stub and skeleton layer of RMI lies just below the actual application and is based on the Proxy design pattern. In the RMI use of the Proxy pattern, the stub class plays the role of the proxy for the remote service implementation. The skeleton is a helper class that is generated by RMI to help the object communicate with the stub across the RMI link. The skeleton carries on a conversation with the stub; it reads the parameters for the method call from the link, makes the call to the remote service implementation object, accepts the return value, and then writes the return value back to the stub.

In short, the Proxy pattern forces method calls to occur through a proxy that acts as a surrogate, delegating all calls to the actual object in a manner transparent to the original caller.

Specific details about the proxy design pattern are beyond the scope of this book, but can be found in Design Patterns, Elements of Reusable Object-Oriented Software Erich Gamma, et al. (ISBN: 0-201-63361-2) or Patterns in Java: A Catalog of Reusable Design Patterns (ISBN: 0-471333-15-8.)

Let's now look at stubs and skeletons in a little more in detail.

Stubs

The stub is a client-side object that represents (or acts as a proxy for) the remote object. The stub has the same interface, or list of methods, as the remote object. However, when the client calls a stub method, the stub forwards the request via the RMI infrastructure to the remote object (via the skeleton), which actually executes it. The following lists the sequence of events performed by the stub in detail:

- Initiates a connection with the remote VM containing the remote object
- Marshals (writes and transmits) the parameters to the remote VM
- Waits for the result of the method invocation
- Unmarshals (reads) the return value or exception returned
- Returns the value to the caller

The stub hides the serialization of method parameters (parameters must be serializable; parameter passing is discussed in detail later) and the network-level communication in order to present a simple invocation mechanism to the caller.

In the remote VM, each remote object may have a corresponding skeleton.

Skeletons

On the server-side, the skeleton object takes care of all of the details of "remoteness" so that the actual remote object doesn't need to worry about them. In other words, we can pretty much code a remote object the same way as if it were local; the skeleton insulates the remote object from the RMI infrastructure. During remote method requests, the RMI infrastructure automatically invokes the skeleton object so it can work its magic. The following bullets list the sequence of events in detail:

- Unmarshals (reads) the parameters for the remote method (remember that these were marshaled by the stub on the client side)
- Invokes the method on the actual remote object implementation
- Marshals (writes and transmits) the result (return value or exception) to the caller (which is then unmarshaled by the stub)

*The JDK contains the `rmic` tool that creates the class files for the stubs and skeletons. Details about `rmic` can be found packaged with the JDK, or online at:
<http://java.sun.com/j2se/1.3/docs/tooldocs/tools.html>*

The Remote Reference Layer

The remote reference layer defines and supports the invocation semantics of the RMI connection. This layer provides a JRMP (Java Remote Method Protocol: see next section)-specific `java.rmi.server.RemoteRef` object that represents a handle to the remote object. A `RemoteStub` uses a remote reference to carry out a remote method invocation to a remote object.

In JDK 1.2 onwards stubs use a single method, `invoke(Remote, Method, Object[], Long)` on the remote reference to carry out parameter marshaling, remote method execution, and un-marshaling of the return value.

The Java 2 SDK implementation of RMI adds a new semantic for the client-server connection: activatable remote objects (as we shall see later). Other types of connection semantics are possible. For example, with multicast, a single proxy could send a method request to multiple implementations simultaneously and accept the first reply (which would improve the response time and possibly improve availability). In the future, Sun is expected to add additional invocation semantics to RMI.

The Transport Layer

The transport layer makes the stream-based network connections over TCP/IP between the JVMs, and is responsible for setting and managing those connections. Even if two JVMs are running on the same physical computer, they connect through their host computer's TCP/IP network protocol stack. RMI uses a wire-level protocol called **Java Remote Method Protocol (JRMP)** on top of TCP/IP (an analogy is HTTP over TCP/IP).

JRMP is specified at <http://java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmi-protocol.doc.html>. It is important to note that JRMP is specific to the Sun "implementation". Alternative implementations, such as BEA Weblogic, NinjaRMI, ObjectSpace's Voyager, and so on, do not use JRMP, but instead use their own wire-level protocol. Sun and IBM have jointly developed the next version of RMI, called RMI-IIOP, which is available with Java 2 SDK version 1.3 (or separately). Instead of using JRMP, RMI-IIOP uses the Object Management Group (OMG) Internet Inter-ORB Protocol (IIOP) to communicate between clients and server. IIOP also enables integration with CORBA objects.

From JDK 1.2 the JRMP protocol was modified to eliminate the need for skeletons and instead use **reflection** to make the connection to the remote service object. Thus, we only need to generate stub classes in system implementations compatible with JDK 1.2 and above. To generate stubs we use the `-v1.2` option with `rmic`.

For a detailed description of how JRMP was modified to remove skeletons see the CallData, Operation, and Hash descriptions in the JRMP specification at <http://java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmi-protocol.doc3.html>.

Reflection is an API in the `java.lang.reflect` package in J2SE that enables Java code to discover information about the fields, methods, and constructors of loaded classes at runtime and operate on them.

The RMI transport layer is designed to make a connection between clients and server, even in the face of networking obstacles. While the transport layer prefers to use multiple TCP/IP connections, some network configurations allow a single TCP/IP connection between a client and server (for example, some browsers restrict applets to a single network connection back to their hosting server). In this case, the transport layer multiplexes multiple virtual connections within a single TCP/IP connection.

The transport layer in the current RMI implementation is TCP-based, but again a UDP-based transport layer could be substituted in a different implementation.

Locating Remote Objects

There is still one crucial question that we haven't answered yet. How does a client find the object? Clients find remote services by using a naming or directory service. This may seem like circular logic. How can a client find the naming service? Simple: a naming or directory service is run on a host and port number that the client is already aware of (for example, a well known port on a public host). RMI can transparently look up these different directory services with the Java Naming and Directory Interface (JNDI), which was described in the previous chapter.

The RMI naming service, a registry, is a remote object that serves as a directory service for clients by keeping a `Hashtable`-like mapping of names to other remote objects. It is not necessary to have a single registry on a particular physical host. An object is free to start its own registry. The behavior of the registry is defined by the interface `java.rmi.registry.Registry`. RMI itself includes a simple implementation of this interface called the RMI Registry, (the `rmiregistry` tool with the JDK can also be started programmatically). The RMI Registry runs on each machine that "hosts" remote objects and accepts queries for services, by default on port 1099.

In simple terms, a remote object is associated with a name in this registry. Any time the clients want to invoke methods on this remote object; it obtains a reference to it by looking up the name. The lookup returns a remote reference, a stub, to the object. RMI also provides another class, the `java.rmi.Naming` class that serves as the client's interaction point with the object serving as the registry on the host for this lookup:

Copyrighted image

Copyrighted image

The `Naming` class's methods take, as one of their arguments, a name that is a URL-formatted `java.lang.String`. Let's look at these methods and their descriptions:

| Method | Description |
|---|--|
| <code>public static void bind(String name, Remote obj)</code> | Binds the remote object to a string name. The name itself is in the RMI URL format described below. |
| <code>public static String[] list(String name)</code> | Returns an array of the names bound in the registry. |
| <code>public static Remote lookup(String name)</code> | Returns a reference, a stub, for the remote object associated with the specified name. |
| <code>public static void rebind(String name, Remote obj)</code> | Rebinds (unbinds the name if it is already bound and binds it again) the specified name if it is already in use to a new remote object. This could be dangerous if different applications use the same name in the registry but is helpful in development. |
| <code>public static void unbind(String name)</code> | Removes the binding with specified name. |

The methods in the `Naming` class and `Registry` interface have identical signatures and throw a variety of exceptions (discussed in the next section). There are a number of things that happen when a client invokes a `lookup` for a particular URL in the `Naming` class. First, a socket connection is opened to the host on the specified port (using a client socket factory if necessary). Next, since the registry implementation on the host itself is a remote object, a stub to that remote registry is returned from the host. This stub acts as the client proxy for the registry. Subsequently, the `Registry.lookup()` is performed on this stub and returns another stub, for the remote object that was registered with it on the server. Finally, once the client has a stub to the requested object, it interacts directly with the object on the port to which it was exported.

The URL takes the form:

```
rmi://<host_name>[:<name_service_port>]/<service_name>
```

Where:

- ❑ `host_name` is a name recognized on the local area network (LAN) or a DNS name on the Internet
- ❑ `name_service_port` needs to be specified only if the naming service is running on a port other than the default 1099
- ❑ `service_name` is the string name that the remote object is associated with in the registry

To facilitate this on the host machine, a server program exposes the remote object service by performing the following sequence of events:

1. Creates a local object
2. Exports that object to create a listening service, that waits for clients to connect and request the service

3. Registers the object in the RMI Registry under a public name

The code snippet below summarizes these steps:

Copyrighted image

Similar to the `Naming` class there is another class, the `java.rmi.registry.LocateRegistry` that has various methods for directly getting a reference to the registry and for starting the registry. To reiterate, starting the registry (either by the tool or programmatically) is nothing but exporting the remote object that implements the `Registry` interface.

Copyrighted image

The JDK has a tool called `rmiregistry` that starts the registry on the host with the following command:

```
rmiregistry -J-Djava.security.policy=<policy file>
```

Alternatively, to start the registry with its default values:

```
start rmiregistry
```

Remember the registry is an object running in a JVM. The `-J` flag is used to pass parameters, such as the `policy file`, to the JVM.

You can also programmatically start the registry with the `LocateRegistry.createRegistry(int port)` method or the more detailed `LocateRegistry.createRegistry(int port, RMIClientSocketFactory csf, RMIServerSocketFactory ssf)`. Both these methods return the stub, implementing the interface `java.rmi.registry.Registry`.

Policy Files

Code is granted permissions in what is called a policy file. If we look in the `%JAVA_HOME%\jre\lib\security` directory, we will find the default policy file for our JVM named `java.policy`. This file can be edited either manually or by using the `policytool` program found in the `%JAVA_HOME%\bin` directory. For more information on policy files see *Professional Java Security*, by Wrox Press, (ISBN 1-861004-25-7).

Before going through an example, we'll take a brief diversion and have a look at exceptions in RMI.

RMI Exceptions

As mentioned earlier, RMI is a distributed system that uses sockets over TCP/IP. In such a networked environment, many things could go wrong. It is important that the client is somehow notified about exceptions so that it can handle them in the appropriate manner, and thereby recover from problems as and when they occur. For example, you don't want a client to indefinitely wait for an input on the socket if the network goes down, or the host cannot be reached.

So that the client is aware of such conditions, every remote method must throw the `java.rmi.RemoteException` (or one of its super classes such as `java.io.IOException` or `java.lang.Exception`). The `RemoteException` is a generic exception and there are specialized subclasses that are thrown (and caught) for specific conditions. These are listed in the table below:

| Exception | Description |
|------------------------------------|---|
| <code>AccessException</code> | Thrown by certain methods of the <code>java.rmi.Naming</code> class (specifically – <code>bind()</code> , <code>rebind()</code> , and <code>unbind()</code>) and methods of the <code>ActivationSystem</code> interface (which we'll encounter shortly), to indicate that the caller does not have permission to perform the action requested by the method call. |
| <code>AlreadyBoundException</code> | Thrown if an attempt is made to bind an object in the registry to a name that already has an associated binding. Note that this exception is not thrown (under the same conditions) if <code>rebind()</code> is used. However, use of <code>rebind()</code> may not be right under all scenarios. |
| <code>ConnectException</code> | Thrown if a connection is refused to the remote host for a remote method call. |
| <code>ConnectIOException</code> | Thrown if an <code>IOException</code> occurs while making a connection to the remote host for a remote method call. |
| <code>MarshalException</code> | Thrown if a <code>java.io.IOException</code> occurs while marshaling the remote call header, arguments, or return value for a remote method call. |
| <code>NoSuchObjectException</code> | Thrown if an attempt is made to invoke a method on an object that no longer exists in the remote virtual machine. If this exception is thrown then it means that the object was garbage-collected or unexported. A common occurrence is if the "client or" clients repeatedly fail to renew their leases, and the leases expire. This could occur if, for example, the network is clogged with traffic or goes down. Also keep in mind that the stub with the client is valid only as long as the RMI server is alive. If you don't want the remote object to be garbage-collected then you should keep it alive in the server JVM by storing a reference to the implementation object in a static variable. As a locally reachable object, it won't be garbage-collected even in the event of extended unreachability of clients. |

Table continued on following page

| Exception | Description |
|------------------------------------|---|
| <code>NotBoundException</code> | Thrown if an attempt is made to <code>lookup()</code> or <code>unbind()</code> in the registry a name that has no associated binding. |
| <code>RemoteException</code> | The common super class for a number of communication-related exceptions that may occur during the execution of a remote method call. |
| <code>ServerError</code> | Thrown as a result of a remote method call if the execution of the remote method on the server machine throws a <code>java.lang.Error</code> . |
| <code>ServerException</code> | Thrown as a result of a remote method call if the execution of the remote method on the server machine throws a <code>RemoteException</code> . |
| <code>StubNotFoundException</code> | Thrown if a valid stub class could not be found for a remote object when it is exported. |
| <code>UnexpectedException</code> | Thrown if the client of a remote method call receives, as a result of the call, a checked exception that is not among the checked exception types declared in the "throws" clause of the method in the remote interface. |
| <code>UnknownHostException</code> | Thrown if a <code>java.net.UnknownHostException</code> occurs while creating a connection to the remote host for a remote method call. |
| <code>UnmarshalException</code> | <p>Can be thrown while unmarshaling the parameters or the results of a remote method call if:</p> <ul style="list-style-type: none"> <li data-bbox="746 1464 1944 1508"><input type="checkbox"/> An exception occurs while unmarshaling the call header. <li data-bbox="746 1522 1944 1566"><input type="checkbox"/> If the protocol for the return value is invalid. <li data-bbox="746 1580 1944 1682"><input type="checkbox"/> If an <code>IOException</code> occurs while unmarshaling parameters (on the server side) or the return value (on the client side). |

Developing Applications with RMI

Writing client-server applications using RMI involves six basic steps:

1. Defining a remote interface
2. Implementing the remote interface
3. Writing the client that uses the remote objects
4. Generating stubs (client proxies) and skeletons (server entities)

5. Starting the registry and registering the object

6. Running the server and client

Let's look at each step by developing a simple application as an example to gain some practical experience and aid our understanding of each step.

Defining the Remote Interface

An interface manifests the exposed operations and the client programmer need not be aware of the implementation (the interface in this case also serves as a marker to the JVM). A remote interface by definition is the set of methods that can be invoked remotely by a client:

- The remote interface must be declared public or the client will get an error when it tries to load a remote object that implements the remote interface, unless that client is in the same package as the remote interface
- The remote interface must extend the `java.rmi.Remote` interface.
- Each method must throw a `java.rmi.RemoteException` (or a superclass of `RemoteException`)
- If the remote methods have any remote objects as parameters or return types, they must be interface types not the implementation classes

Note that the `java.rmi.Remote` interface has no methods. It's just used as a marker by the JVM, in a similar fashion to how the `java.io.Serializable` interface is used to mark objects as being serializable.

For our example, we'll define our remote interface like this:

Copyrighted image

Copyrighted image

Implementing the Remote Interface

The implementation class is the actual class that provides the implementation for methods defined in the remote interface. The `java.rmi.server.RemoteObject` extends the functionality provided by the `java.lang.Object` class into the remote domain by overriding the `equals()`, `hashCode()`, and `toString()` methods.

Remember, the generic `java.rmi.server.RemoteObject` is an abstract class and describes the behavior for remote objects.

The abstract subclass `java.rmi.server.RemoteServer` describes the behavior associated with the server implementation and provides the basic semantics to support remote references (for example, create, export to a particular port, and so on).

`java.rmi.server.RemoteServer` has two subclasses:

- `java.rmi.server.UnicastRemoteObject` – defines a non-replicated remote object whose references are valid only while the server process is alive
- `java.rmi.activation.Activatable` – concrete class that defines behavior for on-demand instantiation of remote objects (see later section on activation)

The following diagram shows the two subclasses and their contained methods in relation to the `java.rmi.server.RemoteServer` class:

Copyrighted Image

An object can exhibit remote behavior as a result of either of the following:

- The class extends `java.rmi.RemoteServer` or one of its subclasses (including other remote objects). The class must, however, invoke one of the superclass constructors so that it can be exported.
- The class explicitly exports itself by passing itself ("this") to different forms of the `UnicastRemoteObject.exportObject()` methods.

The term "exporting" encapsulates the semantics that involve a remote object's ability to accept requests. This involves listening on a TCP (server) socket. Note that multiple objects can listen on the same port (see the section on Sockets for more information).

In addition to these, the class must implement one or more remote interfaces that define the remote methods.

A remote class can define any methods but only methods in the remote interface can be invoked remotely.

The `HelloServer` remote object implementation class for our example looks like this:

```
import java.io.*;
import java.rmi.*;
import java.rmi.server.*;
import java.util.Date;

public class HelloServer extends UnicastRemoteObject
    implements HelloInterface {

    public HelloServer() throws RemoteException {
        super();           // Call the superclass constructor to export this object
    }

    public String sayHello() throws RemoteException {
        return "Hello World, the current system time is " + new Date();
    }
}
```

Note that the source code for this example (and all of the examples in this book) is available for download from <http://www.apress.com/>.

Writing the Client that Uses the Remote Objects

The client performs a lookup on the registry on the host and obtains a reference to the remote object. Note that casting to the remote interface is critical. In RMI, clients always interact with the interface, never with the object implementation:

```
import java.rmi.*;

public class HelloClient {

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

```
if (System.getSecurityManager() == null) {
    System.setSecurityManager(new RMISecurityManager());
}
try {
    HelloInterface obj = (HelloInterface) Naming.lookup("/HelloServer");
    String message = obj.sayHello();
    System.out.println(message);
} catch (Exception e) {
    System.out.println("HelloClient exception: " +e);
}
}
```

Generating Stubs and Skeletons

Now that we have the remote interface and the implementation, we can generate the stubs and skeletons (or only stubs in Java 1.2) with the `rmic` tool *after* we have compiled the classes. Remember to set the directory that you are working from in your classpath; you can then use the following line in a command window:

```
rmic -v1.2 HelloServer
```

Note that the `-v1.2` flag suppresses skeleton generation

Registering the Object

Now that we have the interface and the implementation, we need to make this object available to clients by binding it to a registry. This will allow clients to look the object up on the host by a `String` name. The stubs and skeletons (if any) are needed for registration. After all, it is the object stub that is going to be passed around from the registry to clients.

It is often mistakenly believed that the object must be bound to a registry for it to be used. In fact, this is not true. The object is available for use the moment it is successfully exported. It could behave as a client, invoke a method on another object, and pass itself to that object.

The following code shows how we register our object:

```
import java.rmi.*;
public class RegisterIt {
    public static void main(String args[]) {
        try {
            // Instantiate the object
            HelloServer obj = new HelloServer();
            System.out.println("Object instantiated: " + obj);
            Naming.rebind ("/HelloServer", obj);
        }
    }
}
```

Copyrighted image

There are two methods in the `java.rmi.Naming` class that can bind an object in the registry:

- The `bind()` method binds an object to a string name and throws a `java.rmi.AlreadyBoundException` if the binding already exists
- The `rebind()` method, as used above, replaces any pre-existing binding with the new one

The registry must be running for the object to bind. It can be started by the tool, or programmatically as explained before. We start the executable with the following command line:

```
rmiregistry -J-Djava.security.policy=registerit.policy
```

The security policy is needed because of the Java 2 security model. The registry needs permissions to open sockets (a socket is an endpoint to a TCP connection), which are restricted to standard extensions in the default policy file.

Our policy file for this example, `registerit.policy`, grants all permissions:

Copyrighted image

Of course, we wouldn't use this policy in a production environment. By default there is a single system-wide policy file and a single user policy file. The system policy file is by default located at `%JAVA_HOME%\lib\security\java.policy` (use forward-slash with Solaris), while the user policy file is by default found at `%USER_HOME%\.java.policy`. Look at the tool documentation with your JDK for details about policies if you're not familiar with them. Also, it is worth keeping in mind that while starting the registry all classes and stubs must be available in the classpath, or the classpath should not be set at all, to support dynamic loading.

Running the Client and Server

To run the client we should first compile and run the `RegisterIt.java` file. Then, we need to open yet another command window, so that we have a window running the RMI registry, one running the `RegisterIt` program, and the third to run the `HelloClient` class specifying our security policy file:

```
java      security.policy=registerit.policy HelloClient
```

The result of simple example should look something like this:

Copyrighted image

The RMISecurityManager

The `java.rmi.RMISecurityManager` extends the `java.lang.SecurityManager` class and provides the security context under which RMI applications execute. If no security manager has been set, stubs and classes can only be loaded from the local classpath and not from the host or code base (see the section later on *Dynamically Loading Classes*). This protects applications from downloading unsecure code via remote method invocations.

In JDK 1.3 there really is no need to subclass `RMISecurityManager` due to the policy-based access control. Furthermore, remember that the security manager in JDK 1.3 calls the `AccessController.checkPermission(Permission)` by default and refers to a policy file for permission checking.

There really is no reason to set the security manager to `RMISecurityManager` if an RMI program has a purely server role on all its communication links. `RMISecurityManager` (and user-defined security managers obtained by extending `RMISecurityManager`) are for subjecting the classes that are dynamically loaded by a client application to security control. If the client has access to the object definitions for the host there is no reason to use `RMISecurityManager` on the client side either.

Parameter Passing in RMI

The normal semantics for methods in a single JVM are governed by two rules:

Copyrighted image

When a primitive data type is passed as a parameter to a method the JVM simply copies the value and passes the copy to (or returns it from) the method. An object, on the other hand, resides in heap memory and is accessed by one (or more) references. When passed to a method, a copy of the reference variable is made (increasing the reference count to the object by one) and placed on the stack, and the copy is passed around.

Inside the method, code uses the copy of the reference to access the object. Invoking any method on the reference changes the state of the original object. Altering the reference itself does not affect the original object. However, altering the object that the reference points to alters the object that was initially created.

So when remote method invocation involves passing parameters or accepting a return value, what semantics are used? The answer depends on whether the parameters are primitive data types, objects, or remote objects. Let's look at these parameter types in more depth.

Primitive Parameters

When a primitive data type is passed as a parameter to, or returned from, a remote method, the RMI system passes it by value. A copy of the primitive data type is sent to the remote method and the method returns a copy of the primitive from its JVM. These values are passed between JVMs in a standard, machine-independent format allowing JVMs running on different platforms to communicate with each other reliably.

Object

| | |
|--|--|
| A reference heap and It is actually returns an | an object doesn't make sense across multiple JVMs since the reference points to a value in the JVMs do not share heap memory. RMI sends the object itself, not its reference, between JVMs. object that is passed by value, not the reference to the object. Similarly, when a remote method a copy of the whole object is returned to the calling program. |
| A Java RMI must transform essentially memory | can be simple, or it could refer to other Java objects in a complex graph-like structure. Since the referenced object and all objects it references, it uses RMI Object Serialization to object into a linear format that can then be sent over the network. Object serialization an object and any objects it references. Serialized objects can be de-serialized in the remote JVM and made ready for use by a Java program. |
| Passing arguments implement | object graphs can use a lot of CPU time and network bandwidth. Try keeping object and results from, remote methods simple for optimization. Objects being passed around must <code>java.io.Serializable</code> or the <code>java.io.Externalizable</code> interface. |

Remote

eters

| | |
|-------------------------------------|--|
| Passing A client returned to remote | objects as method arguments or return types is a little different from passing other objects. can obtain a reference to a remote object through the RMI Registry program or it can be client from a method call (see the <code>HelloWorld</code> example in the next section). Passing is very important, especially for remote callbacks. |
|-------------------------------------|--|

Consider code:

Copyrighted image

What when the `MsgInterface` itself is a remote object that has been exported on the server?

RMI does and sends it argument, though it the return a copy of the remote object. It substitutes the stub for the remote object, serializes it, the client. Just as a note, if the client sends this remote object back to the server as another object is still treated as a remote object on the server and not local to the server (even though this may seem like a performance overhead, it is crucial to preserve the integrity of the

Consider case: what happens when the remote method returns a reference to this?

In the server code, `this` refers to the actual server implementation living in the server's JVM. However, clients have no direct contact with the other JVM. They deal with the proxy of the server's object, the stub. Behind the scenes, RMI always checks the input and output parameters from a remote method to see if they implement the `Remote` interface. If they do, they are transparently replaced with the corresponding stub. This gives clients the illusion that they are working with the local objects because even things like `this` can be exchanged between different JVMs.

Copyrighted image

The Distributed Garbage Collector

One of the design objectives for the RMI specifications was to keep the client's perspective of remote objects the same as other objects within its own JVM. This implies that remote objects should also be subjected to garbage collection.

The RMI system provides a reference-counting distributed garbage collection algorithm based on Modula-3's Network Objects. Internally, the server keeps track of which clients have requested access to the remote object. When a reference is made, the server marks the object as dirty and when all clients have dropped the reference, it is marked as being clean. A clean object is marked for garbage collection and reclaimed when the garbage collector runs.

In addition to the reference-counting mechanism on the server, when a client obtains a reference, it actually has a lease to the object for a specified time. If the client does not refresh the connection by making additional dirty calls to the remote object before the lease term expires, the distributed garbage collector then assumes that the remote object is no longer referenced by that client (the reference is considered to be dead) and the remote object may be garbage collected.

A remote object can implement the `java.rmi.server.Unreferenced` interface. This has one method, `unreferenced()`, which is invoked by the RMI runtime when there are no longer any clients holding a live reference. This enables the distributed garbage collector (DGC) to check whether any remote references are still in use.

**The lease time is controlled by the system property `java.rmi.dgc.leaseValue`.
(Its value is in milliseconds and defaults to 10 minutes.)**

Of course all this dirty, clean, and leasing is never visible to users or clients. The DGC mechanism is completely transparent. The DGC mechanism is hidden in the stubs-skeleton layer and is abstracted in the `java.rmi.dgc` package.

It is important to remember that a remote object can be garbage collected, leaving it unavailable to clients (which typically results in a `java.rmi.ConnectException`). Due to these garbage collection semantics, a client must be prepared to deal with remote objects that have "disappeared". On the server, if you don't want your object to "disappear" you should always hold an explicit reference so that it is not garbage collected.

Remember, that the registry (itself a remote object) acts as a client to the server object and hence holds a lease to the object. So even if all the "actual" clients get disconnected from a server, the method `unreferenced()` may not be invoked on a server object while the registry holds a lease.

Consider the following modified version of the same `HelloWorld` example that demonstrates how distributed garbage collection works and how the `unreferenced()` method can be used. We discussed how a client could get a reference to a remote object as a result of a method invocation earlier in the section on parameter passing. This example uses that concept.

We modify the interface to return a remote object instead of a string:

```
import java.rmi.*;  
  
public interface HelloInterface extends java.rmi.Remote {  
    public MsgInterface getMsg() throws RemoteException, Exception;  
}
```

The remote object implementation of this interface is also quite simple:

```
import java.io.*;  
import java.rmi.*;  
import java.rmi.server.*;  
import java.util.Date;  
  
public class HelloServer extends UnicastRemoteObject  
    implements HelloInterface {  
  
    public HelloServer() throws RemoteException {  
        super();  
    }  
  
    public MsgInterface getMsg() throws RemoteException, Exception {  
        return (MsgInterface)new MsgServer();  
    }  
}
```

The `MsgInterface` has no methods and is just used to mark the object being passed around as remote:

Copyrighted Image

The simple implementation of this interface is designed to trap the events of the object by printing out information when the object is created, no longer referenced, finalized, and then deleted. This object also implements the `Unreferenced` interface and implements the `unreferenced()` method. The `unreferenced()` method will be called when there are no client references to the object; `finalize()` is called just before the object is garbage collected:

Copyrighted Image

```
public class MsgServer extends UnicastRemoteObject
    implements MsgInterface, Serializable, Unreferenced {

    // Set a counter for the number of instances of this class
    // that are created
    private static int counter;

    // Hold an id for the object instance
    private int id;

    public MsgServer() throws RemoteException {
        super();
        System.out.println("Created Msg:" + counter);
        counter++;
        setId(counter);
    }

    public void finalize() throws Throwable {
        super.finalize();
        System.out.println("Finalizer called for Msg: " + id);
    }

    public void unreferenced(){
        System.out.println("The unreferenced()method called for Msg: " + id);

        // If we need we can call unexportObject here since no one is using it
        // unexportObject(this, true);
    }

    private void setId(int id){
        this.id=id;
    }
}
```

The registration program remains the same as in the previous examples:

```
import java.rmi.*;

public class RegisterIt {

    public static void main(String args[]) {
        try {
            HelloServer obj = new HelloServer();
            Naming.rebind("/HelloServer", obj);
            System.out.println("HelloServer bound in registry");
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

We modify the client slightly to create multiple instances of the remote object `MsgServer()` on the server (we explicitly instantiate an object in the `getMsg()` method). We need to do this for our demonstration because the JVM runs the distributed garbage collector (like the usual garbage collector), only when it "feels like" there is a need to reclaim memory:

```
import java.rmi.*;

public class HelloClient {

    public static void main(String args[]) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }

        try {
            HelloInterface obj = (HelloInterface) Naming.lookup("/HelloServer");
            for(int i = 0; i < 100; i++) {
                MsgInterface msg = obj.getMsg();
            }
        } catch (Exception e) {
            System.out.println("HelloClient exception: " + e);
        }
    }
}
```

Start the server after compiling the classes and generating the stubs (and skeletons if necessary) for both **HelloServer** and **MsgServer** with the following (changing the codebase path as appropriate, see the next section for more on codebase):

```
javac *.java
rmic -v1.2 HelloServer
rmic -v1.2 MsgServer
rmiregistry -J-Djava.security.policy=registerit.policy
java -Djava.rmi.dgc.leaseValue=1000 -Djava.security.policy=registerit.policy
RegisterIt
```

Again you might want to experiment with the lease value property as well as the **-ms** and **-mx** options to set the heap memory for this example. Here's a snippet from the output when **HelloClient** is executed with the default values:

```
java -Djava.security.policy=registerit.policy HelloClient
```

Copyrighted image

The `leaseCheckInterval` in `sun.rmi.transport.DGCImp` is read in from the `sun.rmi.dgc.checkInterval` property without taking into account the current value of the `leaseValue` variable. This causes a five-minute delay in removing unused objects, even if the `leaseValue` is set to a lower value. To work around this problem set the `sun.rmi.dgc.checkInterval` property to half of the setting of `java.rmi.dgc.leaseValue`.

Dynamically Loading Classes

We talked earlier about how references, stubs, parameters, and socket factories are sent to the client over the wire and how the class definitions are not. Dynamic class loading addresses the latter, making definitions available.

Dynamic class loading has been around for a long time in Java and this ability to dynamically load and instantiate classes is a very powerful concept. Applets, for example, are downloaded to the client browser from a web server and are executed in the client's JVM. This gives the client's runtime the ability to access applications that have never been installed in their system.

It is assumed that the reader is familiar with how applets work and how the `codebase` property is used for them. To summarize, `codebase` is the location from which the class loader loads classes into the JVM. This means that classes can be deployed in a central place, such as a web server, for a distributed system and all applications in the system can download the class files to operate.

There are two important system properties in RMI:

- ❑ `java.rmi.server.codebase`
This specifies the URL (a `file://`, `ftp://`, or `http://` location) from where the classes can be accessed. If an object is passed around as a method argument or return type the client JVM needs to load the class files for that object. When RMI serializes the object it inserts the URL specified by this property alongside the object.
- ❑ `java.rmi.server.useCodebaseOnly`
This property is used to notify the client that it should load classes only from the codebase location.

Let us revisit the process of exporting and registering a remote object and put it under a magnifying glass:

- ❑ When instantiating the remote object and registering it with the registry (our `RegisterIt` program in the previous examples), the codebase is specified by the `java.rmi.server.codebase` property.
- ❑ When the `bind()` call is made, the registry uses this codebase to locate the stub for the object. Remember, the registry itself is a client to the object. Once this is successful the registry binds the object to a name and the codebase is saved along with the reference to the remote object in the registry.
- ❑ When a client requests a reference to the remote object, the registry returns the stub to the client. The client looks for the class definition of the stub in its local classpath (the classpath is always searched before the codebase), then the client loads the local class. If the stub class definition is not there in the classpath, the client will attempt to retrieve the class definition from the remote object's codebase which was stored in the registry.
- ❑ The class definition for the stub and any other classes that it needs, such as socket factories, are downloaded to the client JVM from the `http` or `ftp` codebase.

- When all the class definitions are available, the stub's proxy method calls to the object on the server.

Using the two properties and the downloading mechanism discussed, five potential configurations can be set up to distribute classes:

- **Closed**
There is no dynamic loading and all classes are located in the respective JVMs and loaded from the local classpath (`java.rmi.server.codebase` not set).
- **Dynamic client-side**
On the client, some classes are loaded from the local classpath and others from the codebase specified by the server.
- **Dynamic server-side**
This is similar to the dynamic client-side configuration. Some classes on the server are loaded locally and others from the codebase specified by the client (for example, in the case of callbacks from the server to client).
- **Bootstrapped client**
All of the client code is loaded from the codebase via a small program on the client ("bootstrap" loader).
- **Bootstrapped server**
Same as above, only on the server-side. The server uses a small program (bootstrap loader).

There are two classes, the `java.rmi.RMISecurityManager` and `java.rmi.server.RMIClassLoader` that check the security context before loading the class. The RMI system will only download classes from remote locations if a security manager has been set. The `RMIClassLoader` has one important method:

```
public static Class loadClass(String codebase, String name)
```

This method loads the class from the specified codebase. Other overloaded forms of this method take a URL for a codebase or a string list of multiple URL's

Let's take the `HelloWorld` example and set it up in the bootstrapped client and bootstrapped server configuration. This configuration is usually the most popular one due to its flexibility. We will download:

- All the classes on the client including the client class itself to the client from the web server
- All the server classes including the server class itself from a central web server

We will only need one simple bootstrap class for the client and one for the server.

First, let's tweak the `HelloClient` class a little so that it connects to the server when instantiated:

```
import java.rmi.*;  
  
public class HelloClient {
```

Copyrighted image

```
        System.out.println(message);
    } catch (Exception e) {
        System.out.println("HelloClient exception: " + e);
    }
}
```

That's it. We leave the remote implementation (`HelloServer.java`) and interface (`HelloInterface.java`) the same as our initial example.

Now let's write a bootstrap class (instead of the `RegisterIt.java` file we used earlier) that starts the server. It accesses the `codebase` property and loads the server class from the codebase using `RMIClassLoader`:

```
import java.rmi.Naming;
import java.rmi.Remote;
import java.rmi.RMISecurityManager;
import java.rmi.server.RMIClassLoader;
import java.util.Properties;

public class DynamicServer {

    public static void main(String args[]) {

        // Create and install a security manager
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }

        try {
            Properties p = System.getProperties();
            String url = p.getProperty("java.rmi.server.codebase");
            Class serverclass = RMIClassLoader.loadClass(url, "HelloServer");
            Naming.rebind("/HelloServer", (Remote)serverclass.newInstance());
            System.out.println("HelloServer bound in registry");
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

When the downloaded class is instantiated with a `newInstance()`, the constructor for the class is invoked (our `HelloServer()` constructor) and the object is exported in the constructor and registered with the registry. Steps 1 to 5 that we mentioned earlier in the chapter (*Developing Applications with RMI*), occur in conjunction between the server and the registry.

There is a similar bootstrap program for the client that accesses the `codebase` property and loads the client class. When the downloaded client class is instantiated, it connects to the server and looks up the object. Again, steps 1 to 5 occur between the client and the registry:

```
import java.rmi.RMISecurityManager;
import java.rmi.server.RMIClassLoader;
import java.util.Properties;
```

Copyrighted image

Compile all the classes and generate the stubs. Now we need to place the `HelloWorld` class files in a web server so that the `DynamicServer` can download them at runtime. For this example, we will use a small test HTTP server from Sun called `ClassFileServer` (also included in the source code).

Place the four class files for the `HelloWorld` example in a directory (`\public_html`) under the HTTP server.

First start the RMI registry using:

```
rmiregistry -J-Djava.security.policy=registerit.policy
```

then start the HTTP server. For `ClassFileServer` it would be something like this (assuming `ClassFileServer` is in the classpath):

```
java ClassFileServer 8080 %BOOK_HOME%\Ch03\Dynamic\webserver\public_html
```

The first parameter in this command line specifies the port that the server runs on, and the second parameter is the source for the files to be served. It will probably be a little quicker to set up a batch file to run these commands.

We then run the `DynamicServer` specifying the `codebase` as the web server:

```
java -Djava.security.policy=registerit.policy  
-Djava.rmi.server.codebase=http://localhost:8080/ DynamicServer
```

Finally run the `DynamicClient`, again providing the `codebase`:

```
java -Djava.security.policy=registerit.policy  
-Djava.rmi.server.codebase=http://localhost:8080/ DynamicClient
```

To summarize:

- When we start the **DynamicServer** it accesses the codebase, downloads the implementation class, and exports the object
- The registry uses the codebase to download the stub and bind the object, keeping track of the codebase used
- When you start the **DynamicClient**, it again contacts the codebase and downloads the client class and the stub, and invokes a method on the server object

To summarize the steps to execute the example:

Note: The downloadable source contains sample batch files to execute the code.

- Start the **rmiregistry**:

```
rmiregistry -J-Djava.security.policy=registerit.policy
```

- Start the class file server. Make sure the compiled classes from the first **HelloWorld** example as well as the stubs are available in the **public_html** folder:

```
java ClassFileServer 8080 %BOOK_HOME%\Ch03\Dynamic\webserver\public_html
```

- Start the **DynamicServer**:

```
java -Djava.security.policy=registerit.policy  
      -Djava.rmi.server.codebase=http://localhost:8080/ DynamicServer
```

- Start the **DynamicClient**:

```
java -Djava.security.policy=registerit.policy  
      -Djava.rmi.server.codebase=http://localhost:8080/ DynamicClient
```

Notice that all the **HelloWorld** client, server, and stub files are in one location. The only distribution for the server is one class and the only file needed for client distribution is one class:

Copyrighted image

Remote Callbacks

We discussed earlier how a client can get a reference to a remote object as a result of a method invocation. In fact, we actually saw how a remote object can be dynamically passed around in the DGC example earlier. Remember that we had initially emphasized that a client and server are terms used to describe a role, not physical locations or architectures. A client also can be a remote object. In many situations, a server may need to make a remote call to a client, for example, progress feedback or administrative notifications. A good example would be a chat application where all clients are remote objects.

Copyrighted image

There is nothing really special about peer-to-peer communication or callbacks between remote objects. All that happens is a remote reference is passed around and methods are invoked on that reference.

Let us take the same `HelloWorld` example we looked at first and modify it to demonstrate callbacks.

The remote interface, `HelloInterface`, has one method that takes a `ClientInterface` as an argument:

Copyrighted image

For the client, the `ClientInterface` is also very simple, with only one method:

Copyrighted image

The `HelloServer` implements the `HelloInterface` and invokes the `popup()` method on the `ClientInterface` as shown below:

```
import java.io.*;
import java.rmi.*;
import java.rmi.server.*;
import java.util.Date;

public class HelloServer extends UnicastRemoteObject
    implements HelloInterface {

    public HelloServer() throws RemoteException {
        super();
    }

    public String sayHello(ClientInterface ca) throws RemoteException {
        ca.popup("This is a message from the server!");
        return "Hello World, the current system time is " + new Date();
    }
}
```

The client that invokes methods on the server needs to be a remote object itself. Let us write a simple applet to demonstrate this. The applet exports itself and starts listening for incoming calls by explicitly invoking the `UnicastRemoteObject.exportObject(this)` method (an overloaded form of this method allows you to specify the port as well):

```
import java.applet.*;
import java.awt.*;
import java.io.Serializable;
import java.rmi.*;
import java.rmi.server.*;

public class CallbackApplet extends Applet
    implements ClientInterface, Serializable {

    String message = "-n/a-";
    Frame f = new Frame();
    Label l1 = new Label("                ");

    public void init() {
        f.add(l1);
        try {
            // Export the object
            UnicastRemoteObject.exportObject (this);
            String host = "rmi://" + getCodeBase().getHost() + "/HelloServer";
            HelloInterface obj = (HelloInterface) Naming.lookup(host);
            message = obj.sayHello((ClientInterface) this);
        } catch (Exception e) {
            System.out.println("HelloApplet exception: " + e);
        }
    }

    // Display the message in the applet
    public void paint(Graphics g) {
        g.drawString(message, 25, 50);
    }

    // Implement the interface
    public void popup(String txt) throws RemoteException{
        l1.setText(txt);
        f.setSize(100,100);
        f.show();
    }
}
```

When the applet is loaded it locates the server object and invokes the remote method. It passes itself as an argument to that method. As a result of this, the stub is transported to the server and the server can now reverse roles. It acts as a client to this applet and can invoke the `popup()` method.

We need a simple HTML page for this applet (`Applet.html`):

```
<html>
<applet codebase="http://localhost:8080/Callbacks" code="CallbackApplet.class"
        width=300 height=266>
</applet>
</html>
```

It's as simple as that! We generate the stubs for the client and server by executing `rmic` on `CallbackApplet` and `HelloServer`. Start the registry, and register the `HelloServer` with it using the same `RegisterIt` class as earlier.

The applet
the host
preceding
class file
classpath

be loaded from a web server on the host (remember unsigned applets can connect back to you can use either your own web server or the sample `ClassFileServer` used in the All the classes needed by the client should be accessible on the web server. The stub of the `CallbackApplet` should also either be available on the server objects should be dynamically loaded as we have seen earlier.

You can test the application using  the preferences in Applet Viewer as

Write file with the following code (`AppletViewer.html`):

Run the after starting the registry and the server from the same directory as the HTML file
and class

`-J-Djava.security.policy=registerit.policy AppletViewer.html`

Finally, on this example, we get something that looks like this:


Hello World, the current system time is Fri Aug 17 19:57:11 BST 2001

is a message from the server!

Object Activation

The remote objects discussed so far, are instances of a `java.rmi.UnicastRemoteObject` class, and have one main feature. They are accessible all the time, even when there are no clients executing. Consider the scenario when the number of remote objects, or the resources used by them on a server is high. This was identified as a major performance bottleneck in Java 2 so the concept of object activation was introduced.

Copyrighted image

So, what is the difference between an activatable object and the usual remote object from a client's perspective? None! To the client the entire activation mechanism is transparent and the client is never aware of what is happening behind the scenes.

The references to the remote object themselves can be thought of as lazy or faulting references. References to activatable objects contain persistent handle information that allows the activation subsystem to know that the object should be started if it is not already running. After an activatable reference is used the first time, the remote reference layer switches to a regular remote reference so that it doesn't have to go through the activation system subsequently.

Since the activation system can switch the lazy reference to a live remote reference, references to an activatable object are always available. However, references to a remote object don't survive a crash, or a restart.

To understand the actual semantics of using the activation model, let's take a moment to familiarize ourselves with a few useful terms:

□ **Activator**

The activator is a major component on the server. It facilitates remote object activation by keeping track of all the information needed to activate an object and is responsible for starting instances of JVMs on the server if needed.

□ **Activation Group**

An activation group creates instances of objects in its group, and informs its monitor about the various active and passive states. The closest analogy to an activation group is a thread group. An activation group is essentially a complete, separate instance of the JVM that exists solely to host groups of activated objects. This fresh JVM is started as needed by the activator. There can be multiple activation groups.

□ **Activation Monitor**

Every activation group has an activation monitor that keeps track of an object's state in the group and the group's state as a whole. The activation monitor is created when the group is made active.

□ **Activation System**

The activation system provides a means for registering groups and activatable objects to be activated within those groups. This works closely with the activator, which activates objects registered via the activation system, and the activation monitor, and obtains information about active and inactive objects, and inactive groups.

The activation model and its associated implementations is summarized in the following table:

| Entity | Implementation | Implemented as |
|--------------------|--|---|
| Activator | <code>java.rmi.activation.Activator</code> | Interface (notice that the activator is a remote object itself) |
| Activation Group | <code>java.rmi.activation.ActivationGroup</code> | Abstract class |
| Activation Monitor | <code>java.rmi.activation.ActivationMonitor</code> | Interface |
| Activation System | <code>java.rmi.activation.ActivationSystem</code> | Interface |

The activation mechanism uses identifiers and descriptors. If this seems overwhelming at first , it is worth keeping these important points in mind:

- Every activatable object has an ID and a descriptor.
- Every activatable object belongs to an activation group. The group itself has an ID and a descriptor.

The Activation Group

An activation group, as mentioned before, is used to maintain a group of activatable objects. The activation group is associated with a group identifier (`java.rmi.activation.ActivationGroupID`) and a group descriptor (`java.rmi.activation.ActivationGroupDesc`) that identify and describe the activation group respectively.

An activation group is created explicitly as a result of invoking the `ActivationGroup.createGroup()` method:

```
public static ActivationGroup createGroup(ActivationGroupID id,
                                         ActivationGroupDesc desc,
                                         long incarnation)
```

Where:

- `id` is the activation group identifier
- `desc` is the activation group's descriptor
- `incarnation` is the activation group's incarnation number (zero on a group's initial creation)

The `ActivationGroupID`, besides identifying the group uniquely within the activation system, also contains a reference to the group's activation system. This allows the group to interact with the system as and when necessary. All objects with the same `ActivationGroupID` are activated in the same JVM.

An `ActivationGroupDesc` contains the information necessary to create or recreate the group in which to activate objects. It contains:

- ❑ The group's class name. Remember that `java.rmi.activation.ActivationGroup` is an abstract class and the activator (`rmid`) internally provides its concrete implementation (for example the class `sun.rmi.server.ActivationGroupImpl`).
- ❑ The location of the group's class.
- ❑ A marshaled object that can contain group specific initialization data.

`ActivationGroupDesc` contains an inner class `CommandEnvironment` that specifies the startup environment options for the `ActivationGroup` implementation classes. This allows exact control over the command options used to start the child JVM – a null `CommandEnvironment` refers to the `rmid` default values.

`ActivationGroupDesc` can be created using one of the two constructors specified below:

- ❑ Construct a group descriptor that uses system default for group implementation and code location:

```
ActivationGroupDesc(Properties overrides,  
                    ActivationGroupDesc.CommandEnvironment cmd)
```

- ❑ Specify an alternative group implementation and execution environment to be used for the group:

```
ActivationGroupDesc(String className, String location,  
                    MarshaledObject data, Properties overrides,  
                    ActivationGroupDesc.CommandEnvironment cmd)
```

We have taken a whirlwind tour of the activation groups here. Let us look at some code that summarizes the creation of activation groups:

Copyrighted image

ActivationID

Just as the `ActivationGroupID` is an ID for the group, the `ActivationID` is an ID for the object. Once the object is registered with the activation system, it is assigned an `ActivationID`. It contains two crucial pieces of information:

- ❑ A remote reference to the object's activator
- ❑ A unique identifier for the object

We will see how to register the object shortly.

Activation Descriptor

Every activatable object has a unique identifier containing a reference to the activator that the object is associated with (**ActivationGroupID**) and an ID that uniquely identifies the object (**ActivationID**). The activation descriptor contains all the information the system needs to activate an object:

- The activation group identifier for the object
- The name of the class being activated
- The location of the class for the object
- An optional marshaled object that contains initialization data

Remember all the descriptors and identifiers we talked about until now were associated with the group. Once the group is created we have an identifier for that group in the system. So all that is needed to recreate/activate the object at any time is the activation group identifier and the class detail.

The activation descriptor can be created by using one of four constructors:

- Constructs an activation descriptor for the object with the given **groupID** and **className** that can be loaded from the code location, with the optional initialization information data and restart specifics for if the object is activated on demand or restarted when the activator is restarted:

```
ActivationDesc(ActivationGroupID groupID, String className,
               String location, MarshaledObject data, boolean restart)
```

- Same as above but without the restart information:

```
ActivationDesc(ActivationGroupID groupID, String className,
               String location, MarshaledObject data)
```

- Again similar to the first, only that the activation group defaults to the current **ActivationGroupID** for the current instance of the JVM. It is worth noting that this constructor will throw an **ActivationException** if no current activation group has been explicitly created for this JVM:

```
ActivationDesc(String className, String location, MarshaledObject data,
               boolean restart)
```

- Simpler form of above, again without the restart information:

```
ActivationDesc(String className, String location, MarshaledObject data)
```

One of the useful things about activation is the restart flag in the constructors above. **rmid** remembers the remote objects that have registered with it and can recreate them when it (or the machine itself) restarts. This is possible because **rmid** keeps a log. By default this is in a log in the directory from which **rmid** was started and can be configured with the **-log** option. When **rmid** starts, it consults this log to gather information and restart any objects that were configured for restart.

Once the `ActivationDesc` has been created, it can be registered in one of the following ways:

- By invoking the static `Activatable.register(ActivationDesc desc)` method.
- By instantiating the object itself using the first or second constructor of the `Activatable` class (which take the `ActivationID` as a parameter). This registers and exports the object.
- By exporting the object explicitly via `Activatable`'s first or second `exportObject()` method that takes an `ActivationID`, the remote object implementation, and a port number as arguments. This registers and exports the object.

Behind the scenes, here is what happens:

- When a stub is generated for an activatable object, it contains special information about the object. This information includes the activation identifier and information about the remote reference type of the object.
- This stub for the object uses the activation identifier and calls the activator to activate the object associated with the identifier (the stub is the lazy/faulting reference).
- The activator locates the object's activation descriptor and activation group. If the activation group in which this object should be does not exist, the activator starts an instance of a JVM, creates an activation group, and then forwards the activation request to that group.
- The activation group loads the class for the object and instantiates the object (using special constructors that take several arguments, as we shall soon see).

When the object is activated, the activation group returns an object reference to the activator (this is a serialized or marshaled reference). The activator records the activation identifier and reference pairing and returns the live reference to the stub. The stub then forwards method invocations via this live reference directly to the remote object (the live reference is like any other remote reference).

Let us summarize everything we have covered until now by redoing the `HelloWorld` example we covered earlier to make it activatable.

Making Objects Activatable

As we have seen there are a few cooperating entities in the activation framework that make everything possible:

- The object itself.
- The wrapper program that registers the object. This is similar to the `RegiserIt` program we used earlier. It typically makes a few method calls to the activation system to provide details about how the object should be activated.
- The third entity is the activation daemon that records information like the registry, about when and what to do with the objects (this daemon is `rmid`).

Keeping these entities in mind, the steps to make an object activatable can be summarized as follows:

- The object should extend the `java.rmi.activation.Activatable` class instead of `UnicastRemoteObject` (there is an alternative to this as we shall see later).

- The object should include a special constructor that takes two arguments, its activation identifier of type `ActivationID`, and its optional activation data, a `java.rmi.MarshaledObject`. This is unlike non-activatable remote objects, which include a no argument constructor. This special constructor is called by the RMI system when it activates the object.
- An activation descriptor (`java.rmi.activation.ActivationDesc`) should be created and registered with the activator (`rmid`).

It is worth noting that the remote interface of the object does not need to be changed or modified in any way. This makes sense as we are only changing the implementation of the object's instance, not how the outside world sees the object.

Step 1: Create the Remote Interface

This is no different from what we had earlier:

```
import java.rmi.*;

public interface HelloInterface extends Remote {
    public String sayHello() throws RemoteException;
}
```

Step 2: Create the Object Implementation

This class extends `java.rmi.activation.Activatable` and implements the remote interface (since `Activatable` extends `RemoteObject`). Furthermore, it must contain a two-argument constructor that takes the `ActivationID` and `MarshaledObject` as arguments. This constructor should call the appropriate superclass constructors to ensure initialization:

```
import java.rmi.*;
import java.rmi.activation.*;
import java.util.Date;
```

Copyrighted image

```
public String sayHello() throws RemoteException{
    return "Hello World, the current system time is " + new Date();
}
}
```

Step 3: Register the Object with the System

This class contains all the information necessary to register the object, without actually creating an instance of the object:

```
import java.rmi.*;
import java.rmi.activation.*;
```

```
import java.util.Properties;

public class RegisterIt {

    public static void main(String args[]) throws Exception {
        try {
            //Install a SecurityManager
            System.setSecurityManager(new RMISecurityManager());

            // Create the group
            Properties env = new Properties();
            env.put("java.security.policy",
                    "file://$BOOK_HOME%/Ch03/Activatable/
                     activation/registerit.policy");
            ActivationGroupDesc mygroupdes = new ActivationGroupDesc(env, null);
            ActivationGroupID mygroupid = ActivationGroup.getSystem().
                registerGroup(mygroupdes);
            ActivationGroup.createGroup(mygroupid, mygroupdes, 0);

            // Create the details about the activatable object itself
            ActivationDesc objectdesc = new ActivationDesc("HelloServer",
                    "file://$BOOK_HOME%/Ch03/Activatable", null);

            // Register the activation descriptor with the activator
            HelloInterface myobject = (HelloInterface)Activatable
                .register(objectdesc);

            // Bind the stub to a name in the rmiregistry
            Naming.rebind("helloObject", myobject);

            // Exit
            System.out.println("Done");

        } catch(Exception e) {
            System.out.println("Exception "+ e);
        }
    }
}
```

We're nearly finished. Now we can use the old client that we wrote for our initial example and execute it with the same policy and startup arguments:

```
import java.rmi.*;

public class HelloClient {

    public static void main(String args[]) {

        if (args.length < 1) {
            System.out.println ("Usage: java HelloClient <host>");
            System.exit(1);
        } else {

            try {
                HelloInterface server = (HelloInterface)Naming.lookup(
```

Copyrighted image

Compile using the then generate the stubs and skeletons for the remote object. Start the Activation system utility and then start our program that registers our activatable objects:

```
rmid           security.policy=registerit.policy
```

Now we execute the RegisterIt file:

```
java           .security.policy=registerit.policy
               .rmi.server.codebase=file://%BOOK_HOME%/Ch03/Activatable/
               It
```

Finally, we use the client, as shown in the following output shot:

Copyrighted image

Note that should have three or four command-line windows open for this example:

- rmiregistry
- Rmid
- RegisterIt
- HelloClient

The last two could be done in one window.

Alternative to Extending the Activatable Class

Sometimes it may be not be possible to extend the `Activatable` class if the remote object needs to extend another class. For example, if we are writing an applet that we want to make activatable or a remote object, our applet would need to extend `java.applet.Applet` and in Java there is no such thing as multiple inheritance.

We saw earlier that it was possible to make an object remote by exporting the object directly using one of the export methods in the `java.rmi.UnicastRemoteObject` class rather than extending the `UnicastRemoteObject` class itself.

There is a similar alternative for activatable objects. While talking about activation descriptors we mentioned an `exportObject()` method. It is possible to register and export an object by invoking any of the export methods in the `java.activation.Activatable` class.

Let us rewrite our activatable `HelloServer` class to demonstrate this, keeping everything else the same:

```
import java.rmi.*;
import java.rmi.activation.*;
import java.util.Date;

public class HelloServer implements HelloInterface {

    public HelloServer(ActivationID id, MarshaledObject data)

    {
        // ...
    }

    public String sayHello() throws RemoteException{
        return "Hello World, the current system time is" + new Date();
    }
}
```

Copyrighted image

In the above example, two of the constructors shown take socket factories as arguments. We will cover socket factories in a while. For now, just keep in mind that you can pass socket factories as arguments to your activatable objects

Starting Multiple JVMs other than with rmid

`rmid` or the activator itself is a remote object and executes in a JVM. Every object that is activated is activated in its activation group's JVM but sometimes it may be desirable to spawn a separate JVM for each activatable object (for example, in a large application you would want to eliminate the single point of failure by isolating objects in different JVMs). This is straightforward if you keep in mind what was mentioned when talking about the `ActivationGroupID`, that is, "All objects with the same `ActivationGroupID` are activated in the same JVM".

To start multiple JVMs, the object must have a different `ActivationGroupID` or in other words must be in a separate activation group. The following code shows how the same `HelloServer` object is registered differently so that each object has a different `ActivationGroupID` (and hence a different activation group).

Here's the class that registers our activatable objects:

```

import java.rmi.*;
import java.rmi.activation.*;
import java.util.Properties;

public class RegisterIt {
    public static void main(String[] args) {
        try {
            //Install a SecurityManager
            System.setSecurityManager(new RMISecurityManager());

            // create another one for a new VM
            Properties env = new Properties();
            env.put("java.security.policy",
                    "file://$BOOK_HOME%/Ch03/Activatable/MultiVM/
registerit.policy");

            ActivationGroupDesc mygroupdes = new ActivationGroupDesc(env, null);
            ActivationGroupID mygroupid = ActivationGroup.getSystem()
                .registerGroup(mygroupdes);
            0 ;

            ace myobject = (HelloInterface)Activatable.register(
                objectdesc);
            Naming.rebind("helloObject", myobject);
        }
    }
}

```

Copyrighted image

```

        System.exit(0);

    } catch (Exception e) {
        System.out.println(e);
        e.printStackTrace();
    }
}
}

The client starts the new JVM on the server because of the lazy activation (activation the first time the method is invoked):

```

```

import java.rmi.*;

public class HelloClient {

```

```
public static void main(String args[]) {  
    if (args.length < 1) {  
        System.out.println ("Usage: java HelloClient <host>");  
        System.exit(1);  
    }  
    try {  
  
    } catch (Exception e) {  
        System.out.println("HelloClient exception: " + e.getMessage());  
        e.printStackTrace();  
    }  
}
```

Copyrighted image

If we run this example, and look at the processes running when we run the client, we'll see that two Java VMs are used:

Copyrighted image

Deactivation

Activation allows an object to be started on demand, however it does not automatically deactivate an object. The decision to deactivate the object is left to the object itself. An activatable object can deactivate itself by invoking the `Activatable.inactive(ActivationID id)` method.

It is important to know the difference between un-registering and deactivating an object. Deactivation is a temporary state, allowing the activation system to restart the object later, whereas un-registering the object by `Activatable.unregister(ActivationID id)` permanently removes it from the activation system.

Another useful method is the `Activatable.unexportObject(Remote obj, boolean force)` method that can be used to unexport the object. The `boolean` is used to force the object to be unexported even if there are pending or in-progress calls.

Copyrighted image

Next we'll turn our attention to the Secure Sockets Layer. This section will assume that the reader is familiar with TCP/IP sockets and networking in Java.

Custom Sockets and SSL

A socket is an endpoint for communication. Two collaborating sockets, one on the local machine and the other on the remote machine, form a connection. This distinction between sockets and connections is very important.

There are two types of sockets, **connection sockets** and **listening sockets**, also referred to as client and server sockets, respectively. A connection socket exists on each end of an open TCP connection and is abstracted by the `java.net.ServerSocket` and `java.net.Socket` classes. A listening socket is not associated with any TCP connection, but only exists as an abstraction to allow the TCP kernel to decide which incoming connections are accepted, and who gets the newly accepted connection socket.

At any time, RMI has a small number of listening sockets, one for each listened-to port (usually just one because RMI exports all objects on the "default" port, unless a specific port is specified in the `export()` method discussed earlier). RMI also creates connection sockets for outgoing connections and incoming connections.

The number of outgoing connections only depends on the number of concurrent outgoing calls. The simple rule is:

- If a thread wants to make a remote call, and all the connections to the endpoint are in use, then RMI opens a new connection to carry the call
- If a connection is free (meaning there's no call in progress using that connection), then RMI will reuse it for the next remote call

RMI spawns one thread to listen to each listening socket (also usually one). When RMI accepts a new connection, it creates a new thread – one thread handles the new connection, and the other goes back to accept a new connection. When the connection closes, its associated thread also exits.

The connection-handling threads spawned by RMI are not serialized in any way. If the calls arrive at the same time they will be run in concurrent threads. The calls are still allowed to synchronize on Java objects but RMI does not do such synchronization automatically (the remote object is responsible for its own synchronization either by synchronized methods or by synchronized locking).

A common point of confusion is that if a remote stub is returned by a remote call, the client can sometimes be seen to make two connections to the server. This happens because the distributed garbage-collection subsystem needs to make a `DGC.dirty()` call to notify the server that a new entity holds a reference to the remote object.

You can use `netstat` to monitor listening sockets. The following screenshot shows the sockets just after the registry is started on 1099. `HelloServer` is exported to 2000 and is binding with the registry. The left column lists the open sockets on the machine. The first line is for the registry, the second for the object. The third shows that a socket has been opened by the object to the registry and the last line shows the socket opened by the registry to the object.

Copyrighted image

A significant enhancement that was added to RMI from the Java 1.2 release was the ability to use custom socket factories based on the Factory design pattern. Instead of using the conventional sockets over TCP/IP, each object has the ability to use its own socket type. This allows the object to process data (rather than simply passing it), either before it is sent to, or after it has been received from, the socket.

In JDK 1.1.x, it was possible to create a custom `java.rmi.RMISocketFactory` subclass that produced a custom socket, other than the `java.net.Socket`, for use by the RMI transport layer. However, it was not possible for the installed socket factory to produce different types of sockets for different objects. For example in JDK 1.1, an RMI socket factory could not produce Secure Sockets Layer (SSL) sockets for one object and use the Java Remote Method Protocol (JRMP) directly over TCP for a different object in the same JVM. In addition, before 1.2, it was necessary for `rmiregistry` to use only your custom socket protocol.

The socket factory affected the whole system and not just a single object.

To see what this statement actually means, let us first see where in the RMI architecture sockets are involved, their types and purpose:

| Location | Server socket | Client sockets |
|--|---|---|
| RMI registry | It opens a server socket (default port 1099) and waits for requests from: | It uses a client socket to establish a connection to a server object right after they have registered. |
| | <ul style="list-style-type: none"> <li data-bbox="614 523 1325 611"><input type="checkbox"/> Servers to bind, re-bind and un-bind object implementations <li data-bbox="614 640 1297 679"><input type="checkbox"/> Clients who want to locate servers | |
| RMI servers (Remote implementation) | It opens a server socket on a user specified port via the <code>exportObject()</code> method or defaults to a random local port; it is used to accept connections from the client's (stub's) send responses. | A client socket is used while connecting to the registry, in particular to register the server implementation. |
| RMI Clients (objects invoking remote methods) | Not used | Client sockets are used for the communication with the RMI registry, to locate the server implementation and to make the actual RMI method call to the server |

Copyrighted image

When a client performs a "lookup" operation, a connection is made to the server socket on the `rmiregistry`. In general, a new connection may or may not be created for a remote call. The RMI transport layer caches connections for future use. If there are any existing connections and at least one of them is free, then it is reused; otherwise the current implementation creates additional sockets on demand. For example, if an existing socket is in use by an existing call, then a new socket is created for the new call. Usually there are at least two sockets open since the distributed garbage collector needs to make remote calls when remote objects are returned from the server. A client has no explicit control over the connections to a server, since connections are managed at the RMI transport-layer level. Connections will time out if they are unused for a period.

As you can see, there are a number of communication points and sockets being used in this three-way communication among the client, the registry, and the object. It is also important to note that the registry is only needed to locate an object by its name. Once the object is located there is direct communication between the object and the client.

An object can specify what factory classes are needed for anyone to communicate with its socket type through the `java.rmi.server.UnicastRemoteObject` constructor (or `java.rmi.activation.Activatable` for that matter). Instances of these factories are used to create instances of the desired sockets types:

```
protected UnicastRemoteObject(int port, RMIClientSocketFactory csf,  
                             RMIServerSocketFactory ssf)
```

As explained in the previous table, once the client has located an object, it uses a client socket to connect to the server socket that the object is listening on. The client socket factory must implement the `java.rmi.server.RMISocketFactory` interface. The client socket is created from this instance of the `RMIClientSocketFactory` by invoking the `createSocket()` method that returns an instance of the custom socket type to the specified server and port:

```
public Socket createSocket(String host, int port)
```

The server socket factory must implement the `java.rmi.server.RMIServerSocketFactory` interface and provide an implementation for the `createServerSocket()` method. The server socket is created (to start listening for incoming calls) by invoking the method implementation in the server socket factory class:

```
public ServerSocket createServerSocket(int port)
```

This is particularly useful when you want to encrypt the communication between the objects. For example, using SSL or TLS. Let us look at an example of how this can be done.

First we would need a couple of things, the foremost being a security provider that provides an implementation of the SSL protocol in Java. Many third party products available do this.

Copyrighted image

SSL specifications can be found at <http://home.netscape.com/eng/security/> and TLS specifications can be found in RFC 2246 at <http://www.ietf.org/rfc/rfc2246.txt>. Details about open source SSL implementations (SSLeay/OpenSSL) can be found at <http://www.openssl.org/>.

Sun provides the **Java Secure Sockets Extension (JSSE)** that implements a Java version of SSL and TLS protocols and includes functionality for data encryption, server authentication, message integrity, and optional client authentication.

The example discussed here uses JSSE. You would need to download it separately for J2SE (1.2 or 1.3) and it will be actually packaged with J2SE (1.4), see <http://java.sun.com/products/jsse/> for more information.

There is going to be a performance overhead and the communication is going to be very slow due to the complexity of the handshake and the encryption involved. How the provider implements the algorithms would also affect performance. For example, a native implementation such the GoNative Provider (<http://www.rfm.com/puretls/gonative.html>) is faster than its pure Java counterpart.

Let us first try to see what part of the communication we want/need to encrypt. The naming registry is implemented as a standard RMI service so attempts to register and lookup services will involve network connections being established. If you need the communication with the registry to take place over secure sockets then obviously the naming registry would need to listen on secure server sockets.

A straightforward way to do this is to have your server start its own naming registry. This registry will then benefit from the server's SSL support. The server would need to invoke and set a socket factory explicitly and start the registry:

```
RMISocketFactory.setSocketFactory(some vendor provided factory);
LocateRegistry.createRegistry(someport);
```

The SSL socket factory can then be installed on the client before looking up the object and everything will be SSL-secured.

Copyrighted image

In most cases it is only the communication between the objects that is important and needs encryption, not the communication between the client and the server. This is what we will look at in the following example. In order for communication to occur over SSL or TLS we need two things: keys and certificates. To see more information on these, see the links to the [SSL documentation](#) and [TLS documentation](#).

Before we do anything, we generate the keys and certificates using the JDK utility `keytool`. This is a certificate management tool used to manage public/private key pairs and associated certificates to be used in self-authentication and data integrity and authentication services, using digital signatures. In a commercial environment you would probably use `keytool` to generate a certificate request and use that request to obtain a certificate from a Certificate Authority (CA) like Verisign (<http://www.verisign.com>). For our purposes, we will generate a self-signed certificate (a certificate from the CA authenticating its own public key). In other words, we will use our own keys and certificates using the `-genkey` option in `keytool`.

Run the `keytool` utility with parameters something like this:

```
keytool -genkey -dname "cn=ProJavaServer, ou=Java, o=Apress, c=US" -alias apress
-keypass secret -storepass secret -validity 365 -keystore
%BOOK_HOME%\Ch03\SSL\.keystore
```

We can check the certificate by using `keytool` in a different way:

```
keytool -list -keystore %BOOK_HOME%\Ch03\SSL\.keystore -storepass secret
```

Copyrighted image

- This generates a keystore called "apress" that contains the keys and the self-signed certificate and has the password "secret".

Please refer to the tool documentation packaged with JDK for details about how to use keytool, keys, and certificates in the Java 2 security architecture.

Finally, we also export the generated certificate into a file `clientimport.cer` for use by clients using `keytool`:

```
keytool -export -keystore %BOOK_HOME%\Ch03\SSL\.keystore -storepass secret -file clientimport.cer -alias apress
```

Remember that we are not using browser-based SSL anywhere in these examples. If you want to do that then you need to generate certificates with the RSA algorithm (use the `-keyalg` option in `keytool`). For this you would need to install a provider that offers the RSA algorithm implementation since the JDK 1.2 `keytool` does not support it. For our purposes, the JSSE provider contains an implementation and RSA support is in-built in JDK 1.3. Also, the freeware JCE implementation available from <http://www.openjce.org> is a good security API provider.

Let us now revisit and rewrite the `HelloWorld` example we used earlier, to secure the object-to-object communication.

Again, the remote interface remains unchanged:

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface HelloInterface extends Remote {
    public String sayHello() throws RemoteException;
}
```

We now rewrite the remote implementation of this interface:

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.Date;

public class HelloServer extends UnicastRemoteObject
    implements HelloInterface {
    public HelloServer() throws RemoteException {
        super(0, new MyClientSocketFactory(), new MyServerSocketFactory());
    }
}
```

```
public String sayHello() {  
    return "Hello World, the current system time is " + new Date();  
}  
}
```

Notice that the only difference is that we pass our own custom factories that will be used for this object instead of the default factories.

We now use the same program we used earlier to register this remote object:

```
import java.rmi.*;  
  
public class RegisterIt {  
  
    public static void main(String args[]) {  
        try {  
            // Instantiate the object  
            HelloServer obj = new HelloServer();  
            System.out.println("Object instantiated" .+obj);  
            Naming.rebind ("/HelloServer", obj);  
            System.out.println("HelloServer bound in registry");  
        } catch (Exception e) {  
            System.out.println(e);  
        }  
    }  
}
```

The pieces we passed so easily are the socket factories. Let us look at them in detail now. The **MyServerSocketFactory** class is used to create instances of the server socket on the port that the object was exported on. This class must implement the **java.rmi.server.RMIServerSocketFactory** interface and should be serializable (to facilitate possible transportation over the network). The secure interchange between the client and the server involves an exchange of keys and certificates and the socket created must first initialize communication with the client before exchanging any data. The JSSE implementation allows us to do this in a few lines of code. We open the keystore (that we generated) from the file using its password, initialize the SSL/TLS implementation, and return an instance of a secure socket:

Copyrighted image

```
char[] passphrase = "secret".toCharArray();

// Get a context for the protocol. We can use SSL or TLS as needed.
SSLContext ctx = SSLContext.getInstance("TLS");
KeyManagerFactory kmf = KeyManagerFactory.getInstance("SunX509");

// Open the keystore with the password
// and initialize the SSL context with this keystore.
KeyStore ks = KeyStore.getInstance("JKS");
ks.load(new FileInputStream ("keystore"), passphrase);
kmf.init(ks, passphrase);
ctx.init(kmf.getKeyManagers(), null, null);
ssf = ctx.getServerSocketFactory();
} catch (Exception e) {
    e.printStackTrace();
}
return ssf.createServerSocket(port);
}
```

The client socket factory is used to create instances of the client sockets that connect to the instances of the server sockets generated from the above factory. It is assumed that the JSSE provider is also installed on the client machine:

```
import java.io.*;
import java.net.*;
import java.rmi.server.*;
import javax.net.ssl.*;

public class MyClientSocketFactory implements RMIClientSocketFactory,
    Serializable {

    public Socket createSocket(String host, int port) throws IOException {

        // We get the default SSL socket factory.
        SSLSocketFactory factory =
            (SSLSocketFactory)SSLSocketFactory.getDefault();
        SSLSocket socket = (SSLSocket)factory.createSocket(host, port);
        return socket;
    }
}
```

The client remains unchanged:

```
import java.rmi.Naming;
import java.rmi.RemoteException;

public class HelloClient {

    public static void main(String args[]) {
        if (args.length < 1) {
            System.out.println("Usage: java HelloClient <host>");
            System.exit(1);
    }
}
```

```

try{
    HelloInterface obj =(HelloInterface)Naming.lookup(
        "rmi://" + args[0] + "/HelloServer");
    System.out.println("Got a remote reference" + obj);
    System.out.println(obj.sayHello());
} catch (Exception e) {
    System.out.println(e);
}
}
}
}

```

The `ssl.` file we use here is actually the same as that used in the previous examples:

Copyrighted image

We can now compile the classes and generate the stubs for the `HelloServer` using the `rmic` packaged with the JDK.

Two small configuration details need to be examined before executing the example. In order for the client to interact and exchange keys and certificates with the server, it must trust the provider of the digital certificate. In other words, since our certificate is self-generated the client VM should:

- Explicitly import the certificate into the keystore it is using and mark it as a trusted certificate
- Use the same keystore that we use on the server

For the second option, the JSSE provider must be installed on the client machine (if it is different from the server) in one of two possible ways:

- Statically** in the `java.security` file
- Dynamically** using `Security.addProvider(new com.sun.net.ssl.internal.ssl.Provider());`

For more specific details regarding these installation methods, the reader is referred to the JSSE documentation.

The client can import the certificate from the file we generated earlier, `clientimport.cer`, into the keystore it is using and mark it as trusted using `keytool` with the `-import` statement:

```
keytool -import -file clientimport.cer -alias apress
```

Copyrighted image

Once this is done we first start the registry on the server (execute rmiregistry):

```
rmiregistry -J-Djava.security.policy=ssl.policy
```

then register the object with the registry:

```
java -classpath %CLASSPATH% -Djava.security.policy=ssl.policy -  
Djava.rmi.server.codebase=file:///%BOOK_HOME%/Ch03/SSL/ RegisterIt
```

Remember to carefully install the JSSE. Put the jar files in your <JAVA_HOME>\lib\ext location and install the provider in your java.security file (see the file INSTALL.txt within the JSSE bundle for further details). If you have multiple runtime environments, be sure to edit the right security file. The easiest solution is to set the PATH to point only to whichever JDK you are using.

Copyrighted image

Now run the client and specify the keystore to use using the JSSE javax.net.ssl.trustStore property. This is the keystore into which the certificate has been imported or the same keystore as the server:

```
>  
java -Djava.security.policy=ssl.policy -Djavax.net.ssl.trustStore=.keystore  
HelloClient localhost
```

Copyrighted image

If we want the client to be an applet, a few configuration issues need to be resolved first:

- The client must have a J2SE 1.2 (or newer) JVM. At the time of writing Netscape 6 and 6.1 were the only browsers with J2SE 1.2 support. Hence, we would probably need to install the Java plug-in software. (The Java plug-in is a one-time download that automatically upgrades the browser's JVM to the latest version. It works with both IE and Netscape.)
- The client JVM must have the appropriate keystore and certificates installed.
- The client JVM should have the right policy, allowing sockets to be opened back to the server.
- The client JVM should have the security provider installed, or have the right security permissions so that a security provider can be installed dynamically.

More information and downloads for the Java Plug-in can be accessed at <http://java.sun.com/products/plugin/>.

Let us look at the applet below. It is simple, similar to the example we discussed earlier, except that it is an applet. It is important to realize that the applet has nothing to do with the browser's SSL configuration. The SSL connection happens at the RMI-socket level. The client socket-factory class is downloaded dynamically and the secure connection made to the server:

```

import java.applet.*;
import java.awt.*;
import java.rmi.*;
import java.security.*;

public class SSLHelloApplet extends Applet{

    String message = "-n/a-";

    HelloInterface obj = null;

    // Get a reference to the object in applet initialization
    public void init() {
        try{
            Security.addProvider(new com.sun.net.ssl.internal.ssl.Provider());
            String host="rmi://" + getCodeBase().getHost() + "/HelloServer";
            HelloInterface obj = (HelloInterface)Naming.lookup(
                "rmi://localhost/HelloServer");
            message = obj.sayHello();
        } catch (Exception e){
            System.out.println(e);
        }
    }

    // Display the message in the applet
    public void paint(Graphics g){
        g.drawString(message, 25, 50);
    }
}

```

Below is the HTML page (`index.html`) for the above applet. This page is converted using the HTML converter tool packaged with the Java plug-in. When a client comes across this page, if a compatible version does not exist, the JRE 1.3 will be downloaded from the URL specified. (The Sun web site in this case):

```

<html>
  <head>
    <title>Hello World</title>
  </head>

  <body>
    <center> <h1>Hello World</h1> </center>

    The message from the HelloServer is:
    <p>
    <!-- "CONVERTED_APPLET"-->
    <!-- CONVERTER VERSION 1.3 -->
    <object classid = "clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
           width = 500 height = 120 codebase =
           "http://java.sun.com/products/plugin/1.3/jinstall-13-win32.cab#Version=1,3,0,0">

```

```
<param name = code value = "SSLHelloApplet" >
<param name = codebase value = "http://localhost:8080/hello/" >

<param name ="type" VALUE="application/x-java-applet;version=1.3">
<param name ="scriptable" VALUE="false">
<comment>
<embed type = "application/x-java-applet;version=1.3" code = "SSLHelloApplet"
codebase = "http://localhost:8080/hello/" width = 500 height = 120
scriptable=false pluginspage = "http://java.sun.com/products/plugin/1.3/plugin-
install.html"> <noembed></comment>

</noembed></embed>
</object>

<!--
<applet code = "SSLHelloApplet" codebase = "http://localhost:8080/hello/" width =
500 height = 120>
</applet>
-->
<!--"endConvertedApplet"-->

</body>
</html>
```

When the page above is accessed from the browser, the browser determines from the tags that a plug-in (JRE 1.3) is needed to load this component and prompts you to download it from the location that was specified in the `pluginspage` parameter (or `codebase` depending on the browser) – this is of course assuming that it isn't already installed:

Copyrighted image

Once we download and install the plug-in and enable it from its console on our machine, we can access the same page. The browser will then load it in the new Java 1.3 JVM (rather than the browser's default JVM).

At this point, we need to make sure we add the necessary run-time parameters for the plug-in using the Java Plug-in applet in Control Panel. We need to add the **-Djava.security.policy** and **-Djavax.net.ssl.trustStore** parameters. In this example they will be:

```
-Djava.security.policy=%BOOK_HOME%\Ch03\SSL\ssl.policy  
-Djavax.net.ssl.trustStore=%BOOK_HOME%\Ch03\SSL\.keystore
```

Copyrighted image

Finally, we should get a result that looks something like this:

Copyrighted image

The same URL can be accessed using **appletviewer**:

```
appletviewer -J-Djava.security.policy=ssl.policy  
-J-Djavax.net.ssl.trustStore=.keystore  
http://localhost:8080/hello/index.html
```

Copyrighted image

RMI, Firewalls, and HTTP

If you have worked in any networked enterprise, then you are probably familiar with how firewalls block all network traffic, with the exception of that intended for certain "well-known" ports. They are however necessary to protect the security of the network.

Since the RMI transport layer opens dynamic socket connections between the client and the server, the JRMP traffic is typically blocked by most firewall implementations. RMI provides a workaround to this.

There are three main methods to bypass firewalls:

- HTTP-tunneling
- SOCKS
- Downloaded socket factories

We'll examine each of these methods in turn.

HTTP Tunneling

To get across firewalls, RMI makes use of HTTP tunneling by encapsulating the RMI calls within an HTTP POST request. This method is popular since it requires almost no setup, and works quite well in firewall environments that permit handling of HTTP through a proxy, but disallow regular outbound TCP connections. There are two forms of HTTP-tunneling:

HTTP-to-Port

If a client is behind a firewall and RMI fails to make a normal connection to the server, the RMI transport layer automatically retries by encapsulating the JRMP call data within an HTTP POST request.

Since almost all firewalls recognize the HTTP protocol, the specified proxy server should be able to forward the call directly to the port on which the remote server is listening. It is important that the proxy server configuration be passed to the client JVM. Once the HTTP-encapsulated JRMP data is received at the server, RMI can automatically unwrap the HTTP tunneled request and decode it. The reply is then sent back to the client as HTTP-encapsulated data:

Copyrighted image

The proxy server configuration can be passed using properties, such as:

```
java -Dhttp.proxyHost=hostname -Dhttp.proxyPort=portnumber HelloClient
```

HTTP tunneling can be disabled by setting the property:

```
java.rmi.server.disableHttp=true
```

Http-to-CGI

If the server cannot be contacted from the client, even after tunneling, because it is also behind a firewall, the RMI Transport layer uses a similar mechanism on the server. The RMI Transport layer places JRMP calls in HTTP requests and sends those requests, just as above. However, instead of sending them to the server port it sends them to `http://hostname:80/cgi-bin/java-rmi?forward=<port>`. There must be an HTTP server listening on port 80 on the proxy, which has the `java-rmi.cgi` script. The `java-rmi.cgi` in turn invokes a local JVM, unwraps the HTTP packet, and forwards the call to the server process on the designated port. RMI JRMP-based replies from the server are sent back as HTTP packets to the originating client port where RMI again unwraps the information and sends it to the appropriate RMI stub.

The `java-rmi.cgi` script is packaged with the JDK and can be found in the `bin` directory. To avoid any DNS resolution problems at startup, the host's fully qualified domain name must be specified via a system property as:

```
java.rmi.server.hostname=www.host.domain
```

A servlet implementation of the CGI called a servlet handler is available from Sun at <http://java.sun.com/j2se/1.3/docs/guide/rmi/archives/rmiservlethandler.zip>.

Another alternative to the `java-rmi.cgi` is using a port redirector (for example, DeleGate proxy) on port 80 that accepts connections and immediately redirects them to another port.

Although HTTP tunneling is an alternative, in general it should be avoided for the following reasons:

- ❑ There is significant performance degradation. While tunneling, the RMI application will not be able to multiplex JRMP calls on a single connection, due to the request-response HTTP paradigm.
- ❑ Using the `java-rmi.cgi` script (or servlet) is a big security loophole on the server. The script redirects any incoming request to any port, completely circumventing the firewall.
- ❑ RMI applications tunneling over HTTP cannot use callbacks.

The SOCKS Protocol

SOCKS (SOCKet Server) is a networking proxy protocol that enables hosts on one side of a SOCKS server to gain full access to hosts on the other side of the SOCKS server without requiring direct IP reachability. The SOCKS server redirects connection requests from hosts on opposite sides of a SOCKS server (the SOCKS server authenticates and authorizes the requests, establishes a proxy connection, and relays data).

By default, JDK sockets use a SOCKS server if available and configured. Server sockets however do not support SOCKS so this approach is only useful for outgoing calls from the client to the server.

Downloaded Socket Factories

We have already discussed at length how custom sockets can be used. The factory classes can be coded to work around the firewall. Dynamically downloaded socket factories provide a good alternative to HTTP tunneling, but the code to bypass the firewall must be hard-coded in the factories. This is OK if you have a fixed network configuration and know how that particular firewall works. However, different clients can have different firewalls and there are questions regarding access rights to provide this tunneling and, of course, changing factory classes.

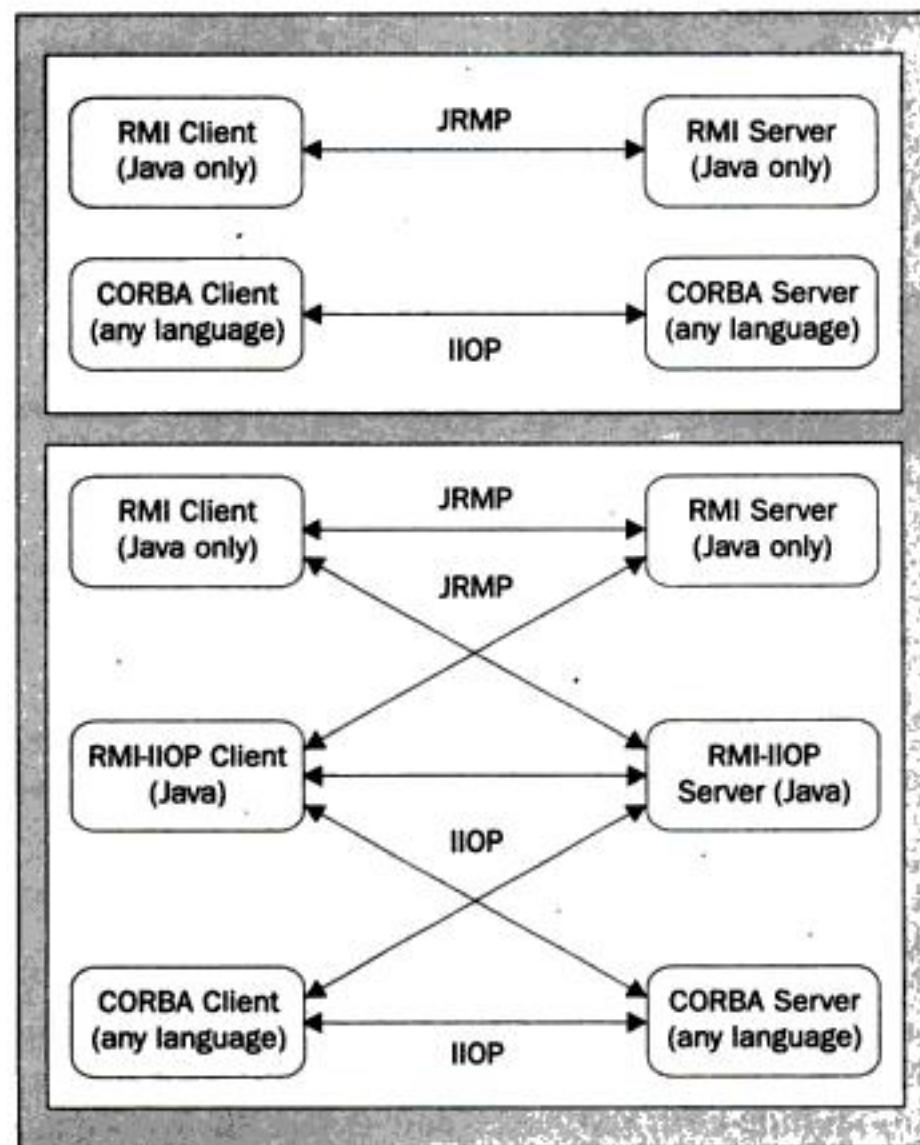
RMI Over IIOP

RMI over Internet Inter-Orb Protocol (IIOP) integrates Common Object Request Broker Architecture (CORBA)-compliant distributed computing directly into Java. RMI over IIOP, developed jointly by IBM and Sun, is a new version of RMI for IIOP that combines RMI's easy programming features with CORBA's interoperability.

RMI and CORBA have developed independently as distributed-objects programming models. RMI was introduced to provide a simple programming model for developing distributed objects whereas CORBA is a well-known distributed-object programming model that supports a number of languages. The IIOP protocol connects CORBA products from different vendors, ensuring interoperability among them.

The Object Management Group (OMG) – see <http://www.omg.org/> – is the official keeper of information for CORBA and IIOP, including the CORBA 2.0 specifications available at <http://www.omg.org/technology/documents/specifications.htm>. Additionally, the IDL to Java mappings are available at http://www.omg.org/technology/documents/formal/corba_language_mapping_specs.htm.

RMI-IIOP is, in a sense, a marriage of RMI and CORBA, since remote interfaces can be written in Java, and implemented using Java RMI APIs. These interfaces however can be implemented in any other language that is supported by an OMG mapping and a vendor-supplied ORB for that language. Similarly, clients can be written in other languages, using IDL derived from the Java remote interfaces.



The figure above summarizes the RMI-IIOP marriage. It may seem that the arrows connecting the JRMP clients/servers to the RMI-IIOP client/servers are misplaced because they are different protocols. These arrows are actually in the right place because RMI-IIOP supports both JRMP and IIOP protocols.

One of the initial design objectives was to make migration to IIOP feasible, indeed easy, and to avoid the need for a third distributed model for developers to learn. The server object created using the RMI-IIOP API can be exported as either a JRMP or IIOP-supporting object by simply changing deployment-time properties (without changing or recompiling code).

RMI-IIOP also supports dual export, meaning that a single server object can be exported to support both JRMP and IIOP simultaneously.

Interoperability with CORBA

An RMI-IIOP client cannot necessarily access all existing CORBA objects. The semantics of CORBA objects defined in IDL are a superset of those supported by RMI-IIOP objects, which is why an existing CORBA object's IDL cannot always be mapped into an RMI-IIOP Java interface. It is only when a specific CORBA object's semantics happen to correspond with those of RMI-IIOP that an RMI-IIOP client can call a CORBA object. The connection between the RMI-IIOP client and CORBA server is sometimes (but not always) possible.

However, this issue should not be over emphasized because it only applies when dealing with existing CORBA objects. Looking at the lower half of the above figure, if we design a new object with an RMI-IIOP Java interface, we can make the following observations:

- The CORBA implementation: you can automatically generate its corresponding IDL with the `rmic` tool. From this IDL file you can implement it as a CORBA object in any acceptable language, like C++ for instance. This C++ object is a pure CORBA object that can be called by a CORBA client as well as an RMI-IIOP client without any limitations. To the RMI-IIOP client, this C++ CORBA object appears as a pure RMI-IIOP object because it is defined by an RMI-IIOP Java interface.
- The RMI-IIOP implementation: the object appears as a CORBA object to a CORBA client (because a CORBA client accesses it through its IDL) and as an RMI object to RMI clients (because they access it through its RMI-IIOP Java interface).

In short, the difference between a CORBA object and an RMI-IIOP object is only an implementation matter.

One of the reasons for the problems with existing objects mentioned above is that two significant enhancements to the CORBA 2.3 specifications were actually made to bring about the RMI-IIOP and CORBA interoperability. OMG accepted these specifications:

- **Objects by Value specification:** This is already defined in Java in the form of object Serialization and is intended to make other languages implement a similar protocol.
- **Java-to-IDL Mapping specification:** This is the mapping used to convert RMI Java interfaces into CORBA IDL definitions. It should not be confused with the IDL-to-Java mapping already defined in CORBA 2.2.

Both of these specifications are available at OMG, and they are also accessible at <http://java.sun.com/products/rmi-iiop/index.html>. OMG has officially accepted both specifications for CORBA 2.3 and JDK 1.3 to include both RMI-IIOP and an IDL-to-Java compiler.

Writing Programs with RMI-IIOP

There are some development differences in syntax though the overall model remains the same, significantly, RMI-IIOP uses the JNDI API to locate and register objects. Although the development procedure for RMI-IIOP is almost the same as that for RMI (JRMP), the runtime environment is significantly different in that communication is made through a CORBA 2.3-compliant ORB using IIOP for communication between servers and clients. Let's briefly look at these differences.

On the Server

There is no significant change in development procedure for RMI objects in RMI-IIOP. The basic steps and their order is still the same as outlined earlier in our `HelloWorld` example, with some changes that also reflect the use of JNDI to lookup and bind objects.

HelloServer.java

Copyrighted image

Implementation class of a remote object:

```
public class HelloServer extends PortableRemoteObject
    implements HelloInterface {
```

The **PortableRemoteObject** is similar to the **UnicastRemoteObject** but provides the base functionality in the IIOP domain.

object
Copyrighted image

- Generate a tie for IIOP with **rmic -iiop**:
With the **-iiop** option the **rmic** compiler generates the stubs and tie classes that support the IIOP protocol. Without this **-iiop** option, **rmic** generates a stub and a skeleton for the JRMP protocol or only stubs if you also use the **-v1.2** option (remember skeletons are not needed in 1.2).
- Run **tnameserv.exe** as a name server:
This server provides the IIOP CosNaming services to for clients to lookup objects.
- Generate IDL with **rmic -idl** for CORBA clients (if our object is also going to be accessed by CORBA clients).
- While starting the server the two important environment variables for the JNDI context must be set up.
 - a. **java.naming.factory.initial**. This is the name of the class to use as the factory for creating the context.
 - b. **java.naming.provider.url**. This is the URL to the naming service similar to the **rmi://** URL format described earlier. The **rmi://** is replaced by **iiop://** and the default port is **900** instead of the **1099** for the registry.

These properties can be specified either while starting the server as shown below or by hard-coding values into **java.util.Properties** and passing that to the **InitialContext** as we do in the next example:

```
java -Djava.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory
-Djava.naming.provider.url=iiop://129.112.1.9:900 RegisterIt
```

In the Client

The client has the same import statement and uses the JNDI context to obtain a reference to the object, rather than using the registry. The JNDI **lookup()** method returns a **java.lang.Object**, which must then be cast using the **narrow()** method of **PortableRemoteObject**.

HelloClient.java

```
import java.rmi.*;
import javax.rmi.PortableRemoteObject;
import javax.naming.InitialContext;

public class HelloClient {
    public static void main(String args[]) {
        try {
            InitialContext ic = new InitialContext();
            PortableRemoteObject.narrow(ic.lookup("HelloServer"), Hello.class);
        }
    }
}
```

Create the JNDI context and bind to it:

```
InitialContext ctx = new InitialContext(); // Create the JNDI context
Object obj = ctx.lookup("/EgServer");
```

This binds the object to the JNDI context just as the `Naming.bind()` method did.

```
HelloInterface myobj = (HelloInterface) PortableRemoteObject
    .narrow(obj, HelloInterface.class);

String message = myobj.sayHello();
System.out.println(message);
} catch (Exception e) {
    System.out.println("HelloClient exception: " + e);
}
}
```

Again, the properties for the JNDI context (the context factory name and provider URL) *must* be passed to the client applications at runtime. That's basically it!

Sun provides a step-by-step guide packaged with the JDK on conversion of existing RMI programs and applets. This can also be accessed online at http://java.sun.com/j2se/1.3/docs/guide/rmi-iiop/rmi_iiop_pg.html.

Note that we can also write a server that supports both IIOP and JRMP clients explicitly by:

- Exporting it to both protocols using the `exportObject()` method. Of course, in this case we do not need to extend either `PortableRemoteObject` or `UnicastRemoteObject`.
- Creating two `InitialContexts`, one to allow binding to the RMI registry and one to the `COSNaming` service, and binding the server in both.
- Not passing the naming service as a command line argument using the `-D` option.

For example if we wanted to make our initial `HelloServer` object accessible over both RMI (JRMP) and RMI-IIOP then it could be modified as shown below:

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.util.Date;
import javax.rmi.PortableRemoteObject;

public class HelloServer implements HelloInterface {

    public void HelloServer() throws RemoteException {
        PortableRemoteObject.exportObject(this); // Export for IIOP
        UnicastRemoteObject.exportObject(this); // Export for JRMP
    }

    public String sayHello() throws RemoteException {
        return "Hello World, the current system time is " + new Date();
    }
}
```

The object can be registered with both contexts by modifying our RegisterIt file:

```

import java.rmi.RMISecurityManager;
import java.util.Properties;
import javax.naming.InitialContext;

public class RegisterIt {

    public static void main (String[] args){
        try{
            if(System.getSecurityManager() == null) {
                System.setSecurityManager (new RMISecurityManager());
            }
            HelloServer obj = new HelloServer (); // Create the object

            // Create a JNDIContext for JRMP and bind the object to the registry
            // Note we are using the RMI registry via JNDI and not the default
            // registry.
            Properties p_1 = new Properties();
            p_1.put("java.naming.factory.initial",
                    "com.sun.jndi.rmi.registry.RegistryContextFactory");
            InitialContext ctx_1 = new InitialContext ();
            ctx_1.rebind ("HelloServer", obj);
            System.out.println ("HelloServer bound in JRMP registry");

            // Repeat the same step for the IIOP registry
            Properties p_2 = new Properties();
            p_2.put("java.naming.factory.initial",
                    "com.sun.jndi.cosnaming.CNCtxFactory");
            InitialContext ctx_2 = new InitialContext (p_2);
            ctx_2.rebind ("RemoteHelloServer", obj);
            System.out.println ("HelloServer bound in IIOP registry");
        } catch(Exception e) {}
    }
}

```

We have swept through quite a lot in this section; so let's recap what we have just described. We have talked about what steps are involved in writing the RMI-IIOP server and client, the differences between them, and how the same server object can be written for both RMI-JRMP and RMI-IIOP clients. What we haven't mentioned, however, is how a CORBA client can request services of this object or how the RMI-IIOP client can access a CORBA object. Let's examine this a little more.

Once we have a Java interface (we are reusing the `HelloInterface` we wrote earlier), we can use the `rmic` with the `-idl` option to generate its IDL source.

In our case `rmic -idl HelloInterface` produces the following `HelloInterface.idl` file:

```

/**
 * HelloInterface.idl
 * Generated by rmic -idl. Do not edit
 * 20 August 2001 18:27:51 BST
 */

#include "orb.idl"

```

```
#ifndef __HelloInterface__
#define __HelloInterface__

interface HelloInterface {

    ::CORBA::WStringValue sayHello();
};

#pragma ID HelloInterface "RMI:HelloInterface:0000000000000000"

#endif
```

This .idl file can be used by any vendor-provided CORBA 2.3-compliant IDL compiler; we can generate a stub in our language of choice (we talk about C++ since it is a very common language for CORBA objects) to generate the stub classes for that language.

The same compiler can be used to generate skeleton/tie classes and the C++ server objects written to these classes, and IDL can be accessed with the RMI-IIOP clients that we wrote.

RMI-IIOP and Java IDL

Java IDL is an Object Request Broker provided with the JDK and can be used to define, implement, and access CORBA objects from Java. Java IDL is compliant with the CORBA/IIOP 2.0 Specification and the Mapping of OMG IDL to Java.

The first question relating to this topic that most developers ask is, "So is RMI being phased out in favor of RMI-IIOP?" The answer is an emphatic no! Another question that sometimes follows is, "Is RMI-IIOP a replacement for Java IDL?" Again, the answer is no. An RMI-IIOP client cannot necessarily access an existing CORBA object. If we want to use Java to access CORBA objects that have already been written, Java IDL is your best choice. With Java IDL, which is also a core part of the Java 2 platform, we can access any CORBA object from Java. The general recommendation for usage of RMI-IIOP and Java IDL is this: "If you want to use Java to access existing CORBA resources use Java IDL. If, conversely, you want to make Java RMI resources accessible to CORBA users you should use RMI-IIOP. If your application is going to be all Java, then use RMI-JRMP."

The J2SE v1.3 includes a new version of the IDL-to-Java compiler idlj that takes an IDL file and generates the Java server and client bindings from it.

The IDL File

OK, remember the IDL we just generated above? Let's take that file (`HelloInterface.idl`) as a starting point for this example and see how we can write CORBA objects and use Java IDL:

```
#include "orb.idl"

#ifndef __HelloInterface__
#define __HelloInterface__

interface HelloInterface {
```

```

    ::CORBA::WStringValue sayHello();
};

#pragma ID HelloInterface "RMI:HelloInterface:0000000000000000"
#endif

```

There are two things that can happen with an IDL file. We can either write the server or we can write a client (as we just mentioned above in the RMI-IIOP example). We will do both in this example using the `idlj` compiler.

The Server Implementation

Let's examine the sequence of events associated with the server implementation:

- Generate the server-side bindings for the IDL file. This is done with the `idlj` compiler:

```
idlj -i %JAVA_HOME%\lib -fserver HelloInterface.idl
```

Where `%JAVA_HOME%` is the J2SE 1.3 installation directory. We include the `orb.idl` file with the `-i` option to the `idlj` compiler because it is referenced in the IDL file (remember that this IDL we started with was auto generated itself). The `-fserver` option tells the compiler to generate the server-side bindings.

- Write the implementation. In CORBA terminology, this is called a servant class.

The servant, `HelloServant`, is the implementation of the IDL interface and is a subclass of `_HelloInterfaceImplBase`, which is generated by the `idlj` compiler (for each `XXX.idl` file the compiler generates a `_XXXImplBase.java` file). The servant contains one method for each IDL operation. In this example, this is just the `sayHello()` method. Note that servant methods are just ordinary Java methods.

```

import java.util.Date;

public class HelloServant extends _HelloInterfaceImplBase {
    public String sayHello() {
        return "Hello World, the current system time is " + new Date();
    }
}

```

- Write the class that binds the implementation (the servant) to the naming service. This class is called the *server* in CORBA terminology.

So, the `HelloServer` class will look like this:

```

import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

public class HelloServer {
    public static void main(String args[]) {
        try {
            // create and initialize the ORB
            ORB orb = ORB.init(args, null);

```

```
// create servant and connect it with the ORB
HelloServant helloRef = new HelloServant();
orb.connect(helloRef);

// get the root naming context
org.omg.CORBA.Object objRef =
    orb.resolve_initial_references("NameService");
NamingContext ncRef = NamingContextHelper.narrow(objRef);

// Bind the servant reference to the Naming Context
NameComponent nc = new NameComponent("Hello", "");
NameComponent path[] = {nc};
ncRef.rebind(path, helloRef);

// Wait for invocations from clients
java.lang.Object sync = new java.lang.Object();
synchronized (sync) {
    sync.wait();
}
} catch (Exception e) {
    System.out.println("Exception: " + e);
}
}
}
```

Thus, it is clear from the code above that the server performs the following actions:

- Instantiates the ORB
- Instantiates the servant and connects it to the ORB
- Gets a CORBA object reference for a naming context in which to register the new CORBA object
- Binds the new object in the naming context under the name "Hello"
- Waits for invocations of the new object

The Client Implementation

Now let's follow the procedure associated with the client implementation. We will need to create a client subfolder before running this step:

- Generate the client-side bindings for the IDL file:

```
idlj -i %JAVA_HOME%\lib -td client -fclient HelloInterface.idl
```

- Write the client class. This can be an applet or a simple Java class.

```
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class HelloClient{
    public static void main(String args[]){
        try{
```

```
// create and initialize the ORB
ORB orb = ORB.init(args, null);

// get the root naming context
org.omg.CORBA.Object objRef =
    orb.resolve_initial_references("NameService");
NamingContext ncRef = NamingContextHelper.narrow(objRef);

// resolve the Object Reference in Naming
NameComponent nc = new NameComponent("Hello", "");
NameComponent path[] = {nc};
HelloInterface helloRef =
    HelloInterfaceHelper.narrow(ncRef.resolve(path));

// call the Hello server object and print results
String hello = helloRef.sayHello();
System.out.println(hello);
} catch (Exception e) {
    System.out.println("Exception : " + e) ;
}
}
```

Finally, we can run this example using the following sequence:

- #### **1. Compile all the java files:**

```
javac *.java
```

- ## **2. Create the stubs for the server**

```
rmic -iiop HelloServer
```

- ### 3. Run the Naming service tnameserv:

```
tnameserv -ORBInitialPort 900
```

- #### 4. Start the server:

```
java -Djava.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory  
-Djava.naming.provider.url=iiop://localhost:900 HelloServer
```

- ## 5. Run the client HelloClient:

```
java -Djava.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory  
-Djava.naming.provider.url=iiop://localhost:900 HelloClient
```

With the previous two examples we have seen how to write an RMI-IIOP object and make it accessible to other CORBA clients. We have also seen how to take an IDL file and generate a CORBA object and access it with a Java-IDL client.

The idlj supports both the inheritance and delegation models.

This means given an interface XX in your IDL file, idlj generates _XXImplBase.java. The implementation for XX that you write must extend from _XXImplBase class. This is known as the inheritance model.

Sometimes this may not be useful if your implementation extends from another class. (Remember, a Java class can implement any number of interfaces but can extend only one class.)

In such a case you can tell the idlj compiler to use the Tie model. In such a case for the interface XX in your IDL file, idlj will generate XX_Tie.java and the constructor to XX_Tie takes a XX as an argument.

The implementation for XX that you provide is only required to implement the interface XX. To use it this implementation with the ORB, you must wrap your implementation within XX_Tie. For example:

```
XXImpl obj = new XXImpl ();
XX_Tie tie = new XX_Tie (obj);
orb.connect (tie);
```

As you can see the drawback of using the Tie model is that it adds an extra step and an extra method call.

The idlj compiler's default server-side model is the inheritance model. To use the delegation model use:

idlj -fserverTIE Hello.idl

or:

idlj -fallTIE Hello.idl

RMI-IIOP and J2EE

RMI-IIOP provides the following benefits to developers over RMI-JRMP:

- Interoperability with objects written in other languages via language-independent CORBA IDL
- Transaction and security contexts can be propagated implicitly because of the use of IIOP, which can implicitly propagate context information
- IIOP-based firewall support via IIOP proxies that can pass IIOP traffic in a controlled and manageable way

With IIOP as the transport protocol, Java RMI finds the support it needs to promote industrial-strength distributed application development, within a Java environment only. RMI-IIOP nevertheless has some weaknesses of its own compared to RMI-JRMP:

- No distributed garbage collection support is present. The RMI DGC interfaces do not represent object IDs as CORBA does, so those interfaces are not sufficient for CORBA/IIOP. You cannot rely on Java RMI/JRMP's features while using RMI-IIOP.

- Java's casting operator cannot be used in your clients directly after getting a remote object reference. Instead, you need to use a special method to get the right type.
- You are not allowed to inherit the same method name into your remote interface from different base remote interfaces.
- All constant definitions in remote interfaces must be of primitive types or Strings and evaluated at compile time.

So, what's the idea behind making RMI-IIOP the de facto protocol in J2EE instead of RMI-JRMP?

Copyrighted image

Enterprise JavaBeans (EJB) are a key constituent of J2EE; EJB components live within EJB containers, which provide the runtime environments for the components. EJB containers are developed by different vendors based on the EJB specifications (which use RMI semantics). Vendors are free to choose the implementation for their containers. To help interoperability for EJB environments that include systems from multiple vendors, Sun has defined a standard mapping of EJB to CORBA, based on the specification of Java-to-IDL mapping from OMG. That is, EJB interfaces are inherently RMI-IIOP interfaces.

*Sun's EJB-to-CORBA mapping can be accessed at
<http://java.sun.com/products/ejb/index.html>.*

As we'll see later in this book, EJBs present a model for creating distributed enterprise applications that can be deployed in a heterogeneous environment. Standard RMI with the JRMP fails to deliver on some aspects, such as:

- It's a Java only solution, and EJBs should also be accessible to other clients in an enterprise environment
- It does not support transaction and security context propagation across distributed JVMs (in fact, it does not support context propagation at all)
- It is not as scalable as IIOP

RMI-IIOP overcomes these limitations of RMI-JRMP. The EJB-to-CORBA mapping not only enables on-the-wire interoperability among multiple vendors' implementations of the EJB container, but also enables non-Java clients to access server-side applications written as EJBs through standard CORBA APIs. The EJB 2.0 specification enables inter-container operability using RMI-IIOP.

Tuning RMI Applications

The RMI API provides a lot of system properties that can be dynamically set at runtime with the `-D` option to the JVM. (For example, `java -Djava.rmi.dgc.leaseValue=30000 MyApp`). These are very useful for fine tuning performance and debugging applications. These properties are listed in the following tables, starting with a summary of all the properties that can be set in the client and server JVMs:

| Property | First Introduced (version) | Description |
|---|----------------------------------|---|
| <code>java.rmi.dgc.leaseValue</code> | 1.1 | The lease duration granted by the DGC to clients that access remote objects in this JVM. Clients usually renew a lease when it is 50% expired, so a very short value will increase network traffic and risk late renewals in exchange for reduced latency in calls to <code>Unreferenced.unreferenced</code> . The default value is 600000 milliseconds (10 minutes). |
| <code>java.rmi.server.codebase</code> | 1.1 | The locations from which classes that are published by this JVM may be downloaded. It can be a URL or space separated list of URLs (for 1.2) and can be set in both client and server JVMs. |
| <code>java.rmi.server.hostname</code> | 1.1 | The host name that should be associated with remote stubs for locally created remote objects, in order to allow clients to invoke methods on the remote object. The default value of this property is the IP address of the local host. |
| <code>java.rmi.server.logCalls</code> | 1.1 | Incoming method calls and exceptions will be logged to <code>System.err</code> if this is true. |
| <code>java.rmi.server.randomIDs</code> | 1.1.8 | If this value is true, object identifiers for remote objects exported by this JVM will be generated by using a cryptographically secure random number generator. The default value is false. |
| <code>java.rmi.server.useCodebaseOnly</code> | 1.1 | If this value is true the JVM cannot automatically load classes from anywhere other than its local classpath and from the server's <code>java.rmi.server.codebase</code> property. This can be set in both client and server JVM's. |
| <code>java.rmi.server.useLocalHostName</code> | 1.1.7 | RMI now uses an IP address to identify the local host when the <code>java.rmi.server.hostname</code> property is not specified and a fully qualified domain name for the localhost cannot be obtained. If set to true RMI is forced to use the fully qualified domain name by default. |
| <code>java.rmi.server.disableHttp</code> | 1.1 | If this value is true, HTTP tunneling is disabled, even when <code>http.proxyHost</code> is set. The default value is false and this property is useful for client JVMs. |

The next table lists the activation-specific properties:

| Property | First Introduced (version) | Description |
|--|-------------------------------|---|
| <code>javsa.rmi.activation.port</code> | 1.2 | The value of this property represents the TCP port on which the activation system daemon, <code>rmid</code> , will listen for incoming remote calls. The default value is 1098. |
| <code>java.rmi.activation.activator.class</code> | 1.2 | The class that implements the interface <code>java.rmi.activation.Activator</code> . This property is used internally to locate the resident implementation of the activator from which the stub class name can be found. |

This final table provides a summary of properties that are specific to the SUN JDK implementation of RMI:

| Property | First Introduced (version) | Description |
|--|-------------------------------|--|
| <code>sun.rmi.activation.execTimeout</code> | 1.2 | The time that <code>rmid</code> will wait for a spawned activation group to start up. The default is 30,000 milliseconds. |
| <code>sun.rmi.activation.snapshotInterval</code> | 1.2 | The number of updates for which the activation system will wait before it serializes a snapshot of its state to the <code>rmid</code> log file on disk. An "update" refers to a persistent change in the state of the activation system (for example, the registration of an activatable object) since the last snapshot was taken. Changing the value of this property can be used to make <code>rmid</code> re-start more quickly (by taking snapshots of the log more often) or to make <code>rmid</code> more efficient (by taking snapshots of the log less often). The default value is 200. |
| <code>sun.rmi.log.debug</code> | 1.2 | If this value is <code>true</code> , details of <code>rmid</code> 's logging activity are sent to <code>System.err</code> . |

Table continued on following page

| Property | First Introduced (version) | Description |
|--|----------------------------------|--|
| <code>sun.rmi.rmid.maxstartgroup</code> | 1.2 | The maximum number of activation group JVMs that <code>rmid</code> will allow to be in the "spawning but not yet active" state simultaneously. If more VMs need to be started, they will queue up until one of the current spawn attempts either succeeds or times out. This property does not limit the maximum number of active VMs; it is intended to smooth out sudden spikes of activity to avoid reaching operating system limits. While setting the value of this property to a lower number may result in a longer startup time for <code>rmid</code> , and setting the value to a higher number could shorten the startup time, setting this value too high can crash <code>rmid</code> , because your system may run out of resources. The default value is 3. |
| <code>sun.rmi.server.activation.debugExec</code> | 1.2 | If this value is <code>true</code> , the activation system will print out debugging information to the command line that is used for spawning activation groups. By default, the value is <code>false</code> , so debugging information is not printed. |
| <code>sun.rmi.dgc.checkInterval</code> | 1.1 | How often the RMI runtime checks for expired DGC leases. The default value is 300,000 milliseconds. |
| <code>sun.rmi.dgc.logLevel</code> | 1.1 | Controls the logging of incoming and outgoing calls related to DGC lease granting, renewing, and expiration. |
| <code>sun.rmi.dgc.server.gcInterval</code> | 1.2 | When it is necessary to ensure that unreachable remote objects are un-exported and garbage-collected in a timely fashion, the value of this property represents the maximum interval that the RMI runtime will allow between garbage collections of the local heap. The default value is 60,000 milliseconds. |
| <code>sun.rmi.loader.logLevel</code> | 1.2 | Controls the logging of each class name and codebase, whenever the RMI runtime attempts to load a class as a result of unmarshaling either an argument or a return value. The codebase that is printed is the annotated codebase, but may not necessarily be the actual codebase from which the class gets loaded; the RMI class loader defers the class loading to the current thread's context class loader, which may load the class from the classpath, rather than the annotated codebase. |

| Property | First Introduced (version) | Description |
|---|-------------------------------|---|
| <code>sun.rmi.server.exceptionTrace</code> | 1.2 | Controls the output of server-side stack traces from exceptions and errors that are thrown by dispatched, incoming remote calls. If this value is <code>true</code> , exception stack traces will be printed. By default (<code>false</code>), exception and error stack traces are not printed. |
| <code>sun.rmi.transport.logLevel</code> | 1.1 | Controls detailed logging throughout the transport layer. |
| <code>sun.rmi.transport.tcp.localHostNameTimeOut</code> | 1.1.7 | Represents the time that the RMI runtime will wait to obtain a fully qualified domain name for the local host. The default value is 10,000 milliseconds. |
| <code>sun.rmi.transport.tcp.logLevel</code> | 1.1 | Controls detailed logging for the TCP-specific transport sub-layer. |
| <code>sun.rmi.transport.tcp.readTimeout</code> | 1.2.2 | Represents the time used as an idle timeout for incoming RMI-TCP connections. The value is passed to <code>Socket.setSoTimeout</code> . This property is used only for cases where a client has not dropped an unused connection as it should. The default value is 7,200,000 milliseconds (2 hrs). |
| <code>sun.rmi.dgc.cleanInterval</code> | 1.1 | Represents the maximum length of time that the RMI runtime will wait before retrying a failed DGC "clean" call. The default value is 180,000 milliseconds. |
| <code>sun.rmi.dgc.client.gcInterval</code> | 1.2 | When it is necessary to ensure that DGC clean calls for unreachable remote references are delivered in a timely fashion, the value of this property represents the maximum interval that the RMI runtime will allow between garbage collections of the local heap. The default value is 60,000 milliseconds. |
| <code>sun.rmi.loader.logLevel</code> | 1.2 | Controls the logging of each class name and codebase, whenever the RMI runtime attempts to load a class as a result of unmarshaling either an argument or return value. The codebase that is printed is the annotated codebase, but may not necessarily be the actual codebase from which the class gets loaded; the RMI class loader defers the class loading to the current thread's context class loader, which may load the class from the classpath, rather than the annotated codebase. |
| <code>sun.rmi.server.logLevel</code> | 1.1 | Controls the logging of information related to outgoing calls, including some connection-reuse information. |

Table continued on following page

| Property | First Introduced (version) | Description |
|---|----------------------------|--|
| <code>sun.rmi.transport.connectionTimeout</code> | 1.1.6 | Represents the period for which RMI socket connections may reside in an "unused" state, before the RMI runtime will allow those connections to be freed (closed). The default value is 15,000 milliseconds. |
| <code>sun.rmi.transport.proxy.connectTimeout</code> | 1.1 | Represents the maximum length of time that the RMI runtime will wait for a connection attempt (<code>createSocket</code>) to complete, before attempting to contact the server through HTTP. This property is only used when the <code>http.proxyHost</code> property is set and the value of <code>java.rmi.server.disableHttp</code> is <code>false</code> . The default value is 15,000 milliseconds. |
| <code>sun.rmi.transport.proxy.logLevel</code> | 1.1 | Controls the logging of events (<code>createSocket</code> and <code>createServerSocket</code>) when the default <code>RMISocketFactory</code> class is used. This type of logging is likely to be useful for applications that use RMI over HTTP. Events in custom socket factories are not logged by this property. |

The Sun RMI properties described above are useful for debugging but it is important to remember – they are not officially supported and are subject to change. Properties that end ".logLevel" have output redirected to `System.err` with possible values of "SILENT", "BRIEF", and "VERBOSE".

Summary

We have covered a lot of ground in this chapter, ranging from the basic RMI architecture to HTTP-tunneling. If you're now feeling somewhat overwhelmed, take it a little slower. Go through the examples and write some test applications – that should get you on the move.

We have covered in the chapter:

- The RMI architecture.
- Developing RMI applications. The stages required for the creation of effective applications.
- Parameter passing in RMI, the types available, and their use.
- Object activation, the benefits of execution on an as-needed basis.
- RMI, firewalls, and HTTP. The different methods designed to allow JRMP traffic access through firewalls.
- The development of RMI over IIOP, and the integration of RMI with CORBA.

Keep in mind that even though RMI-IIOP is the model for J2EE, JRMP is still very much alive. If your application requires just Java-Java communication then RMI-JRMP is a feasible solution.

The inclusion of RMI-IIOP in the core JDK (from the J2SE 1.3 release), and its close association with EJB, establishes it as a foundation technology for enterprise middleware.

In hindsight, RMI has come a long way from its initial days when no distributed model existed in Java. Today it is the core of the Java distributed model in J2EE and is the base for future technologies, such as JINI, which are transforming the way in which devices and systems will be networked together. Though outside the scope of this chapter, Jini essentially takes the RMI model and architecture a quantum leap forward.

The following chapter will deal with database access using the Java Database Connectivity API (JDBC).

Copyrighted image

Copyrighted material

4

Database Programming with JDBC

A relational database is usually the primary data resource in an enterprise application. The **JDBC API** provides developers with a way to connect to relational data from within Java code. Using the JDBC API, developers can create a client (which can be anything from an applet to an EJB) that can connect to a database, execute **Structured Query Language (SQL)** statements, and processes the result of those statements.

If you are familiar with SQL and relational databases, the structure of the JDBC API is simple to understand and use. The API provides connectivity and data access across the range of relational databases. It can do this because it provides a set of generic database access methods for SQL-compliant relational databases. JDBC generalizes the most common database access functions by abstracting the vendor-specific details of a particular database. This result is a set of classes and interfaces, placed in the `java.sql` package, which can be used with any database that has an appropriate **JDBC driver**. This allows JDBC connectivity to be provided in a consistent way for any database. It also means that with a little care to ensure the application conforms to the most commonly available database features, an application can be used with a different database simply by switching to a different JDBC driver.

However, there is more to database connectivity in an enterprise-level environment than simply connecting to databases and executing statements. There are additional concerns to be met including using connection pooling to optimize network resources and the implementation of distributed transactions. While some of the concepts behind these concerns are advanced we will find that addressing them is not itself complex.

We will discuss the **JDBC 3.0 API**, which, at the time of writing, is the latest version of the JDBC API. Version 2.0 of the JDBC API had two parts: the **JDBC 2.1 Core API** and the **JDBC 2.0 Optional Package API**, and although these two APIs have been combined into one in version 3.0, the JDBC classes and interfaces remain in two Java packages: `java.sql`, and `javax.sql`:

- **java.sql**

This package contains classes and interfaces designed with traditional client-server architecture in mind. Its functionality is focused primarily on basic database programming services such as creating connections, executing statements and prepared statements, and running batch queries. Advanced functions such as batch updates, scrollable resultsets, transaction isolation, and SQL data types are also available.

- **javax.sql**

This package introduces some major architectural changes to JDBC programming compared to `java.sql`, and provides better abstractions for connection management, distributed transactions, and legacy connectivity. This package also introduces container-managed connection pooling, distributed transactions, and rowsets.

We will not attempt to cover general database programming concepts in this chapter; basic knowledge of SQL and client-server application development is a prerequisite. There is not the space in a single chapter to cover database programming exhaustively so we will take a high-level approach to the JDBC API. Rather than attempting to discuss, build, and analyze large applications, we will study short snippets of code that illustrate the usage of various classes and interfaces from `java.sql` and `javax.sql`. In particular we will look at:

- Database connectivity using JDBC drivers
- How various SQL operations can be performed using the standard JDBC API including obtaining database connections, sending SQL statements to a database, and retrieving results from database queries
- How to map SQL types to Java
- Advanced features in the JDBC API, including scrollable resultsets and batch updates
- New features in the JDBC 3.0 API, including save points

We will then go on to consider macro-level issues including:

- Using the `javax.sql` package and JNDI to obtain database connections
- Connection pooling – including a discussion on traditional connection pooling versus data source based connection pooling
- Distributed transactions – how the `javax.sql` package, together with the Java Transaction API, enables distributed transactions
- The basic concepts of a rowset, and how they provide both a higher-level of abstraction for database access and the easy serialization of data

We will be mostly concerned with the `java.sql` package, apart from the creation of database connections, and the programmatic control of transactions.

We begin by understanding how Java applications can connect to, and communicate with, databases.

Database Drivers

It is important to understand that the JDBC abstracts database connectivity from Java applications. A database vendor typically provides a set of APIs for accessing the data managed by the database server. Popular database vendors such as Oracle, Sybase, and Informix provide proprietary APIs for client access. Client applications written in native languages such as C/C++ can use these APIs to gain direct access to the data. The JDBC API provides an alternative to using these vendor-specific APIs. Although this eliminates the need for the Java developer to access vendor-specific native APIs, the JDBC layer may still need ultimately to make these native calls. A **JDBC driver** is a middleware layer that translates the JDBC calls to the vendor-specific APIs.

Depending on whether the `java.sql` package or the `javax.sql` package is used, there are different approaches for connecting to a database via the driver. Irrespective of this, to access data you'll need a database driver, probably supplied by the database vendor or by a J2EE server provider. A driver is nothing but an implementation of various interfaces specified in `java.sql` and `javax.sql` packages.

There are four different approaches to connect an application to a database server via a database driver. The following classification is an industry standard and commercial driver products are categorized according to it.

Type 1 – JDBC-ODBC Bridge

Open Database Connectivity (ODBC) was originally created to provide an API standard for SQL on Microsoft Windows platforms and was later enhanced to provide SDKs for other platforms. ODBC is partly based on the X/Open Call-Level Interface (CLI) specification, which is a standard API for database access. This API provides bindings in the C and Cobol languages for database access. CLI is intended to be platform-and database-neutral, which is where ODBC diverges from the specification. Embedded SQL was one of the approaches considered for database access by the SQL Access Group. This involves embedding SQL statements in applications programmed in high-level languages, and preprocessing them to generate native function calls. ODBC defines a set of functions for direct access to data, without the need for embedded SQL in client applications.

The first category of JDBC drivers provides a bridge between the JDBC API and the ODBC API. The bridge translates the standard JDBC calls to corresponding ODBC calls, and sends them to the ODBC data source via ODBC libraries:

Copyrighted image

The process boundaries are marked with a broken line. The **JDBC-ODBC Bridge** translates the JDBC API calls into equivalent ODBC calls. The driver then delegates these calls to the data source. The Java classes for the JDBC API and the JDBC-ODBC Bridge are invoked within the client application process; the ODBC layer executes in another process. This configuration requires every client that will run the application to have the JDBC-ODBC Bridge API, the ODBC driver, and the native-language-level API, such as the OCI library for Oracle, installed.

This solution for data access is inefficient for high-performance database access requirements because of the multiple layers of indirection for each data access call. In addition this solution limits the functionality of the JDBC API to that of the ODBC driver. The use of a JDBC-ODBC should be considered for experimental purposes only.

J2SE includes classes for the JDBC-ODBC Bridge and so there is no need to install any additional packages to use it. However, you do have to configure the ODBC manager by creating **data source names (DSNs)**. DSNs are simply named configurations linking up a database, an appropriate driver, and some optional settings. The JDBC-ODBC bridge should work with most ODBC 2.0 drivers.

Type 2 – Part Java, Part Native Driver

Type 2 drivers use a mixture of Java implementation and vendor-specific native APIs to provide data access. There is one layer fewer to go through than for a Type 1 driver and so in general a Type 2 driver will be faster than a Type 1 driver:

Copyrighted image

JDBC database calls are translated into vendor-specific API calls. The database will process the request and send the results back through the API, which will in turn forward them back to the JDBC driver. The JDBC driver will translate the results to the JDBC standard and return them to the Java application.

As with Type 1 drivers the part Java, part native code driver and the vendor-specific native language API must be installed on every client that runs the Java application. As the native code uses vendor-specific protocols for communicating with the database efficiently, Type 2 drivers are more efficient than Type 1 drivers. In addition we now have full use of the vendor's API. These two factors mean that Type 2 drivers are, in general, preferred over Type 1 drivers.

Type 3 – Intermediate Database Access Server

Type 3 drivers use an intermediate (middleware) database server that has the ability to connect multiple Java clients to multiple database servers:

Copyrighted image

Clients connect to database servers via an intermediate server component (such as a listener) that acts as a gateway for multiple database servers. The Java client application sends a JDBC call through a JDBC driver to the intermediate data access server, which completes the request to the Data Source using another driver (for example, a Type 2 driver).

The protocol used to communicate between clients and the intermediate server depends on the middleware server vendor but the intermediate server can use different native protocols to connect to different databases.

BEA WebLogic includes a Type 3 driver. One of the benefits of using a Type 3 driver is that it allows flexibility on the architecture of the application, as the intermediate server can abstract details of connections to database servers.

Type 4 – Pure Java Drivers

Type 4 drivers are a pure Java alternative to Type 2 drivers:

Copyrighted image

Type 4 drivers convert the JDBC API calls to direct network calls using vendor-specific networking protocols. They do this by making direct socket connections with the database. Type 4 drivers generally offer better performance than Type 1 and Type 2 drivers. Type 4 drivers are also the simplest drivers to deploy since there are no additional libraries or middleware to install. All the major database vendors provide Type 4 JDBC drivers for their databases and they are also available from third party vendors.

A list of available JDBC drivers, of all four types, is available at <http://industry.java.sun.com/products/jdbc/drivers/>. At the time of writing there were over 160 drivers available.

Getting Started

To start using JDBC with our applications, we need to install a driver. Of course, we also need a database server. For the sake of simplicity, in this chapter, we'll use the Cloudscape database for demonstrating the JDBC API as the J2EE Reference Implementation server from Sun includes a version of Cloudscape. Cloudscape is written using Java and can be found at `%J2EE_HOME%\lib\cloudscape\cloudscape.jar`.

You can also obtain an evaluation copy of Cloudscape at <http://www.cloudscape.com>. In order to use the JDBC API and the Cloudscape database, add the `cloudscape.jar` file to your CLASSPATH. Refer to the Cloudscape documentation for instructions on using CloudView, a GUI for accessing the Cloudscape database.

Cloudscape is a small-footprint database management system built in Java and can be used in either embedded or client-server mode. When used in embedded mode, Cloudscape executes within the client application process. This is the simplest mode for database management, as, apart from including the Cloudscape classes in your CLASSPATH, it does not involve any setup. Client-server mode lets you access Cloudscape in the traditional client-server manner.

As Cloudscape is implemented in Java, additional database drivers are not required since JDBC calls are mapped to Cloudscape Java API calls within the same process. In client-server mode, these calls are further mapped to RMI calls to the Cloudscape server process.

If you wish to use another database make sure that you follow the instructions given by the database vendor to set up the database, and install an appropriate database driver.

The `java.sql` Package

The classes in the `java.sql` package can be divided into the following groups based on their functionality:

- Connection Management
- Database Access
- Data Types
- Database Metadata
- Exceptions and Warnings

Let's look at the classes available in each group.

Connection Management

The following classes/interfaces let you establish a connection to the database. In most cases, this involves a network connection:

| Class | Description |
|--|---|
| <code>java.sql.DriverManager</code> | This class provides the functionality necessary for managing one or more database drivers. Each driver in turn lets you connect to a specific database. |
| <code>java.sql.Driver</code> | This is an interface that abstracts the vendor-specific connection protocol. You may find implementations of this interface from database vendors as well as third party database driver vendors. |
| <code>java.sql.DriverPropertyInfo</code> | Since each database may require a distinct set of properties to obtain a connection, you can use this class to discover the properties required to obtain the connection. |
| <code>java.sql.Connection</code> | This interface abstracts most of the interaction with the database. Using a connection, you can send SQL statements to the database, and read the results of execution. |

Database Access

Once you obtain a connection, the following classes/interfaces let you send SQL statements to the database for execution, and read the results:

| Class | Description |
|---|---|
| <code>java.sql.Statement</code> | This interface lets you execute SQL statements over the underlying connection and access the results. |
| <code>java.sql.PreparedStatement</code> | This is a variant of the <code>java.sql.Statement</code> interface allowing for parameterized SQL statements. Parameterized SQL statements include parameter markers (as "?"), which can be replaced with actual values later on. |
| <code>java.sql.CallableStatement</code> | This interface lets you execute stored procedures. |
| <code>java.sql.ResultSet</code> | This interface abstracts results of executing SQL SELECT statements. This interface provides methods to access the results row-by-row. You can use this interface to access various fields in each row. |

The `java.sql.PreparedStatement` and `java.sql.CallableStatement` extend `java.sql.Statement`. All the three interfaces are conceptually similar, each offering a means of sending requests to execute SQL statements (stored procedures in the case of `java.sql.CallableStatement`), and obtaining the results of execution.

Data Types

The `java.sql` package also provides several Java data types that correspond to some of the SQL types. You can use one of the following types as appropriate depending on what a field in a result row corresponds to in the database:

| Class | Description |
|---------------------------------|---|
| <code>java.sql.Array</code> | This interface provides a Java language abstraction of <code>ARRAY</code> , a collection of SQL data types. |
| <code>java.sql.Blob</code> | This interface provides a Java language abstraction of the SQL type <code>BLOB</code> . |
| <code>java.sql.Clob</code> | This interface provides a Java language abstraction of the SQL type <code>CLOB</code> . |
| <code>java.sql.Date</code> | This class provides a Java language abstraction of the SQL type <code>DATE</code> . Although the <code>java.util.Date</code> provides a general-purpose representation of date, the <code>java.sql.Date</code> class is preferable for representing dates in database-centric applications, as this type maps directly to SQL <code>DATE</code> type. Note that the <code>java.sql.Date</code> class extends the <code>java.util.Date</code> class. |
| <code>java.sql.Time</code> | This class provides a Java language abstraction of the SQL type <code>TIME</code> , and extends <code>java.util.Date</code> class. |
| <code>java.sql.Timestamp</code> | This class provides a Java language abstraction of the SQL type <code>TIME</code> , and extends <code>java.util.Date</code> class. |
| <code>java.sql.Ref</code> | This class provides a Java language abstraction of the SQL type <code>REF</code> . |
| <code>java.sql.Struct</code> | This interface provides a Java language abstraction of SQL structured types. |
| <code>java.sql.Types</code> | This class holds a set of constant integers, each corresponding to a SQL type. |

As we shall see later, in addition to the above, the JDBC API also specifies standard mappings between primitive types in Java and SQL.

The JDBC API also includes facilities for dealing with custom database types. These types can be represented as `java.sql.SQLData` objects, and data within these types can be accessed using `java.sql.SQLInput` and `java.sql.Output` interfaces that provide a stream-like interface to accessing data.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

In the following sections, we'll discuss some of the most commonly used classes and interfaces of this API. In particular, we will learn:

- How to load a database driver
- How to open a database connection
- How to send SQL statements to databases for execution
- How to extract results returned from a database query
- What prepared statements are
- The role of JDBC types
- Handling exceptions and warnings

Loading a Database Driver and Opening Connections

The `java.sql.Connection` interface represents a connection with a database. It is an interface because the implementation of a connection is network-protocol- and vendor-dependent. The JDBC API provides two different approaches for obtaining connections. The first uses `java.sql.DriverManager`, and is suitable for non-managed applications such as standalone Java database clients. The second approach is based on the `javax.sql` package that introduces the notion of data sources and is suitable for database access in J2EE applications.

Let's start by considering how connections are obtained using the `java.sql.DriverManager` class. In a single application, we can obtain one or more connections for one or more databases using different JDBC drivers. Each driver implements the `java.sql.Driver` interface. One of the methods that this interface defines is the `connect()` method that establishes a connection with the database, and returns a `Connection` object that encapsulates the database connection.

Instead of directly accessing classes that implement the `java.sql.Driver` interface, the standard approach for obtaining connections is to register each driver with the `java.sql.DriverManager`, and use the methods provided on this class to obtain connections. The `java.sql.DriverManager` can manage multiple drivers. Before going into the details of this approach, we need to understand how JDBC represents the location of a database.

JDBC URLs

The notion of a URL in JDBC is very similar to the way URLs are used otherwise. In order to see the rationale behind JDBC URLs, consider an application using several databases, each being accessed via a different database driver. In such a scenario, how do we uniquely identify a driver? Moreover, databases use different types of parameters for obtaining connections. How are parameters specified while making connections?

JDBC URLs provide a way of identifying a database driver. A JDBC URL represents a driver, and additional driver-specific information to locate a database and to connect to it. The syntax of the JDBC URL is as follows:

`jdbc:<subprotocol>:<subname>`



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Methods to Obtain Connections

The driver manager has three variants of a static method `getConnection()` used to establish connections. The driver manager delegates these calls to the `connect()` method on the `java.sql.Driver` interface.

Depending on the type of the driver and the database server, a connection may involve a physical network connection to the database server, or a proxy to a connection. Embedded databases require no physical connection. Whether or not there is a physical connection involved, the connection object is the only object that an application uses to communicate with the database. All communication must happen within the context of one or more connections.

Let's now consider the different methods for getting a connection:

```
public static Connection getConnection(String url) throws SQLException
```

The `java.sql.DriverManager` retrieves an appropriate driver from the set of registered JDBC drivers. The database URL is specified in the form of `jdbc:subprotocol:subname`. Whether we get a connection with this method or not depends on whether the database accepts connection requests without authentication.

```
public static Connection getConnection(String url,  
                                     java.util.Properties info)  
throws SQLException
```

This method requires a URL and a `java.util.Properties` object. The `Properties` object contains each required parameter for the specified database. The list of properties differs from database to database. Two commonly used properties for Cloudscape are `autocommit=true` and `create=false`. We can specify these properties along with the URL as

`jdbc:cloudscape:Movies;autocommit=true;create=true`, or we can set these properties using the `Properties` object, and pass the `Properties` object in the above `getConnection()` method:

Copyrighted Image

The driver neglects nonexistent properties; if the property value is not valid, you may get an exception.

Note that these properties are driver-specific, and you should refer to your driver documentation for the list of required properties. To learn more about the available properties for Cloudscape 4.0, refer to http://www.cloudscape.com/docs/doc_40/doc/html/coredocs/attrib.htm.

```
public static Connection getConnection(String url, String user,  
                                     String password)  
throws SQLException
```

The third variant takes user and password as the arguments in addition to the URL. Here is an example; the following code uses an ODBC driver, where `Movies` is a DSN set up in the ODBC configuration. This DSN corresponds to a database that requires a user name and password for getting a connection:

Copyrighted image

Note that all these methods are **synchronized**, implying that more than one application thread cannot directly get hold of the same `java.sql.Connection` object. These methods throw a `SQLException` if the driver fails to obtain a connection.

Sometimes it is necessary to specify the maximum time that a driver should wait while attempting to connect to a database. The following two methods can be used to set/get the login timeout:

```
public static void setLoginTimeout(int seconds)
public static int getLoginTimeout()
```

The default value for the login timeout is driver/database-specific.

Methods for Logging

The following methods access or set a `PrintWriter` object for logging purposes:

```
public static void setLogWriter(PrintWriter out)
public static PrintWriter getLogWriter()
```

In addition, client applications can also log messages using the following method:

```
public static void println(String message)
```

This method is used in conjunction with the `PrintWriter` set by `setLogWriter()` method to print log messages.

Driver

In JDBC, each driver is identified using a JDBC URL, and each driver should implement the `java.sql.Driver` interface. For instance in Cloudscape, the `COM.cloudscape.core.JDBCDriver` class implements the `java.sql.Driver` interface, which specifies the following methods:

```
public boolean acceptsURL(String url)
public Connection connect(String url, Properties info)
public int getMajorVersion()
public int getMinorVersion()
public DriverPropertyInfo getPropertyInfo(String url, Properties info)
public boolean jdbcCompliant()
```

The `DriverManager` class uses these methods. In general, client applications need not access the driver classes directly.

Establishing a Connection

To communicate with a database using JDBC, we must first establish a connection to the database through the appropriate JDBC driver. The JDBC API specifies the connection in the `java.sql.Connection` interface. This interface has the following public methods:

| Function | Methods |
|--|--|
| Creating statements | <code>createStatement()</code> <code>prepareStatement()</code> <code>prepareCall()</code> |
| Obtaining database information | <code>getMetaData()</code> |
| Transaction support | <code>setAutoCommit()</code> <code>getAutoCommit()</code> <code>commit()</code> <code>rollback()</code> <code>setTransactionIsolation()</code> <code>getTransactionIsolation()</code> |
| Connection status and closing | <code>isClosed()</code> <code>close()</code> |
| Setting various properties, clearing values and retrieving any warnings generated | <code>setReadOnly()</code> <code>isReadOnly()</code> <code>clearWarnings()</code> <code>getWarnings()</code> |
| Converting SQL strings into database-specific SQL and setting views and user defined types | <code>nativeSQL()</code> <code>setCatalog()</code> <code>getCatalog()</code> <code>setTypeMap()</code> <code>getTypeMap()</code> |

The categories in the above table represent an attempt to break the methods into logical groups according to functionality. In the next few sections, we'll discuss most of these methods.

Here's an example of establishing a JDBC connection to the Cloudscape **Movies** database:

Copyrighted image

```
        connection.close();
    } catch(SQLException e) { // System error?
}
```

In this example, the URL specifies the JDBC URL; `create=true` indicates that Cloudscape should create the `Movies` database if it does not exist. You should also always catch the `SQLException`, and try to close the connection after using the connection for data access.

In the JDBC API, there are several methods that throw a `SQLException`. In this example, the connection is closed in the `finally` block, so that the system resources can be freed up regardless of the success or otherwise of any database operations.

Let us now summarize what has been discussed so far using a simple example. This example registers the cloudscape driver and establishes a connection:

```
// Import required packages
import java.sql.Driver;
import java.sql.DriverManager;
import java.sql.Connection;
import java.sql.DatabaseMetaData;
import java.sql.SQLException;
import java.sql.DriverPropertyInfo;
import java.util.Enumeration;
import java.util.Properties;

public class DriverTest {
    public static void main(String arg[]) {
        String protocol = "jdbc:cloudscape:c:/CloudscapeDB";
        String driverClass = "COM.cloudscape.core.JDBCDriver";

        // Register the Cloudscape driver
        try {
            Class.forName(driverClass);
        } catch (ClassNotFoundException cne) {
            cne.printStackTrace();
        }

        // Check the registered drivers
        Enumeration drivers = DriverManager.getDrivers();
        while (drivers.hasMoreElements()) {
            Driver driver = (Driver) drivers.nextElement();
            System.out.println("Registered driver: "
                + driver.getClass().getName());
        }
    }

    // Does this driver accept the known URL
    if (driver.acceptsURL(protocol)) {
        System.out.println("Accepts URL: " + protocol);
    }
} catch (SQLException sqle) {
    sqle.printStackTrace();
}

// Get a connection from the DriverManger.
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

This includes the driver class name, the accepted URL, the product name, and the driver name. Note that you will find that the driver name is somewhat cryptic. This is because Cloudscape libraries are obfuscated to prevent de-compiling.

Creating and Executing SQL Statements

We can use a `Connection` object to execute SQL statements by creating a `Statement`, a `PreparedStatement`, or a `CallableStatement`. These objects abstract regular SQL statements, prepared statements, and stored procedures respectively. Once we obtain one of these statement objects, we can execute the statement and read the results through a `ResultSet` object.

As shown in the table above, the following methods create statement objects:

```
Statement createStatement() throws SQLException
```

This method creates a `Statement` object, which we can use to send SQL statements to the database. SQL statements without parameters are normally executed using `Statement` objects.

```
Statement createStatement(int resultSetType, int resultSetConcurrency)
    throws SQLException
```

This variant of `createStatement()` requires `resultSetType`, and `resultSetConcurrency` arguments. These arguments apply to `ResultSet` objects created by executing queries.

Of these two arguments, the first argument is used to specify the required `ResultSet` type. As we shall see later, there are three resultset types depending on the scrollability: forward only scrollable (`ResultSet.TYPE_FORWARD_ONLY`), scrollable but insensitive to changes made by other transactions (`ResultSet.TYPE_SCROLL_INSENSITIVE`), or scrollable and sensitive to changes made by other transactions (`ResultSet.TYPE_SCROLL_SENSITIVE`). The `java.sql.ResultSet` interface specifies three constants as result types.

The second argument is used to specify if the `ResultSet` is read-only (`ResultSet.CONCUR_READ_ONLY`), or should be updateable (`ResultSet.CONCUR_UPDATABLE`).

The no-argument `createStatement()` method returns a `ResultSet` that is forward-only scrollable, and read-only.

The `createStatement()` method takes no arguments (except for type and concurrency where applicable) and returns a `Statement` object. The ultimate goal of a `Statement` object is to execute a SQL statement that may or may not return results. We will see an example soon.

```
public PreparedStatement prepareStatement(String sql) throws SQLException
```

We can get a `PreparedStatement` object by calling this method on a `Connection`. Later in this chapter, we shall walk through the usage of prepared statements for executing SQL statements.

```
public CallableStatement prepareCall(String sql) throws SQLException
```

This method is used to call a stored procedure.

The `Statement` interface has the following methods:



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
    } catch(SQLException sqlException) {  
  
        while(sqlException != null) {  
            System.err.println(sqlException.toString());  
            sqlException = sqlException.getNextException();  
        }  
    }  
}
```

You can similarly retrieve warnings (including those that are vendor-specific) received or generated by the driver using the `getWarnings()` method on the connection:

```
SQLWarning warnings = connection.getWarnings();  
  
while(warnings != null) {  
    System.err.println(connection.getWarnings());  
    warnings = warnings.getNextWarning();  
}
```

Here is the complete source:

```
import java.sql.DriverManager;  
import java.sql.Connection;  
import java.sql.Statement;  
import java.sql.PreparedStatement;  
import java.sql.SQLException;  
import java.sql.Date;  
import java.io.BufferedReader;  
import java.io.FileReader;  
import java.io.IOException;  
import java.io.EOFException;  
import java.text.SimpleDateFormat;  
import java.text.ParseException;  
  
// This class requires a text file called catalog.txt containing the  
// input data.  
  
public class CreateMovieTables {  
  
    static String driver = "COM.cloudscape.core.JDBCDriver";  
    static String url = "jdbc:cloudscape:";  
  
    String title, leadActor, leadActress, type, dateOfRelease;  
    Connection connection;  
    Statement statement;  
  
    public void initialize() throws SQLException, ClassNotFoundException {  
        Class.forName(driver);  
  
        connection = DriverManager.getConnection(url + "Movies;create=true");  
    }  
  
    public void createTable() throws SQLException {  
        statement = connection.createStatement();  
        statement.executeUpdate("CREATE TABLE CATALOG" +
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

This method first creates a `Statement` object, which it then uses to call `executeQuery()` with a SQL `SELECT` statement as an argument. The `java.sql.ResultSet` object returned contains all the rows of the `CATALOG` table matching the `SELECT` statement. Using the `next()` method of `ResultSet` object, we can iterate through all the rows contained in the resultset. At any given row, we can use one of the `getXXX` methods in the table above to retrieve the fields of a row.

What happens if the query (or any SQL being sent to the database via the driver for execution) is very expensive and takes a very long time to complete? This may happen, for instance, if the query is complex, or if the database is attempting to return a very large number of results. In order to control the time that the driver waits for the database to complete execution, the `java.sql.Statement` interface has two methods to get or set a maximum timeout.

For instance, we can modify the `queryAll()` method to set a maximum query timeout interval (in seconds) using the `setQueryTimeout()` method. You can use the `getQueryTimeout()` method to find the current query timeout interval in seconds (or the default, if not set explicitly). Once the database exceeds the query timeout interval, the driver aborts the execution and throws a `java.sql.SQLException`. For example, here we set the time out value to 1 second:

```
Statement statement = connection.createStatement();
statement.setQueryTimeout(1);
// Set SQL for statement
ResultSet rs = statement.executeQuery(sqlString);
```

ResultSetMetaData Interface

The `ResultSet` interface also allows us to find out the structure of the resultset. The `getMetaData()` method helps us to retrieve a `java.sql.ResultSetMetaData` object that has several methods to describe resultset cursors:

| | |
|-------------------------------------|-------------------------------------|
| <code>GetCatalogName()</code> | <code>getScale()</code> |
| <code>getTableName()</code> | <code>getPrecision()</code> |
| <code>GetSchemaName()</code> | <code>isNullable()</code> |
| <code>GetColumnCount()</code> | <code>isCurrency()</code> |
| <code>GetColumnName()</code> | <code>isSearchable()</code> |
| <code>GetColumnLabel()</code> | <code>isCaseSensitive()</code> |
| <code>GetColumnType()</code> | <code>isSigned()</code> |
| <code>GetColumnTypeNames()</code> | <code>isAutoIncrement()</code> |
| <code>GetColumnClassName()</code> | <code>isReadOnly()</code> |
| <code>GetColumnDisplaySize()</code> | <code>isDefinitelyWritable()</code> |
| <code>isWritable()</code> | |

Given a resultset, we can use the `getColumnCount()` method to get the number of columns in the resultset. Using this column number, we can get the meta-information of each column.

For example, the following method in our example prints the structure of the resultset:



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Savepoints

The JDBC 3.0 API introduces save points. A **save point** is similar to a temporary marker set at a specific position during a set of database operations within a transaction. A savepoint allows you commit or rollback transactions partially. In order to understand the notion of a savepoint, consider using a word processor to edit a file. During the editing process, you can save the document several times. Each save operation is like setting a new save point. After a save operation, you can revert to the previously saved version by simply negating all the edits and reopening the same file. In a similar fashion, the `java.sql.Savepoint` mechanism lets you commit/abort at various save points.

Consider the following sequence of database operations:

```
Connection connection = null;

// Get a connection
...
try {
    // Begin a transaction
    connection.setAutoCommit(false);

    // Perform the first set of database operations
    ...

    // Perform the second set of database operations
    ...

    // Perform the third set of database operations
    ...
} catch (SQLException e) {
    // Handle exceptions here
} finally {
    // Close the statement and connection
}
```

Suppose that the three sets of operations are logically related. For instance, if there is a failure while performing the second set of operations, there is no way to undo only the second set, and do some other set of corrective operations. The `commit()` and `rollback()` methods apply to the three sets of operations as a whole. You can overcome this limitation with save points.

Consider how the above can be implemented using save points:

```
Connection connection = null;

// Get a connection
...
try {
    // Begin a transaction
    connection.setAutoCommit(false);

    // Perform the first set of database operations
    ...

    Savepoint spl = connection.setSavepoint("First Batch");
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

`isFirst()` returns true if the cursor is at the first row of the resultset. `isLast()` returns true if the cursor is at the last row of the resultset.

```
public void beforeFirst() throws SQLException  
public void afterLast() throws SQLException
```

These two methods returns true if the cursor position is before the first row or after the last row of the resultset respectively.

Methods for Scrolling

```
public boolean first() throws SQLException  
public boolean last() throws SQLException
```

These methods move the cursor to the first row and last row of the resultset respectively.

```
public boolean absolute(int row) throws java.sql.SQLException
```

`absolute()` moves the cursor to the specified row in the resultset. The row argument can be positive or negative. A negative argument moves the cursor in the backward direction, while a positive argument moves the cursor in the forward direction.

```
public boolean relative(int rows) throws SQLException
```

This method moves the cursor by the specified number of rows relative to the current position of the cursor. If the argument is positive, this method moves the cursor forwards by the specified number of rows. If it is negative, this method moves the cursor backwards by the specified number of rows. If the specified number (positive or negative) is more than the available rows (forward or backward), this method positions the cursor at the first or the last row in the resultset. This method should not be called when the cursor position is invalid. For instance, after calling `next()` recursively till `next()` returns `false` (signifying the end of the resultset), we should not call `relative()`, as the current position is outside the resultset. Before calling `relative()`, we should bring the cursor to a valid position using the `absolute()`, `first()`, or `last()` methods.

```
public boolean previous() throws SQLException
```

`previous()` moves the cursor to the previous row, relative to the current position. Unlike the `relative()` method, this method can be called even when there is no current row. For instance, after calling `next()` recursively until it returns `false`, we can call `previous()` to bring the cursor to the last row.

Fetch Direction and Size

```
public void setFetchDirection(int direction) throws SQLException
```

This method lets us set a fetch direction. The `java.sql.ResultSet` interface specifies three fetch direction constants – `ResultSet.FETCH_UNKNOWN`, `ResultSet.FETCH_FORWARD`, and `ResultSet.FETCH_REVERSE`. These types indicate the current fetch direction for the results.

The resultset may internally optimize the result structures for the specified type of scrolling.

```
public int getFetchDirection() throws SQLException
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Deleting a Row

We can use the `deleteRow()` method to delete the current row from the resultset as well as from the underlying database. The `rowDeleted()` method indicates whether a row has been deleted using the resultset. Note that a deleted row may leave a visible hole in a resultset. The `rowDeleted()` method can be used to detect such holes. While scrolling through a resultset that may have deleted rows, you can call this method to detect such deleted rows.

Inserting a Row

To insert a row in the resultset, and the underlying database, we should first use `moveToInsertRow()`, and then one or more of `updateXXX()` methods.

A call to the `moveToInsertRow()` method moves the cursor to the insert row. The insert row is a special buffer row associated with an updateable resultset. After moving the cursor to this row, we can use the usual `updateXXX()` methods to set the elements of this row. At the end of these calls, you should call `insertRow()` to finally insert the row in the database, and to clear the buffer row.

Batch Updates

Apart from scrollable resultsets, the JDBC API 2.1 also specifies support for batch updates. This feature allows multiple update statements (`INSERT`, `UPDATE`, or `DELETE`) in a single request to the database. Batching large numbers of statements can result in significant performance gains.

For example, we can re-modify the table for the `Movies` database but this time we can perform all `INSERT` statements in a single batch. We can also use prepared statements in a batch. When compared to the `insertData()` in the example, the only differences in this implementation are the calls to the `addBatch()` method, and `executeBatch()` method on the statement for batch update:

```
public void insertBatchData() throws SQLException, IOException {
    BufferedReader br = new BufferedReader(new FileReader("catalog.txt"));
    statement = connection.createStatement();

    try {
        do {
            title = br.readLine();
            if(title == null) break;
            leadActor = br.readLine();
            leadActress = br.readLine();
            type = br.readLine();
            dateOfRelease = br.readLine();

            String sqlString = "INSERT INTO CATALOG VALUES('" + title + "','" +
                               leadActor + "','" + leadActress + "','" +
                               type + "','" + dateOfRelease + "')";

            statement.addBatch(sqlString);
        } while(br.readLine() != null);

        statement.executeBatch();

    } catch (EOFException e) {
    } finally {
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
import java.sql.Statement;
import java.sql.ResultSet;
import java.sql.DatabaseMetaData;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;

// Import javax.sql classes
import javax.sql.DataSource;
// Import JNDI classes
import javax.naming.Context;
import javax.naming.NamingException;
```

Note the two additional import statements to import classes from `javax.sql`, and `javax.naming` packages.

The other change is in the `initialize()` method in which we originally loaded the driver, and created a connection. Instead of this, we should now obtain a `javax.sql.DataSource` object, and create a connection:

```
public void initialize() throws SQLException, NamingException {
    // Rather than registering the driver as below
    /*   Class.forName (driver);
    // lookup and obtain a DataSource object from JNDI
    Context initialContext = new InitialContext();
    dataSource = (DataSource) initialContext.lookup("jdbc/Cloudscape");

    // Replace
    //   connection = DriverManager.getConnection(url);
    // with using the DataSource object to obtain a connection.
    connection = dataSource.getConnection();
}
```

The first statement creates a `javax.naming.Context` object. The `InitialContext` class implements the `javax.naming.Context` interface and provides the starting point for resolution of names in the JNDI service.

In the above example, we use `initialContext` to `lookup jdbc/Cloudscape`. This is a name that we specify while configuring the application server, so that the application server can create an instance of `javax.sql.DataSource`, and bind it with the name `jdbc/Cloudscape` in the JNDI service. `jdbc` is the standard naming sub-context for all `DataSource` objects.

The `lookup()` method returns an object implementing the `javax.sql.DataSource` interface. Since the `lookup()` method actually returns a `java.lang.Object`, we need to cast it to `javax.sql.DataSource`.

The rest of the database operations are done as usual. In order to test this application, we require a J2EE application server that supports data sources for client applications.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Leaving aside the details of which classes are used to hold connections, this approach is neither robust nor scalable. The fundamental flaw in classic connection pooling is that such a pool does not work with distributed transactions. It also plays havoc with container-managed environments such as web containers, since it forces you to have connections held in static variables, on which the container cannot enforce access control.

Connection Pooling with the javax.sql Package

The `javax.sql` package provides a transparent means of connection pooling. With this approach, the application server and/or the database driver handle connection pooling internally. As long as we use `DataSource` objects for getting connections, connection pooling will automatically be enabled once we configure the J2EE application server.

Before we go into details of this approach, here is a schematic view:

Copyrighted image

When compared to the first figure, the only change is the additional connection pool maintained by the application server in coordination with the JDBC driver. This means that there is no additional programming requirement for JDBC client applications. Instead, the administrator of the J2EE server will be required to configure a connection pool on the application server. The exact syntax and the names of classes are implementation-dependent. With a JDBC 2.0 compliant application server and database driver, however, the server administrator typically specifies the following:

- ❑ A class implementing the `javax.sql.ConnectionPoolDataSource` interface
- ❑ A class implementing the `java.sql.Driver` interface
- ❑ Size of the pool (minimum and maximum sizes)
- ❑ Connection time out
- ❑ Authentication parameters (login, password etc.)



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

What is a Transaction?

Enterprise applications often require concurrent access to distributed data shared among multiple components, to perform operations on data. Such applications should maintain integrity of data (as defined by the business rules of the application) under the following circumstances:

- **Distributed access to a single resource of data**

For example, we may have two application components implementing two parts of a business transaction – both using the same database, but different connections.

- **Access to distributed resources from a single application component**

For example, a component updating multiple databases in a single business transaction. This operation would require multiple `Connection` objects.

In both these cases, the methods provided by the core JDBC API are not adequate. This is because the methods provided on the `java.sql.Connection` for starting and ending transactions are associated with a single `Connection` object. In the above situations, however, this is not the case.

The concept of a transaction, and a transaction manager (or a transaction processing service) simplifies developing such applications to maintaining integrity of data in a unit of work as described by the **ACID** properties, guaranteeing that a transaction is never incomplete, the data is never inconsistent, concurrent transactions are independent, and the effects of a transaction are persistent. For more information on transactions refer to Chapter 17.

In the following section you will learn what ACID stands for, and how ACID properties are maintained by transaction processing systems.

Brief Background

In the distributed transaction-processing domain, the X/Open Distributed Transaction Processing (DTP) model is the most widely adopted model for building transactional applications. Almost all vendors developing products related to transaction processing, relational databases, and message queuing, support the interfaces defined in the DTP model.

This model defines three components:

- Application programs
- Resource managers
- Transaction manager

This model also specifies functional interfaces between application programs and the transaction manager (known as the **TX interface**), and between the transaction manager and the resource managers (the **XA interface**). Using TX-compliant transaction managers and XA-compliant resource managers (such as databases), we can implement transactions with the two-phase commit and recovery protocol to comply with the requirements for transactions.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The following are the typical responsibilities of an application component:

- Create and demarcate transactions
- Propagate transaction context
- Operate on data via resource managers

Resource Managers

A resource manager is a component that manages persistent and stable data storage systems, and participates in the two-phase commit and recovery protocols with the transaction manager. Examples are database systems, message queues, etc.

In addition, a resource manager is typically a driver or a wrapper over a storage system, with interfaces for operating on the data (for the application components). This component may also, directly or indirectly, register resources with the transaction manager so that the transaction manager can keep track of all the resources participating in a transaction. This process is called resource enlistment. The resource manager should also implement supplementary mechanisms (for example, logging) that make recovery possible.

Resource managers provide two sets of interfaces: one set for the application components to get connections and perform operations on the data, and the other set for the transaction manager to participate in the two-phase commit and recovery protocol.

The following are the typical responsibilities of resource managers:

- Enlist resources with the transaction manager
- Participate in two-phase commit and recovery protocol

Transaction Manager

The transaction manager is the core component of a transaction-processing environment. Its primary responsibilities are to create transactions when requested by application components, allow resource enlistment and de-listing, and to perform the commit/recovery protocol with the resource managers.

A typical transactional application begins by issuing a request to a transaction manager to initiate a transaction. In response, the transaction manager starts a transaction and associates it with the calling thread. The transaction manager also establishes a transaction context. All application components and threads participating in the transaction share the transaction context. The thread that initially issued the request for beginning the transaction, or, if the transaction manager allows, any other thread, may eventually terminate the transaction by issuing a commit or rollback request.

Before a transaction is terminated, any number of components and/or threads may perform transactional operations on any number of transactional resources known to the transaction manager. If allowable by the transaction manager, a transaction may be suspended and then resumed before finally being completed.

Once the application issues the commit request, the transaction manager prepares all the resources for a commit operation (by conducting a vote), and based on whether all resources are ready, issues a commit or rollback request.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Notice we import the `javax.transaction` package in addition to the `javax.sql`, and `javax.naming` packages.

In the above snippet, the JNDI lookup looks for `javax.transaction.UserTransaction`. This is the standard name associated with the `javax.transaction.UserTransaction` interface. The application server, which implements this interface, binds an object implementing this interface in the JNDI service.

Database Operations

Apart from certain constraints on calling specific methods on the `java.sql.Connection` object, the rest of the procedure for JNDI lookup for a data source object, creating connections, and executing SQL statements is the same. The following code snippet gives a snapshot:

```
DataSource dataSource = null;
UserTransaction ut = null;

// Create a context
...
try {
    Context context = new InitialContext();
    dataSource = (DataSource) context.lookup("jdbc/x");
    ut = (UserTransaction)
        context.lookup("javax.transaction.UserTransaction");
} catch (NamingException ne) {
    // Failed to create the context or lookup.
}

Connection connection = dataSource.getConnection();

// ...

// Perform usual database operations

// ...

// Rollback the transaction in case of any failure of business condition
if(...){
    // Get the UserTransaction.
    ut.setRollbackOnly()
}

connection.close()

// Do something that uses another data source and another
// connection object
doSomething();

// ...
```

The above code is similar to what we would do in local database transactions. However, the implementation of the application logic can decide to raise a condition for rollback of the transaction.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Application objects implementing the `javax.sql.RowSetListener` interface can register and deregister for these events with the rowset, using the following methods on `javax.sql.RowSet`:

```
public void addRowSetListener(RowSetListener listener);
public void removeRowSetListener(RowSetListener listener);
```

These methods are similar to those that we find in AWT and Swing. We can dynamically add more than one listener. We can also remove these listeners dynamically.

When the Listener objects are notified, the Listener objects receive the RowSet object encapsulated in the `javax.sql.RowSetEvent`.

Command Execution and Results

After setting the properties, and the listeners, we can call the `execute()` method on the rowset to populate the results, and use any of the methods specified in the `java.sql.ResultSet` interface for scrolling and modifying the results. The `execute()` method internally obtains a database connection, prepares a statement, and creates a resultset.

Creating and using RowSet objects is very simple, and follows the JavaBeans model. In order to populate a resultset there are three tasks that we are required to do:

- Create a RowSet object
- Set its properties
- Execute it

Types of RowSet Objects

The JDBC API does not specify standard interfaces implementing the `javax.sql.RowSet` interface. The early access release from Sun, however, identifies three possible implementations. Although currently these are not part of the specification, in the future they could be added to the specification as part of a revised set of interfaces.

The following are three possible implementations of this interface:

- Cached rowsets (`sun.jdbc.rowset.CachedRowSet`)
- JDBC rowsets (`sun.jdbc.rowset.JdbcRowSet`)
- Web rowsets (`sun.jdbc.rowset.WebRowSet`)

These are specified in the `sun.jdbc.rowset` package, which is not part of the JDBC API. Currently, Sun provides a reference implementation of these RowSet objects in an Early Access release (currently release 4), which can be downloaded from Sun's Java Developer Connection web site (<http://java.sun.com/jdc/>). This early-access release includes implementations of these tree types of RowSet object.

All the above classes extend from the `sun.jdbc.rowset.BaseRowSet`, and implement the `javax.sql.RowSet` interface. The `BaseRowSet` class provides the common implementation for properties, events, and methods for setting parameters for RowSet objects (since rowset objects implement the `java.sql.ResultSet` interface).



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Summary

The purpose of this chapter has been to provide an overview of database programming using the JDBC API, and then introduce certain advanced concepts related to transactions and databases that are more relevant in an enterprise platform such as J2EE.

As mentioned at the beginning of this chapter, the goal of the first part of the chapter has been to discuss how to perform typical SQL operations using this API. The JDBC API is quite extensive and provides more facilities than can be covered in one chapter. Most of these features (such as batch updates, scrollable resultsets, etc.) are comparatively new, and most of the database vendors are only beginning to support these features. We're encouraged to check the documentation from the vendor before we plan to implement these features.

To summarize, in the first part of this chapter, we discussed the following:

- Database drivers, and how to load JDBC drivers
- Creating tables, and inserting data
- Prepared statements
- Mapping between SQL and Java types
- Batch updates
- Scrollable resultsets
- Save points

In the second part of this chapter, we discussed four advanced features of the JDBC API:

- The JDBC client applications need not be hard-wired to specific JDBC drivers, and to use vendor-specific JDBC URLs. Instead, the `DataSource` object, combined with the JNDI-based binding and lookup decouples vendor-specific database details from the client applications.
- The `javax.sql` package shifts the responsibility of connection pooling to J2EE application servers and JDBC drivers. With this approach, connection pooling is a matter of configuring the application server – there is no need to implement custom connection pools, as this mechanism offers a more reliable means of connection pooling.
- Distributed transaction processing has never been so simple. Using the `javax.transaction.UserTransaction` interface, we can explicitly demarcate transactions, while still using the standard JDBC API for database access.
- The `RowSet` interface and its implementations are an addition to the JDBC API. The goal of the `RowSet` is to provide for more flexible, bean-like data access. Sun is also working on Java Data Objects (JDO), presumably based on Microsoft's Active Data Objects. JDOs are expected to complement the JDBC API and provide database access at a higher level without using SQL. The JDO API is currently under development, and we could expect more advanced data access facilities in future. Look for announcements at <http://java.sun.com/products/jdbc/>.

In the next chapter we'll take a look at web applications, and the containers that manage them.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The J2EE architecture offers a richly featured and flexible programming model for building dynamic web applications. As we discussed in Chapter 1, the J2EE architecture provides **web containers**, the Java Servlet API, and the JavaServer Pages API for the building and management of web applications. The web container provides the basic runtime environment and a framework for providing runtime support for web applications. The Java servlet and JSP technologies form the core material for the development of web applications.

This chapter introduces us to J2EE web containers, and the basics of web application development, in three distinct steps:

- We will discuss the basic anatomy of a web container. Starting with a brief introduction to HTTP, the emphasis will be on introducing the concepts of web containers, Java servlets, and JSP pages. We'll continue to learn more about these topics in subsequent chapters.
- We will walk through the creation of a simple but complete web application.
- We will discuss, in detail, the inner working of the example web application.

We begin by looking at how clients and servers in a web application communicate using the HTTP protocol.

The HTTP Protocol

In distributed application development, the communication protocol determines the nature of clients and servers and the relationship between them. This is also true in the case of web-based applications. The complexity of many of the features possible in our web browser and on the web server depends on the underlying protocol – that is, the **Hypertext Transfer Protocol (HTTP)**.

Copyrighted image

Clients (such as web browsers) send requests to the server (such as the web server of an online store) to receive information (such as downloading a catalog), or to initiate specific processing on the server (such as placing an order).

HTTP Request Methods

As an application-level protocol, HTTP defines the types of request that clients can send to servers and the types of response that servers can send to clients. The protocol also specifies how these requests and responses are structured.

HTTP/1.0 specifies three types of request methods: **GET**, **POST**, and **HEAD**. HTTP/1.1 has five additional request methods: **OPTIONS**, **PUT**, **TRACE**, **DELETE**, and **CONNECT**. Of these, the GET and POST requests meet most common application-development requirements.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

To better understand the role of application logic in the request-response process, consider a web-based mail server. When we log in to our favorite web-based mail server, the server should be able to send a page with links to our mail, our mail folders, our address book, and so on. This information is dynamic, in the sense that each user expects to see their individual mailbox. To generate such content, the server must execute complex application logic to retrieve the mail and compose a page.

Clients can send client-specific or context information in the requests to the server, but how can the server decide how the content should be generated? HTTP does not define a standard means for embedding application logic during the response generation phase. There is no programming model specified for such tasks. HTTP defines how clients can request information, and how servers can respond, but it is not concerned with how the response could be generated.

This is where the benefits of server-side technologies such as Java servlets and JavaServer Pages become apparent. With these technologies, we can embed custom application logic during the request processing and response generation stages.

Java servlets are not user-invoked applications. Instead, the web container in which the web application containing the servlets is deployed invokes the servlets based on incoming HTTP requests. When a servlet has been invoked, the web container forwards the incoming request information to the servlet, so that the servlet can process it, and generate a dynamic response. The web container interfaces with the web server by accepting requests for servlets, and transmitting responses back to the web server.

When compared to CGI, and proprietary server extensions such as NSAPI or ISAPI, the servlet framework provides a better abstraction of the HTTP request-response paradigm by specifying a programming API for encapsulating requests and responses. In addition, servlets have all the advantages of the Java programming language, including platform-independence. Java servlet-based applications can be deployed on any web server with built-in (in-process) or connector-based (out-of-process) web containers, irrespective of the operating system and the hardware platform.

In order to understand how a servlet interacts with a web server via a web container, let's consider the basic invocation process with the web server receiving an HTTP request. As we've seen before, the HTTP protocol is based on a request-response model. A client connects to a web server and sends an HTTP request over the connection. Based on the request URL, the following sequence of events happens in a typical sequence (the rightward arrows indicate requests and leftward arrows indicate responses):

Copyrighted image



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The **WEB-INF** directory is a private area and the container will not serve contents of this directory to users. The files in this are meant for container use only. **WEB-INF** contains, among other things, the deployment descriptor, a **classes** directory, and a **lib** directory. The **classes** directory is used to store compiled servlets and other utility classes. If an application has packaged JAR files (for example, a third-party API packaged as a JAR file), they can be copied into the **lib** directory. The web container uses these two directories to locate servlets and other dependent classes.

Types of Web Containers

There are essentially three ways to configure web containers. Recall from Chapter 1 that a web container either implements the underlying HTTP services, or delegates such services to external web servers:

- **Web container in a J2EE application server**
Most of the commercial J2EE application servers such as WebLogic, Inprise Application Server, iPlanet Application Server, and WebSphere Application Server now include web containers built in.
- **Web containers built into web servers**
This is the case with pure Java web servers that contain integrated web containers. Jakarta Tomcat (from <http://jakarta.apache.org>), which is the web container reference implementation, also falls into this category. Tomcat includes a web server along with a web container.
- **Web container in a separate runtime**
In the non-J2EE world, this is the most common scenario. Web servers such as Apache, or Microsoft IIS require a separate Java runtime to run servlets, and a web server plug-in to integrate the Java runtime with the web server. The plug-in handles communication between the web server and the web container. Commercially available servlet/JSP engines such as JRun from Allaire, and ServletExec from New Atlanta (now part of Unify's eWave product family) provide plug-ins to integrate with web servers.

The choice of a type of container depends entirely on the requirements of our application. For instance, if we were only interested in building a new Java-based web application, we would choose the second or the third configuration. However, with the unification of web and enterprise computing standards via the J2EE, the first configuration is becoming increasingly common.

A Simple Web Application

Now that we've seen the broad description of the J2EE web container architecture and web applications, let's build a web application so that we can walk through the basic aspects of developing web applications with J2EE.

Our web application will do two things:

- Prompt the user for a name and an e-mail address
- Print a welcome greeting based on the time of the day



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

| Purpose | Class/Interface |
|------------------------|---|
| Servlet Implementation | <code>javax.servlet.Servlet</code> <code>javax.servlet.SingleThreadModel</code> <code>javax.servlet.GenericServlet</code> <code>javax.servlet.http.HttpServlet</code> |
| Servlet Configuration | <code>javax.servlet.ServletConfig</code> |
| Servlet Exceptions | <code>javax.servlet.ServletException</code> <code>javax.servlet.UnavailableException</code> |
| Requests and Responses | <code>javax.servlet.http.HttpServletRequest</code> <code>javax.servlet.http.HttpServletRequestWrapper (*)</code> <code>javax.servlet.http.HttpServletResponse</code> <code>javax.servlet.http.HttpServletResponseWrapper</code> <code>javax.servlet.ServletInputStream (*)</code> <code>javax.servlet.ServletOutputStream</code> <code>javax.servlet.ServletRequest</code> <code>javax.servlet.ServletRequestWrapper (*)</code> <code>javax.servlet.ServletResponse</code> <code>javax.servlet.ServletResponseWrapper (*)</code> |
| Session Tracking | <code>javax.servlet.http.HttpSession</code> <code>javax.servlet.http.HttpSessionActivationListener (*)</code> <code>javax.servlet.http.HttpSessionAttributeListener (*)</code> <code>javax.servlet.http.HttpSessionBindingListener</code> <code>javax.servlet.http.HttpSessionBindingEvent</code> <code>javax.servlet.http.HttpSessionEvent (*)</code> <code>javax.servlet.http.HttpSessionListener (*)</code> |
| Servlet Context | <code>javax.servlet.ServletContext</code> <code>javax.servlet.ServletContextAttributeEvent (*)</code> <code>javax.servlet.ServletContextAttributeListener (*)</code> <code>javax.servlet.ServletContextEvent (*)</code> <code>javax.servlet.ServletContextListener (*)</code> |
| Servlet Collaboration | <code>javax.servlet.RequestDispatcher</code> |
| Filtering | <code>javax.servlet.Filter (*)</code> <code>javax.servlet.FilterChain (*)</code> <code>javax.servlet.FilterConfig (*)</code> |
| Miscellaneous | <code>javax.servlet.http.Cookie</code> <code>javax.servlet.http.HttpUtils</code> |



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

- Performing cleanup tasks, such as closing any open resources, closing a connection pool, or even informing another application/system that the servlet will no longer be in service
- Persisting any state associated with a servlet

The getServletConfig() Method

```
public ServletConfig getServletConfig()
```

This method should be implemented to return the `ServletConfig` that was passed to the servlet during the `init()` method.

The getServletInfo() Method

```
public String getServletInfo()
```

This method should return a `String` object containing information about the servlet (for example, author, creation date, description, and so on). This is available to the web container, should it wish to display, for example, a list of servlets installed together with their descriptions.

The GenericServlet Class

```
public abstract class GenericServlet implements Servlet, ServletConfig,  
Serializable
```

The `GenericServlet` class provides a basic implementation of the `Servlet` interface. This is an abstract class, and all subclasses should implement the `service()` method. This abstract class has the following methods in addition to those declared in `javax.servlet.Servlet`, and `javax.servlet.ServletConfig`:

```
public init()  
public void log(String message)  
public void log(String message, Throwable t)
```

The `init(ServletConfig config)` method stores the `ServletConfig` object in a private transient instance variable (called `config`). You can use the `getServletConfig()` method to access this object. However, if you choose to override this method, you should include a call to `super.init(config)`. Alternatively, you can override the overloaded no-argument `init()` method in the `GenericServlet` class.

The `GenericServlet` class also implements the `ServletConfig` interface. This allows the servlet developer to call the `ServletConfig` methods directly without having to first obtain a `ServletConfig` object. These methods are `getInitParameter()`, `getInitParameterNames()`, `getServletContext()`, and `getServletName()`. Each of these methods delegates the calls to the respective methods in the stored `ServletConfig` object.

The `GenericServlet` class also includes two methods for writing to a servlet log, which call the corresponding methods on the `ServletContext`. The first method, `log(String msg)`, writes the name of the servlet and the `msg` argument to the web container's log. The other method, `log(String msg, Throwable cause)`, includes a stack trace for the given `Throwable` exception in addition to the servlet name and message. The actual implementation of the logging mechanism is container-specific, although most of the containers use text files for logging purposes.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Servlet Configuration

In the Java Servlet API, `javax.servlet.ServletConfig` objects represent the configuration of a servlet. The configuration information contains initialization parameters (a set of name/value pairs), the name of the servlet, and a `javax.servlet.ServletContext` object, which gives the servlet information about the container. The initialization parameters and the name of a servlet can be specified in the deployment descriptor (the `web.xml` file), for example:

```
<web-app>
  <servlet>
    <servlet-name>Admin</servlet-name>
    <servlet-class>com.apress.admin.AdminServlet</servlet-class>
    <init-param>
      <param-name>email</param-name>
      <param-value>admin@admin.apress.com</param-value>
    </init-param>
    <init-param>
      <param-name>helpURL</param-name>
      <param-value>/admin/help/index.html</param-value>
    </init-param>
  </servlet>
<web-app>
```

This example registers a servlet with name `Admin`, and specifies two initialization parameters, `email` and `helpURL`. The web container reads this information, and makes it available to the `com.apress.admin.AdminServlet` via the associated `javax.servlet.ServletConfig` object. If we want to change these parameters, we can do so without having to recompile the servlet. We'll learn more about this approach in Chapter 9.

The deployment descriptor mechanism was introduced in the Servlet 2.2 specification. Web containers (then called servlet engines) complying with older versions of the specification provide vendor-specific means (such as properties files) to specify the initialization parameters.

The `ServletConfig` Interface

```
public interface ServletConfig
```

As we saw when we looked at the `GenericServlet` class, this interface specifies the following methods:

```
public String getInitParameter(String name)
public Enumeration getInitParameterNames()
public ServletContext getServletContext()
public String getServletName()
```

The `getInitParameter()` Method

```
public String getInitParameter(String name)
```

This method returns the value of a named initialization parameter, or `null` if the specified parameter does not exist. In the above example, calling the `getInitParameter()` method with "email" as an argument returns the value "admin@admin.apress.com".



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The container creates a servlet instance in response to an incoming HTTP request, or at container startup. After instantiation, the container initializes the instance by invoking its `init()` method. After initialization, the servlet instance is ready to serve incoming requests. The purpose of this initialization process is to load any initialization parameters required for the servlet. We'll see how this can be accomplished in the next section.

During the initialization process, a servlet instance can throw a `ServletException`, or an `UnavailableException`. The `UnavailableException` is a subclass of `ServletException`. While the `ServletException` can be used to indicate general initialization failures (such as failure to find initialization parameters), `UnavailableException` is for reporting non-availability of the instance for servicing requests. For example, if your servlet depends on an RMI server, and you're verifying if the server is reachable for service, your servlet instance can throw a `UnavailableException` to indicate that it is temporarily or permanently unavailable. If your servlet instance determines that the unavailability should be temporary, it may indicate so while constructing the `UnavailableException`, by specifying the number of seconds of unavailability to the exception's constructor. When such a failure occurs, the container suspends all requests to your servlet for the specified period, and brings it back to an available state at the end of the period. If you don't specify an unavailability setting internally, the servlet will be unavailable when the container restarts. One of the applications of this exception is when one of your backend components or systems is not accessible temporarily. In general, if your application logic is such that certain failures can be resolved by retrying after a while, you can use this exception.

The container guarantees that before the `service()` method is called, the `init()` method will be allowed to complete, and also that before the servlet is destroyed, its `destroy()` method will be called.

The servlet may throw a `ServletException` or an `UnavailableException` during its `service()` method, in which case the container will suspend requests for that instance either temporarily or permanently. It is important for you to design your servlets considering temporary or permanent failures.

In theory, there is nothing to stop a web container from performing the entire servlet lifecycle each time a servlet is requested. In practice, web containers load and initialize servlets during the container startup, or when the servlet is first called, and keep that servlet instance in memory to service all the requests it receives. The container may decide at any time to release the servlet reference, thus ending the servlet's lifecycle. This could happen, for example, if the servlet has not been called for some time, or if the container is shutting down. When this happens, the container calls the `destroy()` method.

In the typical servlet lifecycle model, the web container creates a single instance of each servlet. But what happens if the servlet's `service()` method is still running when the web container receives another request? For servlets that do not implement the `javax.servlet.SingleThreadModel` interface, the container invokes the same servlet instance in each request thread. Therefore, it is always possible that the `service()` method is being executed in more than one service thread, requiring that the `service()` method be thread-safe. Apart from not accessing thread-safe resources (such as writing to files), we should also consider keeping our servlets stateless (that is, not define any attributes in your servlet classes). When defining instance variables in our servlets, we should make sure that such variables are manipulated in a thread-safe manner.

We've previously discussed the behavior of servlets implementing the `javax.servlet.SingleThreadModel` interface. Recall that, for such servlets, the container could either serialize requests, or maintain a pool of servlet instances and allocate each request to a different instance in the pool, or use a combination of these two.

The following diagram shows the sequence of events showing a servlet being loaded, servicing two requests in quick succession, and then being unloaded when the server is shut down:



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Copyrighted image

If the generated random number is `false`, `FreakServlet` generates a page displaying the states recorded so far. It uses methods in the `HttpServletRequest` interface to set the content type of the response, and generate the HTML:

```
states.add(createState("Service"));

response.setContentType("text/html");
PrintWriter out = response.getWriter();

// Send acknowledgment to the browser
out.println("<html>");
out.println("<meta http-equiv=\"Pragma\" content=\"no-cache\">");
out.println("<head><title>");

out.println("FreakServlet: State History");
out.println("</title></head>");
out.println("<body>");
out.println("<h1>FreakServlet: State History</h1>");

out.println("<a href=\"/lifeCycle/servlet/freak\">Reload</a></p>");

for (int i = 0; i < states.size(); i++) {
    out.println("<p> " + states.elementAt(i) + "</p>");
}

out.println("</body></html>");
out.close();
}
```

There are four steps for generating the response:

- The first step is to set the content type for the response. The receiving application (the browser) uses this information to know how to treat the response data. In this case, since we're generating HTML output, the content type is being set to "`text/html`".



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Methods for Buffered Output

You can also send buffered response from your servlets. The following methods are useful for controlling the buffering.

If you're sending a large amount of data in the response, you should consider setting the buffer size to smaller values, so that the user can start receiving the data quickly.

Buffering also allows you to abort the content generated so far, and restart the generation.

The setBufferSize() Method

```
public void setBufferSize(int size)
```

This method sets the preferred buffer size for the body of the response. Note that the web container will use a buffer at least as large as the size requested.

The getBufferSize() Method

```
public int getBufferSize()
```

This method returns the actual buffer size used for the response.

The resetBuffer() Method

```
public void resetBuffer()
```

This method clears content of the underlying buffer without clearing the response headers or the status code. If the response has already been committed, this method throws an `IllegalStateException`.

The flushBuffer() Method

```
public void flushBuffer() throws java.io.IOException
```

This method forces any content in the buffer to be written to the client.

The isCommitted() Method

```
public boolean isCommitted()
```

This method returns a boolean indicating if the response in the buffer has been committed.

The reset() Method

```
public void reset()
```

This method is useful for resetting the buffer thereby discarding the content in the buffer.

The ServletResponseWrapper Class

```
public class ServletResponseWrapper implements ServletResponse
```

The class provides a convenient implementation of the `javax.servlet.ServletResponse` interface. Except for a constructor, this class does not introduce any new methods. We shall shortly see the role of these wrapper classes.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

A placeholder for a copyrighted image, likely a screenshot of a software interface.

That's all we need to set in the wizard so you can either press the **Finish** button now, or keep going to the end of the wizard to see the deployment descriptor. Once you exit the wizard, you'll see the newly created web component in the main window:

A placeholder for a copyrighted image, likely a screenshot of a software interface showing the newly created web component.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Copyrighted image

We should have set the JNDI name for the datasource when we created the application, but in case you didn't you can add it in here. Press **Next** to move to the next screen:

Copyrighted image



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Summary

The Java Servlet API is both simple and powerful. It allows us to extend the functionality of any web server, with the help of a simple programming model.

The objective of this chapter has been to introduce the Servlet API and the lifecycle of servlets, and to demonstrate how to write servlet-based web applications. In this process, we've covered the following:

- ❑ When building a servlet, you need to implement the `Servlet` interface. You can do this by extending either `GenericServlet` or `HttpServlet`.
- ❑ The `HttpServlet` class extends `GenericServlet`, and provides additional HTTP-specific functionality.
- ❑ Servlets can implement the `SingleThreadModel` interface for enforcing synchronized access to the service methods. However, servlets that do not implement this interface should make sure that any servlet instance variables are accessed in a thread-safe manner. We've discussed the implications with the `FreakServlet` and the `TechSupportServlet`.
- ❑ The servlet lifecycle involves the `init()`, `service()`, and `destroy()` methods. The `FreakServlet` demonstrates the states associated with the lifecycle of a servlet instance.
- ❑ We use deployment descriptors to specify initialization parameters. This helps us avoid hard-coding such parameters within the servlets.
- ❑ You can use error pages to automatically send pre-designed HTML pages in response to HTTP errors and exceptions. This is a very flexible approach, and it is a good practice to do adequate error handling to prevent unfriendly container-generated messages from being sent to clients.

In the next chapter, we'll look at another part of the Servlet API, which deals with sessions, context and servlet collaborations.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

For implementing flexible business transactions across multiple requests and responses, we need two facilities:

Session

The server should be able to identify that a series of requests from a single client form a single working 'session'. By associating a specific request to belong to a specific working session, the shopping cart or the online banking application can distinguish one user from another.

State

The server should be able to remember information related to previous requests and other business decisions that are made for requests. That is, the application should be able to associate state with each session. In the case of the shopping cart application, possible state could include the user's favorite item categories, user profile, or even the shopping cart itself.

However, in the case of HTTP, connections are closed at the end of each request, and hence HTTP servers cannot use the notion of connections to establish a session. HTTP is a stateless protocol, concerned with requests and responses, which are simple, isolated transactions. This is perfect for simple web browsing, where each request typically results in downloading static content. The server does not need to know whether a series of requests come from the same, or from different clients, or whether those requests are related or distinct. But this is not the case with web applications, where we need the ability to perform business transactions across multiple requests and responses.

With HTTP, in a transaction that spans multiple requests and responses, the web server cannot determine that all the requests are from the same client. A client, therefore, cannot establish a dialogue with the web server to perform a business transaction. However, the goal of HTTP has been to provide speedy and light information retrieval across the Internet, and a stateless protocol is the most suitable for such requirements.

Apart from being able to track users based on the notion of sessions, the server should also be able to remember any necessary information within a session. That is, the application programmer should be able to specify what data should be remembered within a given session, so that the application can use such information to make more informed decisions. This is very useful for transactions or processes spanning multiple requests and responses. With stateful protocols, the server associates a state with the connection: it remembers who the user is, and what it is that they're doing, with the help of the connection. However, this cannot be achieved with HTTP, as the lifetime of a connection is limited to a single request.

Over time, several strategies have evolved to address session tracking, and to manage state within a session. The Java Servlet API provides facilities for tracking sessions and maintaining state within a given session. With the help of these facilities, the server can associate all of the requests together and know that they all came from the same user. It can also associate a state with the connection: it remembers who the user is, and what it is that they're doing.

Note that HTTP 1.1 (<http://www.w3.org/Protocols/rfc2068/rfc2068>) provides for 'persistent connections', in which case clients and servers can use the same connection object for multiple requests/responses. When supported by both clients and servers, persistent connections reduce the latency associated with creating TCP/IP connections. Persistent connections are useful in cases where requests and responses occurring in a quick succession can be optimized using a single connection instead of one connection per request. However, persistent connections do not carry over session state. The discussion in this chapter remains the same with persistent connections.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Here **Name** is the name of the cookie, and **VALUE** is the value of Name. **Max-age** (optional) specifies the maximum life of the cookie in seconds. **Domain** (optional) and **Path** (optional) specify the URL path for which the cookie is valid, and **secure** (optional) specifies if the cookie can be exchanged over HTTP. For cookies implemented as per RFC 2109, the **Version** should be set to 1. The **Version** should be set to 0 for cookies implemented as per the original Netscape specification. **Comment** is an optional parameter that can be used to document the intent of the cookie.

Here is an example of a cookie:

```
Set-cookie: uid=joe; Max-age=3600; Domain=".myserver.com"; Path="/"
```

The above response header sends a cookie with name **uid** and value **joe**. The lifetime of this cookie is 3600 seconds, and is valid for the **myserver.com** domain (including all sub-domains) for the URL path **/**. The browser should discard this cookie after 3600 seconds. If the **Max-age** attribute is missing, the browser discards the cookie when you exit the browser.

Instead of specifying **.myserver.com** as the domain, you can also specify specific sub-domains such as **www.myserver.com** or **some.myserver.com** etc., in which case, the browser returns the cookie only to the specified sub-domains. For instance, information portals such as Yahoo specify entire the entire **.yahoo.com** domain for when setting cookies, so that Yahoo can identify/track you whether you're visiting **http://my.yahoo.com**, or shopping at **http://shop.yahoo.com**, or checking e-mail at **http://mail.yahoo.com**.

When a browser client receives the above response header, it can either reject it or accept it. For instance, you can configure your browser to accept or reject cookies. Let's consider that the browser accepts this cookie. When the browser sends a request to the **http://www.myserver.com** domain within the next one hour, it also sends a request header:

```
Cookie: uid=joe
```

The server can read this cookie from the request, and identify that the request corresponds to a client identified by **uid=joe**. This completes the token exchange necessary for tracking sessions.

A cookie is specific to a domain or a sub-domain, and can be set by the **domain** attribute in the cookie. Browsers use the **domain** and **path** attributes to determine if a cookie should be sent in the request header, and with what name-value attributes. Once accepted by a browser, the browser stores the cookie against the domain and the URL path.

If the client chooses to reject a cookie, the client does not send the cookie back to the server in the request header, and therefore the server fails to track the user session.

Note that the main advantage of this approach is that the cookies are not mixed with the HTML content and the HTTP request and response bodies. The container can transparently set cookies in the response headers, and extract cookies from request headers.

The Servlet API specification requires that web containers implement session tracking using the cookie mechanism. In this approach, the web container automatically sets a session tracking cookie with name **jsessionid**. When the container receives client requests, it checks for existence of this cookie, and accordingly track sessions.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
        String lifeCycleURL = "/session/servlet/lifeCycle";
        out.print("<br><a href=\"" + lifeCycleURL + "?action=invalidate\">");
        out.println("Invalidate the session</a></td></tr>");
        out.print("<br><a href=\"" + lifeCycleURL + "\">");
        out.println("Reload this page</a>");
        out.println("</body></html>");
        out.close();
    }
}
}
```

Consider the highlighted parts of this source code. Let's look at the final `else` block first. This block is executed when servlet is invoked without any parameters, and performs the following steps:

- Calls `getSession()` on the `HttpSession`, with boolean `true` as an argument.
- Calls `getId()` to get the session ID.
- Calls `getCreationTime()` to get the creation time of the session. Since this method returns the creation as milliseconds since January 1, 1970 00:00:00 GMT, we need to convert this into a `Date` object using the new constructor.
- Calls `getLastAccessedTime()` to get the session's last accessed time.
- Calls `getMaxInactiveInterval()` to get the current max-inactive-interval setting.

After printing the above, the servlet also generates two links: one to reload the page, and the other to invalidate the session. The first link simply points to the same page. The `no-cache` meta-tag in the generated HTML lets you reload the page by clicking on this link.

Note that the second link has a query string `action=invalidate` appended. When you click on this link, the `if` block of the `doGet()` method will be executed. In order to see this servlet in action, compile this servlet and create the following deployment descriptor:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC
  "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  "http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">

<web-app>
  <servlet>
    <servlet-name>lifeCycle</servlet-name>
    <display-name>lifeCycle</display-name>
    <servlet-class>SessionLifeCycleServlet</servlet-class>
  </servlet>

  <session-config>
    <session-timeout> <!-- In minutes -->
      5
    </session-timeout>
  </session-config>
</web-app>
```

From now on I will assume that you know how to deploy a web component into the J2EE Reference Implementation server, and so I will only highlight those steps that are particular to each example. Refer back to the preceding chapters for more complete instructions on the process.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
int itemCount = 0;
ArrayList cart = (ArrayList) session.getAttribute("cart");
if (cart != null) {
    itemCount = cart.size();
}

res.setContentType("text/html");
PrintWriter out = res.getWriter();

out.println("<html><head><title>Simple Shopping Cart "
+ "Example</title></head>");
out.println("<body><table border=\"0\" width=\"100%\"><tr>");
out.println("<td valign=\"top\"><img "
+ "src=\"/cart/images/logo.gif\"></td>");
out.println("<td align=\"left\" valign=\"bottom\">");
out.println("<h1>APRESS Book Store</h1></td></tr></table><hr>");
out.println("<p>You've " + itemCount + " items in your cart.</p>");
out.print("<form action=\"\"");
out.println(res.encodeURL("/cart/servlet/cart"));
out.println("\" method=\"post\"");
out.println("<table cellspacing=\"5\" cellpadding=\"5\"><tr>");
out.println("<td align=\"center\"><b>Add to Cart</b></td>");
out.println("<td align=\"center\"></td></tr><tr>");
out.println("<td align=\"center\">");
out.println("<input type=\"Checkbox\" name=\"item\""
+ " value=\"Pro Java Programming, Second Edition\"></td>");
out.println("<td align=\"left\">Item 1: "
+ " Pro Java Programming, Second Edition</td></tr><tr>");
out.println("<td align=\"center\">");
out.println("<input type=\"Checkbox\" name=\"item\""
+ " value=\"Beginning Java Objects\"></td>");
out.println("<td align=\"left\">Item 2: "
+ " Beginning Java Objects</td></tr><tr>");
out.println("<td align=\"center\">");
out.println("<input type=\"Checkbox\" name=\"item\""
+ " value=\"Professional Java Server Programming\"></td>");
out.println("<td align=\"left\">Item 3: Professional Java "
+ "Server Programming</td></tr>");
out.println("</table><hr>");
out.println("<input type=\"Submit\" name=\"btn_submit\" "
+ "value=\"Add to Cart\">");
out.println("</form></body></html>");

out.close();
}
}
```

This servlet obtains the number of books from data stored in the `HttpSession`. It starts by using the `getSession()` method on the `HttpServletRequest` object to get the current session:

```
HttpSession session = req.getSession();
```

This is the first step required for creating and tracking HTTP sessions. You'll also find a similar `req.getSession()` call in the `ShoppingCart` servlet.

It then uses a `HttpSession` attribute "cart" to find and display the number of books in the cart. As we'll see in a moment, the `ShoppingCart` servlet stores and updates this attribute when items are added to the cart, using an `ArrayList` object to store the list of selected books.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

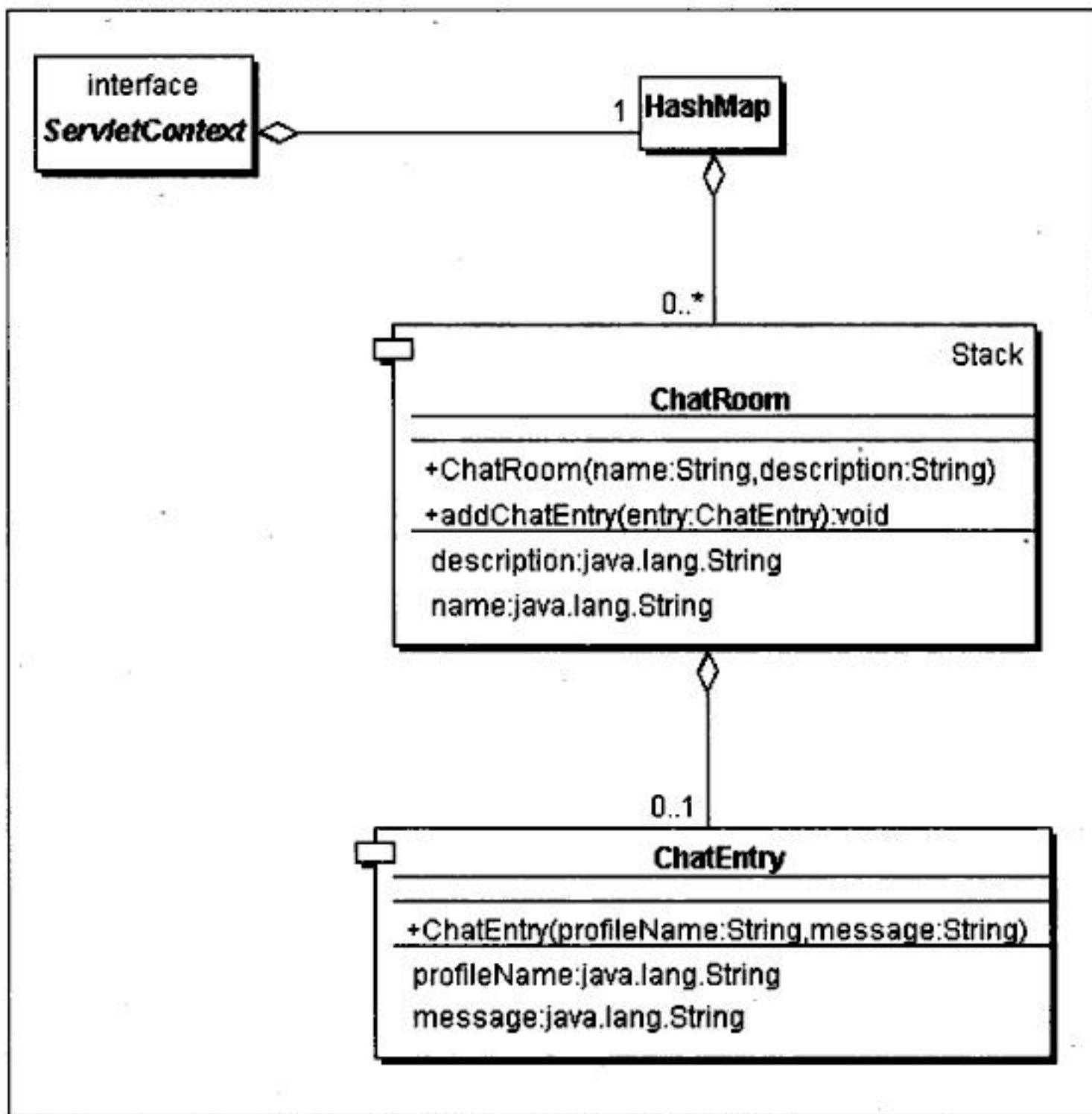


You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

In order to build such an application, we need to model certain classes:

- A `ChatRoom` class to represent a chat room.
Since a `ChatRoom` is shared across multiple users (and therefore multiple sessions), the `ServletContext` is the right place to store a `ChatRoom`. Since there can be more than one `ChatRoom`, we need to maintain a list of `ChatRooms` in the `ServletContext`.
- A `ChatEntry` to represent a chat message.
In a chat room, multiple messages are exchanged between users. A `ChatEntry` is part of a `ChatRoom`, and the `ChatRoom` class maintains a list of `ChatEntry` objects.

The diagram below gives an overview of relationships between these classes:



A `java.util.HashMap` class is used to hold a map of `ChatRoom` objects, and each `ChatRoom` contains a stack of `ChatEntry` objects. The `HashMap` object is the entry point to all the other objects, so a reference to the `HashMap` is stored as a `ServletContext` attribute under the name `roomList`.

The `HashMap`, `ChatRoom`, and `ChatEntry` objects together represent the state of the chat application. These classes represent the data model for this application.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
String chatRoomPath;
String listRoomsPath;
String chatAdminPath;

public void init() throws ServletException {
    ServletContext context = getServletContext();
    chatRoomPath = context.getInitParameter("CHATROOM_PATH");
    listRoomsPath = context.getInitParameter("LISTROOMS_PATH");
    chatAdminPath = context.getInitParameter("ADMIN_PATH");
    if(chatRoomPath == null || listRoomsPath == null ||
       chatAdminPath == null) {
        throw new UnavailableException("Application unavailable.");
    }
}

public void doGet (HttpServletRequest req, HttpServletResponse res)
    throws IOException, ServletException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
    out.println("<html>");
    out.println("<head><title>Chat Room Administration</title></head>");
    out.println("<body>");
    out.println("<h1>Chat room administration</h1>");
    out.println("<form method=\"POST\" action=\""
               + res.encodeURL(chatAdminPath) + "\">");
    // Check for existing chat rooms
    HashMap roomList =
        (HashMap) getServletContext().getAttribute("roomList");
    if (roomList != null) {
        Iterator rooms = roomList.keySet().iterator();

        if (!rooms.hasNext()) {
            out.println("<p>There are no rooms</p>");
        } else {
            out.println("<p>Check the rooms you would like to remove,"
                       + "and press Update List.</p>");

            while (rooms.hasNext()) {
                String roomName = (String) rooms.next();
                ChatRoom room = (ChatRoom) roomList.get(roomName);
                out.println("<input type=checkbox name=remove value='"
                           + room.getName() + "'>" + room.getName() + "<br>");
            }
        }
    }

    // Add fields for adding a room
    out.println("<p>Enter a new room and the description</p>");
    out.println("<table>");
    out.println("<tr><td>Name:</td><td><input name=roomname " +
               "size=50></td></tr>");
    out.println("<tr><td>Description:</td></tr>");
    out.println("<td><textarea name=roomdescr cols=40 rows=5>" +
               "</textarea></td></tr>");
    out.println("</table>");

    // Add submit button
    out.println("<p><input type=submit value='Update List'>\"");
    out.println("<p><a href=\"" + listRoomsPath + "\">Chat Now</a>\"");
    out.println("</form>");

    out.println("</body></html>");
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```

// Add radio boxes for selecting a room
out.println("Select the room you would like to enter,"
           + " or click on a name to see the description:<p>");

Iterator rooms = roomList.keySet().iterator();
boolean isFirst = true;
while (rooms.hasNext()) {
    String roomName = (String) rooms.next();
    ChatRoom room = (ChatRoom) roomList.get(roomName);
    String listRoomsURL = listRoomsPath + "?expand="
                           + URLEncoder.encode(roomName);
    listRoomsURL = response.encodeURL(listRoomsURL);

    out.println("<input type=radio name=roomName value=\"" + roomName
               + "\" " + (isFirst ? " CHECKED" : "") + ">"
               + "<a href=\"" + listRoomsURL + "\">" + roomName
               + "</a><br>");
    isFirst = false;

    // Show description if requested
    if (expand != null && expand.equals(roomName)) {
        out.println("<blockquote>");
        if (room.getDescription().length() == 0) {
            out.println("No description available.");
        } else {
            out.println(room.getDescription());
        }
        out.println("</blockquote><br>");
    }
}

// Add a field for the profile name
out.println("<p>Enter your name: ");
out.println("<input name=profileName value='"
           + profileName
           + "' size=30>");

// Add submit button
out.println("<p><input type=submit value='Enter'>");
out.println("</form>");
}

out.println("</body></html>");
out.close();
}
}

```

The HTTP GET method is used to invoke the servlet, so we use the `doGet()` method to generate an HTML page with a form containing the list of rooms, the user name field, and an Enter button.

To generate the list of chat rooms, this method calls the `getAttribute()` method on the `ServletContext` to obtain the list of chat rooms. The servlet also creates a session for each user. Note that, as discussed in the previous example, all URLs are encoded to make use of URL rewriting in case cookies are not being accepted by chat users.

The ChatRoomServlet Class

As seen above, the `ListRoomsServlet` generates a page with a `<form>` tag with the action attribute set to the name of the `ChatRoomServlet`. The form contains a radio button control that holds the name of the selected room and a text field with the user's name, and uses the `POST` method to invoke the servlet again.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```

        out.println("<html>");
        out.println("<head><meta http-equiv=\"refresh\" content=\"5\"></head>");
        out.println("<body>");
        out.println("  <b>Room: " + room.getName() + "</b><br>  <b>Identity: " +
                   + profileName + "</b><br>");

        // List all messages in the room
        if (room.size() == 0) {
            out.println("<font color=red>There are no messages in this room " +
                       "yet</font>");
        } else {
            Iterator entries = room.iterator();
            while (entries.hasNext()) {
                ChatEntry entry = (ChatEntry) entries.next();

                String entryName = entry.getProfileName();
                if (entryName.equals(profileName)) {
                    out.print("<font color=blue>");
                }
                out.println(entryName + " : " + entry.getMessage() + "<br>");
                if (entryName.equals(profileName)) {
                    out.print("</font>");
                }
            }
        }
        out.println("</body></html>");
    }
}

```

When compared to the rest of the classes in this application, this servlet seems rather more complicated. The following screenshots shows you a chat session in progress:



The following key points should help you navigate through this class:

- ❑ Each screenshot above has two HTML frames in it. Of these, the top frame is used to display the messages posted in the room, while the bottom frame is used for the user to enter new chat messages. The HTML frameset is generated in the `writeFrame()` method, invoked in `doPost()`.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Enter `http://localhost:8000/chat` in your browser to access the welcome file created above. Use the administration link to create chat rooms.

To view the list of chat rooms and to join a chat room, you can use the Chat Now button on the admin page, or enter the URL `http://localhost:8000/chat/servlet/listRooms` directly. Use the radio buttons to select a chat room, enter your name, and click on Enter to enter the chat room.

In Chapter 9, we shall see how to protect the admin page, so that only a designated administrator can add/delete chat rooms.

Servlet Collaboration

In the typical servlet model, a servlet receives an HTTP request, executes some application logic, and prepares the response. This completes one request-response trip for the client. However, there are several scenarios in which this basic model is not adequate:

- ❑ A servlet receives an HTTP request from a client, processes application logic, and a JavaServer Page drives the response. In this case the servlet is not responsible for response generation. Instead, the JSP page is responsible for dynamic content.
- ❑ A servlet receives an HTTP request from a client, processes application logic partially, and hands over the request to another servlet. The second servlet completes the application logic, and either prepares the response, or requests a JSP page to drive the response.

In both the scenarios, the servlet is not completely responsible for processing a request. Instead, it delegates the processing to another servlet (or a JSP page, which is equivalent to a servlet at run time).

There are two types of solution for addressing the above requirements:

- ❑ **Servlet chaining**
This was once a very widely used approach, and supported by some of the servlet engine vendors. Although this is not supported by the Java Servlet API specification, for the sake of completeness, you'll find a short description below.
- ❑ **Request Dispatching**
Request dispatching allows one servlet to dispatch the request to another resource (a servlet, a JSP page, or any other resource). Prior to version 2.2 of the Servlet API, this approach used to be called 'inter-servlet communication'. The API used to provide a method to get an instance of another servlet using a name. See the documentation of the now deprecated `getServlet()` method of the `javax.servlet.ServletContext` interface. From version 2.2 of the API, request dispatchers replace this functionality.

Servlet Chaining

Servlet chaining predates J2EE and its component model. The idea of servlet chaining is very simple: you design a set of servlets, each of which does a single task. After developing these servlets, you configure your servlet engine to specify a chain of servlets for a given URL path alias. Once the servlet engine receives a request for this alias, it invokes the servlets in the specified order. This is similar to piping on Unix, where output of one program becomes input for another program in the pipe.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
<td>
    <select name="software" SIZE="1">
        <option value="Word">Microsoft Word</option>
        <option value="Excel">Microsoft Excel</option>
        <option value="Access">Microsoft Access</option>
        <option value="Outlook">Microsoft Outlook</option>
    </select>
</td>
<td align="right">Operating System:</td>
<td>
    <select name="os" size="1">
        <option value="95">Windows 95</option>
        <option value="98">Windows 98</option>
        <option value="NT">Windows NT</option>
        <option value="2KPro">Windows 2000 Pro</option>
        <option value="2KServer">Windows 2000 Server</option>
        <option value="XP">Windows XP</option>
    </select>
</td>
</tr>
</table>

<br>Problem Description
<br>
<textarea name="problem" cols="50" rows="4"></textarea>

<hr><br>
<input type="Submit" name="submit" value="Submit Request">
</form>
</center>
</body>
</html>
```

This produces the form shown below:

Technical Support Request

Email:

Software:

Operating System:

Copyrighted Image

Problem Description



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

RegisterCustomerServlet

This servlet registers customer information. Upon registration, this servlet forwards the request to ResponseServlet:

```

// Import servlet packages
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.UnavailableException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpSession;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

// Import java packages
import java.io.PrintWriter;
import java.io.IOException;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import javax.sql.DataSource;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class RegisterCustomerServlet extends HttpServlet {

    public void doPost(HttpServletRequest request,
                        HttpServletResponse response)
        throws ServletException, IOException {

        HttpSession session = request.getSession();
        String fname = request.getParameter("txtFname");
        String lname = request.getParameter("txtLname");
        String email = (String) session.getAttribute("email");
        String phone = request.getParameter("txtPhone");

        Connection connection = null;
        String insertStatementStr = "INSERT INTO CUSTOMERS VALUES(?, ?, ?, ?)";
        try {
            InitialContext initial = new InitialContext();
            DataSource ds = (DataSource) initial.lookup("jdbc/TechSupport");
            connection = ds.getConnection();

            PreparedStatement insertStatement =
                connection.prepareStatement(insertStatementStr);
            insertStatement.setString(1, email);
            insertStatement.setString(2, fname);
            insertStatement.setString(3, lname);
            insertStatement.setString(4, phone);

            insertStatement.executeUpdate();

        } catch (NamingException ne) {
            throw new ServletException("JNDI error", ne);
        } catch (SQLException sqle) {
            throw new ServletException("Database error", sqle);
        }
        finally {
            if (connection != null) {
                try {
                    connection.close();
                } catch (SQLException sqle) {}
            }
        }
    }
}

```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```

</servlet>

<servlet>    <!-->
  <servlet-name>register</servlet-name>
  <display-name>register</display-name>
  <servlet-class>RegisterCustomerServlet</servlet-class>
</servlet>

<welcome-file-list>
  <welcome-file>techsupp.html</welcome-file>
</welcome-file-list>

<resource-ref>
  <res-ref-name>jdbc/TechSupport</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>

</web-app>

```

Of course we also need to modify the database. Delete the existing table and recreate it using the following SQL:

| | |
|---------------------------------------|-------------------------------------|
| CREATE TABLE SUPP_REQUESTS(REQUEST_ID | INT DEFAULT AUTOINCREMENT INITIAL 1 |
| | INCREMENT 1 NOT NULL, |
| EMAIL | VARCHAR(40), |
| SOFTWARE | VARCHAR(40), |
| OS | VARCHAR(40), |
| PROBLEM | VARCHAR(256)); |
| CREATE TABLE CUSTOMERS(EMAIL | VARCHAR(40) PRIMARY KEY, |
| FNAME | VARCHAR(15), |
| LNAME | VARCHAR(15), |
| PHONE | VARCHAR(12)); |

If you still have the Tech Support application from the previous chapter deployed, undeploy it and replace it with this one. Remember to make sure the datasource resource is referenced in the deployment.

To test the application, enter the URL <http://localhost:8000/techSupport>. You should be able to navigate through the flow depicted at the beginning of this section. Don't forget that you need to have the database running!

Using RequestDispatchers for Collaboration

In this application, we used the `RequestDispatcher` for the following purposes:

Forwarding a Request to ResponseServlet

The `TechSupportServlet` checks in the customer database whether the customer is already registered. If so, the request is forwarded to the `ResponseServlet`:

```

if(rs.next()) {
    String firstName = rs.getString("FNAME");
    String lastName = rs.getString("LNAME");
    request.setAttribute("firstName", firstName);
    request.setAttribute("lastName", lastName);
}

```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

8

Filters for Web Applications

Filters are the most recent addition to the Java servlet technology. The Servlet specification version 2.3 introduces filters as a flexible means of interacting with HTTP requests and responses before and after the web container invokes web resources including servlets.

In the range of J2EE technologies, filters stand apart. As discussed in Chapter 1, J2EE application components are container-managed. Remember that when a client makes a request to a J2EE component, such as a servlet, JSP page, or EJB, the container receives the requests and then invokes the appropriate component instances managed in the container runtime. In the case of the web container, it receives HTTP requests and invokes various web application resources including servlets, JSP pages, HTML files, images, and so on. That is, the web container is the *interceptor* for HTTP requests to resources. The interception process includes various steps including identifying the web application, finding path mappings to resources, checking for authentication and authorization (more about these in the next chapter), and invoking resources. With the various servlet features, there is no facility to participate in this interception process. You cannot change, control, or peek into the way the container invokes resources.

For instance, consider the basic need to monitor requests to a specific kind of static file (such as certain types of images or documents) on the container, and log the requests to a specific database. In order to build this functionality, what you need is a facility to plug in some logic during the HTTP request handling process. That is, you need a facility to participate in the interception process. The purpose of a filter is to provide this facility.

Why are filters special? As of J2EE 1.3, filters provide the only mechanism by which we can plug in code to participate in the container interception process. Although filters are limited to the web container (that is, we can filter HTTP requests, but we can't filter requests to EJBs, and so on), we can perform several useful tasks with the help of filters in the web container.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```

FilterConfig config;

public void init(FilterConfig config) {
    this.config = config;
}

public void doFilter(ServletRequest request,
                     ServletResponse response,
                     FilterChain chain)
    throws IOException, ServletException {

    ServletContext context = config.getServletContext();

    Integer count = (Integer) context.getAttribute("count");
    if(count == null) {
        count = new Integer(0);
    }

    count = new Integer(count.intValue() + 1);
    context.setAttribute("count", count);

    // Invoke the next filter (if any)
    chain.doFilter(request, response);
}

public void destroy() {}
}

```

This class creates a filter by implementing the `javax.servlet.Filter` interface, an interface that is very similar to that of a servlet. It has the usual `init()` and `destroy()` methods. The only difference is that the filter has a `doFilter()` method that is invoked during the interception process. We'll shortly study these methods in more detail. For the time being, we only need consider the code in the body of this method. This code uses the `FilterConfig` object to access the servlet context. When invoked first time, this filter sets a context attribute "count". During subsequent invocations, the filter updates the count in the context. It then requests that the container processes subsequent filters by calling the `doFilter()` method on the `chain` object. Since we have only one filter in this example, the last call results in sending the contents of the `index.html` file over the response.

But how does the container invoke the filter in the first place? Let's come back to this question after creating the rest of application.

The next step is to create the servlet to display the counter:

```

import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import java.io.IOException;
import java.io.PrintWriter;

public class DisplayCount extends HttpServlet {

```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Let's now study these interfaces and classes individually in more detail.

The Filter Interface

```
public interface Filter
```

This interface from the `javax.servlet` package encapsulates the lifecycle methods of a filter. The lifecycle methods include `init()` and `destroy()`, which are invoked during initialization and destruction of the filter, and a `doFilter()` method, which is invoked whenever there is a request/response pair to be filtered.

The init() Method

```
public void init(FilterConfig config) throws ServletException
```

The container invokes this method before putting the filter into service. During the initialization, the container sends the configuration information (initialization parameters) to the filter via a `FilterConfig` object.

The doFilter() Method

```
public void doFilter(ServletRequest request, ServletResponse response,
                     FilterChain chain) throws IOException, ServletException
```

The container invokes this method while processing a filter. Using the `FilterChain` object, each filter may instruct the container to process the rest of the filter chain. We shall see this process in detail shortly. Notice that this method is similar to a servlet's `service()` method with the difference of the third parameter.

The destroy() Method

```
public void destroy()
```

The container invokes this method before taking the filter out of service. This could happen either at container shut down, or when the application is undeployed.

All these methods are container-invoked methods used during a filter lifecycle. The lifecycle of a filter consists of the following steps. Compare this with the lifecycle of a servlet.

- **Instantiate**

The container instantiates each filter class either at the container startup or sometime before a filter instance is required for invocation. For each web application, the container maintains one instance per filter. This is similar to servlets that do not implement the `SingleThreadedServlet` interface. In the case of distributed web containers, the container maintains one instance per application per virtual machine.

- **Initialize**

After instantiating a filter, the container calls the `init()` method. During this call, the container passes on initialization parameters to the filter instance.

- **Invoke**

For those requests that require a filter to process, the container invokes the `doFilter()` method that is required to implement the filter logic.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
<web-app>

    <!-- Declare filter definitions -->
    <filter>
        <filter-name> Name of the filter </filter-name>
        <filter-class> Class name for the filter </filter-class>
        <init-param>
            <param-name> Name of the parameter </param-name>
            <param-value> Value of the parameter </param-value>
        </init-param>
        <!-- More initialization parameters -->
    </filter>

    ...

    <!-- Map filters to requests -->
    <filter-mapping>
        <filter-name> Name of the filter </filter-name>
        <url-pattern> URL pattern </url-pattern>
    </filter-mapping>

    ...

    <!-- Declare servlets -->

</web-app>
```

The Chat Application with Filters

Let us now consider certain enhancements to the chat application that we developed in the previous chapter, and see how filters can be used to implement these enhancements:

- **Message Logging**
It is usual to log chat messages in a persistent data source for monitoring purposes. How do we implement such a logging for all chat messages?
- **Message Moderation**
How can we implement a message moderation system where messages containing objectionable or offensive words can be detected? When the application receives messages containing certain words, we may replace the actual message with a standard warning message.

One way to implement these features is to alter the `ChatRoomServlet` class:

- To log each message in a persistent data source such a database
- To scan each message for moderated words, and replace messages containing such words with a warning message

However, this involves modifying the `ChatRoomServlet` adding further complexity to it. In addition, such a mechanism will not be declarative. Instead, let's consider using filters to implement these features such that:



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```

        .getResourceAsStream(fileName);
BufferedReader reader = new BufferedReader(
                new InputStreamReader(is));
wordList = new ArrayList();
while(true) {
    String word = reader.readLine();
    if(word == null) {
        break;
    }
    wordList.add(word.toLowerCase());
}
reader.close();

warningMessage = config.getInitParameter("warning_message");
if(warningMessage == null) {
    warningMessage = "**** Message restricted";
}
} catch(IOException ie) {
    ie.printStackTrace();
    throw new ServletException("Error reading moderated_words.txt");
} catch(Exception e) {
    e.printStackTrace();
}
}

```

This method opens a file whose name is specified as an initialization parameter for this filter. The filter opens this file using the `getServletContext()` method of the `FilterConfig` object. Using this method, we can open files and other resources available within the web application. The `init()` method reads the words and stores them in a list. This method also reads a standard warning message via another initialization parameter.

Let us now consider the `doFilter()` method:

```

public void doFilter(ServletRequest request,
                     ServletResponse response,
                     FilterChain chain)
                     throws IOException, ServletException {
String message = ((HttpServletRequest) request).getParameter("msg");
if(message != null) {
    boolean setWarning = false;
    for(int i = 0; i < wordList.size(); i++) {
        if(message.toLowerCase().indexOf((String) wordList.get(i)) != -1) {
            setWarning = true;
            break;
        }
    }
    if(setWarning) {
        message = warningMessage;
        // How do we change the request parameter? We'll see shortly!
    }
}

// Invoke the next filter.
chain.doFilter(request, response);
}

```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
// Invoke the next filter  
chain.doFilter(request, response);  
}  
  
public void destroy() {}  
}
```

Deployment Descriptor

The next step is to modify the deployment descriptor to add the filter declarations and filter mappings. Here is the modified deployment descriptor:

```
<?xml version="1.0" encoding="ISO-8859-1"?>  
  
<!DOCTYPE web-app  
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"  
"http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">  
  
<web-app>  
  
    <display-name>chat</display-name>  
    <context-param>  
        <param-name>ADMIN_PATH</param-name>  
        <param-value>/chat/servlet/chatAdmin</param-value>  
    </context-param>  
  
    <context-param>  
        <param-name>LISTROOMS_PATH</param-name>  
        <param-value>/chat/servlet/listRooms</param-value>  
    </context-param>  
  
    <context-param>  
        <param-name>CHATROOM_PATH</param-name>  
        <param-value>/chat/servlet/chatRoom</param-value>  
    </context-param>  
  
    <filter>  
        <filter-name>MessageLogFilter</filter-name>  
        <display-name>MessageLogFilter</display-name>  
        <filter-class>MessageLogFilter</filter-class>  
    </filter>  
  
    <filter>  
        <filter-name>MessageModeratorFilter</filter-name>  
        <display-name>MessageModeratorFilter</display-name>  
        <filter-class>MessageModeratorFilter</filter-class>  
        <init-param>  
            <param-name>moderated_words_file</param-name>  
            <param-value>moderated_words.txt</param-value>  
        </init-param>  
        <init-param>  
            <param-name>warning_message</param-name>  
            <param-value>*** Message filtered by the moderator.</param-value>  
        </init-param>  
    </filter>
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Let us now consider certain variations:

- In order to remove message moderation, simply remove the mapping for `MessageModeratorFilter` in the deployment descriptor.
- In order to remove message logging, simply remove the corresponding mapping from the deployment descriptor.
- With the above code, moderated messages will be logged instead of the original message. What if you instead want to log the original message before moderation? To do this, we can modify the `MessageLogFilter` to read the message using the `getAttribute()` call instead of the `getParameter()` call.
- We may also enhance this example to replace objectionable words with "****" instead of replacing the entire message.

You may try these variations on your own as an exercise.

Summary

Filters provide a powerful mechanism to programmatically alter how the container serves web resources. The filter mechanism is independent of the type of resource. It applies equally well to dynamic resources such as servlets or JSP pages, or static resources such as HTML files, images, and so on.

The examples in this chapter have illustrated two main points:

- We can introduce one or more filters within the request-response process.
- A filter can modify the environment for servlets and JSP pages. As illustrated by the `MessageModeratorFilter`, we can create new request, response, or even session objects for servlets or JSP pages. This feature opens up several possibilities for us to build more intelligent web applications.

When you plan to develop your own web applications, consider if some of the functionality qualifies for implementation in filters. You may also consider filters when you're adding new features to existing web applications. In general, if you find certain logic common to one or more servlets/JSP pages, or when you want to include dynamic behavior before invoking static content, consider using filters. Filters provide a declarative, and therefore loosely coupled, mechanism with which to add useful features.

In the next chapter, we will complete our exploration of the Servlet specification and look at some other topics involved with developing web applications.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

A public resource is any file that is accessible to the client of the web application. Typical public resources include:

- ❑ HTML, XML, and JSP documents
- ❑ Image, audio, and video files
- ❑ Java applets – classes as well as JAR files containing applets
- ❑ Microsoft Office documents – Word, Excel, etc.

Client browsers can download these resources unchanged and render them as required. For example, when an HTML page includes an <APPLET> tag that refers to an applet in a JAR file, the web container delivers the JAR file to the client browser unchanged.

Although JSP pages are included as public resources in a web application, the container does not show them directly to clients. As we'll learn in Chapter 10, the container automatically converts JSP pages into servlets. The servlet is then compiled and invoked to generate the response for the client. However, since a JSP page is often considered more akin to an HTML document than to a Java class, JSP pages are also included under public resources.

Typically, the container reads the static resource from the file system, and writes the contents directly to the network connection. The interpretation of the content of these resources is the responsibility of the client browser. In order to help the browser render the file correctly, the container sends the MIME type of the file. These MIME types can be set in the deployment descriptor of the web application.

Of course, there are certain resources that clients should not download directly. These may be files that the client should not view, such as configuration files, or they may be files that the container must perform some processing on before sending the response to the client, like servlets. These private resources are stored in the WEB-INF directory, or its subdirectories.

The types of resources that clients should not download directly include:

- ❑ Servlets – these components include application logic, and possibly access to other resources such as databases
- ❑ Any other files in the web application that servlets access directly
- ❑ Resources that are meant for execution or use on the server side, such as Java class files and JAR files for classes used by your servlets
- ❑ Temporary files created by your applications
- ❑ Deployment descriptors and any other configuration files

These resources are private and, as such, are accessible only to their own web application and the container. In order to accommodate private resources, the web container requires the WEB-INF directory to be present in your application. If your web application does not contain this directory it will not work. WEB-INF includes:

- ❑ A **web.xml** file. This is the deployment descriptor file.
- ❑ A subdirectory named **classes**. This directory is used to store server-side Java class files such as servlets and other helper classes, structured according to the usual Java packaging rules.
- ❑ A **lib** subdirectory to contain JAR files used by the web application.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

An HTTP request always contains a Uniform Resource Identifier (URI) identifying the requested resource. The terms Uniform Resource Locator (URL) and Uniform Resource Identifier (URI) are often used inconsistently in books, specifications, and other documents. This is partly because the URL is the one of the most commonly used subset of URIs for specific network protocols. We will use the terms as they are used in the Servlet API specification, which is largely the same as their use in the HTTP specification:

- A URI is any string used to identify an Internet resource, in a name space. Indeed, the address of any Internet resource can be encoded in the form of a URI.
- A URL is such a string, in a format for a specific protocol such as `http`, `ftp`, `telnet`, `news`, etc. This format includes a scheme (for example, `http`), the domain name of the server, a path, and possibly query string parameters.
- A URL path is the part of the URL that identifies the resource within a specific server; in other words, just the part that denotes the path.

For example, the URI `http://www.apress.com/Consumer` is a URL for the scheme `http`, located at the server `www.apress.com`, and the URL path of this resource is `/Consumer`.

From the above description, URLs and URIs seem very similar. However, the main difference is that a URI *may or may not* describe a physical resource, whereas a URL *must* describe a physical resource.

For instance, consider the DOCTYPE declaration of the web application deployment descriptor. The DOCTYPE declaration specifies a DTD for the deployment descriptor that the web container uses to validate the deployment descriptor. What this declaration contains is a URI, for example:

`http://java.sun.com/j2ee/dtds/web-app_2_3.dtd`. The URI specified in this declaration may or may not exist physically. Although in the case of this particular example you will actually find a DTD at this location, this need not always be true. The XML parser that parses the deployment descriptor may instead map this URI to a DTD available in the local file system (using an entity resolver).

Web containers use the same concept to identify web applications, and the resources within them. With the help of the URL path, web containers can identify the application, the associated servlet context, and the resource within the application. For this to be possible, each web application must be mapped to a unique URL path prefix, for instance `/chat`. This path prefix specifies a unique namespace for all resources with the web application. The web container uses this prefix to map requests to resources within the web application.

In order to see how this mechanism works, consider a `help.html` file within the `help` subdirectory of an application, with its root deployed with a context path `/catalog` on a web container located at `http://localhost:8000`. This resource can thus be accessed as `http://localhost:8000/catalog/help/help.html`. This is similar to having a `help.html` file in the `/catalog/help` subdirectory under the document root of a web server located at `http://localhost:8000`. In this regard, the container behavior is similar to that of a web server, except that the container uses the context path name to identify the application.

The path name serves as the document root for locating resources within the application. For instance, consider two applications deployed on the same server: one with the context path `/techSupport` and the other with a context path `/chat`. When the web container receives an HTTP request with a path starting with `/chat` the container can determine that the request should be handled by the Chat application, and similarly it can determine that all requests starting with `/techSupport` will be handled by the Tech Support application.

This mechanism can be extended to servlets within applications. In the same way as the context path is used to map a request to a web application, URL path mappings can be used to map requests to servlets. There are two steps in this mapping process:



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

`http://hoststring/ContextPath/servletPath/pathInfo`

It is easy to move an application to a different place in the web container's name space as all URI paths within a context are relative to the context's root URI path. So if we introduce a new version of the Chat application, and we want to use the `/chat` prefix for the new version while still providing access to the old version at `/oldchat`, all we have to do is change the web container's context mappings. In this case we would map `/chat` to the new version and `/oldchat` to the old version.

Securing Web Applications

Controlled access to web applications, in order to guard valuable data and resources, is a very common requirement. The most commonly used approach for web application security is to use login forms, and authenticate users based on their login names and passwords. This approach is usually combined with Secure Sockets Layer (SSL) with server-side digital certificates obtained from certificate vendors such as VeriSign (<http://www.verisign.com/>).

SSL is a security protocol and specifies a process for web servers and browsers to establish a secure communication channel over HTTP. This protocol includes steps by which browsers and servers exchange certain private keys before exchanging application data. Browsers and servers use these private keys to encrypt and decrypt application data over HTTP. Since this protocol allows information to be exchanged in an encrypted form, it is more secure than sending information unencrypted.

Although more secure implementations are possible, and sometimes required, we will concentrate on how web containers and web applications can be configured to implement user authentication. Note that most web servers provide mechanisms to create realms, and access control lists to control access to various web resources. In this section, we're not concerned about such features, and instead we'll focus on the features possible with the servlet specification.

Before discussing details of how web containers can be set up to provide secure access, it is useful to understand the traditional approach that did not rely on the container. This was usually implemented as follows:

- Program the application to have one or more entry points with login forms. For example, in the Chat application, we can provide a login page, with a form to accept the login name and password of the Chat administrator.
- Authenticate the user when they submit the login form, preferably over HTTPS. For this purpose, you should verify against a previously created user database or a file that contains user login names and passwords. For example, in the case of the Chat admin servlet, we could maintain all users with administrative capabilities in a database. A new servlet would be required, called `LoginServlet`, to receive this request, connect to the database, and check if the user name and password match.
- Once the login is successful, the user would be forwarded to the admin servlet.

What safeguards can be introduced to prevent a user from directly accessing the admin servlet? There are two possibilities:

- The admin servlet should somehow be able to know that the user has already logged in. We can use a session attribute, `username`, which is set by the login servlet when the login is successful. The admin servlet can verify if this attribute exists in the session, and if not, can send some content explaining that the user cannot use the admin servlet.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Let's designate Joe as the user holding the `chatAdministrator` role. Joe can also have other roles such as `sysAdministrator`. The objective is to allow only those users that have a role called `chatAdministrator` access to invoke the chat admin servlet.

We begin with the deployment descriptor and would add the following to the `web.xml` file of our Chat application:

```

<web-app>
    <!-- Insert servlet and context definitions -->

    <security-constraint>
        <web-resource-collection>
            <web-resource-name>Chat Administration</web-resource-name>
            <description>Chat Administration</description>
            <url-pattern>/admin/*</url-pattern>
            <url-pattern>/servlet/chatAdmin</url-pattern>
            <http-method>GET</http-method>
        </web-resource-collection>
        <auth-constraint>
            <description>Chat Administrator</description>
            <role-name>chatAdministrator</role-name>
        </auth-constraint>
        <user-data-constraint>
            <transport-guarantee>NONE</transport-guarantee>
        </user-data-constraint>
    </security-constraint>

    <login-config>
        <auth-method>FORM</auth-method>
        <realm-name>
            Form-Based Authentication for Chat Administration.
        </realm-name>
        <form-login-config>
            <form-login-page>/login.html</form-login-page>
            <form-error-page>/error.html</form-error-page>
        </form-login-config>
    </login-config>
</web-app>

```

The above deployment description specifies the security constraint and the authentication scheme. More specifically, the `<security-constraint>` tag includes definition of a resource collection, and the authorization requirement. The resource collection in the above includes all `GET` and `POST` requests with the URL-pattern `/admin/*`. The `<user-data-constraint>` tag specifies how data should be transmitted across the wire. The possible values are `NONE`, `INTEGRAL`, and `CONFIDENTIAL`. The value `NONE` used here means that there is no transport requirement; a value of `INTEGRAL` would indicate the underlying data transmission should guarantee integrity of data, and a value of `CONFIDENTIAL` would require that the underlying transmission should prevent other entities from observing the data. Support for these values depends on both the clients and servers. For instance, SSL-enabled client browsers and web containers can participate in `CONFIDENTIAL` transmission.

The `<security-constraint>` element also includes an `<auth-constraint>` tag for the resource collection. In the example above, this element includes the `<role-name>` element with value `chatAdministrator`. Hence, only users with role `chatAdministrator` are allowed to access the resources.

The next step is to define a login configuration. The `<login-config>` tag specifies how authentication should be performed. In our example, we are using a form-based authentication, which requires us to specify a login-page and an error-page. Whenever login is required, the web container automatically sends the login page. Whenever a login fails, the web container automatically sends the error page. In our example, we are using `login.html` and `error.html` pages respectively to achieve this.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Servlet Initialization Parameters

Just as parameters can be set for the context, initialization parameters can be specified for each servlet. This is done using the `<init-param>` element. The following deployment descriptor snippet sets two initialization parameters:

```
<web-app>
  ...
  <servlet>
    <servlet-name>Jack</servlet-name>
    <servlet-class>mypackage.Jack</servlet-class>
    <init-param>
      <param-name>name</param-name>
      <param-value>Jill</param-value>
      <description>Name of Jill. Jill is Jack's brother.</description>
    </init-param>
  </servlet>
</web-app>
```

An initialization parameter called `name` is set for the `mypackage.Jack` servlet. Use the `getInitParameter()` method on the `ServletConfig` object to retrieve this initialization parameter.

Unlike context initialization parameters, servlet initialization parameters are specific for each servlet. Initialization parameters can be used to specify startup information for a servlet. You may recall that our `FreakServlet` example of Chapter 6 had an initialization parameter to specify an interval for servlet unavailability. The `FreakServlet` would use the following code in its `init()` method to read the value of this parameter:

```
public void init() throws ServletException {
    states.add(createState("Initialization"));
    String waitIntervalString =
        getServletConfig().getInitParameter("waitInterval");
    if(waitIntervalString != null) {
        waitInterval = new Integer(waitIntervalString).intValue();
    }
}
```

The same initialization procedure applies to filters also. As we saw in the last chapter, you can add `<init-param>` elements for each filter to specify initialization parameters and their values. Refer to the relevant deployment descriptor for the filters introduced in the previous chapter.

Loading Servlets on Startup

Loading a servlet involves loading the servlet class from the deployment archive, instantiation of the servlet, and servlet initialization. By default, the container does not guarantee any order in which servlets are loaded. The container may wait until a request is received before loading a servlet.

Depending on your application design, you may want to specify that one or more servlets be loaded during the startup of the web application. Furthermore, these servlets may need to be loaded in a particular order. The optional `<load-on-startup>` element lets you specify just such a requirement:

```
<web-app>
  ...
  <servlet>
    <servlet-name>Jack</servlet-name>
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Both of these methods will set the HTTP response status code and commit the response. The first of these methods accepts a status code, while the second one also accepts a user-defined message to further clarify the error.

No further output should be written after executing the `sendError()` method because after `sendError()` is called, the web container sends an HTTP status code and HTTP status message in the response headers. If configured correctly, the web container may send a particular document explaining the error. The container may also log the error. The client can extract the status code and status message from the response headers.

The following table describes the most common HTTP error codes. All these error codes are defined as constants in the `HttpServletResponse` interface:

| HTTP Code | Error Code | Description |
|-----------|---------------------------------------|---|
| 400 | <code>SC_BAD_REQUEST</code> | The request was syntactically incorrect |
| 401 | <code>SC_UNAUTHORIZED</code> | Indicates that the request requires HTTP authentication |
| 403 | <code>SC_FORBIDDEN</code> | The server understood the request but refused to fulfill it |
| 404 | <code>SC_NOT_FOUND</code> | The requested resource is unavailable |
| 500 | <code>SC_INTERNAL_SERVER_ERROR</code> | An error within the HTTP server caused the request to fail |
| 501 | <code>SC_NOT_IMPLEMENTED</code> | The HTTP server does not support the functionality needed to fulfill this request |
| 503 | <code>SC_SERVER_UNAVAILABLE</code> | The HTTP server is overloaded and cannot service the request |

Throwing `ServletException`

The second approach to indicate a failure is to throw a `javax.servlet.ServletException` or one of its subclasses. You could design a set of classes that extend `ServletException`, and throw these from within your servlets as appropriate.

Consider, for example, a servlet that performs database access using the JDBC API. A common exception that the servlet might encounter is `java.sql.SQLException`. Since the container cannot gracefully handle non-servlet exceptions such as this, the exception cannot be re-thrown to the container. Your servlet must therefore throw a subclass of `javax.servlet.ServletException` whenever it receives a `java.sql.SQLException`. For example, consider the following snippet of code from the servlet:

Copyrighted image



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

In theory, distribution is a simple approach to distribute load across multiple processes running on multiple servers. However, in practice setting up distribution is not as simple as it sounds, and requires advanced vendor-specific configurations. Besides configuring the cluster in a network, which could be the hardest for a novice, one of the configuration steps is to choose a **load-distribution strategy**. This strategy is the method by which the container is instructed to distribute load across nodes in a cluster. One of the most commonly used strategies is called "round-robin", which involves sending each request to the next node in the cluster. Containers also provide strategies that are more advanced. Such strategies take into consideration the processing capacity of each machine, and their current process loads, priorities, etc. Refer to your container's documentation for more specific details.

Another consequence of clustering is to provide **failure-recovery**. Failure-recovery is to do with how to allow seamless processing of requests in spite of a server crash. Load distribution mechanisms usually have the capability to detect a server crash (by polling or some other form of communication) and send requests to one of the functional nodes so that new requests can be processed without failure.

Perhaps one of the most obvious effects of clustering is that, in a cluster, a request from a client can be processed by any node. There is one exception to this, as we shall see shortly.

Once a cluster is set up, the process of enabling distribution for web applications is quite simple. Add the element `<distributable>` at the beginning of the deployment descriptor (before specifying any context parameters):

```
<web-app>
  ...
  <distributable />
  ...
</web-app>
```

This element indicates to the container that this application may be distributed in a cluster.

Although this process of specifying that an application is distributable is simple, both load distribution and failure recovery get complex when it comes to web applications maintaining session and context. As we saw in the previous chapters, we can store information in session and context objects. So what happens to these objects in a cluster where any node can process a request?

When a web container receives an HTTP request to a web application, as we discussed previously, the container creates and manages a session for the sender of the request. Servlets and JSP pages handling the request can store information (state) in the session. Since session itself is an object, it is maintained within the JVM of this node. What happens when the next request from the same user is sent to a different node in the cluster? For the other node to process this request seamlessly, it should have the same session available. Otherwise, all session/state information will be lost. In order to allow this, web containers replicate the session information into other nodes too. Whether the container replicates the session information to all nodes in the cluster or not depends on the container implementation. Some containers designate a secondary node to each node, and limit replication only to the secondary node. Others may replicate the state to all nodes in the cluster.

In order for the session information to be replicable, the data stored in the session must be serializable. When you attempt to store non-serializable objects within the session of a distributable web application, the container may throw an `IllegalArgumentException`.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

- A JSP container specific implementation class that implements the `javax.servlet.jsp.JspPage` interface and provides some basic page-specific behavior. (Since most JSP pages use HTTP, their implementation classes must actually implement the `javax.servlet.jsp.HttpJspPage` interface, which itself extends `javax.servlet.jsp.JspPage`).
- Specified by the JSP author via an `extends` attribute in the page directive.

The `javax.servlet.jsp.JspPage` interface contains two methods:

```
public void jspInit()
```

This is invoked when the JSP is initialized, and is analogous to the `init()` method in servlets. Page authors may provide initialization of the JSP by implementing this method in their JSP pages.

```
public void jspDestroy()
```

This is invoked when the JSP is about to be destroyed by the container, and is analogous to the `destroy()` method in servlets. Page authors may provide cleanup of the JSP by implementing this method in their JSP pages.

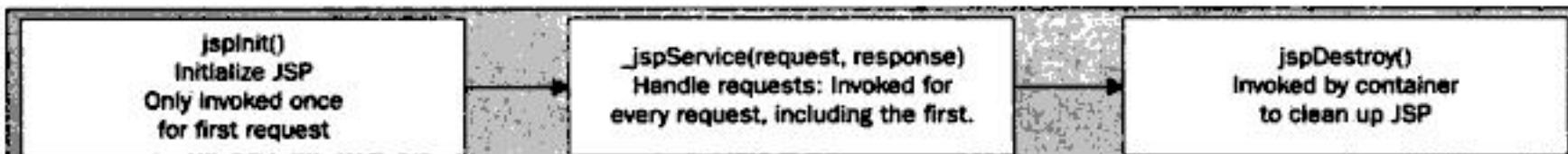
The `javax.servlet.jsp.HttpJspPage` interface contains a single method:

```
public void _jspService(HttpServletRequest request,  
                         HttpServletResponse response)  
        throws ServletException, IOException
```

This method corresponds to the body of the JSP page and is used for threaded request processing, just like the `service()` method in servlets.

The implementation of this method is generated by the container and should never be provided by page authors.

These three major life event methods work together in a JSP, as seen below:



- The page is first initialized by invoking the `jspInit()` method, which may be defined by the page author. This initializes the JSP in much the same way as servlets are initialized, when the first request is intercepted and just after translation.
- Every time a request is made to the JSP, the container-generated `_JSP pageservice()` method is invoked, the request is processed, and the JSP generates the appropriate response. This response is taken by the container and passed back to the client.
- When the JSP is destroyed by the server (for example at shut down), the `jspDestroy()` method, which may be defined by the page author, is invoked to perform any cleanup.

Note that in most cases the `jspInit()` and `jspDestroy()` methods don't need to be provided by the JSP author, and can be omitted.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Example

Consider the simple JSP below, declaration.jsp:

```
<%!
    int numTimes = 3;

    public String sayHello(String name) {
        return "Hello, " + name + "!";
    }
%>

<html>
    <head>
        <title>Declaration test page</title>
    </head>
    <body>
        <h1>Declaration test page</h1>

        <p>The value of numTimes is <%= numTimes %>.</p>
        <p>Saying hello to reader: "<%= sayHello("reader") %>".</p>
    </body>
</html>
```

This declares an `int` variable called `numTimes`, and a `sayHello()` method that greets the requested person. Further down the page, expression elements (to be covered shortly) are used to return the value of `numTimes` to the browser and to invoke the `sayHello()` method.

Save `declaration.jsp` to the `JSPEexamples` directory, and view `http://localhost:8000/JSPEexamples/declaration.jsp`; will then see the following:

Copyrighted image

The generated servlet contains the declaration code:

```
import javax.servlet.*;
// ... more import statements

public class _0002fdeclaration_jsp extends HttpJspBase {

    int numTimes = 3;
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
Hello World!
<%= "This JSP has been accessed " + i + " times" %>

</body>
</html>
```

An int variable `i` is declared, and initially has the value 0. Each time this instance of the generated servlet is called, for instance when the browser requests `http://localhost:8000/JSPExamples/expression.jsp`, the variable is incremented by the scriptlet. Finally, an expression is used to print out the value of `i`, together with some surrounding text.

After this JSP has been requested seven times, the browser will display:

Copyrighted image

Note that while code in scriptlets follows normal Java grammar, ending statements with a semicolon, a JSP expression, however complicated it may be, does not end with a semicolon.

Comments

JSP supports hidden comments. Any content between the delimiters `<%--` and `--%>` will be ignored by the container at translation time. JSP comments don't nest so any content except a terminating `--%>` is legal within JSP comments.

If we want to see comments in the generated output, we can always use HTML or XML comments with the `<!--` and `-->` delimiters; these will simply be treated as normal template data by the container.

We can use JSP comments to document details of our implementation as we do in Java code. However, we could choose to use just JSP comments rather than HTML comments to document our markup structure as well. This means that we can be as verbose as we like without bloating the generated content.

We are also free to use the standard Java comment syntax within JSP scriptlet elements.

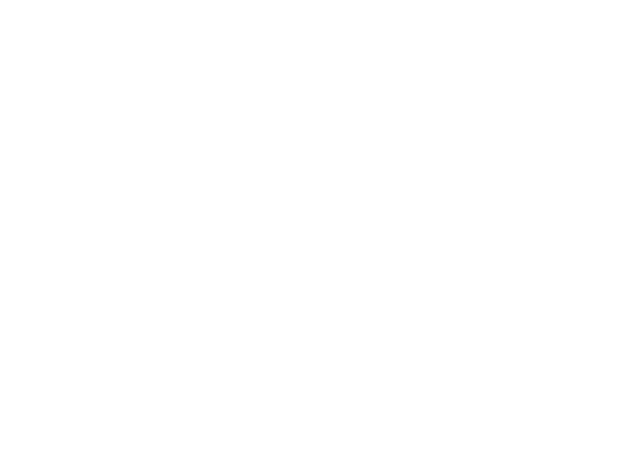
Standard Actions

Standard actions are well-known tags that affect the runtime behavior of the JSP and the response sent back to the client. They must be implemented to behave the same way by all containers. Unlike custom tags these actions are available by default and do not require taglib declarations in order to use them.

When the container comes across a standard action tag during the conversion of a JSP page into a servlet, it generates the Java code that corresponds to the required predefined task.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The `<jsp:setProperty>` action uses the Java bean's introspection mechanism to discover what properties are present, their names, whether they are simple or indexed, their type, and their accessor and mutator methods.

The syntax of the `<jsp:setProperty>` action is:

```
<jsp:setProperty name="beanName" propertydetails />
```

where `propertydetails` is one of:

- `property="*"`
- `property="propertyName"`
- `property="propertyName" param="parameterName"`
- `property="propertyName" value="PropertyValue"`

Note that `PropertyValue` is a string or a scriptlet. The attributes are:

| Attribute | Description |
|-----------------------|--|
| <code>name</code> | The name of a bean instance, which must already have been defined by a <code><jsp:useBean></code> tag. The value of this attribute in <code><jsp:setProperty></code> must be the same as the value of the <code>id</code> attribute in <code><jsp:useBean></code> . |
| <code>property</code> | <p>The name of the bean property whose value is being set.</p> <p>If this attribute has the value "*", the tag looks through all the parameters in the <code>request</code> object and tries to match the request parameter names and types to property names and types in the bean. The values in the request are assigned to each matching bean property unless a request parameter has the value "", in which case the bean property is left unaltered.</p> |
| <code>param</code> | <p>When setting bean properties from request parameters, it is not necessary for the bean have the same property names as the request parameters.</p> <p>This attribute is used to specify the name of the request parameter whose value we want to assign to a bean property. If the <code>param</code> value is not specified, it is assumed that the request parameter and the bean property have the same name.</p> <p>If there is no request parameter with this name, or if it has the value "", the action has no effect on the bean.</p> |
| <code>value</code> | <p>The value to assign to the bean property. This can be a request-time attribute, or it can accept an expression as its value.</p> <p>A tag cannot have both <code>param</code> and <code>value</code> attributes.</p> |

When properties are assigned from string constants or request parameter values, conversion is applied using the standard Java conversion methods; for example, if a bean property is of type `double` or `Double` the `java.lang.Double.valueOf(String)` method is used. However, request-time expressions can be assigned to properties of any type, and the container performs no conversion. For indexed properties, the value must be an array.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Enter your name and choose your favored language, then click the Submit information button:

Copyrighted image

The advantage of this approach over that used in previous examples, in which Java code and HTML were freely mixed, should be clear. By removing the logic to the `LanguageBean` class the JSP page is much more readable, and therefore easily edited by someone who is a skilled web designer but does not necessarily understand the details of Java programming.

Using JavaBeans is just one way to avoid having excessive amounts of Java code in JSP pages. Other strategies include using servlet controllers and JSP tag extensions.

<jsp:param>

The `<jsp:param>` action is used to provide other, enclosing, tags with additional information in the form of name-value pairs. It is used in conjunction with the `<jsp:include>`, `<jsp:forward>`, and `<jsp:plugin>` actions, and its use is described in the relevant sections that follow. The syntax is:

```
<jsp:param name="paramname" value="paramvalue" />
```

The available attributes are:

| Attribute | Description |
|--------------|--|
| Name | The key associated with the attribute. (Attributes are key-value pairs.) |
| Value | The value of the attribute. |

<jsp:include>

This action allows a static or dynamic resource, specified by a URL, to be included in the current JSP at request-processing time. An included page has access only to the `JspWriter` object, and it cannot set headers or cookies. A request-time exception is thrown if this is attempted. This constraint is equivalent to that imposed on the `include()` method of `javax.servlet.RequestDispatcher` used by servlets for this type of inclusion.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
    }
    out.write("\r\n      ");
    {
        String _jspx_qStr = "";
        out.flush();
        pageContext.include("included2.jsp" + _jspx_qStr);
    }
    out.write("\r\n\r\n  </body>\r\n</html>\r\n");
}

// ...
}
```

The highlighted lines in the above code show how the container in-lines the resources for the `include directive`, and invokes them dynamically for the `include action`. To reiterate the difference, see what happens when the included resources are changed, without changing the parent JSP that includes them. Change `included2.html` so that it contains:

```
<p>This is some new text in the html file</p>
```

and change `included2.jsp` so that its contents are now:

```
<p>This is the new JSP</p>
```

When the page is requested again, the output will look like this:

Copyrighted image

The parts included using the `include directive` are not altered, because the parent JSP (`includeAction.jsp`) has not changed and hence is not recompiled; however, the parts included using the `include action` are changed, because the `include action` performs the inclusion afresh each time the parent JSP is requested.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The available attributes for the `<jsp:plugin>` tag are:

| Attribute | Details | Required |
|--------------------------|---|--|
| <code>type</code> | Identifies the type of the component: a JavaBean, or an Applet. | Yes |
| <code>code</code> | Same as HTML syntax. | Yes |
| <code>codebase</code> | Same as HTML syntax. | No |
| <code>align</code> | Same as HTML syntax. | No |
| <code>archive</code> | Same as HTML syntax. | No |
| <code>height</code> | Same as HTML syntax. | No, but some browsers do not allow an object of zero height due to security issues |
| <code>hspace</code> | Same as HTML syntax | No |
| <code>jreversion</code> | The Java runtime environment version needed to execute this object. Default is "1.1". | No |
| <code>name</code> | Same as HTML syntax. | No |
| <code>vspace</code> | Same as HTML syntax. | No |
| <code>title</code> | Same as HTML syntax. | No |
| <code>width</code> | Same as HTML syntax. | No, but some browsers do not allow an object of zero width due to security issues |
| <code>nspluginurl</code> | URL where the Java plugin can be downloaded for Netscape Navigator. Default is implementation defined. | No |
| <code>iepluginurl</code> | URL where the Java plugin can be downloaded for Internet Explorer. Default is implementation defined. | No |

The following example shows the `<jsp:plugin>` standard action in use, and the HTML generated by the RI for an Internet Explorer 5.5 client running on Windows 2000. We took the applet from the examples shipped with the JDK version 1.3.0. To run it, take the whole MoleculeViewer sample tree from the JDK, place it in a WAR, and place the following JSP (`applet.jsp`) in the root of the MoleculeViewer directory.

```

<p>This is the molecule viewer example from the
JDK 1.3.
</p>

<jsp:plugin type="applet"
  code="XYZApp.class"
  codebase="./"
  width="300" height="300">
  <jsp:params>
    <jsp:param
      name="model"
      value=".models/HyaluronicAcid.xyz" />

```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The object reference is discarded upon completion of the current `Servlet.service()` invocation (in other words, when the page is fully processed by the servlets generated from the JSP). When generating the servlet, the servlet engine creates an object in the `service()` method, which follows the usual object scope convention in Java. This object is created and destroyed for each client request to the page.

This is the default scope for objects used with the `<jsp:useBean>` action.

Request Scope

Request scope means that the object is bound to the `javax.servlet.ServletRequest`, and can be accessed by invoking the `getAttribute()` methods on the implicit `request` object.

The object reference is available as long as the `HttpServletRequest` object exists, even if the request is forwarded to different pages, or if the `<jsp:include>` action is used. The underlying, generated servlet relies on binding the object to the `HttpServletRequest` using the `setAttribute(String key, Object value)` method in the `HttpServletRequest`; this is transparent to the JSP author. The object is distinct for every client request (in other words it is created afresh and destroyed for each new request).

Session Scope

An object with session scope is bound to the `javax.servlet.jsp.PageContext`, and can be accessed by invoking the `getValue()` methods on the implicit session object.

The generated servlet relies on binding the object to the `HttpSession` using the `setAttribute(String key, Object value)` method. This too is transparent to the JSP author. The object is distinct for every client, and is available as long as the client's session is valid.

Application Scope

Application scope means that the object is bound to the `javax.servlet.ServletContext`. An object with this scope can be accessed by invoking the `getAttribute()` methods on the implicit `application` object.

This is the most persistent scope. The generated servlet relies on binding the object to the `ServletContext` using the `setAttribute(String key, Object value)` method in the `ServletContext`. This is not unique to individual clients and, consequently, all clients access the same object, as they all access the same `ServletContext`.

When accessing application variables, take care your code is thread-safe. Application variables are often populated on application startup, and read-only thereafter.

JSP Pages as XML Documents

The JSP 1.1 specification introduced an alternative, XML-based syntax for JSP pages, replacing directives and scripting elements with XML-compliant alternatives. The JSP 1.2 specification takes this a step further by requiring JSP engines to accept JSP pages as XML documents, and basing the new tag library validation mechanism (discussed in Chapter 11) on the XML representation of JSP pages.

The XML syntax is not intended for hand authoring of JSP pages, and so will never replace the JSP syntax we have discussed so far. However, an XML-based syntax may be preferable when JSP pages are machine-authored, and makes it easy to manipulate JSP pages.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Copyrighted image

an to

Application Design

The broad structure of the application is roughly the same as before, except that JSP pages replace the servlets and the logic is removed to a JavaBean:

Copyrighted image

RegisterCustomerServlet by register.jsp, ResponseServlet by response.jsp, and BannerServlet by banner.jsp.

sp,



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The TechSupportBean

So, how does `TechSupportBean` do all this work for `techSupport.jsp`? Let's start with the easy bit – the class declaration, member variables, and the property getter and setter methods:

```
// TechSupportBean

package com.apress.techsupport;

import java.sql.*;

public class TechSupportBean {

    // JNDI name of the data source this class requires
    private static final String DATA_SOURCE_NAME = "techSupportDB";

    private String email;
    private String software;
    private String os;
    private String problem;
    private String firstName;
    private String lastName;
    private String phoneNumber;

    private boolean registered;

    private JDBCHelper jdbcHelper;

    public TechSupportBean() {
        jdbcHelper = new JDBCHelper(DATA_SOURCE_NAME);
    }

    public void setEmail(String email) {
        this.email = email;
    }

    // ... and similarly for the software, os problem, firstName, lastName,
    // and phoneNumber properties

    public String getEmail() {
        return email;
    }

    // ... and similarly for the software, os problem, firstName, lastName,
    // and phoneNumber properties

    // ... more properties and methods ...
}
```

In a production-quality application, the driver class name and the JDBC URL might be read from context parameters in the deployment descriptor. (Better still, we could use a database connection pool managed by the servlet engine, or access the database only through a layer of EJBs.)

We have also provided setter and getter methods for several properties we haven't dealt with yet: `firstName`, `lastName`, and `phoneNumber`. These details will either be retrieved from the database (if the user is already registered), or obtained in a moment from the registration form. Remember that this bean will be used throughout the user's session, not just from `techSupport.jsp`.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
<jsp:getProperty name="techSupportBean" property="lastName" />
<hr>
XYZ Corporation, Customer Service at 1.800.xyz.corp.<br>
```

Note how much more convenient it is to generate the banners using JSP than servlets (as in Chapter 7).

The Error Page

Finally, we provide a simple error page, `error.jsp`, which is referenced by all the JSP pages that might fail during processing:

```
<%-- error.jsp --%>

<%@ page isErrorPage="true" %>

<html>
<head>
    <title>XYZ Corporation, IT Department</title>
</head>
<body>
    <h1>Technical Support</h1>

    <p>We're sorry, an error occurred processing your request.</p>
    <p>You got a <%= exception %>

</body>
</html>
```

Deploying the Application

All that remains is to ensure that the necessary files are in the correct directories, and that the Java classes are compiled:

- ❑ The JSP and HTML pages are placed in the `JSPTechSupport` directory.
- ❑ Place `web.xml`, as usual, in the `JSPTechSupport/WEB-INF` directory. This ensures we have a valid web application that can be deployed in any container.
- ❑ The Java source file `TechSupportBean.java` goes in `JSPTechSupport/src`.
- ❑ Compile the Java source file into the `JSPTechSupport/WEB-INF/classes` directory by running the command:

```
javac -d ..WEB-INF/classes *.java
```

from the `src` directory.

- ❑ Now we need to create the WAR file for the application. Enter the following command in the `JSPTechSupport` folder:

```
jar -cvf JSPTechSupport.war **
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The Page-View architecture is handy for prototyping, as it is easy to get quick results. It is the approach most developers use when first learning JSP. However, it is too unsophisticated to be used in any but trivial real-world applications.

Page-View with Bean

This architecture is a refinement of the same basic approach when the Page-View architecture becomes too cluttered with business-related code and data access code. The architecture now evolves into a more sophisticated design, as shown in the figure below:

Copyrighted image



The Java code representing the business logic and simple data storage implementation has migrated from the JSP to the JavaBean worker. This refactoring leaves a much cleaner JSP with limited Java code, which can be comfortably owned by an individual in a web-production role, since it encapsulates mostly markup tags.

Additionally, a less technical individual could be provided with property sheets for the JavaBean workers, providing a listing of properties that are made available to the JSP page by the particular worker bean, and the desired property may simply be plugged into the JSP <jsp:getProperty> action to obtain the attribute value.

Moreover, we now have created a bean that a software developer can own, so that its functionality may be refined and modified without the need for changes to the HTML or markup within the JSP source page. We have created cleaner abstractions in our system by replacing implementation with intent.

The JSP Technical Support sample application we have just discussed used a form of the Page-View with Bean architecture.

The Front Controller Pattern

The Page-View with Bean approach is a significant improvement on the basic Page-View architecture. However, it still leaves JSP pages needing both to perform (or, at least, to initiate) request processing and to present content. These tasks are quite distinct for many pages, and request processing may be complicated.

If a JSP is a view, the idea that it should handle incoming requests to the application it belongs to is clearly flawed. How do we know that this JSP is the correct view in all cases? How should it react to recoverable errors? (Presenting an error page may be an over-reaction.) Some JSP pages may need to manipulate session, application-wide resources, or state before the response can be generated. Others may need to examine request values to check whether to redirect the request. Simply mapping request properties onto a bean (see the discussion of JSP beans below) is not a universal solution, especially if redirection may be required.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```

    * @return the URL of the view that should render the response
    * (probably a JSP), or null to indicate that the response has been
    * generated already and processing is complete.
    */
String handleRequest(HttpServletRequest request,
                      HttpServletResponse response)
throws ServletException, IOException;
}

```

The interface will be implemented by application-specific helper classes to which a controller servlet will delegate request processing.

The implementations of this interface will be the core of the web tier of our application and enable the controller to delegate application-specific logic. The return value of the `handleRequest()` method will probably be the URL within our application of a view JSP; in some cases it could be the URL of a static HTML page, or a servlet within the same web application. A return value of `null` will have a special meaning, indicating to the controller that the `RequestHandler` implementation built the response itself. This is necessary to support those cases when a JSP view *isn't* appropriate: for example, if we need to generate binary data.

The controller servlet will examine incoming requests, and call one of its registered request handlers for each. Let's choose the request handler to use based on the request's URL within the web application, as returned by calling the `HttpServletRequest.getServletPath()` method. The controller's `init()` method will build a hash table of `RequestHandler` instances, keyed by the request URL. If the controller doesn't find a handler for a particular request, it will send an HTTP response code 404 ("Not Found"). (Of course, in real applications we might want to be more helpful to the user in this situation. For example, a more sophisticated controller with access to the user's session state might call a non application-specific method asking the session state for the most appropriate view to prompt the user to continue their session.)

The `ControllerServlet` listing follows. Note that for the sake of simplicity we've hard-coded the initialization of the handler hash table in the `init()` method: this wouldn't happen in a real implementation, which would use an external configuration file and therefore avoid dependencies on application-specific classes. We've also used a `System.out.println` statement in the `doGet()` method to show the servlet path of incoming requests. A production application would use a logging package such as Apache log4J instead of relying on console output:

```

package com.apress.proj2ee.ch10;

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

import javax.servlet.*;
import javax.servlet.http.*;

/**
 * Simple fragment of a controller Servlet using
 * mappings from application URL to a number of
 * RequestHandler helper classes.
 */
public class ControllerServlet extends HttpServlet {

    // Hash table of RequestHandler instances, keyed by request URL
    private Map handlerHash = new HashMap();

    // Initialize mappings: not implemented here
    public void init() throws ServletException {

```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```

/** Getter for property email.
 * @return Value of property email.
 */
public String getEmail() {
    return email;
}

/** Setter for property email.
 * @param email New value of property email.
 */
public void setEmail(String email) {
    this.email = email;
}
}

```

Since the data lookup process is not web-specific, we provide a helper class to implement it. Note that a real implementation of `DataStore` would look up an enterprise data source such as a database, or perhaps connect to an EJB server. In this case, we've simply populated the data values inline:

```

package com.apress.proj2ee.ch10.app;

import java.util.*;

public class DataStore {

    // HashMap of DataModelBean objects keyed by user name
    private static Map dataMap = new HashMap();

    static {
        dataMap.put("rod", new DataModelBean("Rod", "Johnson",
                                             "rod.johnson@interface21.com"));
        dataMap.put("detective", new DataModelBean("Sherlock", "Holmes",
                                                 "supersleuth@221b.co.uk"));
        dataMap.put("watson", new DataModelBean("John", "Watson",
                                                "sidekick@221b.co.uk"));
    }

    // Creates new DataStore
    public DataStore() {
    }

    public DataModelBean getInfo(String username) {
        return (DataModelBean) dataMap.get(username);
    }
}

```

Don't feel that you need to use the `<jsp:useBean>` tag to instantiate JSP beans. Remember there is a very useful form of this tag that does not specify an implementing class, which throws an exception if no such object is defined in the HTTP session or request. Simply omit the `class` attribute of the `<jsp:useBean>` tag, and specify only the type, like this:

```
<jsp:useBean id="dataBean" scope="request" type="DataModelBean"/>
```

Beans used in such pages can be instantiated through a complete `<jsp:useBean>` tag in another JSP, or can be placed in the session or request by a controller.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

JSP Tag Extensions

Probably the most important addition to the JSP 1.1 specification was support for **tag extensions** (or **custom tags**). This support is further enhanced in the JSP 1.2 specification, confirming tag extensions as one of the most important features of JSP.

Tag extensions look a lot like standard HTML or XML tags embedded in a JSP page, but they have a special meaning to the JSP engine at translation time, and allow custom functionality to be invoked without having to write Java code within scriptlets. This allows a JSP developer to invoke application functionality without having to know any of the details of the underlying Java code. Tag libraries also offer portable runtime support, authoring/modification support, and validation.

In this chapter, we'll look at the basics of writing our own tags, including:

- The basic ideas of tag extensions
- The anatomy of a tag extension
- How to deploy a tag library
- How to create custom tag extensions
- The enhancements to tag extension support in the JSP 1.2 specification

We'll begin by looking at what we can accomplish with tag extensions.

Tag Extensions

Let's consider the `<jsp:forward>` standard action that is provided by the JSP specification. This tag dispatches the current request to another JSP page within the current web application. It can be invoked with the following syntax:

```
<jsp:forward page="next.jsp" />
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

To understand how this works we should remember the important directories in the WAR:

- **WEB-INF**
This contains the `web.xml` file, in which the TLD URI-location mapping must be specified.
- **WEB-INF/classes**
This contains Java classes required to implement tag libraries or otherwise support the functionality of the web application.
- **WEB-INF/lib**
This contains JAR files containing additional classes required to support the functionality of the web application.
- **WEB-INF/tlds**
By convention (although not mandated in any specification) this contains the tag library descriptors (but not tag handler classes) that will be made available to JSP pages in the `web.xml` file. The TLDs could actually be placed anywhere in the WAR (so long as a mapping is included in the `web.xml` file), but adhering to this convention makes it easier to understand the WAR's structure.

Use this mechanism for tag libraries that are part of a particular application, rather than standard deployment units that will be reused elsewhere.

Packaged Tag Library JAR

A tag library may be distributed in a JAR file in which the `/META-INF` subdirectory contains the tag library descriptor, named `taglib.tld`. The JAR file should also contain the classes required to implement the tags defined in the tag library, but will not contain any JSP pages that use the tag library. In this case, the `taglib` directive in JSP pages should refer to this JAR, which will probably be within a WAR. This enables custom tags to be supplied in self-contained units – a vital precondition for the successful distribution of third-party custom tags.

The `taglib` directive will look like this:

```
<%@ taglib uri="/WEB-INF/lib/hellotags.jar" prefix="examples" %>
```

When including such a tag library JAR in a web application, we place it under the `/WEB-INF/lib` directory, to ensure that the classes it contains will be on the application classpath.

Default Mapping

The simplest means of deployment is to simply place the tag library descriptor under the server's document root (or to make it available to the application as an external URL), and ensure that the Java classes required to implement the tags are on the application's classpath. There is no attempt to package a tag library or an application. In this case, the `taglib` directive will look like this:

```
<%@ taglib uri=".//hello.tld" prefix="examples" %>
```

The URI is simply a path on the host server, which may be relative (as in this example) or absolute. In this approach, the tag library descriptor (although not the classes implementing the tag handler) is always publicly available; anyone could view it by simply typing in its URL. It's also easy to forget to make the supporting classes available to the application. For these reasons this approach isn't recommended, although it's easy to get started by using it.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

When we specify a reader parameter we see a runtime attribute value. To see this in action, navigate to <http://localhost:8000/tagext/helloAttribute?reader=yourName>:

Copyrighted image

Attributes are an excellent way of controlling tag behavior at run time, and are especially valuable in ensuring that tags are generic and reusable.

As elegant as the attribute/property mechanism is, there is one annoying problem with passing string attributes to tags. Specifying certain characters in attributes is messy. The double quote character, for example, is illegal in an attribute, and we must use the entity reference " if we want to include it. This rapidly becomes unreadable if the data includes multiple quotation marks. Attributes are also unsuited to handling lengthy values, as they soon become unreadable.

So there are limits to what can sensibly be achieved with attributes. Where complicated markup is concerned, consider the alternatives:

- Processing markup and expressions in the body of the tag, possibly repeatedly
- Defining a subtag that configures its ancestor
- Implementing the tag to read its markup from a template file or URL

The most elegant of these solutions (although not always the most feasible) is to manipulate the tag body. This will only be useful if the tag defines scripting variables that the tag body can use.

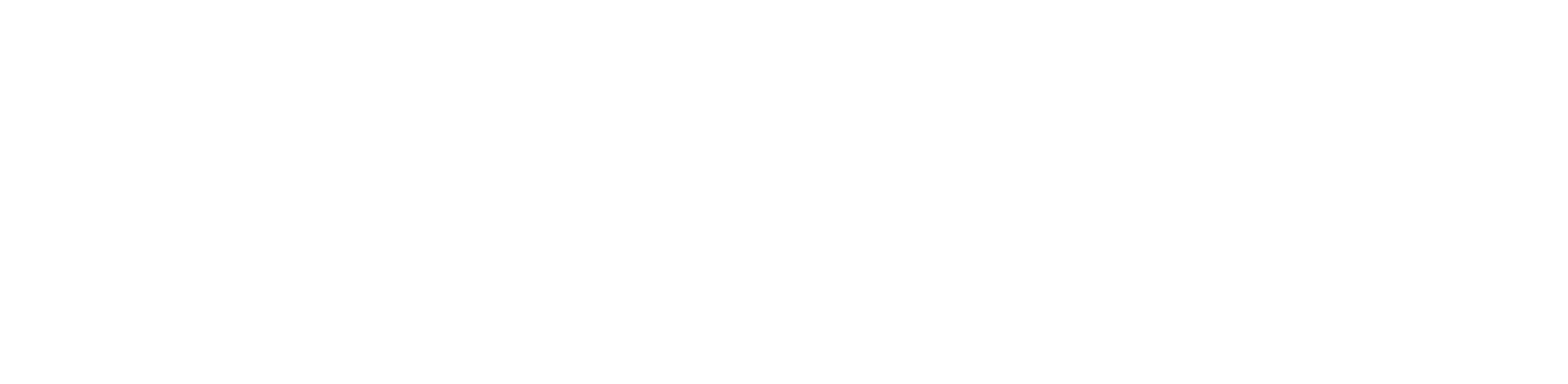
There is a curious and confusing inconsistency in JSP syntax when non-String tag attributes are the results of JSP expressions. Let's suppose we want to pass an object of class examples.Values (a kind of list) to a tag extension. The syntax:

`<apress:list values="<% =values%>">`

is problematic, because we know from the JSP specification that an expression "is evaluated and the result is coerced to a string which is subsequently emitted into the current out JspWriter object." In the case of the custom tag above, however, the value of the expression is not coerced to a string, but passed to the tag handler as its original type.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

We then output what we're building up while iterating over the body content:

```
private StringBuffer output = new StringBuffer();
```

Getter and setter for the names property/attribute:

```
public List getNames() {  
    return names;  
}  
  
public void setNames(List names) {  
    this.names = names;  
}  
  
public int doStartTag() throws JspTagException {  
    setLoopVariables();
```

The return value is the same as for a normal, non-iterating, tag:

```
    return EVAL_BODY_INCLUDE;  
}
```

The JSP engine will call this method each time the body content of this tag has been processed. If it returns SKIP_BODY, the body content will have been processed for the last time. If it returns EVAL_BODY AGAIN, the body will be processed and this method called at least once more:

```
public int doAfterBody() throws JspTagException {
```

If we still haven't got to the end of the list we continue processing:

```
if (++index < names.size()) {  
    setLoopVariables();  
    return EVAL_BODY_AGAIN;  
}
```

If we get to here, then we've finished processing the list:

```
    return SKIP_BODY;  
}
```

This method is implemented to direct the JSP engine to evaluate the rest of the calling JSP page, and to expose the two variables defined with scope AT_END:

```
public int doEndTag() {  
    pageContext.setAttribute("className", getClass().getName());  
    pageContext.setAttribute("date", new Date());  
    return EVAL_PAGE;  
}
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



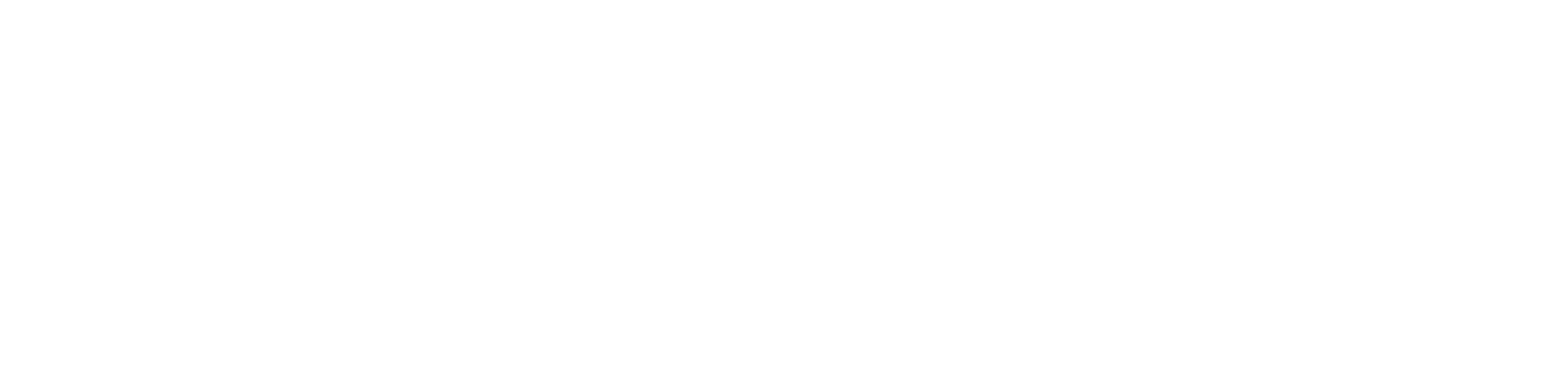
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



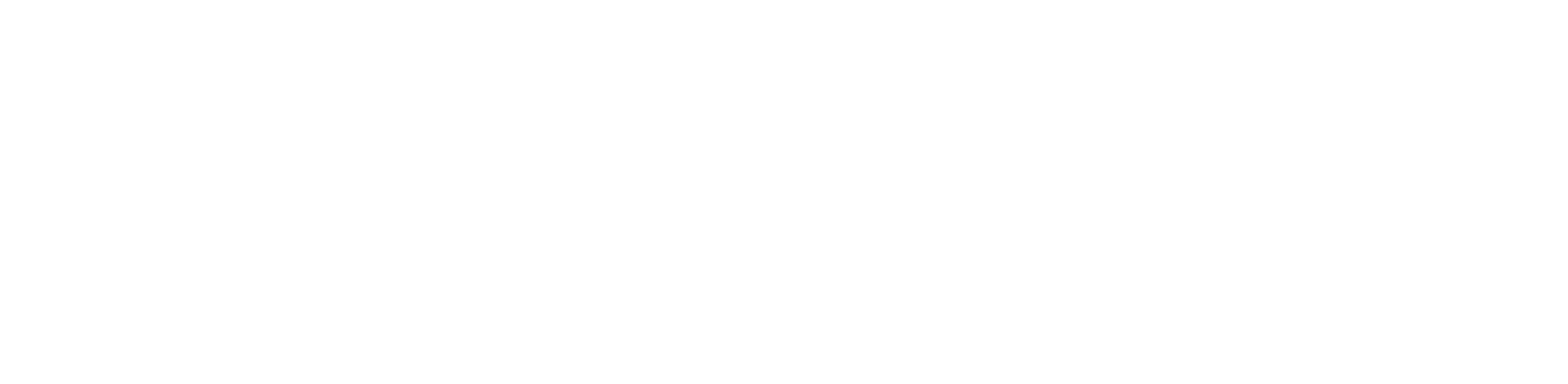
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
    public Date getCreationDate() {
        return new Date(timeCreated);
    }
}
```

Since the `ExampleTLDListener` class has been registered as a servlet context listener, all tags in the examples library can assume that the shared object will be available through the `PageContext` object available to them at run time. The following simple tag obtains a reference to the shared object, and prints out its creation date to JSP pages that use it:

```
package tagext.listeners;

import java.io.IOException;
import java.util.Date;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.TagSupport;

public class SharedObjectAccessTag extends TagSupport {

    public int doEndTag() throws JspTagException {
        String dateString = new Date().toString();
        try {

```

Get hold of the shared object:

```
            DummySharedObject dso =
                (DummySharedObject) pageContext.getAttribute(
                    ExampleTLDListener.SHARED_OBJECT_NAME,
                    PageContext.APPLICATION_SCOPE);
            if (dso == null) {
                throw new JspTagException(
                    "Error: shared object not found in servlet context. " +
                    " It should have been made available by the " +
                    "ExampleTLDListener");
            }
            pageContext.getOut().write("I got hold of the shared object, " +
                "and it was created at " +
                dso.getCreationDate());
        } catch (IOException ex) {
            throw new JspTagException("Fatal error: " +
                "hello tag could not write to JSP out");
        }
    }
}
```

This return value means that the JSP engine should continue to evaluate the rest of this page:

```
        return EVAL_PAGE;
    }
}
```

Before we can access the tag from JSP pages, we will need to add the following `<tag>` element to our TLD, `hello.tld`. It uses exactly the same syntax as the `<tag>` elements we've seen before, as the new tag handler's access to the shared object is a Java implementation detail that doesn't impact its use from JSP pages:

```
<tag>
    <name>listenerTest</name>
    <tag-class>tagext.listeners.SharedObjectAccessTag</tag-class>
    <body-content>JSP</body-content>
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Tag extensions are a valuable way of separating presentation from content in JSP interfaces. Since they are a standard part of JSP, there are a growing number of third-party tag libraries, which are becoming increasingly valuable building blocks in JSP development.

Importantly, once the initial concepts are grasped, tag extensions are remarkably easy to develop. The tag extension mechanism allows us to build, portable, simple, and expressive extensions to our JSP pages – all built upon existing concepts and machinery.

In the next chapter we will go on to look at some more complex examples of tag extensions.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



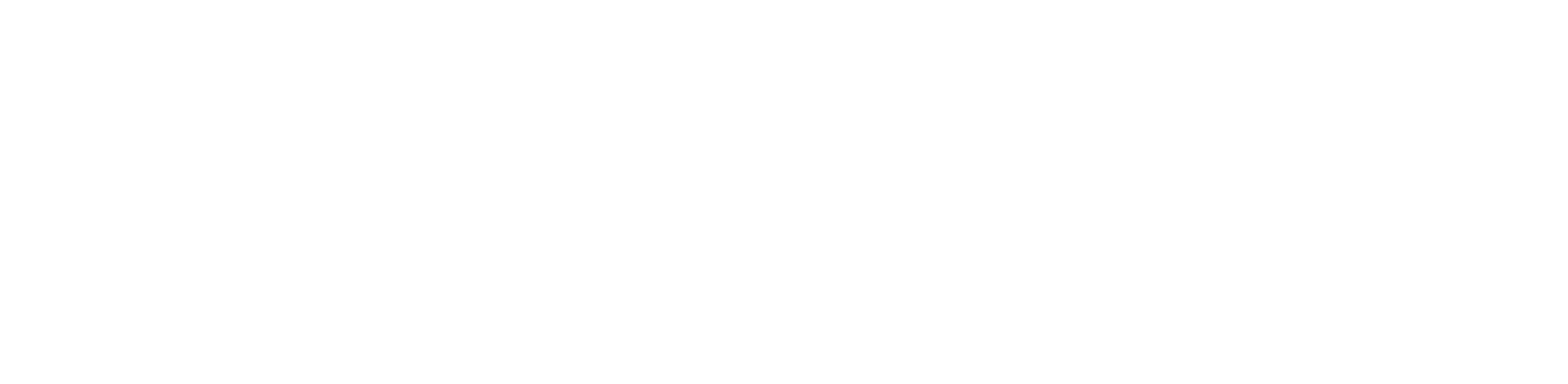
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

- **How a tag collaborates with its environment.** Tags usually collaborate with their environment implicitly or explicitly. Tags collaborate implicitly with their environment by implementing a well-defined interface that can be exposed by the ancestor tag hierarchy (through the `TagSupport.findAncestorWithClass(...)` method). Tags collaborate explicitly by exposing information to the environment in which they operate. Traditionally, within a JSP scripting environment, this has been achieved by exposing one or more scripting variables.

The JSPTL design team has elected not to expose information using scripting variables as a general case. Rather, explicit collaboration is achieved by exposing information via scoped attributes, set using the `ServletContext` class.

With these design considerations in mind, the next section introduces the most basic tags that make up the JSPTL.

Some Basic Tags

The following table shows the JSPTL tags that are used in conjunction with the other tags in the JSPTL. These tags are concerned with returning the value of an expression-language expression value, either directly onto the JSP page, or through a scoped attribute. As a knock-on effect of the decision of the JSPTL team to make the JSPTL tags create scoped attributes on demand, through the `ServletContext`, these tags are concerned with making scoped attributes available to the JSP page or the traditional JSP scripting environment.

| Tag | Attribute | Description |
|-------------------|----------------------|---|
| <code>expr</code> | | Evaluates an expression-language expression and outputs the result to the JSP page. Similar in behavior to JSP scriptlet expressions, <code><%= ... %></code> . Example: <code><jx:expr value="\$employee.address.postcode"/> <jx:forEach var="color" items="\$colors"> <TR bgcolor="#<jx:expr value='\$color.RGBValue' />"> <TD><jx:expr value="\$color.name"/></TD> <TD><jx:expr value="\$color.RGBValue"/></TD> </TR> </jx:forEach></code> |
| | <code>value</code> | An expression-language expression, as expected by the configured expression-language. |
| | <code>default</code> | A default expression to evaluate and return the value of, if the expression fails to evaluate. The body of the <code>expr</code> tag can also be used to contain a default or fallback value if the expression did not evaluate. |
| <code>set</code> | | Sets the result of an expression into a named, scoped attribute. Example: <code><jx:set var="postCode" value="\$employee.address.postCode"/> <jx:set var="userName" value="\$user.userName" scope="request"/></code> |

Table continued on following page



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

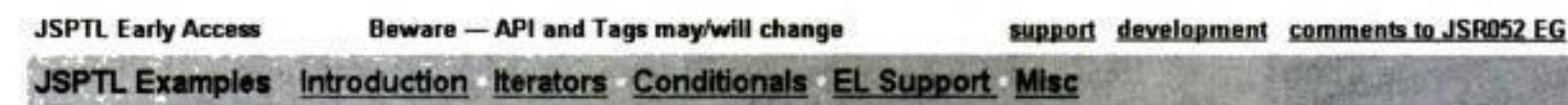


You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The examples WAR file for the JSPTL requires the latest version of Tomcat to run (available from <http://jakarta.apache.org/tomcat/index.html>). To get the examples running, simply drop the `jsptl-examples.war` into the `webapps` directory of Tomcat and restart the server. The following screen shows the welcome page of the JSPTL examples application:



Welcome to the jsptl-examples web application!

This web application includes a variety of sample JSP pages that showcase the JSPTL tags currently being specified within the JSR-052 Expert Group.

WARNING: This is EARLY ACCESS (EA). The goal of this EA program is to keep the community informed of the EG's progress as well as to give the community a chance to experiment with the standard tag library (JSPTL) early in the specification process so that valuable feedback can quickly be channelled back to the Expert Group.

Documentation

Copyrighted image

Documentation on the JSPTL tags is available at <http://jakarta.apache.org/taglibs/doc/jsptl-doc>. It is also available as the `jsptl-docs` web application of the jsptl EA1 release.

Mailing Lists

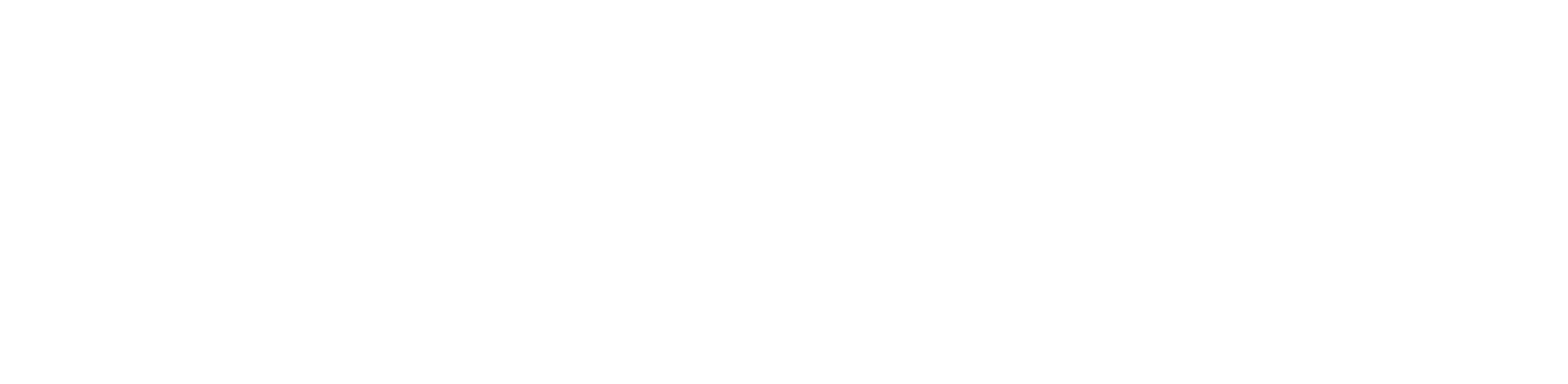
There are three ways to obtain information from or send your comments to the JSR052 Expert Group:

1. **Support** For support/usage questions, please use the [user mailing list](#) of jakarta-taglibs.
2. **Development** For bugs, development related questions, please use the [developer mailing list](#) of jakarta-taglibs
3. **Comments to the JSR052 EG**: To provide the Expert Group with feedback on the JSPTL, use either the [developers mailing list](#) at Jakarta, or you may contact the expert group privately at jsr052-comments@sun.com. All comments will be read, but we cannot guarantee a personal reply to all messages received.

Examples

The JSPTL examples have been divided in the following category:

- [Iterator Tags](#) (`<forEach>`, `<forToken>`)
- [Conditional Tags](#) (`<if>`, `<choose>`)
- [Expression Language Support Tags](#) (`<expr>`, `<set>`, `<define>`)
- [Miscellaneous](#) (various tests)



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

To round off the discussion about the JSPTL iteration tags let's discuss the following issues:

- Within the iteration cycle, obtaining information about the current status of that cycle
- Extending the iteration tags to provide custom forms of iteration

These two issues are discussed in the next sections.

Iteration Status

The `forEach` and `forTokens` tags each provide an object that can be used to establish the status of the iteration process. The name of the scoped attribute created can be specified via the `status` attribute. The type of the object is `javax.servlet.jsp.tagext.IteratorTagStatus`. The following table shows the main properties of the `IteratorTagStatus` object created:

| Property | Type | Description |
|----------------------|----------------------|---|
| <code>current</code> | <code>Object</code> | The current item of the list of items iterated over. |
| <code>index</code> | <code>int</code> | The zero-based index of the current item in the list of items iterated over. |
| <code>count</code> | <code>int</code> | <p>The one-based position of the current item in the set of items eligible to be iterated over. This value increases by one for each item, regardless of the <code>begin</code>, <code>end</code>, or <code>step</code> attributes of the iteration tag.</p> <p>For example, if <code>begin=1</code>, <code>end=10</code> and <code>step=2</code> then the set of items eligible for iteration over is 1, 3, 5, 7, and 9. These have position counts of 1, 2, 3, 4, and 5 respectively.</p> |
| <code>first</code> | <code>boolean</code> | Indicates whether the current item is the first item in the iteration cycle. |
| <code>last</code> | <code>boolean</code> | Indicates whether the current item is the last item in the iteration cycle. |

The iteration tag status example, run and inspected in the last section, showed how the status information was put to good use.

Iteration Tag Extensibility

Most iterative tasks can easily be achieved with the JSPTL iteration tags. However, failing that, the iteration tag mechanism is extensible. If you can't achieve what you need with the JSPTL tags, you can extend the standard tags and define your own.

The following table shows the three types the JSPTL provides to developers to aid in the creation of custom iteration tags:

| Type | Description |
|--|--|
| <code>interface IteratorTag</code> | The main interface for writing custom iteration tags. |
| <code>interface IteratorTagStatus</code> | The JSPTL iteration tags provide the status of the iteration cycle within the scope of the tag. Custom iteration tags that conform to the <code>IteratorTag</code> interface must also provide this information through the <code>IteratorTagStatus</code> object, as discussed in the previous section, <i>Iteration Status</i> . |



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The `handleRequest()` method is what's interesting here. It contains the 'rules' we discussed, for mapping requests to the JSP pages invoked. We can see how the URLs shown in the diagram above are mapped to their corresponding JSP. For example, how `/registration/register` is mapped to the `registerNewUser` method that sets up some data (user, roles, and regions) and then forwards on to `/registration/regForm.jsp`, which shows the registration form.

Let's now take a look at the other JSP pages of the application.

The Registration Form Page

As a new user, the first step within a registration system is to record some information about you. In our application you can achieve this through option one of the welcome page, or directly via <http://localhost:8000/jspTagApps/registration/register>.

The controller servlet maps the request to a form for entering new registration details, `regForm.jsp`, shown below:

```
<%@page import="java.util.*" %>
<%@page import="java.text.SimpleDateFormat" %>

<!-- Bring in the HTML Helper tag library -->
<%@taglib uri="/html" prefix="html" %>

<jsp:useBean id="user" type="writingjsps.registration.User"
    scope="request" />

<html>
<head>
<title>Registration</title>
</head>
<body>
<font face="arial, helvetica" size="4" color="#996600">Registration</font>

<html:form method="post" actionURI=".//regFormSubmit">
    <table border="0">
        <tr>
            <td>User Name:</td>
            <td>
                <html:textInput name="username"
                    value="<% user.getUsername() %>"
                    numberOfColumns="20" />
            </td>
        </tr>
        <tr>
            <td>First Name:</td>
            <td>
                <html:textInput name="forename"
                    value="<% user.getForename() %>"
                    numberOfColumns="20" />
            </td>
        </tr>
        <tr>
            <td>Last Name:</td>
            <td>
                <html:textInput name="surname"
                    value="<% user.getSurname() %>"
                    numberOfColumns="20" />
            </td>
        </tr>
    </table>
</html:form>
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```

...
<tr>
  <td>Password:</td>
  <td>
    <html:passwordInput name="password" numberOfRows="20" /><br>
  </td>
</tr>
<tr>
  <td>Confirm Password:</td>
  <td>
    <html:passwordInput name="password" numberOfRows="20" /><br>
  </td>
</tr>
</table>

<input type="submit" value="Register!">
</html:form>

```

The registration form, `regForm.jsp`, uses the `form`, `textInput`, `passwordInput`, and `list` tags of the library. When the form is submitted, the registration controller servlet processes the form and performs a check with the chosen username. If a user with the chosen username does not already exist, it stores the user's details in the user database and forwards the new registrant to a "thank you" page. If the chosen username is already taken, the registration controller forwards to a registration error page, accordingly.

Both the thank you page and the registration error page are discussed in the next two sections.

Thanks for Registering!

Once registered, a thank you page is displayed. As one might expect, the `thankyou.jsp` page is really quite small, and yet it will introduce a very powerful concept in JSP pages. It's a concept not seen too often in web pages – not often enough, really. Lets start by taking a look at the JSP code:

```

<%@page import="writingjsps.registration.*" %>
<%@taglib uri="/internationalisation" prefix="i18n" %>
<jsp:useBean id="user" type="writingjsps.registration.User" scope="request" />
<%-- Load the resource bundle for the page --%>
<i18n:resourceBundle name="writingjsps.registration.RegistrationResources">
<%-- Display the greeting message, inserting the user properties --%>
<i18n:message name="greetingMessage" arg0="<% user.getForename() %>">
  arg1="<% user.getUsername() %>" /><br>
<html>
<head>
<title>Thank You</title>
</head>
<body>
<p>Thanks for registering!<br>
<p>Be sure to check out your user profile by<br>
clicking <a href="displayDetails">here</a>.
<p>In the meantime, have fun working with JSP tag library applications.

```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

| Tag | Attribute | Description |
|-------------------|------------------|---|
| | value | The initial value of the numeric. This can be a numeric constant, primitive or wrapper type of type short , integer , long , float , or double . Defaults to zero. |
| | scope | The scope of the variable: page , request , session , or application . Defaults to page scope. |
| Boolean | | Declares a boolean script variable and optionally assigns a value to it. |
| | id | The name of the scripting variable to be created. |
| | value | The initial value of the boolean variable: true or false . Defaults to false . |
| | scope | The scope of the variable: page , request , session , or application . Defaults to page scope. |
| dateFormat | | Outputs a string representation of a date, formatted according to the specified format parameter for the specified locale and time zone. |
| | date | The java.util.Date to be formatted. |
| | format | The format that the string representation of the date is to take. This is based entirely on the patterns provided by the java.text.SimpleDateFormat class in the Java 2 Standard Edition. For example, "yyyy.MM.dd hh:mm:ss" would produce output of the form "1997.10.02 15:10:25" and "EEE, MMM d, yyyy" would produce output of the form "Wed, August 19, 1998". |
| | timeZone | The name of a known time zone. Specified to offset the time to a specific time zone and cater for daylight saving time. |
| | locale | A locale is used to sensitize formatting of the date to a particular locale. If not specified, the clients preferred locale is used, if possible, otherwise the default locale of the server is used. |
| log | | Logs its body or specified message using the Servlet Logging API. |
| | message | Specifies the message to be logged. |
| | throwable | Specifies a java.lang.Throwable to be logged. |
| encode | | A tag that URL-encodes its body. Refer to the java.net.URLEncoder class for further information. |

The declaration tags, **number** and **boolean**, provide an easy way for JSP authors to declare and use scripting variables within their JSP pages. This is particularly useful, over declaration of variables in JSP scriptlets, when the page author is not necessarily an expert in Java or for creating scripting variables whose scope is beyond that of the JSP page on which they are declared.

The **number** tag creates a numeric scripting variable without paying too much attention to the explicit numerical type. In fact, because the tag is based on the **java.lang.Number** type, it can represent the values of any numeric primitive or primitive wrapper type. Here's the example taken from the user profile display page:



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Summary

We've covered a lot in this chapter, looking at the benefits to developers, vendors, and businesses of standardizing some of the most commonly-used tags today. There's a lot of work going into this process at the moment, and the introduction of the JSPTL promises to be exciting and beneficial all round.

After reading this chapter, you've hopefully gained a firm understanding of some of the common patterns and usage of the Java 2, Standard and Enterprise Edition APIs that can be exploited through a set of common and consistent tags. We looked at:

- The benefits of using custom tag libraries
- Some of the current vendor-specific tag libraries out there today
- The proposed introduction of the JSP Standard Tag Library

We then took a glimpse at a registration application, used throughout this chapter, as the main example. The registration application was built to use and exploit to the fullest the sample JSPTL-like tags outlined in the sections that followed.

Having introduced the main example, we looked at several aspects of day-to-day JSP development, covering tag libraries for:

- Fundamental tags, such as those for declaration, control flow, data and time handling, server-side includes, and other miscellaneous tags
- Internationalization
- Wrapping HTML forms
- Creating XML views of data contained in a relational database

With these concepts in mind, we're now ready to start using common JSP tags within our applications. Look out for the JSPTL; it's going to exploit all of these concepts and, in doing so, create a storm on the Java and web technology fronts.

In the next chapter, we move on to look at the JavaMail API, and how we can use it.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



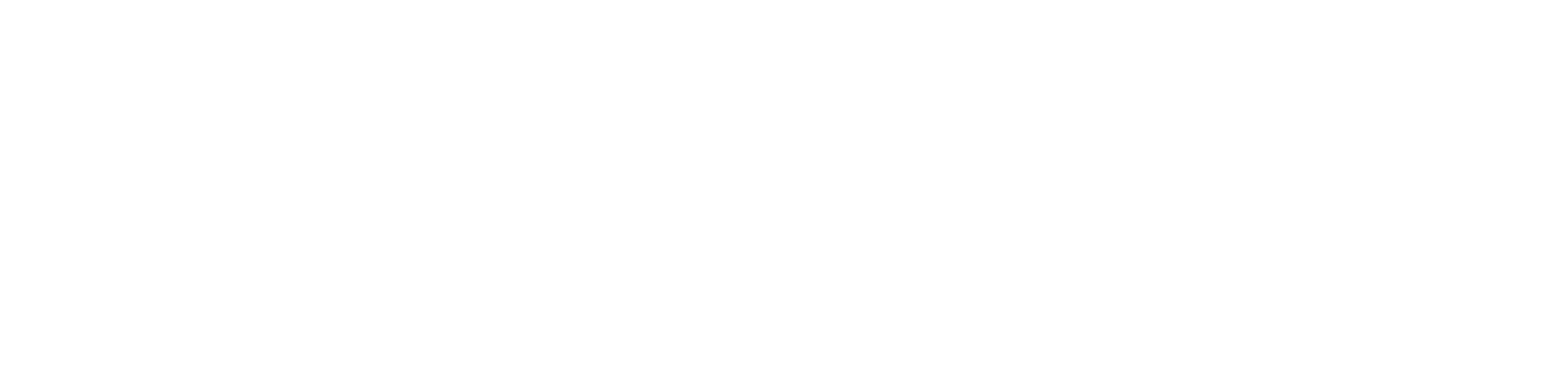
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



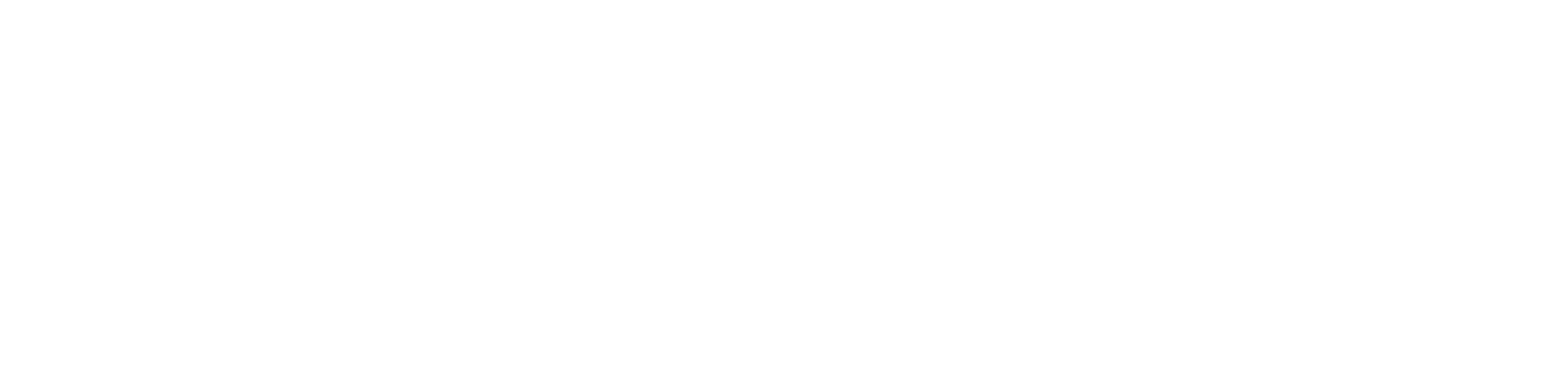
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



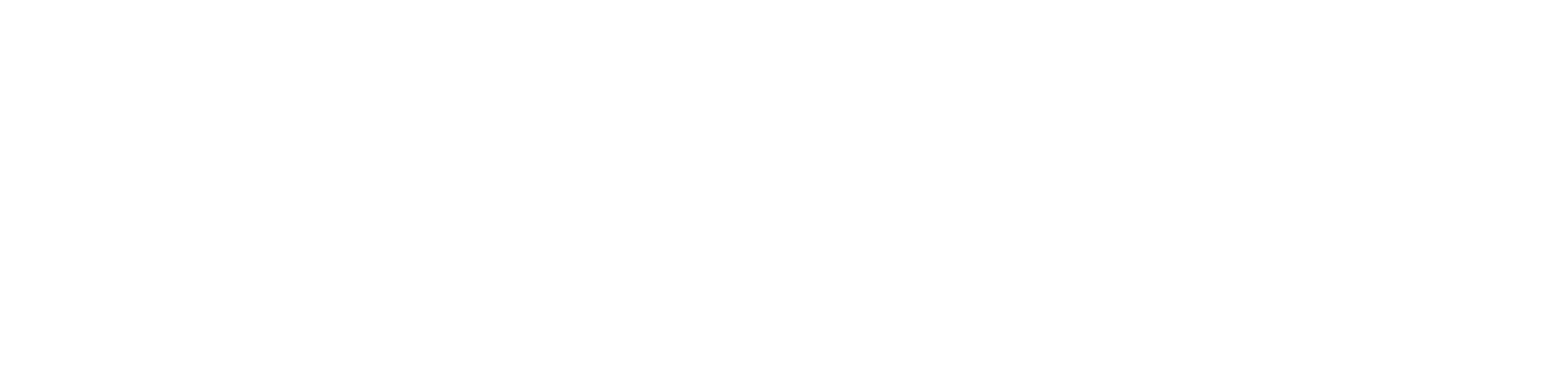
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



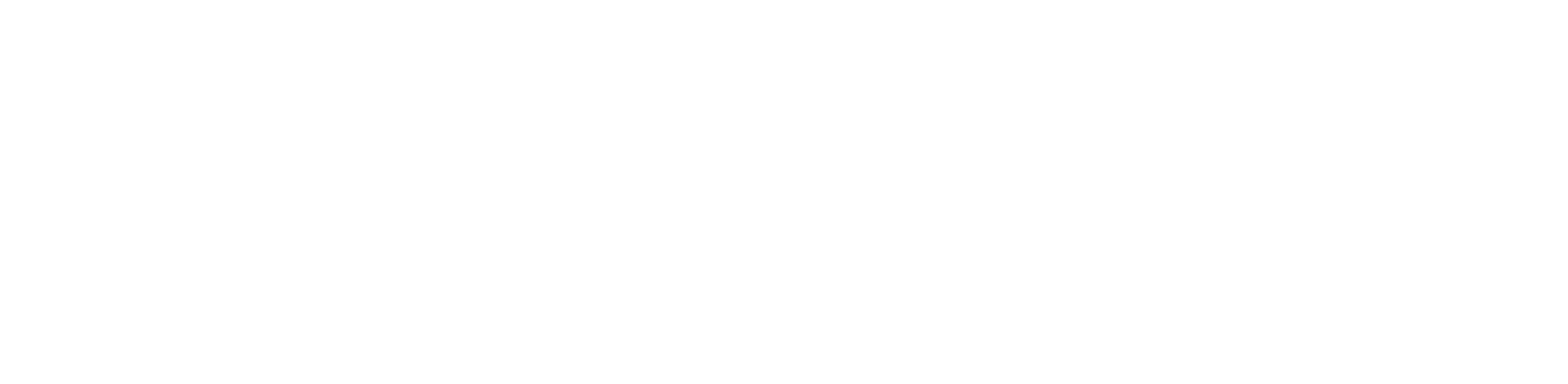
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

To see some of these details in action, let's look at a quick example that, on the surface, looks complicated but, as we will see, is actually very easy to implement. Let us assume that we wanted to search for all messages from `alan@n-ary.com` or `javamailhelp@apress.com` that have the phrase `JavaMail` somewhere in the subject header:

```
SearchTerm st1 = new OrTerm(new FromStringTerm("alan@n-ary.com"),
                           new FromStringTerm ("javamailhelp@apress.com"));

SearchTerm st2 = new AndTerm(st1, new SubjectTerm("JavaMail"));
Message[] messageList= thisFolder.search(st2);
```

As we can see, it is a simple matter of building up the logical comparison using the various classes available through the `javax.mail.search` package.

javax.mail.Transport

The final class in our exploratory look at the JavaMail API is the class that is responsible for the delivery of messages, `javax.mail.Transport`. Most of the time we will be using the SMTP protocol for delivery and, as a convenience, the `Transport` class offers a static method for the sending of messages, as we have seen earlier in the chapter.

```
Transport.send(msg)
```

However, if we wish to have a little more control over the delivery of the message, consider the example below that implicitly gets the specific `Transport` object instance, manually connects, and then performs a send on the message.

```
try {

    Transport myTransport = session.getTransport("smtp");
    myTransport.connect();
    myTransport.sendMessage(msg, msg.getAllRecipients());
    myTransport.close();

} catch (SendFailedException sfe){

    Address[] list = sfe.getInvalidAddresses();
    for (int x=0; x < list.length; x++) {
        System.out.println("Invalid Address: " + list[x]);
    }

    list = sfe.getUnsentAddresses();
    for (int x=0; x < list.length; x++) {
        System.out.println("Unsent Address: " + list[x]);
    }

    list = sfe.getValidSentAddresses();
    for (int x=0; x < list.length; x++) {
        System.out.println("Sent Address: " + list[x]);
    }
}
```

The advantage of this mechanism, as opposed to the static call, is that if we are sending large numbers of messages, the underlying protocol doesn't have to keep connecting to the server for each message. Instead the same connection is used. The important feature to notice here is the `try ... catch` block.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

We can compile and run the program in the usual way; here's a sample of the output below:

Copyrighted image

Now that we've seen how easy it is to send messages and attachments, let's take a look at another aspect of mail management – reading mail.

Reading Mail

Receiving e-mail is as simple as sending it, as long as we follow the proper steps. With this in mind, there are a number of things we can demonstrate. With the following example, we'll illustrate the major concepts of dealing with mail, by building a simple command-line access tool for POP3 mail.

This will be a very simple tool: nothing too fancy and it most certainly will not replace our Outlook or Eudora clients! So don't expect too much.

Essentially it will list all the messages held on a POP server and allow the user to interact with this list. But first of all let's build the framework for this application.

```
import java.util.*;
import java.io.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;

public class javamail_pop extends Object {

    public static void main(String args[]) {

        if (args.length != 1) {
            System.out.println("Usage: javamail_pop <urlname>");
            System.exit(1);
        }
    }
}
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
import java.util.*;
import java.io.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;

public class javamail_sendAll extends Object {

    public static void main(String args[]){
        if (args.length == 0) {
            System.out.println("Usage: javamail_sendAll <directory>");
        } else {
            new javamail_sendAll(args[0]);
        }
    }

    public javamail_sendAll(String spoolDirectory){
        processMailList(spoolDirectory);
    }

    private void processMailList(String spoolDirectory) {
        File rootDir = new File(spoolDirectory);
        String listOfFile[] = rootDir.list(new fileFilter());

        for (int x=0; x < listOfFile.length; x++) {
            File thisFile = new File(rootDir, listOfFile[x]);

            sendMail(thisFile);

            thisFile.delete();
        }
    }

    private void sendMail(File filename) {
        String To = filename.getName(), From = "", Subject="";
        String mailServer = "";

        try {
            // Load in the file
            BufferedInputStream in = new BufferedInputStream(
                new FileInputStream(filename));

            // Set the Session and Server properties
            Session mailSession = Session.getInstance(new Properties());

            MimeMessage msg = new MimeMessage(mailSession, in);
            in.close();

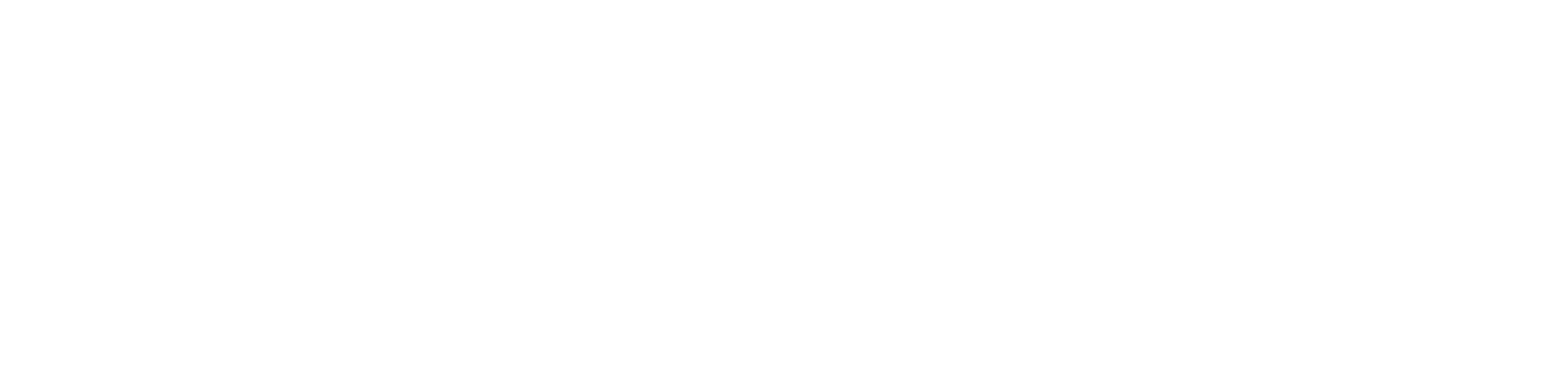
            // Message is now in, deal with the custom header
            mailServer = msg.getHeader("X-Server", ",");
            msg.removeHeader("X-Server");

            // Now we need to deliver it
            Properties props = mailSession.getProperties();
            props.put("mail.smtp.host", mailServer);

            // Set the To, From
            To = msg.getHeader("To", ",");
            From = msg.getHeader("From", ",");
            Subject = msg.getSubject();
        }
    }
}
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Copyrighted image

EJB components, servlet components, and JSP components have more in common with each other, than any of them has with JavaBeans. The following chart will help you to understand how EJB technology is related to other Java component technologies.

| Component | Purpose of component | Tier of execution | Typical services provided by the runtime execution environment |
|----------------------|---|---|---|
| JavaBeans | General-purpose component architecture | Any | Java libraries. Other services vary greatly with the container (for example, Microsoft Word document, Java application). |
| Servlets | Implements a request-response paradigm, especially for web protocols | Server (specifically the web tier) | Lifecycle services, network services, request decoding, response formatting. |
| JavaServer Pages | Provides for the generation of dynamic content, especially for web environments | Server (specifically the web tier) | All servlet services, scripting, tag extensions, response buffering. |
| Enterprise JavaBeans | Provides for the encapsulation and management of business logic | Server (specifically the business-logic tier) | Persistence, declarative transactions and security, connection pooling, lifecycle services, support for messaging services. |

Varieties of Beans

The primary goal of EJB technology is to provide standard component architecture for the creation and use of distributed object-oriented business systems. This ambitious goal could not be met with only one model for a component, so the EJB 2.0 specification provides three models for callback methods and run-time lifecycles. The three types of EJB that implement these models are **entity beans**, **session beans**, and **message-driven beans**.

The differences between these three types of EJB are quite complex, and so each is covered in a chapter that explains the callback methods of the EJB as well as where and when they should be used. (Chapter 15 covers session beans; Chapter 16 entity beans, and message-driven beans are discussed in Chapter 19.) However, as this distinction is so fundamental, we'll review the key differences here.

A session bean is intended for use by a single client at a time, so we can think of it almost as an extension of the client on the server. It may provide business logic, such as calculating the rate of return for an investment, or it may save state, such as a shopping cart for a web client. A session bean's lifespan is no longer than that of its client. When the client leaves the web site, or the application is shut down, the session bean is free to disappear. It is no longer available for access by other clients.

Think of an entity bean as an object-oriented representation of data in a database. Like a database, it can be accessed simultaneously by multiple clients. An entity bean might represent a customer, a product, or an account. The lifespan of an entity EJB is as long as that of the data it represents in the database.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

We will not address the technical requirements of the specification from the container developer's point of view but we will take a closer look at those aspects of EJB development that are directly relevant to the server-side programmer.

The Client Developer's View

A client is any user of an Enterprise JavaBean, and could be a Java client-side application, a CORBA app, a servlet, or even another EJB. A server-side programmer designing a web application, or using a servlet to mediate communication with an EJB, needs to understand how EJBs are accessed and used. In large projects, it is quite likely that the web programmer and the EJB programmer are different people.

The client programmer has a smaller set of concerns than a bean developer with regard to using EJBs. They need to know how to find or create the bean, how to use its methods, and how to release its resources. A client always uses the same procedure for object creation, lookup, method invocation, and removal, regardless of how an EJB is implemented or what function it provides to the client. The client doesn't need to worry about the implementation of the EJB, callbacks that the EJB container will make on the EJB, or the nature of the services provided to the EJB.

Session and entity beans have the following interfaces:

- A **home interface**, primarily for lifecycle operations such as creating, finding, and removing EJBs. The home interface isn't associated with a particular bean instance, just with a type of bean.
- A **remote interface** is for business methods. Logically, it represents the client's view of a particular bean instance on the server. The remote interface also provides some infrastructure methods associated with a bean instance, rather than a bean type. Alternatively, the bean may use a **local interface**, which is similar to the remote interface, but is only accessible by EJBs in the same deployment unit.

It is important to understand that session and entity beans must have a home interface, but it's optional whether they have either a remote interface or a local interface (or both).

A client programmer will acquire a home interface through JNDI (which is explained in Chapter 2). This home interface can then be used to:

- Create or find an instance of a bean, which will be represented to the client as a remote interface
- Execute business methods on that instance of the bean
- Get a serializable reference to the bean, known as a **handle**
- Remove the bean

The removal of a bean can mean radically different things depending on the type of bean and it's important to be clear on these differences. In the case of a stateful session bean, it means the developer has finished using it. The server can then release the resources associated with that bean. In the case of a stateless session bean, it means the same thing – although the server probably wasn't consuming any resources for the bean because it lacked state to begin with. Since stateless bean instances are pooled, it could also mean that the instance that's in use is returned to the pool of available session beans. In the case of an entity bean, removing the bean means removing its object view representation in the persistent data store – in other words, deleting it from the database.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
EJBHome getEJBHome()
    throws java.rmi.RemoteException;

Handle getHandle()
    throws java.rmi.RemoteException;

Object getPrimaryKey()
    throws java.rmi.RemoteException;

boolean isIdentical(EJBObject obj)
    throws java.rmi.RemoteException;

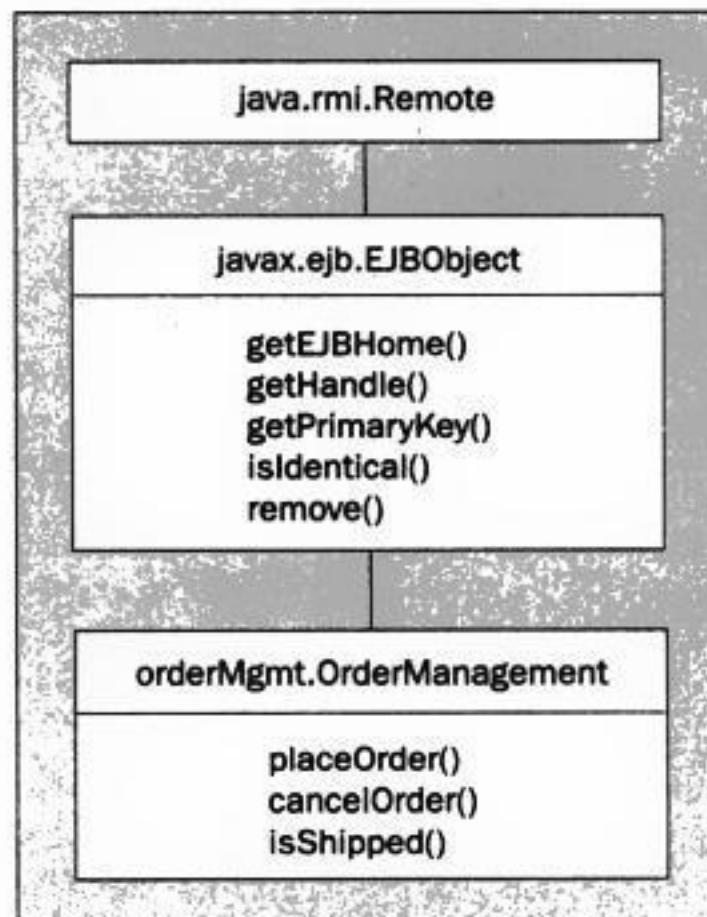
void remove()
    throws java.rmi.RemoteException, javax.ejb.RemoveException;

}
```

For a local interface, the only difference is that the methods do not throw RemoteException in their signature and that they extend EJBLocalObject instead of EJBObject.

The EJB developer, in general, doesn't need to write code to support these methods: they are available to the client developer in every bean that they use. Some won't make sense in certain contexts, in which case calling them results in an exception.

Here is a class diagram of the relationship between the three classes, using our OrderManagement interface for the business interface. Note that `java.rmi.Remote` is a tagging interface with no methods:





You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You should see something like:

Copyrighted image

The client program has run successfully, but hasn't output anything to screen. At first this might seem strange, after all we called the `placeOrder()` method, which includes the command:

```
System.out.println("Order placed for " + quantity + " copies of " +
    prodName + " to be shipped to " + custName);
```

So we might expect to see output to the screen detailing the order that the client placed. However, the method is called on the EJB, which runs on the server, not the client. So, the `placeOrder()` method has been executed on the server. If we look at the command prompt that is running the WebLogic server we see that `placeOrder()` was indeed successfully executed:

Copyrighted image

We can also see that `setSessionContext()` and `ejbCreate()` are called by the EJB container. However, `ejbRemove()` does not appear to have been called despite the fact that we call `remove()` on the instance of the EJB in the client program. This is because it is up to the EJB container when it calls this method.

What an EJB Cannot Do

We've talked at length about the benefits of programming to the EJB specification if you are developing a transactional system. As application developers, we are relieved of system-level programming tasks. This is a great advantage for programmer productivity, application capability, and system reliability. Nevertheless, to really benefit, we must agree to work within the framework of EJB technology, and this means there are certain things we cannot do without losing some degree of portability. Specifically, the EJB 2.0 specification prohibits EJBs from doing the following:



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

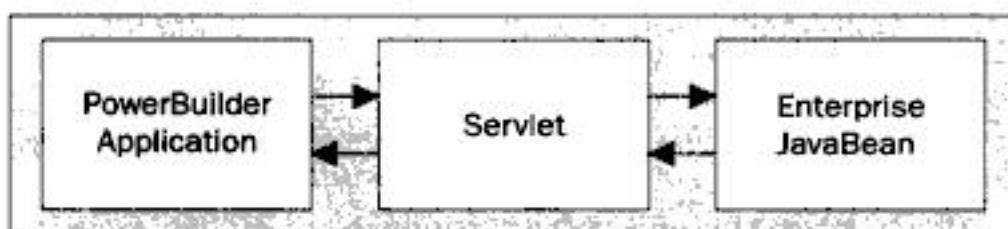
Client-Tier Access to EJBs

Sometimes the presentation logic and workflow are handled on the client tier, as in the case of a Visual Basic application. Such an application might still implement the model-view-controller pattern on the client side, but there would not necessarily be a need to provide this level of structure on the server. There might still be a need to take advantage of services that an EJB container can provide. In the cases where web services are not needed, a client application will probably access EJBs directly.

In the simplest case, a client can access an Enterprise JavaBean remotely using RMI. This is certainly the case for Java GUI clients. A client written in a language other than Java can also access an EJB directly using IIOP (or Internet Inter-ORB Protocol, which is a standard protocol that allows CORBA Object Request Brokers (ORBs) to interoperate).

For this to happen the application server must also function as a CORBA ORB. IIOP communication transport is made mandatory in the EJB 2.0 draft specification, and all compliant application servers will have ORB functionality.

Another possibility that provides maximum flexibility in terms of client access is a model EJB whose data is transformed by a 'view' servlet. The Servlet API provides for servlets that are capable of sending and receiving arbitrary data in a 'request/response' format. A client that requires that a particular format be used to communicate with the server can use such a servlet to mediate communications with the EJBs it needs:



So when would this direct access architecture be appropriate (with or without a mediating 'view' object)? This is the design that would be used to implement a client-server application. Choosing between direct access to EJBs and web access to EJBs is often the same as choosing between a 'thin-client' web app and a 'fat-client' (but still three-tier) client-server app. Many factors will influence this decision, such as client-interface requirements and application-distribution issues.

One rule of thumb is that Internet access (or any distributed access) by a large group of users will typically indicate that a web application is required, and access to enterprise data should be mediated by a view component and a controller component on the web tier. Another rule of thumb is that users on a local network will frequently expect a level of client services that may require a stand-alone GUI client, rather than a web browser; this stand alone GUI can access the Enterprise JavaBeans tier directly. Java applets are the wildcard, combining the distribution advantages of a web application with the presentation abilities of a stand alone GUI. Rather than call a method on an EJB directly using RMI, they will sometimes access EJBs through a servlet, using HTTP to bypass corporate firewalls.

Of course, there's nothing to stop us from directly accessing an EJB from a servlet to provide a simple web page, rather than using the model-view-controller architecture discussed above. If we have a web application of limited scope, and an existing EJB that we want to reuse, this approach can be appropriate. In addition, if we have a web application with limited presentation and navigation needs, but heavy business logic and data access requirements, we might find ourselves directly accessing an EJB from a servlet or a JSP page's helper class. Before we take this approach in a web application, we should consider carefully if there is any chance for the scope of the project to increase – now or in the future.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



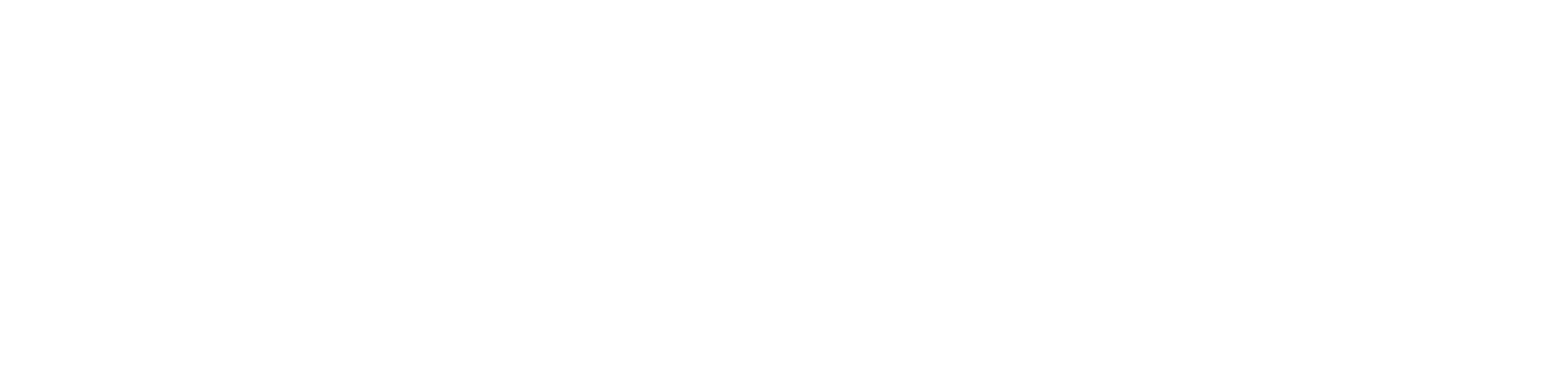
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Compare this diagram to the earlier one for a stateless session bean:

Copyrighted image

The additional complexity is from the container's need to effectively manage per-client state: `ejbActivate()`, `ejbPassivate()`, time outs, and the client's requirement to control the creation and destruction of instances.

The Client for the Stateful Financial Need Calculator

Here is the new client. Unlike the previous version, there is no dependency in the order in which the methods are called. The stateful session bean has taken responsibility to perform the calculations in the correct order at the end of the workflow. Alternatively, it could process the information as it is received, and enforce the workflow by throwing an exception if one of its methods were called out of the correct order:

```
package finCalc.stateful;

import java.util.Properties;
import javax.naming.Context;
import javax.naming.InitialContext;

public class TestClient {

    // Test data

    public static void main(String[] args) {
        try {
            Properties prop = new Properties();
            prop.put(Context.INITIAL_CONTEXT_FACTORY,
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



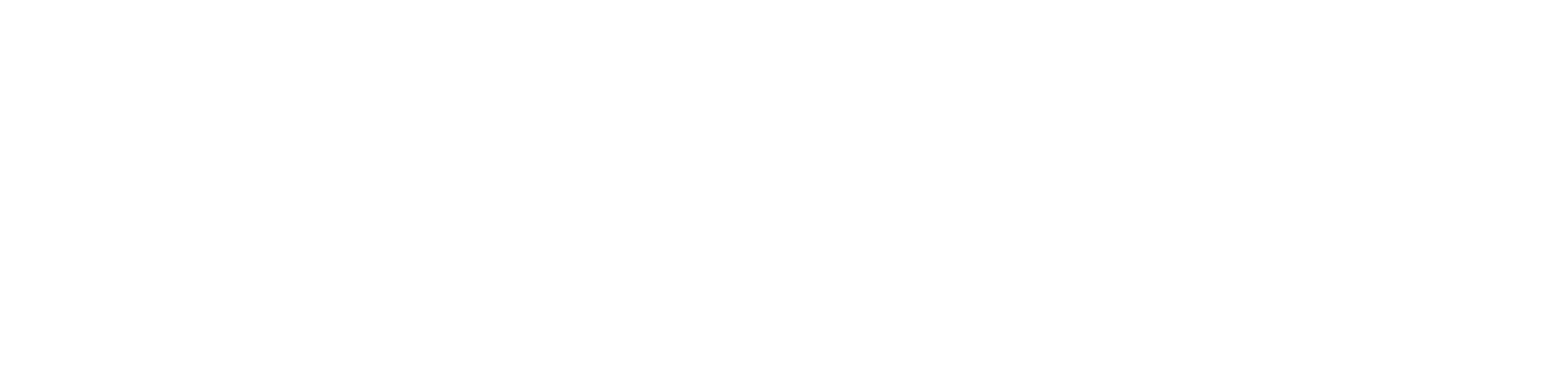
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Copyrighted image



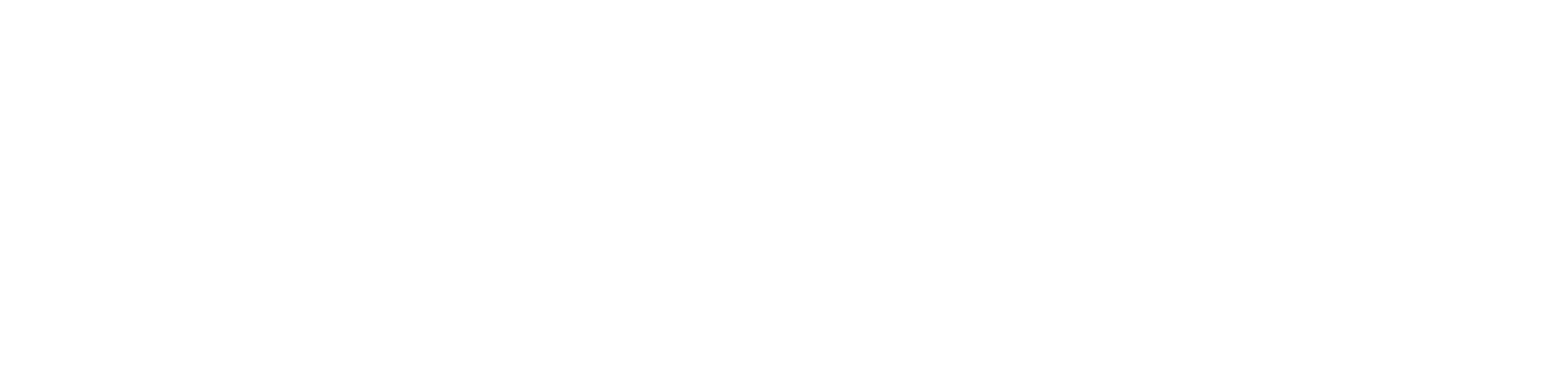
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

These four features are made possible by the introduction of a new actor in the EJB model. Initially called the "Persistence Manager" in early drafts of the EJB 2.0 specification, this actor eventually lost its identity and became part of the container. Therefore, all the EJB 2.0 containers have to implement its services.

What this means for the container is that the tools you use to process your Enterprise JavaBeans (called ejbc in BEA WebLogic, for example) will generate more code in order to provide all those services. In particular, as you will see in the next three sections, the implementation of the above features is made through abstract methods that the container will implement in the generated code.

Abstract Accessors

A CMP entity bean is essentially defined by its CMP fields: Java fields that are mapped to tables in the database. The EJB 1.1 specification was very vague in how these fields were supposed to be represented on the bean class. The EJB 2.0 specification imposes that these CMP fields be defined by abstract accessors instead of Java fields. For example, a CMP field called nickName would be declared as follows in an EJB 2.0 entity bean:

```
public abstract String getNickName ();
public abstract void setNickName(String nickname);
```

Notice that in our database, we might not have a column named nickName; this is OK, since it is only in the deployment descriptor that we specify how to map CMP fields. We'll see this later.

This looks like an innocuous change but it has many consequences on the CMP model in general from a performance standpoint. The most important point is that the container now has total control on the access of fields. Since it will generate the code for each accessor, it is able to make all kinds of decisions and optimizations on how to retrieve the data. The EJB 2.0 specification doesn't mandate any particular implementation, but here are some benefits that EJB 2.0 containers typically provide thanks to this new design:

- **Lazy loading of beans**

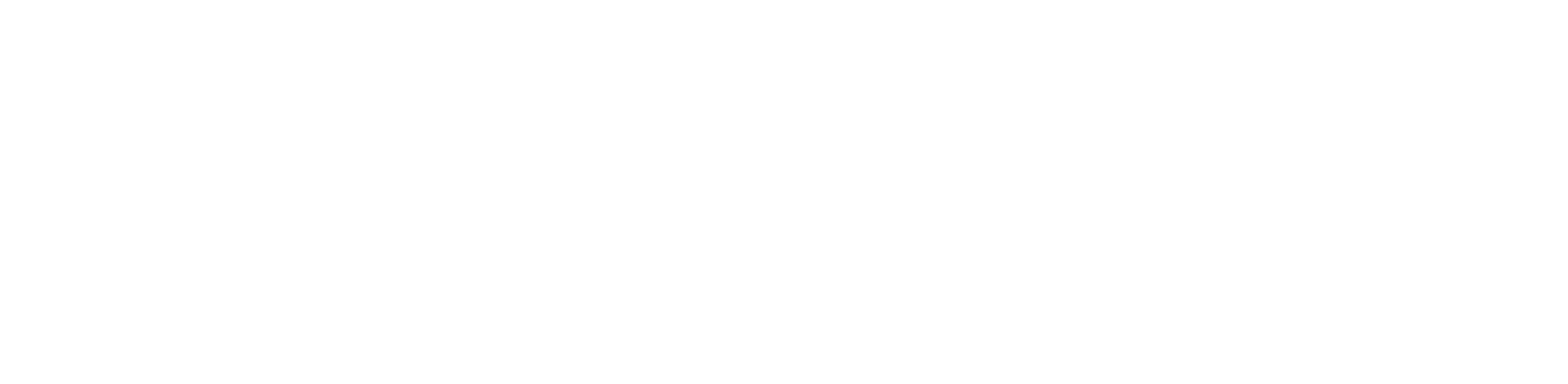
Now that the container knows exactly when and what fields are being accessed, it no longer has to load the entire state of the bean on `ejbLoad()`. It can decide to do nothing when the bean is initially fetched and only decide to start loading the state on the first call of one of the getters. This can be especially important when your application starts and hundreds of beans are being fetched in memory on start-up. It is also a very important feature when it is coupled to "group loading" (see below).

- **Tuned updates**

A major flaw of the EJB 1.1 CMP model and how it failed to define access to CMP fields is that the container couldn't tell what fields had been modified during a transaction. Consequently, container vendors started providing proprietary extensions to the model in order to get this information. The most common one is the introduction of a method `boolean isModified()` which returned `true` if the bean had changed one of its fields during the transaction. This approach has two severe drawbacks:

- The bean provider must implement it, which involves keeping track of the fields that have been modified
- It is not fine grained: all the container knows is that the bean has been modified, but it doesn't know specifically what fields

The EJB 2.0 abstract accessors address all these shortcomings. The bean provider no longer has to track which fields were modified (after all, this is container-managed persistence, so we should rightfully expect the container to do all this work for us) but, most of all, it knows exactly what fields have been modified. Consequently, when the transaction commits, it is able to issue a "tuned update" to the database: a SQL request that will update only columns corresponding to fields modified during the transaction.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

A bean that uses bean-managed persistence would need to use the entity context to retrieve its associated primary key in the implementation of `ejbLoad()`, using the `setEntityContext()` / `unsetEntityContext()` pair of callbacks when they are invoked by the container to save the entity context for use. Add the following methods to the bean-managed persistence version only of the sports team entity bean:

```
public void setEntityContext(EntityContext ctx) {
    this.ctx = ctx;
}

// The container will call this method. Any resources
// that you allocate in setEntityContext() should be freed here.
public void unsetEntityContext () {
    ctx = null;
}
```

The implementation for these methods may be empty for the container-managed persistence version:

```
public void setEntityContext(EntityContext ctx) {}

public void unsetEntityContext() {}
```

Here is the BMP version of `ejbLoad()` for the sports-team bean. Notice how we use the saved EntityContext to retrieve the primary key:

```
public void ejbLoad() {
    SportTeamPK primaryKey = (SportTeamPK) ctx.getPrimaryKey();

    Connection con = null;
    try {
        con = getConnection();
        PreparedStatement statement =
            con.prepareStatement("SELECT OWNERNAME, FRANCHISEPLAYER " +
                               "FROM SPORTSTEAMS WHERE SPORT = ? " +
                               "AND NICKNAME = ? ");
        statement.setString(1, primaryKey.getSport());
        statement.setString(2, primaryKey.getNickName());

        ResultSet resultSet = statement.executeQuery();
        if (!resultSet.next()) {
            throw new EJBException("Object not found.");
        }
        sport = primaryKey.getSport();
        nickName = primaryKey.getNickName();
        ownerName = resultSet.getString(1);
        franchisePlayer = resultSet.getString(2);
        resultSet.close();
        statement.close();
    } catch (SQLException sqle) {
        throw new EJBException(sqle);
    } finally {
        try {
            if (con != null) {
                con.close();
            }
        } catch (SQLException sqle) {}
    }
}
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



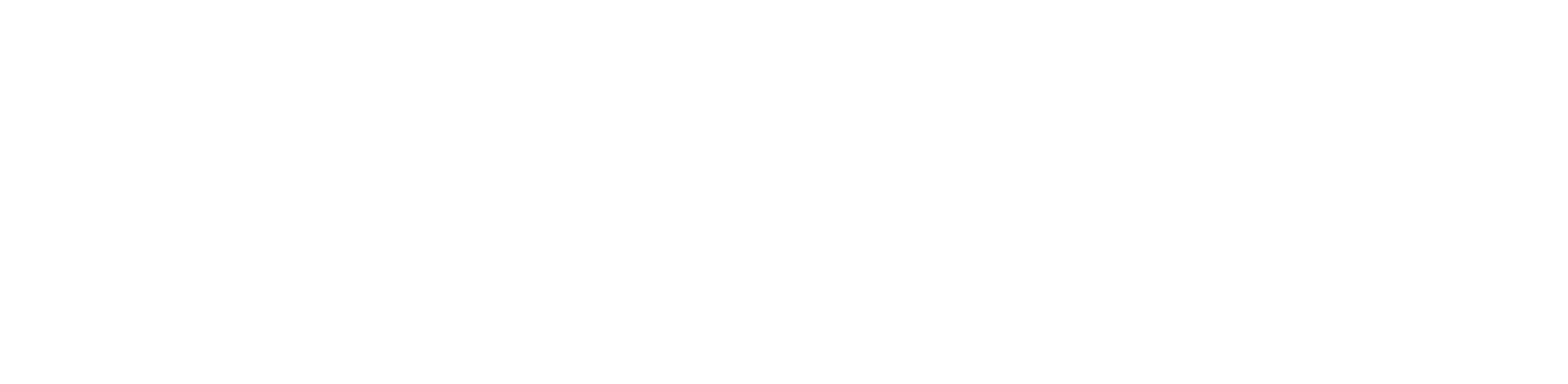
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



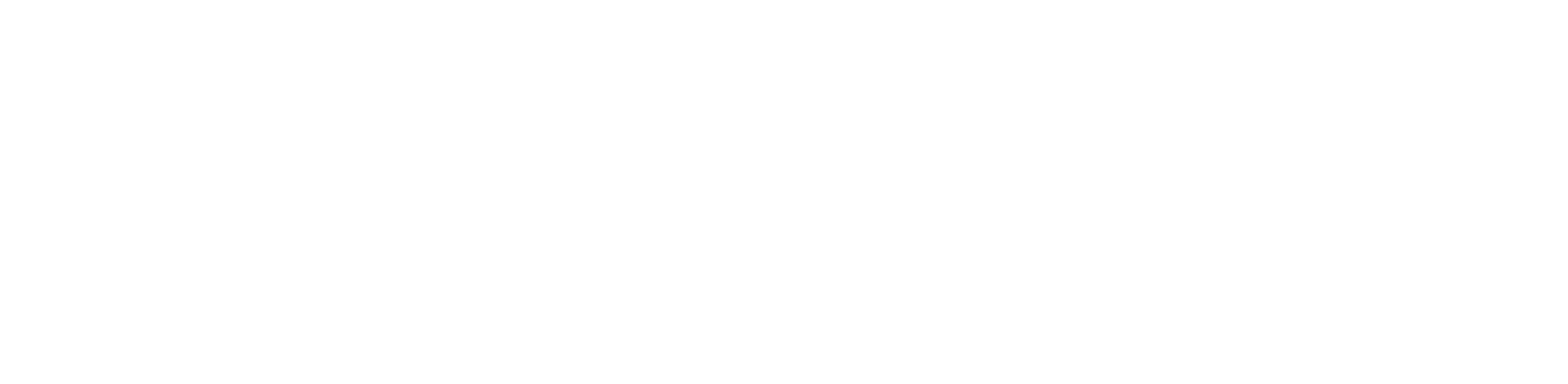
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



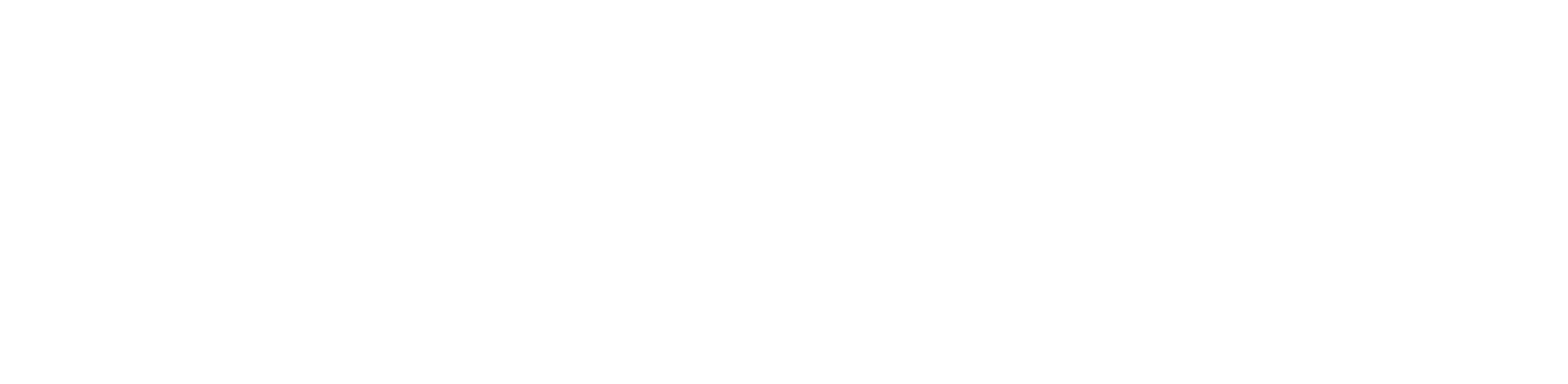
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
    }
    OrderPK otherPK = (OrderPK) other;

    return (salesDivision == otherPK.salesDivision)
        && (orderNumber == otherPK.orderNumber);
}
}
```

The public state fields (`salesDivision` and `orderNumber`) must be named exactly the same and have the same type as the corresponding fields in the bean.

One thing to note is that the order status is stored as a single letter in the database and in the cached state of the order bean. However, the client expects to get a human-readable string when it asks about the status. We have defined these strings in an outside class for simplicity, although the best approach is probably to load them from a resource (to support internationalization, for example):

```
package factory.order;

public class StatusStrings {
    public static final String OPEN = "open";
    public static final String CANCELED = "canceled";
    public static final String IN_PROCESS = "in process";
    public static final String COMPLETED = "completed";
}
```

The letter stored in the database is called `internalStatus` (and it has its own CMP field) but we expect our clients to use `getStatus()` to get the human-readable string. We translate the status when the client asks for it in the `getStatus()` method. We could have cached the status in human-readable form in a transient variable (and worked with it in the business logic in that form) by adding logic in `ejbLoad()` to translate it from its database format. Then in `ejbStore()` we could retranslate it into database form and save it in a container-managed non-transient field. This seemed like a lot of work for little value. In some other, more complicated, cases, this might be the way to go.

The OrderNotCancelableException

The `OrderNotCancelableException` thrown from the `cancelOrder()` method is an unremarkable class already discussed briefly in the last chapter. It is defined as follows:

```
package factory.order;

public class OrderNotCancelableException extends Exception {
    public OrderNotCancelableException() {}
}
```

The Product Bean

The Product bean has three state properties. Two are strings: `product` (which is the key) and `name` (which is the human-readable name of the product), and the third is a list of routing instructions. Routing instructions are the steps through which the product must pass in order to be created. A routing instruction is defined as follows:

```
package factory.product;

import java.io.Serializable;
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Exceptions

Although designing an application error-handling strategy is the responsibility of the business logic programmer, the EJB container provides a certain amount of support for the task. Specifically, there are two primary goals of the EJB specification regarding exception handling:

- An **application exception** thrown by the EJB or related code should be handled by business logic
- An **unexpected exception or system exception** thrown by the EJB or related code should be handled by the EJB container

Typical handling for an exception is a call to `commit()` or `rollback()` the relevant transaction. In addition, resources specific to that bean instance need to be kept consistent. The business logic is responsible for resource consistency with application exceptions, and the container is responsible for unexpected exceptions.

Application Exceptions

An application exception is any exception that is declared in the `throws` clause of a method in the home or remote interface of an EJB, except for `java.rmi.RemoteException`. An application exception must not be derived from `java.lang.RuntimeException` or `java.rmi.RemoteException` because these are defined as system exceptions and will be treated differently by the EJB container. Furthermore, since these exceptions are defined and thrown by the application, they must be sent back to the client, as opposed to `EJBExceptions`, which have a more severe impact at the container level (discarding instances of beans, and so on). An application exception can be derived from `java.lang.Exception`, or from other exception classes in an exception-handling hierarchy. Exactly how exceptions are used in an application is left to the architects of that application. There are, however, several application exceptions predefined in the EJB specification related to various container callbacks: `CreateException`, `DuplicateKeyException`, `FinderException`, `ObjectNotFoundException`, and `RemoveException`. These exceptions were mentioned in the previous two chapters, and are summarized at the end of this section.

To ensure that an application exception can be handled by EJB or client code, the specification requires that an application exception thrown by an EJB instance should be reported to the client precisely. In other words, the client gets exactly the same exception that the EJB threw. This may seem obvious: what other exception might it get? However, we will see that this rule does not hold for unexpected (non-application) exceptions. Remember that there is an interposition layer between the client and the EJB, and the EJB's code is never called directly. So it is possible for the EJB container to arbitrarily transform exceptions from one type to another. The specification makes this option illegal.

Note that this means that any application exception declared in the EJB implementation class must also be declared in the bean's remote interface. The reverse, however, is not true. Just because a bean's remote interface declares that it throws an exception does not mean that the implementation class must do likewise.

An application exception thrown from a method participating in a transaction does not necessarily roll that transaction back. Again, this is to give the caller's business logic a chance to take steps to recover. If the developer of the code that throws the exception knows that recovery is impossible, they can mark the transaction so that it is rolled back.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Message-driven beans do not have home and remote interfaces, but instead receive asynchronous messages. To communicate with a message-driven bean, a resource reference would be used for a topic or queue.

The `<home>` element indicates the fully qualified class name of the home interface of the referenced EJB. The `<remote>` element indicates the fully qualified class name of the remote interface of the referenced EJB. The optional `<ejb-link>` reference can contain an EJB name, which indicates a particular EJB deployment to which this `<ejb-ref>` should refer.

The following fragment of a deployment descriptor illustrates how we can use `<ejb-ref>`. In this example, our EJB `ManageOrders` needs to access an EJB `Orders`. Therefore, we declare the following in the `ManageOrders` section of our deployment descriptor:

```
<ejb-ref>
  <ejb-ref-name>ejb/Order</ejb-ref-name>
  <ejb-ref-type>Entity</ejb-ref-type>
  <home>factory.order.OrderHome</home>
  <remote>factory.order.Order</remote>
  <ejb-link>Orders</ejb-link>
</ejb-ref>
```

Our `ManageOrders` EJB can now reference the `Orders` EJB with the following code:

```
OrderHome home =
  (OrderHome) javax.rmi.PortableRemoteObject
    .narrow(initial.lookup("java:comp/env/ejb/Order"),
    OrderHome.class);
```

<ejb-local-ref>

The `<ejb-local-ref>` element is used to reference an EJB that is accessed through its local home and local interfaces. Other than this difference, the semantics are similar to `<ejb-ref>`. The deployer creates a map between real resources (such as database connections) and the resource reference placeholders listed in the deployment descriptor. Of course, if it's obvious what a particular placeholder means, a smart application server can do this mapping without asking for any more information. For example, if there is only one database connection type available, all database resource references must map to that type.

Along with this EJB-level indirection, there is also application-level indirection. The security role that a particular EJB references with the `isCallerInRole()` method is linked to an application-level role declared in the deployment descriptor. But this application-level role is still an abstraction, not an actual real-world security system object.

In order to be independent of the underlying operating system, J2EE introduces the concept of a **principal**. A principal is a simple name that is used to determine the privileges of a user. Since this is a J2EE abstraction, there still needs to be a mapping of this principal to the name that is used in the underlying operating system to determine the level of security. This indirection will finally be resolved at deployment time. Consider the indirection for our `ATM` role that we referenced.

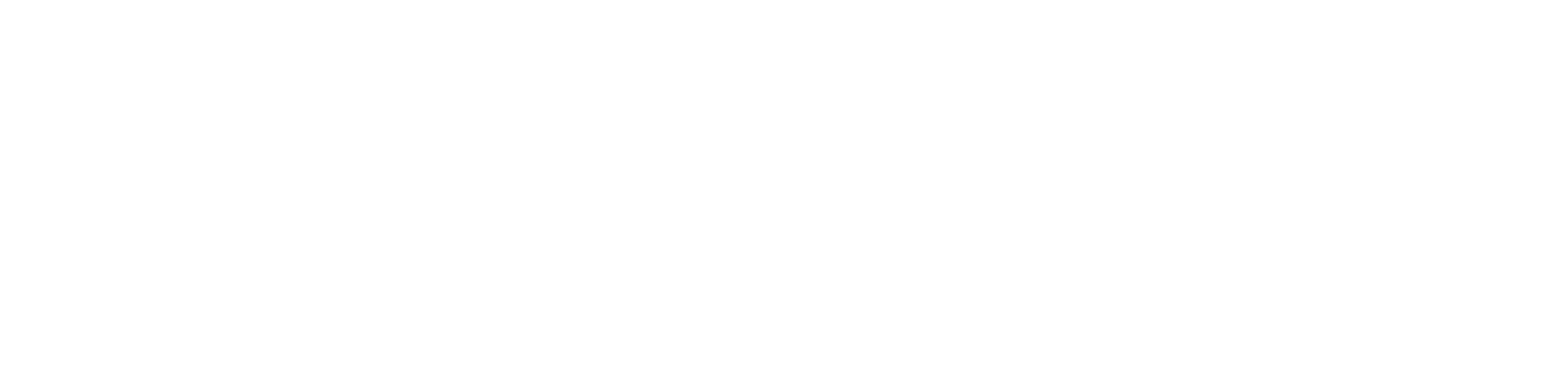
The `ATM` security role refers to the `UnmediatedAccess` security role, which at deployment time will be matched up to one or more real groups or principals in some security implementation. Neither the bean developer nor the application assembler needs to worry about the actual security implementation.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

- The values of the environment entries.
- The description field for any entry.

Here is the final version of our application's deployment descriptor:

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans
2.0//EN' 'http://java.sun.com/dtd/ejb-jar_2_0.dtd'>

<ejb-jar>
  <display-name>ISVs</display-name>
  <enterprise-beans>
    <session>
      <display-name>AccountManager</display-name>
      <ejb-name>AccountManager</ejb-name>
      <home>apress.some_isv.AccountManagerHome</home>
      <remote>apress.some_isv.AccountManager</remote>
      <ejb-class>apress.some_isv.AccountManagerEJB</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
```

The `<ejb-ref>` element now links to the Account bean:

```
<ejb-ref>
  <ejb-ref-name>ejb/GenericAccount</ejb-ref-name>
  <ejb-ref-type>Entity</ejb-ref-type>
  <home>apress.some_isv.AccountHome</home>
  <remote>apress.some_isv.Account</remote>
  <ejb-link>Account</ejb-link>
</ejb-ref>
<security-role-ref>
  <description>
    This role refers to automated customer withdrawals
    from the account; no bank intermediary is involved.
  </description>
  <role-name>ATM</role-name>
```

The role reference now links to a particular role defined in this deployment descriptor:

```
  <role-link>UnmediatedAccess</role-link>
</security-role-ref>
</session>

<entity>
  <display-name>Account</display-name>
  <ejb-name>Account</ejb-name>
  <home>apress.some_isv.AccountHome</home>
  <remote>apress.some_isv.Account</remote>
  <ejb-class>apress.some_isv.AccountEJB</ejb-class>
  <persistence-type>Container</persistence-type>
  <prim-key-class>java.lang.Integer</prim-key-class>
  <reentrant>False</reentrant>

  <cmp-version>2.x</cmp-version>
  <abstract-schema-name>AccountBean</abstract-schema-name>
  <cmp-field>
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
public static final String ORDER_TYPE_OVERDUE_TEXT = "overdue orders";
public static final String SALES_DIVISION_PARAM = "salesdivision";
public static final String ORDER_NUMBER_PARAM = "ordernumber";
public static final String MESSAGE_ATTRIB = "message";
public static final String PRODUCT_ID_PARAM = "product_id";
public static final String PRODUCT_NAME_PARAM = "product_name";
public static final String ROUTING_SEQUENCE_PARAM = "sequence";
public static final String ROUTING_ACTION_STEP_PARAM = "routing";
public static final String ORDER_SALES_DIV_PARAM = "sales_div";
public static final String ORDER_NUM_PARAM = "order_num";
public static final String ORDER_PROD_PARAM = "prod";
public static final String ORDER_DUE_DATE_PARAM = "due_date";
public static final String CELL_PARAM = "cell";
public static final String SHIP_METHOD_PARAM = "shipping_company";
public static final String SHIP_LOADING_DOCK_PARAM = "loading_dock";
}
```

The ModelManager class is the web-tier proxy for the EJB-tier model:

```
package factory;

import javax.ejb.EJBException;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.rmi.PortableRemoteObject;
import javax.servlet.http.HttpSession;
import javax.servlet.ServletContext;

import java.util.Date;
import java.util.Iterator;
import java.util.LinkedList;

import factory.manage_orders.DuplicateOrderException;
import factory.manage_orders.OpenOrderView;
import factory.manage_orders.OverdueOrderView;
import factory.manage_orders.ManageOrders;
import factory.manage_orders.ManageOrdersHome;
import factory.manage_orders.NoSuchOrderException;
import factory.manage_orders.NoSuchProductException;
import factory.manufacture.BadStatusException;
import factory.manufacture.Manufacture;
import factory.manufacture.ManufactureHome;

import factory.order.OrderNotCancelableException;

public class ModelManager {
    private ServletContext context;
    private HttpSession session;
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Once an order has been chosen for manufacture, the routing steps are displayed one at a time:

Copyrighted image

The view is provided by `manufacturerooute.jsp`:

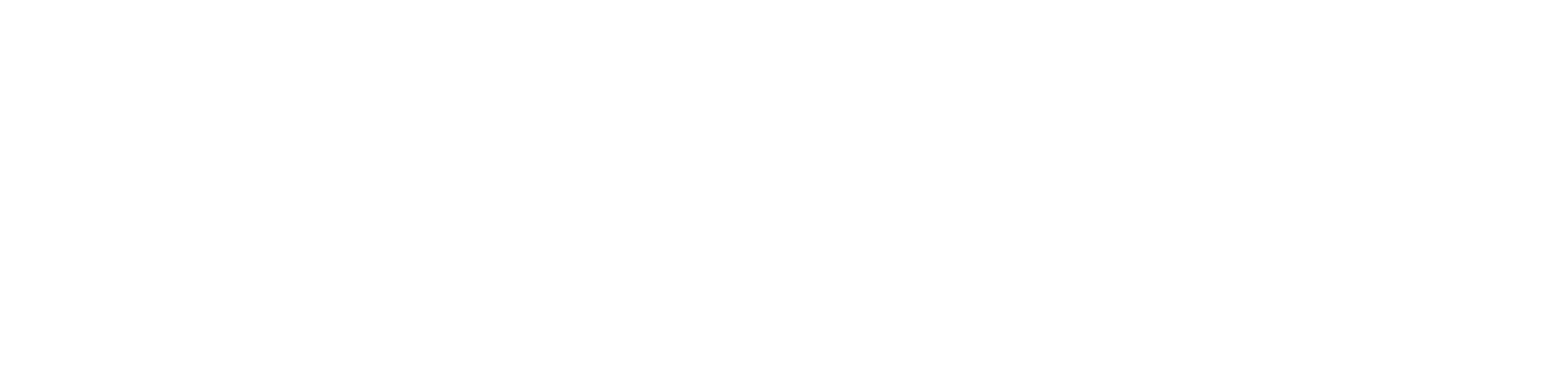
```
<jsp:useBean  
    id="modelManager"  
    class="factory.ModelManager"  
    scope="session"  
/>  
  
<html>  
  <head>  
    <title>Apress Sample Code - Routing Step</title>  
    <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">  
  </head>  
  
  <body bgcolor="#FFFFFF">  
    <p>Here is the next step in the manufacture of this product:</p>  
    <p><jsp:getProperty name="modelManager" property="nextRouting"/></p>  
    <p><a href="manufacturerooute">Click here when completed.</a></p>  
    <p>&nbsp; </p>  
    <p>&nbsp; </p>  
  </body>  
</html>
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```

Topic topic = null;
TopicConnectionFactory topicConnectionFactory = null;
TopicConnection topicConnection = null;

try {
    Context jndiContext = new InitialContext();
    topicConnectionFactory = (TopicConnectionFactory)
        jndiContext.lookup("TopicConnectionFactory");
    topic = (Topic) jndiContext.lookup("ApressPublications");
} catch (NamingException nEx) {
    System.out.println(nEx.toString() + "\nDoes the topic exist?");
    System.exit(1);
}

try {
    topicConnection = topicConnectionFactory.createTopicConnection();
    TopicSession topicSession =
        topicConnection.createTopicSession(false,
            Session.AUTO_ACKNOWLEDGE);
}

```

Everything until now has been the same, barring of course the topic/queue replacements. Next, we create four **TopicSubscribers**; the first is a **durable subscriber** (more about this later) that will listen for messages where the **MetaData** property contains the word **J2EE**. The second **TopicSubscriber** is set up to listen for books published in "Alsacien" (a German dialect spoken in eastern France). The third listens for any book with "Java" in it, and the last just listens for everything (the default):

```

TopicSubscriber topicSubscriberDurableJ2EE =
    topicSession.createDurableSubscriber(topic,
        "Apress",
        "MetaData LIKE '%J2EE%'",
        false);

TopicSubscriber topicSubscriberAlsacien =
    topicSession.createSubscriber(topic,
        "Languages LIKE '%Alsacien%'",
        false);

TopicSubscriber topicSubscriberJava =
    topicSession.createSubscriber(topic,
        "MetaData LIKE '%Java%'",
        false);

TopicSubscriber topicSubscriberAll =
    topicSession.createSubscriber(topic);

```

In case you are wondering what the last parameter is, the "**noLocal**" parameter allows messages from the local (its own) connection to be accepted or blocked, in this case **false** means that local messages are permitted.

Now we need to set up four separate listeners. The constructor of the listener class takes a name parameter to enable identification of a particular instance:

```

topicSubscriberDurableJ2EE.setMessageListener(
    new ApressTopicListener("Durable,J2EE"));
topicSubscriberAlsacien.setMessageListener(
    new ApressTopicListener("alsacien"));
topicSubscriberJava.setMessageListener(new ApressTopicListener("Java"));
topicSubscriberAll.setMessageListener(new ApressTopicListener("All"));

```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Make sure at this point that you are working with an EJB 2.0-compliant app server, otherwise you will not see the "message-driven bean" options. Some third-party JMS providers offer MDB features that complement application services; as time goes on though there are more and more 'real' MDB-compliant application servers.

The following instructions relate specifically to the J2EE Reference Implementation; other application servers will differ in their deployment techniques, and the relevant documentation should be referred to.

Once the deployment tool has loaded, select **File | New | Application**, and enter **ApresApp**, then select **File | New | Enterprise Bean**. This will take you through a series of dialogs:

Copyrighted image

Select your new application and enter **ApressMDBean** as the bean name. Click on **Edit** and add your **MessageDrivenApressBean** class (the class file not the Java source), then click on **Next**.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
<connectionfactory-interface>
    javax.sql.DataSource
</connectionfactory-interface>
<connectionfactory-impl-class>
    com.sun.connector.blackbox.JdbcDataSource
</connectionfactory-impl-class>
```

- Connection implementation class – the resource-adapter provider specifies the fully-qualified name of the Java interface and implementation class for the connection interface:

```
<connection-impl-class>
    com.sun.connector.blackbox.JdbcConnection
</connection-impl-class>
```

- Transactional support – the resource-adapter provider specifies the level of transaction support provided by the resource-adapter implementation. The level of transaction support is usually one of the following: `NoTransaction`, `LocalTransaction`, or `XATransaction`. For example:

```
<transaction-support>NoTransaction</transaction-support>
```

- Configurable properties of the `ManagedConnectionFactory` instance – the resource-adapter provider defines name, type, description, and an optional default values for the properties that have to be configured for each `ManagedConnectionFactory` instance (note that we are using the default Cloudscape database – `CloudscapeDB` – in this example):

```
<config-property>
    <config-property-name>ConnectionURL</config-property-name>
    <config-property-type>java.lang.String</config-property-type>
    <config-property-value>
        jdbc:cloudscape:rmi:CloudscapeDB:create=true
    </config-property-value>
</config-property>
```

- Authentication mechanism – the resource-adapter provider specifies all the authentication mechanisms supported by the resource-adapter. This includes the support provided by the resource-adapter implementation but not by the underlying EIS instance. The standard values are `BasicPassword` and `Kerby5`, for example:

```
<authentication-mechanism>
    <authentication-mechanism-type>
        BasicPassword
    </authentication-mechanism-type>
    <credential-interface>
        javax.resource.security.PasswordCredential
    </credential-interface>
</authentication-mechanism>
```

- Re-authentication support – the resource-adapter provider specifies whether its resource-adapter supports re-authentication of an existing connection:

```
<reauthentication-support>false</reauthentication-support>
```

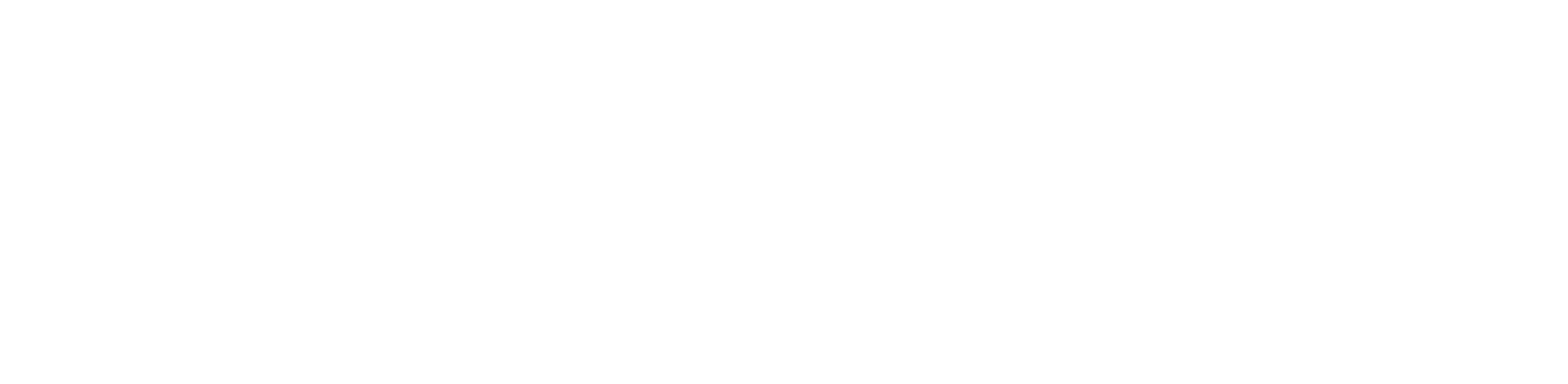
The following example illustrates a complete deployment descriptor for a black-box resource-adapter – with no transaction support (included in `.rar` file):



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
    return result;
}

public void ejbRemove() {
    try {
        deleteRow(accountid);
    } catch (Exception ex) {
        throw new EJBException("ejbRemove: " + ex.getMessage());
    }
}

public void setEntityContext(EntityContext context) {
    this.context = context;
    try {
        makeConnection();
    } catch (Exception ex) {
        throw new EJBException("database failure " + ex.getMessage());
    }
}

public void unsetEntityContext() {

    try {
        con.close();
    } catch (SQLException ex) {
        throw new EJBException("unsetEntityContext: " + ex.getMessage());
    }
}

public void ejbActivate() {
    accountid = (String)context.getPrimaryKey();
}

public void ejbPassivate() {
    accountid = null;
}

public void ejbLoad() {
    try {
        loadRow();
    } catch (Exception ex) {
        throw new EJBException("ejbLoad: " +
                               ex.getMessage());
    }
}

public void ejbStore() {

    try {
        storeRow();
    } catch (Exception ex) {
        throw new EJBException("ejbStore: " +
                               ex.getMessage());
    }
}

public void ejbPostCreate(String accountid, String firstName,
                         String lastName, double balance) { }

private void makeConnection() throws NamingException, SQLException {
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The Home Interface

`BookStoreHome` simply defines a `create()` method to return a reference to the `Book` remote interface:

```
import javax.ejb.*;
import java.rmi.RemoteException;

public interface BookStoreHome extends EJBHome {
    BooksStore create() throws RemoteException, CreateException;
}
```

The Remote Interface

`BookStore` contains the definition for two business methods as follows:

```
import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface BookStore extends EJBObject {
    public void insertBooks(String name, int quantity) throws RemoteException;
    public int getBooksCount() throws RemoteException;
}
```

The Bean Implementation Class

`BookStoreBean` is the bean implementation class, which uses CCI. `BookStoreBean` imports the CCI interfaces (that is, `javax.resource.cci.*`), and also the interface classes (`com.sun.connector.cciblackbox.*`) specific to the black-box adapter, along with `javax.resource.ResourceException`:

```
import java.math.*;
import java.util.*;
import javax.ejb.*;
import javax.resource.cci.*;
import javax.resource.ResourceException;
import javax.naming.*;
import com.sun.connector.cciblackbox.*;

public class BookStoreBean implements SessionBean {
```

In the `setSessionContext()` method, the bean uses environmental variables for the username and password to instantiate a `ConnectionFactory` for the CCI black-box adapter:

```
private SessionContext sc;
private String user;
private String password;
private ConnectionFactory cf;

public void ejbRemove() {}
public void ejbActivate() {}
public void ejbPassivate() {}
public void ejbCreate() throws CreateException {}

public void setSessionContext(SessionContext sc) {
    try {
        this.sc = sc;
        // Establish a JNDI initial context
        Context ic = new InitialContext();
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
con.prepareStatement("INSERT INTO BOOKS VALUES (?,?)");
ptstmt.setString(1, name);
ptstmt.setInt(2, qty);
ptstmt.executeUpdate();
ptstmt.close();
} catch(Exception ex){
    ex.printStackTrace();
}
}
```

And use the following ALIAS:

```
CREATE METHOD ALIAS INSERTBOOKS FOR BookProcs.insertBooks;
```

Deploying the Adapter

We'll use a similar method to before, except this time of course we'll be deploying the `cciblackbox-tx.rar` file.

With the J2EE 1.3 Reference Implementation server running, open the deployment tool and create a new application called `CCIAAdapter`:

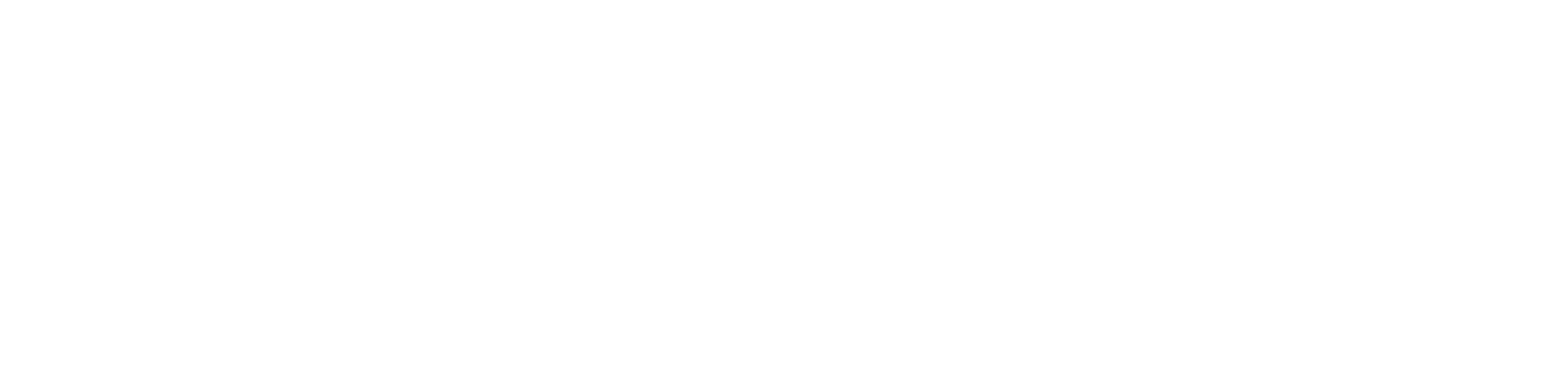
Copyrighted image

Then add an existing resource-adapter `.rar` file as before but this time select the `%J2EE_HOME%\lib\connector\cciblackbox-tx.rar` file before deploying the enterprise application.

Expand the server's node until you can see your running server (most likely `localhost`) and select the server in the left-hand pane. On the right-hand pane, switch to the **Resource-adapters** tab, and hit the **New** button. On the **New Connection Factory** dialog, select the `CCIAAdapter:cciblackbox-tx.rar` as the **Resource-adapter** and give it a JNDI Name of `eis/ApressCCIEIS`, as shown opposite:



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The World is Still Changing

When software vendors – who are usually at war – agree on something, is it a good omen or a bad one? Consider the enterprise application platforms promoted by Sun and Microsoft. Viewed from a high level, the Java 2 Platform Enterprise Edition (J2EE) and the .NET Framework (previously incarnated as Windows Distributed InterNet Architecture, or DNA) look frighteningly similar. Both combine multiple tiers, thin or thick clients, distributed object protocols, standard data access APIs, messaging services, middle-tier component environments, and transactions.

The arrival of e-commerce and Internet timescales has changed the way that most business applications are defined and developed. The requirements are more demanding than before and the timescales shorter. There is an increasing need for solutions to become adaptable since tomorrow's business requirements will almost certainly not be the same as today's. To design and develop applications under these conditions, we need serious (some would say 'professional') help.

This help usually takes two forms:

- A **standardized framework** on which applications can be built and deployed. The framework should provide appropriate levels of functionality and should also help to automate the creation of standard 'plumbing' to plug the application into itself.
- A set of **best practices** for using that framework. Software developers do not have an infinite amount of time to spend on contemplating the philosophy of design or learning the most efficient ways of using certain APIs. What they need are guidelines to help them write good applications using the framework.

In Java terms, the framework for development of distributed business applications is J2EE. The features and functionality of the platform allow the creation of scalable, distributed, flexible, and component-based applications.

The buzzwords listed above are just a few of those regularly applied to J2EE in the average marketing 'blurb'. They are very easy to promise but far more difficult to deliver. When creating applications or application components, requirements such as the level of scalability must be specified and designed. The underlying application architecture is a key element in determining whether or not such requirements are achievable. J2EE provides the foundations on which the architecture of modern, Java-based enterprise and e-commerce applications can be built.

Now, with the 1.3 version, the J2EE platform delivers further refinements in the use of Enterprise JavaBeans (EJB), native XML support, improved security, and a generic resource connector framework. Although this brings with it a few more tools and strategies, it does not change the fundamental way that J2EE is applied. Design strategies remain pretty much the same. Indeed, it is the nature of design that the best application of tools and technologies is discovered only after they have been used for a time. The discussion of patterns throughout this chapter is a good reflection of this.

The move from J2EE version 1.2 to version 1.3 has indeed added more functionality, but is this a benefit or a bane? Recently, even the vendors have come to understand that developers need help in exploiting the ocean of functionality that has already been delivered. Sun Microsystems has built the Sun Blueprints Design Guidelines for J2EE (hereafter referred to as the 'J2EE Blueprints') and the associated Java Pet Store to provide illustration of the best practices for the J2EE platform. The J2EE Blueprints can be found at <http://java.sun.com/j2ee/blueprints/>.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The model of the system is reasonably simplistic and there are no doubt many aspects of the model that we might elaborate on, or where we may organize things somewhat differently. Remember, however, that all solutions reflect the context of their design, and one of the main forces in this case is that the model should be simple enough to be readily understood. The intention at this early stage is to capture the essence of the system and not necessarily the detail. This is sufficient to discuss the move from model to system architecture. We do not need to build a complete working model and hence become bogged down in the domain detail of purchasing systems. You can see a more involved (or, in fact, evolved) solution for an n-tier e-commerce system in the Java Pet Store provided as part of the J2EE Blueprints.

Exploring the Model

A quick look at the model of the system reveals that the building of a purchase order revolves around the **Product** class. Products can be listed on some basis (product code, category) and presented to the user. The products selected by the user will be stored in a **ShoppingCart** represented by a series of **LineItems**. The interaction with the user will be governed by the **OrderingWorkflow**, which encapsulates the user interface logic for displaying products, selecting products, and submitting a purchase order. As part of the ordering process, the purchase order must be sent to the appropriate manager. This will require some form of identification of the user and also of the manager. Credentials can be gathered from the user for this purpose. When the purchase order is submitted, it will be checked against departmental limits of various sorts, including how much money is left in the departmental budget.

The purchase order produced will be stored somewhere, pending retrieval by the manager. The manager will progress through the **ApprovalWorkflow** user-interface logic to view and approve or reject purchase orders that have been submitted.

This, in essence, is the application to be implemented. However, there are many more decisions that must be taken to evolve this model into an implementation using J2EE technologies.

Elaborating the Context

Once the problem-domain model has been built, a solution-domain model must be evolved from it. This model will be subject to the "real-world" forces of application development such as the capabilities of the tools and environments applied, the topology of the infrastructure on which the application is to be deployed, and the skills available for development.

Fitting the Terrain

Consider a 'real' architect designing a house. The architect is given a set of requirements stating that the house must contain four bedrooms, a bathroom, living room, garage, and so on. If the house is to be built on the side of a hill, the design will differ from a house designed for a flat piece of land. The design of the former may make use of the slope of the ground to site the garage underneath the rest of the house, or it may build parts of the house into the hillside to improve its thermodynamic efficiency.

Similarly in software, the solution that evolves from a basic model will differ depending on the "terrain" onto which it will be deployed. A pattern that is prevalent in this environment may suggest a convenient solution to a particular problem (such as the ground-floor garage described above). In this way, the environment will shape the model in the same way that the initial requirements do.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

J2EE Patterns

Of most interest in this book are patterns that apply specifically to J2EE. Probably the most visible incarnation of this interest is the work done by some of the Java architects from the Sun Java Center. They have created a set of J2EE-specific patterns based on several years' worth of practical implementation of J2EE-based systems by Sun's Professional Services arm. The set of patterns are hosted online at the Java Developer Connection and are being evolved and refined through discussion on the J2EE Patterns e-mail list (j2eepatterns-interest@java.sun.com).

The Sun Java Center's J2EE patterns are targeted firmly at n-tier business systems and are categorized based on the tier in which they reside, namely Presentation, Business, and Integration. The table below introduces some of the more common example patterns from the catalog.

| Tier | Pattern | Description |
|--------------|--------------------|--|
| Presentation | Front Controller | Introduce a central controller (servlet or JSP) that controls the management of system services and navigation as used by a typical web-based workflow. |
| Presentation | Composite View | Provide flexibility in the display of information by building the overall web page view from a set of sub-components. |
| Business | Session Façade | Reduce coupling and network traffic by using an EJB session bean to implement common, related use cases on the serverside. |
| Business | Service Locator | Aid decoupling of business components from the underlying implementation by hiding the EJB-specific lookup and creation required when using them. |
| Business | Value Object | Reduce the network traffic associated with the retrieval of related values from an EJB by passing a serialized Java object containing a snapshot of those values. |
| Business | Business Delegate | Reduce coupling between presentation and business tiers by introducing a presentation-tier proxy to hide the details of the interaction with the business tier. |
| Integration | Data Access Object | Provide pluggable access to different data sources, by creating an interface for data access that is independent of the underlying data source. This interface can then be implemented by various objects that provide access to specific data sources without requiring changes to the code that uses the data. |

This list provided is only part of the list of the J2EE patterns identified by the Sun Java Center. J2EE patterns are also identified as part of the J2EE Blueprints (http://java.sun.com/j2ee/blueprints/design_patterns/catalog.html). Many of these are the same as those identified by the Sun Java Center, but there are others that are not, such as the Fast-Lane Reader (described later in this chapter) aimed at accelerating the read-only access to data by bypassing EJB-based access. At the time of writing, efforts are ongoing at merging the Blueprints patterns with the Java Center patterns. There are also other online sources that contain J2EE-based patterns, such as The Server Side (<http://theserverside.com/patterns/>) and O'Reilly's onJava web site (<http://www.onjava.com/design/>).



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Copyrighted image

Note that the Service Locator is typically implemented as a singleton in order to share the benefits of caching.

Value Object Pattern

Simple data objects will commonly expose their data as properties. A client will access those properties using getter or setter methods as and when required. The relatively low overhead associated with in-process method calls has made this common practice, particularly in "draggy-droppy-pointy-clicky" environments such as Integrated Development Environments (IDEs) for creating user interfaces based around JavaBeans. However, once a network is introduced between the client and the data object, such property-based programming leads to an ugly 'sawtooth' effect as data is constantly passed back and forth across the network. When this happens, communication dominates computation, most of the time required to access data is spent waiting for network calls to return, and network performance degrades due to the large amount of data passed.

This issue can be seen when accessing data or data-centric services exposed by EJBs. The effect is amplified if the required data is spread across multiple EJBs, since each must be individually accessed across the network.

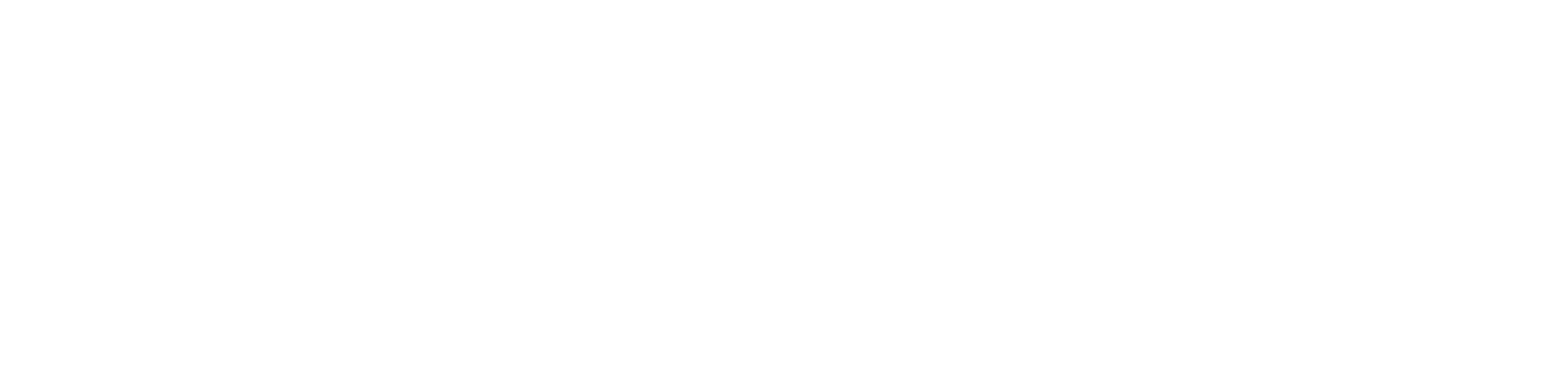
Another consideration here is that most data access is for reading rather than for writing.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Start at the Beginning

We must start the examination of the system somewhere, so let us start with the first thing the user needs: the listed products. Since the purchase order system is to be used as part of an intranet, we will need to provide an HTML interface for it. To generate the required HTML, the user interface, such as the product listing functionality, could be implemented as a set of JavaServer Pages (JSP pages) or servlets. Of the two technologies, JSP pages are best suited to the generation of HTML, so the user interface for the Ordering Workflow and Approval Workflow in the purchase order system (refer to our original UML diagram, earlier in the chapter) will consist largely of a set of JSP pages. However, servlets will also have an important part to play in the provision of common services and control of workflow for these parts of the system.

Note that the following is not a discussion of the way that servlets and JSP pages actually work, since this was given sufficient coverage in Chapters 5 to 12. Instead, some aspects of these technologies will be revisited from a design viewpoint.

Displaying Product Data to the User

We can explore some of the main JSP and servlet design issues by considering how certain parts of the Ordering Workflow could be implemented. When listing products for the user to select, use of one or more JSP pages would work well. The product data could be obtained directly from the database via JDBC. This simple architecture is shown below:

Copyrighted Image



Even at this simple stage, design choices can be made to create a more flexible and maintainable system. One aspect is the use of database connection pools to aid scalability by more effectively sharing the database connections between the JSP pages or servlets that use them. The issue of scalability and resource recycling is discussed later in the chapter.

The access to the product data will most likely be required from within JSP pages. JSP pages make it relatively easy to combine visual user interface design with the Java code required to extract the product data from the database.

Stepping back for a moment, we can define some guiding principles that can benefit even this simple design:

- Abstract the data access
- Separate out functionality from presentation
- Partition the control of user interaction from presentation and data handling

Let us examine each of these points in turn.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



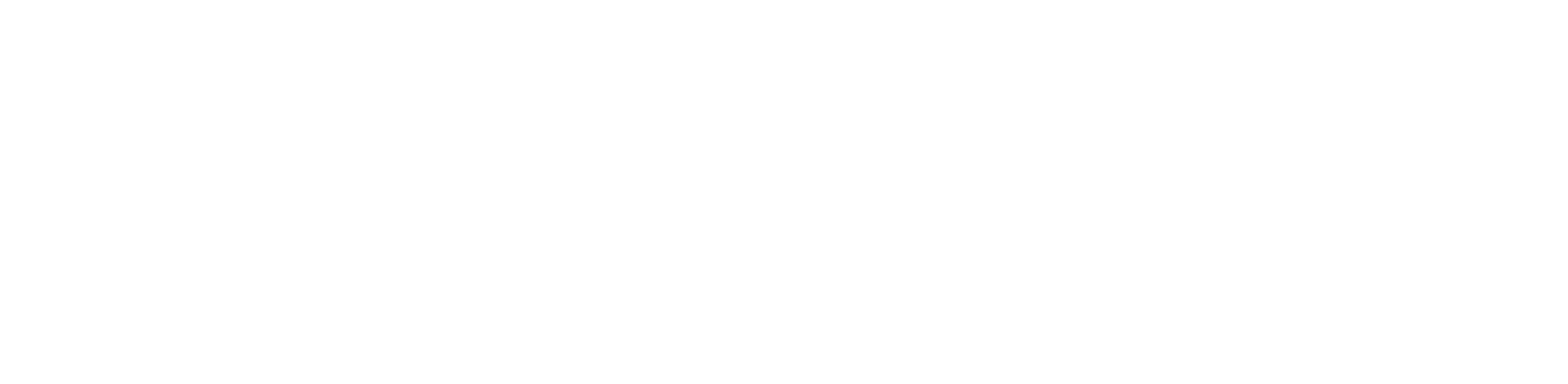
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```

    xmlns:xsd="http://www.w3.org/1999/XMLSchema">

    <import
        location="http://localhost:8080/wsdl/StockQuote_Service-interface.wsdl"
        namespace="http://www.stockquoteservice.com/StockQuote-interface">
    </import>

    <service
        name="StockQuote_Service">
        <documentation>
            IBM WSTK 2.0 generated service definition file
        </documentation>
        <port
            binding="StockQuote_ServiceBinding"
            name="StockQuote_ServicePort">
            <soap:address location="http://localhost:8080/soap/servlet/rpcrouter"/>
        </port>
    </service>

</definitions>

```

As can be seen from the above listing, the WSDL file declares a service called `StockQuote_Service`. It also declares the location of the `rpcrouter` servlet that is used for routing the service requests to the appropriate Java class. You will need to modify this WSDL file to set the location for the interface file to the current folder. Modify the `import` tag's `location` attribute to the following:

```
location="StockQuote_Service-interface.wsdl"
```

We will now look at the generated interface file.

The StockQuote_Service-interface.wsdl File

The `wsdlgen` tool generates the service interface file that is listed below:

```

<?xml version="1.0" encoding="UTF-8"?>
<definitions name="StockQuote_Service"
    targetNamespace="http://www.stockquoteservice.com/StockQuote-interface"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://www.stockquoteservice.com/StockQuote"
    xmlns:xsd="http://www.w3.org/1999/XMLSchema">

    <message
        name="IngetStockPriceRequest">
        <part name="meth1_inType1"
            type="xsd:string"/>
    </message>

    <message
        name="OutgetStockPriceResponse">
        <part name="meth1_outType"
            type="xsd:string"/>
    </message>

```



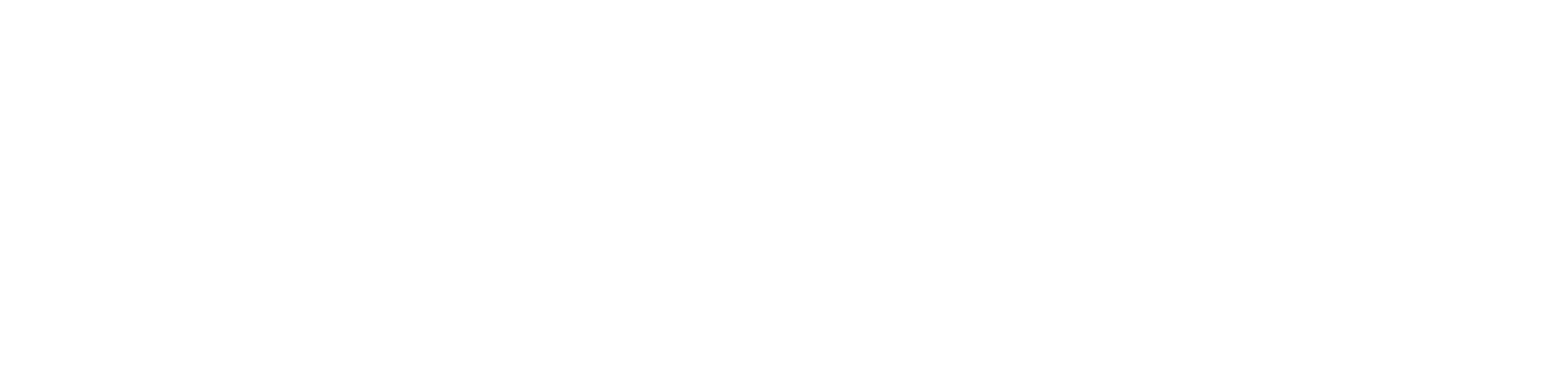
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

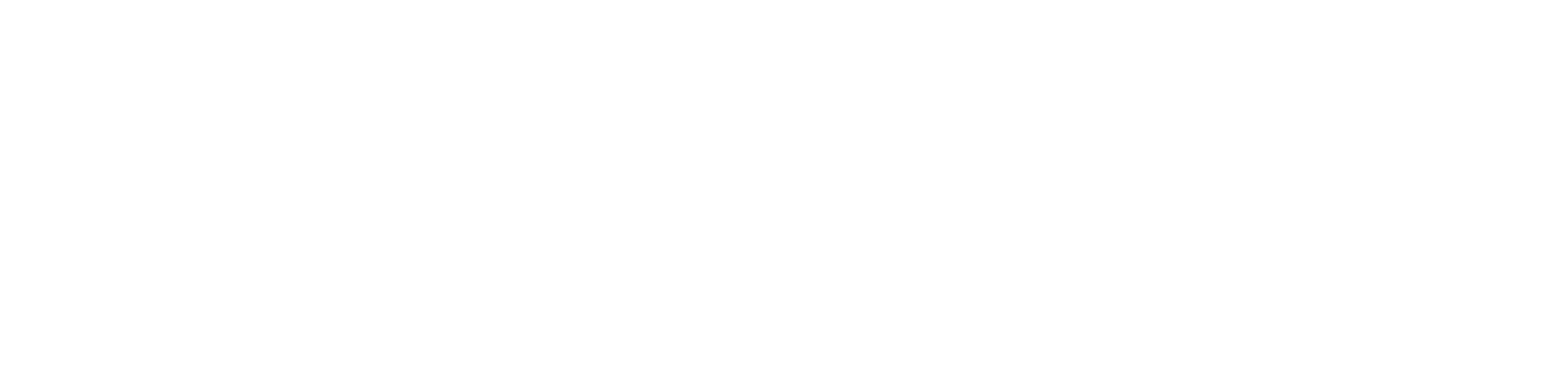
Copyrighted image



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



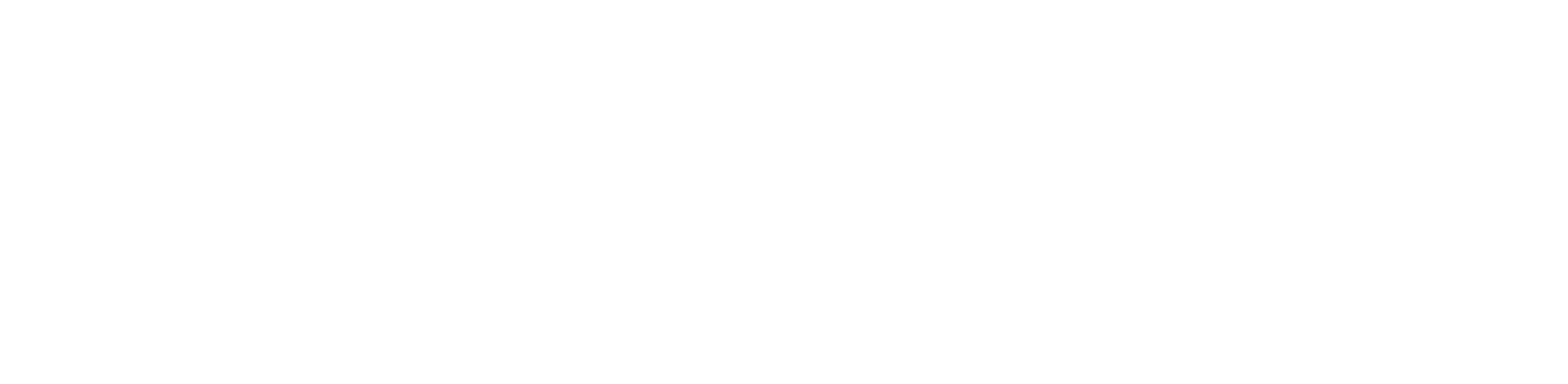
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



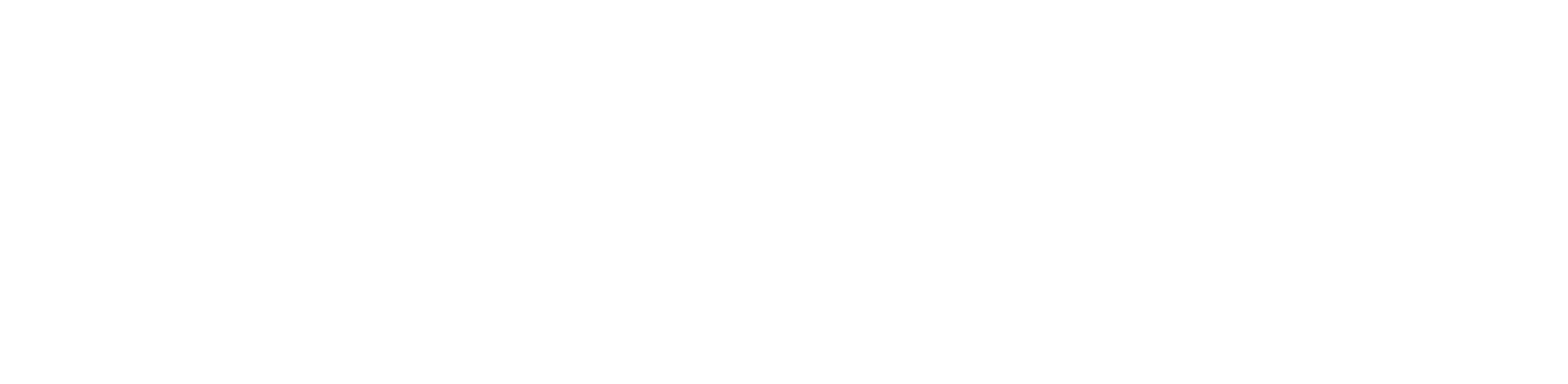
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Index

A Guide to the Index

The index is arranged alphabetically, word by word, with symbols preceding the letter A. Angle-brackets and hyphens have been ignored in the sorting process (thus `loadClass()` method will be found before `<load-on-startup>` element). Acronyms used in the text - rather than their expansions - have been preferred as main entries, on the grounds that unfamiliar acronyms are easier to construct than to expand. Unmodified entries denote the main coverage of a topic, with sub-headings indicating a more partial treatment.

Inevitably, indexes to Java books are as swollen about the letter 'J' as Scottish telephone directories are about 'M'. Readers should be aware that plural forms of main entries may be found at some distance from compound phrases and camel-and Pascal-cased derivatives beginning with the singular form (e.g `servlet context`, `ServletContext interface` and `servlets`).

Symbols

\$ (dollar) character
distinguishing expressions from string literals, 571
<% ... %> tags
JSP code delimiter, 444, 452
<%— ... —%> tags
JSP comment delimiter, 455
<%! ... %> tags
JSP declaration delimiter, 450
<%@ ... %> tags
JSP directive delimiter, 445
<%= ... %> tags
JSP expression delimiter, 454
.NET framework see **Microsoft Corporation**.
<>> brackets
representing a UML stereotype, 725
3-tiered architectures see **three-tier architecture**.

A

absolute() method
`java.sql.ResultSet interface`, 194
abstract accessors, CMP [2.0](#), [802](#)
group loading, 803
lazy loading, [802](#)
tuned updates, [802](#)
abstracting data access
design principle, 1089
abstraction layers, RMI, [84](#)
acceptChanges() method
`sun.jdbc.rowset.CachedRowSet class`, 228
access control, [51](#)
data driven, rather than operation dependent, 886
in EJB specification, 882

programmatic alternative to method permissions, 886
services needed for a secure system, 881
user interfaces reflecting allowed functionality, 888
access control layer, 734
Financial Aid Calculator bean example, 755
access to resources
EJB in unspecified transaction context, 871
AccessException class, java.rmi package, [91](#)
account transfer example
using transactions, 868
with and without transactions, 864
without transactions, 867
loss of data integrity, 868
AccountManager class
`debitFunds() method`, 890, 891
ACID (Atomicity, Consistency, Isolation and Durability)
properties
stateful session beans, 795
transaction data integrity and, [214](#)
transactions, 862
ACL (Access Control Lists), [207](#), [419](#)
LDAP servers and binding, [55](#)
Acme Multinational plc see **purchasing system J2EE design example**.
actions, JSP see **standard actions, JSP**.
Activatable class, java.rmi.activation package, [94](#), [116](#), [117](#)
alternatives to extending, [119](#)
`exportObject() method`, [116](#), [120](#)
`inactive() method`, [123](#)
`register() method`, [116](#)
specifying socket factories, [126](#)
`unexportObject() method`, [123](#)
`unregister() method`, [123](#)

activation
see also **deactivation, RMI.**
ejbActivate() method, 828
entity beans, 828
session beans, 738
Activation descriptors, 115
activation groups, 113
RMI object activation model, 112
activation monitors
RMI object activation model, 112
activation systems
RMI object activation model, 112
ActivationDesc class, java.rmi.activation package
ActivationDesc() method, 115
ActivationGroup class, java.rmi.activation package
createGroup() method, 113
ActivationGroupDesc class, java.rmi.activation package
ActivationGroupDesc() method, 114
ActivationID, 114
activation-specific properties, RMI, 151
Activator interface, java.rmi.activation package
class property, 151
activators
RMI object activation model, 112
actors
UML use cases, 718
analysis model views of interface objects and, 727
manufacturing EJB example, 723
Add() method
javax.naming.directory.BasicAttributes class, 67
addBatch() method
java.sql.Statement interface, 198
addBodyPart() method
javax.mail.Multipart class, 651, 671
addFrom() method
javax.mail.Message class, 646
adding entries
LDAP server, using JNDI, 66
adding users
form-based authentication, 425
addRecipients() method
javax.mail.Message class, 646
Address class, javax.mail package, 655
addRowSetListener() method
javax.sql.RowSet interface, 226
administration services
application servers, future prospects, 1157
administered objects
JMS architecture component, 972
ADS (Active Directory Services), 41
afterLast() method
java.sql.ResultSet interface, 194
airplane design
illustrating design context, 1064
aliasing
servlets, 416
Allaire Corporation see **JRun Application Server**.
<alt-dd> tag
deployment descriptors, 1181
AlreadyBoundException class, java.rmi package, 21
analysis model views
control objects and entity objects, 728
interface objects and control objects, 728
manufacturing EJB example, 727
use case actors and interface objects, 727
analysis objects
three types in UML use cases, 718
Apache log4J package, 495
Apache Software Foundation *see also Tomcat servlet engine.*
Apache Struts generic controller framework, 504, 592
Jakarta Taglibs, 569
API calls
role in traditional distributed computing, 19
APIs (Application Programming Interfaces)
container-specific, 22
implementation-independent, list of, 20
J2EE environment provides, 18
list of those J2EE is required to support, 19
service APIs
single standard for access to technologies, 26
applet containers, 22
packaging and deployment, J2EE, 1168
applet lifecycle, 24
Applet.html page, 110
applet.jsp, 471
applets
clients, in custom sockets example, 132
demonstrating remote callbacks, 109
including in JSP pages using <jsp:plugin>, 470
MoleculeViewer sample, 471
Application and Web Services block
Sun ONE (Sun Open Net Environment), 1142
application architectures *see architectures*.
application assembler role, 34, 1169
adding test clients running with security identity, 923
deployment descriptors
banking example, 917
additions, 923
relevant information, 916
modifying developer info, 916
overview, 916
packaging components, 926
application client containers
packaging and deployment, J2EE, 1167
Application Client JAR files
packaging and deployment, J2EE, 1168
application clients
viewed as fat clients, 233
application component provider role, 34, 1169
Resource Adapter, 1022
application components
clients access via the container, 24
instantiated and initialized within container JVM, 24
lifecycle managed by the container, 24
lifecycle management by containers, 27
modelling business rules in J2EE application
development, 33
packaging into modules, 33
programs hosted by J2EE containers, 22
provided by developers in containers, 23
responsibilities for transaction processing, 217
application contracts
CCI (Common Client Interface) API, 1014
Resource Adapter, 1014
application deployment
two stages of, 36
application deployment and installation
understanding the environment, 862
application deployment tool
J2EE Reference Implementation, 278
application design *see design process*.
application developer role
pressures on, in modern enterprises, 11

- application developments**
 four steps specified in J2EE, 33
 individuals' roles in, 34
- application domain expert** *see bean provider role.*
- application exceptions**
 derivable from Exception class, 889
 description, 889
 error handling by business logic, 889
 predefined, in EJB specification, 892
 rollback not an inevitable consequence of, 890
- application infrastructure**
 specified by J2EE, 12
- application lifecycle events**
 JSP 1.2 new feature, 510, 557
- application logic**
 request-response process, 238
- application objects**
 JSP, 474
 multiple, in enterprise system architecture, 16
- application scope**
 JSP objects, 475
- application server vendor role**
 Resource Adapter, 1022
- application servers**
 additional functionality, 1154
 clustering, 1155
 integrating .NET with Java classes, 1155
 integrating COM with Java classes, 1155
 integrating CORBA with Java classes, 1155
 integrating J2EE with different IDEs, 1155
 load balancing, 1155
 architecture, 1149
 bundling with operating systems, 1161
 commercial JMS implementations packaged with, 967
 communication between, 862
 competition among vendors, 1151
 component behavior, 1148
 configuring distributed transactions, 221
 custom authentication, 425
 deployment descriptors, 1154
 EJB containers, 1148
 enterprise applications, 1147
 future prospects, 1156
 administration services, 1157
 caching, 1157
 content management, 1157
 integration platform, 1157
 personalization, 1157
 portal capabilities, 1157
 user entitlements, 1157
 workflow, 1157
 integration with enterprise systems, 1148
 J2EE APIs, 1153
 J2EE implementation, 1147
 message-driven bean support, 999
 network protocols, 1154
 open source products, 1152
 Enhydra, 1152
 JBoss, 1152
 JonAS, 1152
 scalability, 1148
 SPIs (Service Provider Interfaces), 1153
 web containers, 1148
 WebLogic application server, 1149
 WebSphere application server, 1149
- <application> tag**
 deployment descriptors, 1177
- application.xml file**, 36
- application-client containers**, 22
- application-managed sign-on**
 system-level contracts, 1016
- applications**
 packaging modules into, 35, 1166
- application-specific logging**, 385
- architectural templates**, 1071
- architectures**
see also n-tier; three-tier; two-tier and individual tier names.
 characteristics of n-tier, 15
 choosing and refining, 1071
 compared to design, 1061
 enterprise architecture as a development of n-tier architecture, 16
 interaction with design, 1072
 J2EE, shown diagrammatically, 21
 JSP page-view architectures, 490
 JSP page-view with bean architectures, 491
 layered and tiered styles distinguished, 1061
 RMI layered architecture, 85
 RMI, involving sockets, 124
 service-based architectures, 1063
 three distributed architectures in Java, 52
 three-tier as an intermediate stage, 14
 tiered and layered styles distinguished, 1061
 tiered, advantages of, 1063
 two-tier as basis for client/server systems, 14
- arius.com sample data**, 57
- Array interface, java.sql package**, 164
- ArrayList object, java.util package**,
 shopping cart example application, 343
- ASP (Active Server Pages)**
 compared with JSP, 240
- asynchronous communication**
 availability and scalability advantages of, 1101, 1108
 disadvantages of, 1102
 distributed designs and, 1100
 JMS, 969
 JMS and JavaMail compared, 32
 JMS example, 979
 message-driven beans and, 995
- atomic transactions**
see also ACID properties.
 succeeding or failing as a group, 863
- attachments** *see e-mail attachments.*
- Attribute interface, javax.naming.directory package**
 getAll() method, 62
 getID() method, 62
- attribute values**
 accepted by JSPTL tag libraries, 571
- <attribute> sub-tags**
 TLD <tag> elements, 525
- attributeAdded() method**
 HttpSessionAttributeListener interface, 340
 ServletContextAttributeListener interface, 349
- AttributeHelloTag class**, 529
- attributeRemoved() method**
 HttpSessionAttributeListener interface, 340
 ServletContextAttributeListener interface, 349
- attributeReplaced() method**
 HttpSessionAttributeListener interface, 340
 ServletContextAttributeListener interface, 349
- attributes**
 associating with sessions
 methods of the HttpSession interface, 326

attributes (continued)

attributes (continued)

JSP tag extensions, 508
alternatives, 532
processing, 529
uses reflection, 529
JSPTL tag library, kept few, 576
LDAP, 43
attribute matching rules, 44
non-serializable, warning about use in servlet sessions, 334
restricting display, LDAP searching, 64
AttributeServlet class, 334
authenticated searching, 62
authentication, 51
see also binding.
biometric authentication, 56
custom authentication, 425
form-based authentication, 422
JavaMail, 642
possible sources of, 882
services needed for a secure system, 881
types in the servlet API, 421
Authenticator class, javax.mail package, 642
authorization
LDAP support for, 52
possible use of filters, 385
auto-reloading facilities
web containers, 414
availability
advantages of asynchronous communications, 1101, 1108
availability services
J2EE implementation, 1153
AWT (Abstract Window Toolkit)
EJB 2.0 specification prohibitions, 712, 713

B

B2B (Business-to-Business) integration
JCA (Java Connector Architecture), 1056
banking applications
on-line banking demands stateful protocols, 318
requirements contrasted with those of e-commerce, 1071
banking example
Account entity bean
bean class, 911
AccountManager session bean
bean class, 909
adding test clients running with security identity, 924
bean provider role, 903
deployment descriptors, 914
additions, 923
final version, 917
entity beans, 903
home interfaces, 904
remote interfaces, 903
indirection, developer-product relationship, 906
mapping beans and references into JNDI namespace, 933
mapping logical database accesses to actual database, 931
mapping logical security roles to users/groups, 928
method permissions, 885
packaging components, 926
security roles, 883
session beans, 904
home interfaces, 904
remote interfaces, 904

banner.jsp, 486

BannerServlet, 377
JSP equivalent is banner.jsp, 479
Banyan Vines
JNDI SDK has no service provider for, 50
base names
resource bundles, 605
base, LDAP trees, 57
BasicAttributes class, javax.naming.directory package, 67, 72
basic authentication, 421
batch updates
performance improvement from, 198
batched methods
design pattern for networked applications, 1070
BatchUpdateException class, java.sql package, 165
BEA
see also WebLogic.
support for web services, 1144
complex web services, 1145
bean class
banking example
Account entity bean, 911
AccountManager session bean, 909
Bookstore CCI example, 1046
DemoConnector Adapter example, 1031
EAR files example, 1178
EJB central development point, 704
Financial Aid Calculator bean example, 742, 749, 756
instantiate() method, 457
Order EJB, 840
Product EJB, 846
bean provider role
Account entity bean example
bean class, 911
AccountManager session bean example
bean class, 909
banking example, 903
bean info content fields, deployment descriptor, 913
deployment descriptors
banking example, 914
overview, 912
EJB container contracts and, 694
<ejb-local-ref> element, 908
<ejb-ref> element, 907
indirection, overview and example, 906
overview, 902
producing JAR file, structure, 903
reference types, providing, 906
<resource-ref> element, 907
security role reference, 908
specifying entity bean's primary key class, 914
bean-managed persistence *see BMP*.
bean-managed transaction demarcation, 864
requires session beans, 864
session bean controlled transaction demarcation, 875
bean-managed transactions
message-driven beans, 997
beans.html page, 460
beans.jsp page, 461
beforeFirst() method
java.sql.ResultSet interface, 194
begin() method
javax.transaction.UserTransaction interface, 220
UserTransaction interface, 875
BeginManufacture client
manufacturing EJB example, 768

best practices
 e-commerce/internet applications, 1060

bind() method
 java.rmi.Naming class, 89, 97, 104
 javax.naming.directory.DirContext interface, 70

binding
 attributes to servlet sessions, 339
 LDAP, 55
 provision of unique names in JNDI, 202

binding element
 WSDL document, 1119

biometric authentication, 56

Black Box Resource Adapters
 CCI Black Box Adapter, 1045
 configuring database, 1024
 DemoConnector Adapter example, 1023
 deploying adapters, 1024, 1025
 JCA (Java Connector Architecture), 1023
 local transactions, 1024
 selecting Black Box adapter, 1024
 testing adapter, 1030
 XA transactions, 1024

Blob interface, java.sql package, 164

block structure
 JavaMail messages, 648

BMP (bean-managed persistence)
 CRUD callbacks, 819
 caching, 822
 clients, 833
 compared to CMP (Container-Managed Persistence), 799, 819, 892
 deployment descriptors, 821
 ejbCreate() method, 812
 ejbLoad() method, 815, 816
 ejbPostCreate() method, 812
 ejbRemove() method, 818
 ejbStore() method, 817
 findByPrimaryKey() method, implementation, 826
 finder methods, 825, 826
 primary keys, 807
 specifying isolation levels, 877
 SportBean laboratory example, 812
 utility methods, 831

body content
 JSP tag extensions, 533
 filtering, 544
 iteration and manipulation of, 540
 powerful use of, 562

<body-content> tag
 sub-elements in TLDs, 525

BodyContent class, javax.servlet.jsp.tagext package, 521

BodyPart class, javax.mail package, 648, 652
 message content composition and, 648

BodyTag interface, javax.servlet.jsp.tagext package, 518
 doAfterBody() method, 520
 doInitBody() method, 520, 544
 doStartTag() method, 519
 setBodyContent() method, 520

BodyTagSupport class, javax.servlet.jsp.tagext package, 521

Bookstore CCI example
 CCI Black Box Adapter, 1045
 clients, 1050
 deploying adapter, 1050, 1052
 configuring database, 1051
 stored procedures, 1051

stateful session beans, 1045
 bean class, 1046
 deployment descriptors, 1049
 home interfaces, 1046
 remote interfaces, 1046
 testing application, 1054

Boolean tag
 foundation tags, used in displayDetails.jsp page, 616

booleanInput tag
 HTML tag library, registration example, 601

bootstrapped client configuration
 class distribution in RMI, 105

bootstrapped server configuration
 class distribution in RMI, 105

boundary objects *see Interface objects*.

bracket-matching
 problem with scriptlets, 579

browsers
 role in web application example, 251
 specific HTML for, using <jsp:plugin>, 470

buffered output
 JSP pages, 464, 519
 ServletResponse interface methods for, 294

built-in web containers, 242

business context
 purchasing system J2EE design example, 1066

Business Delegate Pattern, 1082
 compared to Session Façade Pattern, 1097
 hiding an EJB from the client, 1107
 minimizing network traffic, 1108
 possible use in purchasing order system
 exception handling, 1097
 using to abstract services, 1108

business layer *see business logic tier*.

business logic
 component of n-tier architecture, 16
 EJB container as component framework for, 692
 interfaces, 762
 manufacturing EJB example, 762
 web services, 1130
 integrating other resources, 1130

business logic interface
 starting point for EJB design, 701

business logic tier
 EJBs are executed in, 688
 three-tier architecture, 14, 1062

business methods
 manufacturing EJB example, 775, 785, 840, 846
 Order EJB, 840
 Product EJB, 846

business services
 abstraction with Business Delegate Patterns, 1082

business transactions
 characteristics favoring stateful protocols, 319

BytesMessage interface, javax.jms package, 989

C

cached rowsets, 227
 situations favoring use, 227

caching
 application servers, future prospects, 1157
 BMP (bean-managed persistence), 822
 CMP (Container-Managed Persistence), 822

caching (continued)

caching (continued)

- CRUD callbacks and, 822
- entity beans, 822
 - problems with, 822
- caching data**
 - server-side includes, 620
 - using `ejbLoad()`
 - appropriateness, 874
- CallableStatement interface, java.sql package, 163, 175**
 - invoking stored procedures with, 185
- callback methods**
 - distinguish the three varieties of EJBs, 688
 - impossible with Http tunneling, 137
 - JSP tag handlers, 511
 - remote callbacks, 108
 - session bean callback used in OrderManagement example, 704
- CallbackApplet class, 109**
- cancelOrder() method**
 - Order EJB, 844
- cancelRowUpdates() method**
 - java.sql.ResultSet interface, 197
- cardinality**
 - relationships, entity beans, 837
- case sensitivity**
 - JSP pages, 444
 - META-INF directory, 701
- casting operator**
 - not available with RMI-IIOP, 149
- CatalogServlet, 341**
- CC and BCC fields**
 - e-mail message headers, 646
- CCI (Common Client Interface) API**
 - advantages of using, 1039
 - application contracts, 1014
 - CCI Black Box Adapter, 1045
 - Connection interface, 1041
 - connection interfaces, 1040
 - connection with managed-application, 1042
 - connection with non-managed application, 1043
 - ConnectionFactory interface, 1040
 - ConnectionSpec interface, 1041
 - data representation interfaces, 1040
 - description, 1039
 - exception and warning classes, 1040
 - Interaction interface, 1041
 - interaction interfaces, 1040
 - InteractionSpec interface, 1041
 - JCA (Java Connector Architecture), 1011, 1012, 1039
 - LocalTransaction interface, 1041
 - metadata interfaces, 1040
 - Record interface, 1041
 - RecordFactory interface, 1041
- CCI Black Box Adapter, 1045**
 - Bookstore CCI example, 1045
 - clients, 1050
 - stateful session beans, 1045
 - deploying adapter, 1050, 1052
 - configuring database, 1051
 - testing application, 1054
- Certificate Authority, 127**
- certificates** see digital certificates.
- challenges**
 - offered by modern enterprise application developments, 11
- chat.war and chat.ear files, 405**
- chatAdministrator role, 423**

ChatAdminServlet

- chatroom example application, 353
- path mapping, 418

ChatAdminServlet class

- `doGet()` method, 354
- `doPost()` method, 354

ChatEntry class

- chatroom example application, 353

ChatRoom class

- `addChatEntry()` method, 352
- chatroom example application, 352
- synchronized and thread-safe, 353

chatroom example application

- aliasing servlets, 416
- application requirements, 350
- ChatRoomServlet, 357, 359
- class relationships, 351
- deploying and testing, 365
- deployment descriptor, 364
- enhanced to use filters, 395
 - deploying the application, 405
 - possible variations, 407
 - running the application, 406
- enhanced to use form-based authentication, 422
- ListRoomsServlet, 357
- modeling requirements, 350
- setup stages, 364
- URL path mapping, 416
 - using servlet context and sessions, 350

ChatRoomServlet, 357, 359

ChatRoomServlet class

- `doGet()` method, 364
- `doPost()` method, 364
- filters as an alternative to modifying, 395
- `writeFrame()` method, 363
- `writeMessages()` method, 364

checkbox tag

- HTML tag library, registration example, 602

child classloaders, 1170

choose tag

- JSPTL conditional tag, 587
- `displayAllRegisteredUsers.jsp` page, 625

Class class, java.lang package

- `forName()` method, 167
- `getResourceAsStream()` method, 713

class diagrams, UML

- javax.servlet package, 260
- javax.servlet.http package, 260
- OrderManagement EJB example, 702

class loading schemes, 1170

- child classloaders, 1170
- description, 1170
- EAR (Enterprise Archive) files and, 1171
 - before EJB 2.0, 1171
 - EJB 2.0 specification, 1173
 - local interfaces, 1173
- parent classloaders, 1170
 - WEB-INF\lib directory, 1171

class property, java.rmi.activation.activator package, 151

class relationships

- chatroom example application, 351

ClassFileServer, 107

ClassLoader creation

- EJB 2.0 specification prohibitions, 712

ClassLoader object, java.lang package, 167

CLASSPATH

- needn't be reset for web applications, 412
- searched before RMI codebase property, 104

Class-Path: manifest entry

dependency libraries packaging example, 1188
packaging dependency libraries, 1185

clearBody() method

`javax.servlet.jsp.tagext.BodyContent` class, 521

CLI (Call Level Interfaces), 159

client authentication, 421

Client Connectivity Interface API, 713**client ID**

durable subscribers, 986

client sockets, 123**client/server designs, JSP, 489**

Front Controller Pattern compared to, 493

client/server systems

practical problems with, 18

two-tier system architecture as basis of, 14

clientimport.cer file, 128, 131**ClientInterface interface**

remote callbacks, 109

clients

access to EJBs, stages, 696

applets as, custom sockets example, 132

application components accessed via the container, 24

BMP (bean-managed persistence), 833

Bookstore CCI example, 1050

business object access with Business Delegate Patterns, 1082

client-tier access to EJBs, 717

CMP (Container-Managed Persistence), 831

contracts with EJBs under EJB 2.0 spec., 694

DemoConnector Adapter example, 1037

entity beans, 831

executing in an EJB container, 926

Financial Aid Calculator bean example, 741, 752, 755

Javax.sql package initiates JDBC drivers for, 205

JDBC/JNDI datasource connections, diagram, 203

manufacturing EJB example, 766

possible EJBs client types, 687

requests intercepted by Front Controller Pattern, 1075

RMI, meaning of servers and, 84

RMI, running the server and, 97

RMI, use of sockets, 125, 126

SportBean laboratory example, 831

storing state with the client

purchasing system J2EE design example, 1099

two types in J2EE architecture, 22, 233

view of the EJB specification, 698

web services

returning results to client, 1130

StockQuoteService web service example, 1139

Clob interface, java.sql package, 164**close() method**

Connection interface, 1043, 1045

CreateMovieTables class, 180

java.sql.Statement interface, 175

javax.mail.Folder class, 662

javax.sql.PooledConnection interface, 211

closed configuration

class distribution in RMI, 105

Cloudscape database, 297

ships with J2EE Reference Implementation, 299

testing the TechSupportServlet, 312

used for JDBC examples, 162

clustering

application servers, additional functionality, 1155

distributable containers, 28

distributed web containers, 435

CMP (Container-Managed Persistence)

caching, 822

clients, 831

CMP 2.0, improvements in, 800

compared to BMP (bean-managed persistence), 799, 819, 892

CRUD callbacks, 819

deployment descriptors, 821

ejbCreate() method, 811

ejbLoad() method, 815

ejbPostCreate() method, 811

ejbRemove() method, 818

ejbStore() method, 817

finder methods, 825

isolation level management, 877

primary keys, 808

SportBean laboratory example, 811

CMP 2.0

abstract accessors, 802

EJB QL, 804

improvements in persistence in, 800

local interfaces, 803

relationships, 803

CMR (Container Managed Relationship) fields

relationships, entity beans, 837

code libraries

using LDAP with, 53

code maintenance

in JSPs, 488

code reuse, 13

codebase property, `java.rmi.server` package, 103-104, 150

colon separator

JDBC URLs, 167

COM (Component Object Model)

integrating COM with Java classes

application servers, additional functionality, 1155

Combined Attributes see **Value Object Pattern**.**command-line operations**

see also rmic tool; rmid utility; rmiregistry tool.

j2eeadmin, 975

JavaMail examples, 667

POP3 access tool, 672

packaging an application as a WAR file, 413

comments, JSP, 455

commit() method

`java.sql.Connection` interface, 188, 213

`javax.jms.Session` interface, 974

`javax.transaction.UserTransaction` interface, 220, 223

`UserTransaction` interface, 875

warning about, in distributed transactions, 223

commitment

making transactions durable, 863

rollback or, as transaction endpoint, 863

communication technologies

see also asynchronous communication.

J2EE, 32

synchronous communication, JMS example, 977

compare command, LDAP

not supported by JNDI, 47

ComparisonTerm class, javax.mail.search package, 665

compilation
JSPs into servlets at translation time, 442
strategies for JSP static includes, 467
using Java IDL and idlj, 145

CompleteManufacture client
manufacturing EJB example, 770

complex data
JSP tag extensions for exposing, 563

complex web services, 1145
list of characteristics, 1145

component behavior
application servers, 1148

component contracts
J2EE containers, 23, 24

component lifecycle
managed by the container, 24

component technologies, J2EE
JSB, servlets and EJBs as, 28

components
co-locating those that communicate often, 1070

Composite Entity Pattern
Sun Java Center discussion, 1107

Composite View Pattern, 1076
purchasing system J2EE design example, 1089, 1090
user functionality separation, 1107

composition see packaging.

compound statements
recommended for JSP scriptlets, 453

concurrency
JavaMail, 661

concurrency and data integrity
management when using cached rowsets, 228

concurrency types
java.sql.ResultSet interface, 197

conditional tags
JSPTL, 587
jsptl-examples.war, 588

config object, JSP, 474

config.xml file
configuring WebLogic Server 6.1, 835

configuration
applications, using LDAP, 53
web applications, 425

configuration information
Java servlets, using ServletConfig interface, 270

confirmation page
Tech Support servlet example, 302, 303

connect() method
java.sql.Driver interface, 166, 169
javax.mail.Store class, 657

ConnectException class, java.rmi package, 91

ConnectIOException class, java.rmi package, 91

Connection interface, java.sql package, 163, 877, 1020
commit() method, 188, 213
createStatement() method, 174
full list of public methods, 170
getMetaData() method, 173, 877
getTransactionIsolation() method, 877
getWarnings() method, 178
implementation of connection pooling, 213
loading a database driver, 166
predefined isolation levels in, 878
prepareCall() method, 174, 185
prepareStatement() method, 174, 184
releaseSavePoint() method, 191
Resource Adapter, 1020
rollback() method, 188, 213

setAutoCommit() method, 188, 213
setTransactionIsolation() method, 877

Connection interface, javax.jms package, 989
setClientID() method, 986

Connection interface, javax.resource.cci package, 1041
CCI (Common Client Interface) API, 1041
close() method, 1043, 1045
createInteraction() method, 1041, 1043, 1045

connection interfaces
Common Client Interface API, 1040

connection management
system-level contracts, 1014

connection management classes, java.sql package, 163
limitations addressed by javax.sql, 199

connection pooling
application characteristics favouring use, 207
client responsibility in traditional approach, 209
compared with physical database connection, 212
database connection pooling, 1153
full list of classes and interfaces, 211
implementation, 213
javax.sql package, 207
diagram, 210
responsibility change in javax.sql package, 199
traditional approach to, 208
weaknesses of traditional approach, 209

connection sockets, 123

connectionClosed() method
javax.sql.ConnectionEventListener interface, 212

connectionErrorOccured() method
javax.sql.ConnectionEventListener interface, 212

ConnectionEvent class, javax.sql package, 212

ConnectionEventListener interface, javax.sql package
connectionClosed() method, 212
connectionErrorOccured() method, 212

ConnectionFactory interface, javax.jms package, 990
CCI (Common Client Interface) API, 1040
getConnection() method, 1041, 1043
getRecordFactory() method, 1041

ConnectionFactory interface, javax.resource.cci package, 1040

ConnectionMetaData interface, javax.jms package, 990

ConnectionPooledDataSource interface, javax.sql package
getPooledConnection() method, 211

connections
JMS architecture component, 972

ConnectionSpec interface, javax.resource.cci package, 1041
CCI (Common Client Interface) API, 1041

Connector architecture see JCA.

consistent transactions
see also ACID properties.
failure leaves data intact, 863

consumers see message consumers.

container service APIs, 25
container components, 23

container/application server vendor role
choice criteria, application servers, 938
overview, 938

Container-Managed Persistence see CMP.

Container Managed Relationship (CMR) fields
relationships, entity beans, 837

container-managed sign-on
system-level contracts, 1016

container-managed transaction demarcation, 215, 864
declarative semantics, 871
entity beans must use, 864

container-managed transactions
 message-driven beans, 997
containers, 21
see also EJB containers; web containers.
 architecture and components, 23
 compared to modules, 1166
 component lifecycle managed by contracts, 24
 contract between applications and, 19
 declarative container services, 861
 delegation of incoming requests, 27
 ensuring servlet instances are invoked in a separate thread, 267
 interposing a new service, 27
 packaging and deployment, 1166
 runtime services provided by, 27
 scalability of EJB and web containers, 693
 servlet lifecycle management by, 273
content headers, HTTP, 236
content management
 application servers, future prospects, 1157
 possible use of filters, 385
context initialization parameters, 426
Context interface, javax.naming package
 service providers must implement, 49
context parameters, 354
context paths
 identifying web applications using, 415
context propagation
 RMI-IIOP advantage over RMI/JRMP, 148, 149
<context> tag
 server.xml files, 414
contextDestroyed() method
 javax.servlet.ServletContextListener interface, 558
 ServletContextListener interface, 349
contextInitialized() method
 javax.servlet.ServletContextListener interface, 558
 ServletContextListener interface, 349
<context-param> elements
 deployment descriptors, 426
ContextPropertiesFactory class
 manufacturing EJB example, 767
<context-root> tag
 deployment descriptors, 1178
contexts
 JSP implicit objects scope relies on, 474
contracts
 application component management, 24
 application interfaces seen as, 17
 between containers and APIs, 19
 between web containers and servlets, 264
 container components, 23
 EJB container service provision and, 693
 Resource Adapter, 1014
 application contracts, 1014
 system-level contracts, 1014
control flow
 using JSP scriptlets, 453
control flow tags
see also conditional tags; iteration.
 benefits for iteration and conditional logic, 579
 control flow example using, 579
 examples using scriptlets and JSPTL, 578
control logic
 servlets more suitable than JSP for performing, 489
control objects, 734
 advantages over simpler object models, 720
 aggregation during implementation, 721
 analysis model views of entity objects and, 728
 analysis model views of interface objects and, 728
 manufacturing EJB example, 726, 730
 stereotype icons, 727
 UML analysis objects, 719
controller components, MVC, 715
controller designs, JSP, 489
controller servlets, 489
 in Front Controller implementation, 493
 JSP design implementing using, 492
 MVC model of a web server, 592
 web service created by, 627
ControllerServlet class, 495
convenience classes, javax.servlet.jsp.tagext package, 521
conversational state, 735, 737
 storing using stateful session beans, 737
Cookie class, javax.servlet.http package, 264, 324
 creating cookies in servlet API, 264
cookies
 example demonstrating session lifecycle, 328
 retrieving from HttpServletRequest and HttpServletResponse objects, 324
 session tracking using, 320, 322
copyMessages() method
 javax.mail.Folder class, 664
CORBA (Common Object Request Broker Architecture), 45
see also IIOP.
 component model and EJB specification, 687
 IDL interfaces in OTS model, 215
 integrating CORBA with Java classes
 application servers, additional functionality, 1155
 mapping to EJBs, four types described, 896
 one of three Java distributed architectures, 53
 popularity, a reason to use JNDI, 45
 RMI interoperability, 139, 148
 terminology, servers and servants, 145
 Value Object Patterns and interoperability with, 1095
COS (CORBA Object Service)
 COSNaming, 45, 141
 service provider for, in JNDI SDK, 50
count property
 IteratorTagStatus object, 586
CounterFilter class, 386
 deployment, 389
Create function
 CRUD callbacks, 810
 ejbCreate() method, 811
 ejbPostCreate() method, 811
 entity beans, 810
 SportBean laboratory example, 810
CreateException class, javax.ejb package, 892
 predefined in EJB specification, 892
createGroup() method
 java.rmi.activation.ActivationGroup class, 113
createIndexedRecord() method
 RecordFactory interface, 1043, 1045
createInteraction() method
 Connection interface, 1041, 1043, 1045
createManagedConnection() method
 ManagedConnectionFactory interface, 1044
createMappedRecord() method
 RecordFactory interface, 1043, 1045
CreateMovieTables class, 175, 178
 close() method, 180
 createTable() method, 176, 180
 initialize() method, 180, 206
 insertBatchData() method, 198

CreateMovieTables class (continued)

CreateMovieTables class (continued)

insertData() method, 176, 180
insertPreparedStatement() method, 184
main() method, 180
modifying to use datasources, 205

CreateProducts client
manufacturing EJB example, 774

createRegistry() method
`java.rmi.registry.LocateRegistry class, 90, 127`

createResultSet() method
RecordFactory interface, 1043, 1045

createSender() method
`javax.jms.QueueSession interface, 974`

createServerSocket() method
`java.rmi.server.RMIServerSocketFactory interface, 126`

createSocket() method
`java.rmi.server.RMISocketFactory interface, 126`

createStatement() method
`java.sql.Connection interface, 174`

createSubscriber() method
`javax.jms.TopicSession interface, 985`

createTable() method
CreateMovieTables class, 176, 180

createXXXSession() methods
transacted JMS sessions and, 993

CRUD (Create Read Update Delete) callbacks, 809

BMP (bean-managed persistence)
compared to Container-Managed Persistence, 819

caching and, 822

CMP (Container-Managed Persistence)
compared to bean-managed persistence, 819

Create function, 810
`ejbCreate() method, 811`
`ejbPostCreate() method, 811`

Delete function, 818
`ejbRemove() method, 818`

Read function, 814
`ejbLoad() method, 814`

Update function, 817
`ejbStore() method, 817`

CSI (Common Secure Interoperability)
CORBA, in CORBA EJB mapping, 896

curly braces
use with JSP scriptlets, 453

current property
`IteratorTagStatus object, 586`

currentSystemTimeMillis tag
foundation tags
`displayAllRegisteredUsers.jsp page, 626`

cursor position
methods of ResultSet interface, 193

cursorMoved() method
`javax.sql.RowSetListener interface, 225`

custom authentication
vendor-specific APIs, 425

custom sockets, 123
RMI form Java 1.2 release, 124

custom tags
see also JSP tag extensions.
extensibility
JSPTL expected to provide methods for, 631
generic and application-specific tags, 631
interactions with parents and environment
JSPTL expected to define, 631

customer registration
technical support application enhanced to offer, 369

customized HTTP environments

possible use of filters, 385

D

DAO (Data Access Objects) *see Data Access Object Pattern.*

data access

complexity of providing programmatically, 691
design principle of abstracting, 1089

Data Access Object Pattern, 1082

advantages and disadvantages of, 1089
providing common client interfaces, 1108

data binding, XML

JAXB (Java Architecture for XML Binding), 1123

data confidentiality

by encryption of authentication, 881

data integrity

ACID properties, 214

management when using cached rowsets, 228
transactions necessary for, 864

data layers *see data tier.*

data representation interfaces

CCI (Common Client Interface) API, 1040

data sources *see datasources.*

data tier

choosing EJB types for, 1094

n-tier architecture, 16

three-tier architecture, 14, 1062

data types

complex, using JDBC prepared statements for, 184

JAF interface to JavaMail, 649

`java.sql package, 164`

mapping SQL to Java, 186

database access

cached rowsets, 227

web rowsets and, 229

database access classes, `java.sql package, 163`

database connection pooling

J2EE implementation, 1153

database connectivity

Connector Architecture and JDBC, 31

DataSource interface, 201

Enterprise Java Beans, 783

database drivers

function and types, 159

industry standard categorization, 159

initialization the responsibility of `javax.sql package, 201, 205`

JDBC, availability, 162

JDBC, identifying with URLs, 166

JDBC, registering, 167

loading with the Connection interface, 166

managing by methods of DriverManager class, 168

part Java, part native drivers, 160

pure Java database drivers, 161

responsibilities in distributed transactions, 219

support for scrollable resultsets, 196

supporting JDBC rowsets, 227

vendors supporting `java.sql package, 200`

database metadata classes, `java.sql package, 165`

DatabaseMetaData interface, `java.sql package, 165, 877`

`getDefautTransactionIsolation() method, 877`

`othersInsertsAreVisible() method, 193`

`othersUpdatesAreVisible() method, 193`

`ownDeletesAreVisible() method, 193`

- DatabaseMetaData interface, java.sql package (continued)**
- ownInsertsAreVisible() method, 193
 - ownUpdatesAreVisible() method, 193
 - supportsResultSetType() method, 192
 - supportsSavePoint() method, 191
 - supportsTransactionIsolationLevel() method, 877
- databases**
- see also* **Cloudscape database; datasources.**
 - access encapsulated using rowsets, 223
 - access via JNDI in javax.sql package, 199
 - adding fields to, in locking strategies, 880
 - authenticated users, 419
 - chatroom example application, 396
 - common user accounts and connection pooling, 207
 - component of data layer in three-tier system, 15
 - connection pooling explained, 212
 - connections
 - closing with SQLException, 172
 - duration of, and connection pooling, 207
 - database programming with JDBC, 157
 - design considerations for Java interaction, 1106
 - establishing connections through a JDBC driver, 170
 - programmatic transactions and, 222
 - resultsets, encapsulation using rowsets, 200
 - storage, Tech Support servlet example, 299
 - storing state in
 - purchasing system J2EE design example, 1099
 - storing technical support requests, 297
 - technical support application
 - enhanced to offer customer registration, 379
 - Types interface defining column types, 183
 - updating
 - performance gains with batch updates, 198
 - updating a row
 - methods in ResultSet interface, 197
 - use in asynchronous communication, 1100
 - XML interface with, 31
- DataHandler class**
- methods, 649
- DataModelBean class, 498**
- datasource access**
- abstracting with Data Access Object Pattern, 1082
 - javax.sql package, 200
 - JNDI access control, 204
- datasource configuration**
- TechSupportServlet example, 308
- DataSource interface, javax.sql package, 201, 1019, 1027**
- DemoConnector Adapter example, 1027
 - equivalent to java.sql.DriverManager, 200
 - getConnection() method, 201
 - getLoginTimeout() method, 201
 - getLogWriter() method, 202
 - implementation of connection pooling, 213
 - Resource Adapter, 1019
 - responsibilities under the API, 203
 - setLoginTimeout() method, 202
 - setLogWriter() method, 202
- datasources**
- caching content of with Value List Handler Pattern, 1085
 - connections via JDBC/JNDI, diagram, 203
 - creating, using the javax.sql package, 203
 - creation, J2EE application server role in, 204
 - movie catalog example, 205
 - retrieval, by JDBC clients after JNDI binding, 205
- DataStore class, 499**
- DataTruncation class, java.sql package, 165**
- date and time representation**
- java.sql package and, 185
- Date class, java.sql package, 164, 185**
- Date class, java.util package, 185, 977, 979**
- Date field**
- e-mail message headers, 647
- dateFormat tag**
- foundation tag
 - displayAllRegisteredUsers.jsp page, 626
 - displayDetails.jsp page, 616
- DD see deployment descriptors.**
- deactivation, RMI, 123**
- debitFunds() method**
- AccountManager class, 890, 891
- debugging**
- JSP difficulties, 504
 - RMI, 149
 - RMI, properties specific to Sun JDK implementation, 154
- declaration.jsp, 451**
- declarations, JSP**
- compared to scriptlets, 454
 - defining class-wide variables and methods, 450
 - thread safety problems, 454
 - XML equivalent syntax, 476
- declarative invocation**
- explicit invocation and, 26
- declarative nature**
- filter enabling and disabling, 394
 - J2EE applications, 13
- declarative security, 421**
- EJB container service, 692
- declarative semantics**
- container-managed transaction demarcation, 871
 - EJB containers, 695
- declarative services**
- container components, 23, 861
 - EJB containers, 861
 - feature of J2EE services, 26
- declarative transactions**
- declarative transaction demarcation, 215
 - container managed, 871
 - deployment descriptor, 864
 - EJB container service, 692
- declare element**
- JSPTL basic tag, 578
 - displayAllRegisteredUsers.jsp page, 625
 - sub-element of <variable>, 535
- definitions element**
- name attribute, 1117
 - targetNamespace attribute, 1117
 - WSDL document, 1116
- Delete proxy**
- port redirector, 137
- delegation**
- incoming requests to application components, 27
 - servlet API 2.3 wrapper classes, 296
- delegation event model**
- distributed events and, 1106
- delegation model**
- idlj compiler, 147
- Delete function**
- CRUD callbacks, 818
 - ejbRemove() method, 818
 - entity beans, 818
 - SportBean laboratory example, 818

deleteRow() method

deleteRow() method
java.sql.ResultSet interface, 198

deleteSingleMethod() method
JavaMail POP3 access example, 676

deleting entries
LDAP server, using JNDI, 74

delimiters
e-mail header fields, 648

DeliveryMode interface, javax.jms package, 990

demarcation, transactions.
programmatic and declarative approaches, 215

DemoConnector Adapter example
Black Box Resource Adapters, 1023
clients, 1037
configuring database, 1024
DataSource interface, 1027
deploying adapters, 1024, 1025
entity beans, 1030
bean class, 1031
deploying bean, 1037
deployment descriptors, 1036
home interfaces, 1030
ProcessingException class, 1036
remote interfaces, 1031
local transactions, 1024
selecting Black Box adapter, 1024
testing adapter, 1030
XA transactions, 1024

dependency libraries
EAR class loading schemes, 1172
ambiguities in J2EE specification, 1173
before EJB 2.0, 1172
enterprise applications package structure, 1175
impact of dependency on packaging, 1188
packaging and deployment, J2EE, 1183
problems with, 1183
using Class-Path: manifest entry, 1185
using extension mechanism, 1184
using single unified package, 1183
WEB-INF/lib directory, 1183
packaging example, 1185
Class-Path: manifest entry, 1188
deployment descriptors, 1187

dependent objects
discussion on using entities as, 1107

deployer role, 34, 1169
adding users/groups to operational environment, 929
mapping beans and references into JNDI namespace, 933
mapping limitations, reference server, 932
mapping logical database accesses to actual database, 931
mapping logical security roles to users/groups, 928
overview, 927
Resource Adapter, 1022

deployment
chatroom example application, 405
CounterFilter example application, 389
effects on design models, 1069
final stage of J2EE application development, 34
WAR files, 413
web application example, 248

deployment and packaging see **packaging and deployment**.

Deployment APIs
implementing future standards, 1156

deployment configuration
J2EE common approach, 436

deployment descriptor tags
<allt-dd> tag, 1181
<application> tag, 1177
<context-root> tag, 1178
<module> tag, 1177
optional tags, 1181
<security-role> tag, 1181
<servlet-mapping> tag, 416
<web-uri> tag, 1178

deployment descriptors, 241
application assembler role, 916
application servers, 1154
authenticated access to servlets, 421
bean info content fields, 913
BMP (bean-managed persistence), 821
CMP (Container-Managed Persistence), 821
configuring, 1177
context parameters, 354, 426
CounterFilter class, 388
declarative services based on, 23, 26
declare transaction demarcation type, 864
dependency libraries packaging example, 1187
developers must specify for application component groups, 26
DOCTYPE declaration, 1177
EAR (Enterprise Archive) files, 1177, 1180
EJB container and, 695, 870, 1148
EJB specific security roles defined in, 883
enterprise applications, 1175, 1177
entity beans, 819, 820
error handling, 283, 430, 432

example applications
banking example, 914, 917, 923
Bookstore CCI example, 1049
chatroom example application, 364, 403, 423
DemoConnector Adapter example, 1036
Financial Aid Calculator bean example, 746, 753, 758
FreakServlet example, 277
GreetingServlet class, 254
manufacturing EJB example, 848, 961
message-driven bean example, 1005
OrderManagementBean example, 705
shopping cart example application, 344
SportBean laboratory example, 821
technical support application, 308, 378

filters, 393-394

information types, 913

loading servlets on startup, 427

method permissions defined in, 884

MIME mappings, 428

modifying for AttributeServlet, 336

modules, 1177

overview, 913

primary keys, 809

provided by developers in containers, 23

relationships, entity beans, 838

required for an EJB, 701

Resource Adapter, 1019

responsibilities division, EJB developer - application assembler, 913

servlet and JSP definitions and mappings, 241

servlet API 2.3 specification for, 426

servlet initialization parameters, 270, 427

SessionLifeCycleServlet, 330

set order of filter invocation, 394

specifying timeouts, 327, 428

specifying an error page, 283

top-level element, sub-elements, 913

deployment descriptors (continued)

- transactional attributes specified in, 872
- web applications, specified by J2EE, 237, 1178
- web clients should not download, 411
- web containers, 1148
- welcome files, 429

deploytool, J2EE RI, 245

- advantages of using, 254
- EAR files example, 1181
- JSPTL iteration example, 576
- message-driven bean example, 1005
- registration and authentication example, 630
- use with message-driven bean example, 998

deregisterDriver() method

- `java.sql.DriverManager` class, 168

descendant tags

- JSP tag extensions
- retrieving data only at need, 546

design

- compared to architecture, 1061
- interaction with architecture, 1072

design context, 1063**design patterns, 714, 1073**

see also J2EE Patterns.

sources of further information, 1075

design process, 1059

- acceptance of change, 1072
- analysis objects and implementation components, 720
- EJB tier, 718
- iteration and feedback in, 1072
- JSP design strategies, 488
- principles, 1088, 1107
 - database interaction, 1106
 - resources giving help, 1061
 - security and, 888
 - separation from deployment and packaging, 409

Destination interface, javax.jms package, 990**destination queues**

- persistent in J2EE, 979

destroy() method

- `FreakServlet` class, 285
- `javax.servlet.Filter` interface, 391
- `Servlet` interface, 265

destroySubContext() method

- `javax.naming.directory.DirContext` interface, 74

development communities

- J2EE implementation, 1161

DGC (Distribute Garbage Collector), 100

- not available with RMI-IIOP, 148

digest authentication, 421**digital certificates, 52, 56**

- Certificate Authority, 127
- HTTPS client authentication, 421

DirContext interface, javax.naming.directory package

- `bind()` method, 70
- `destroySubContext()` method, 74
- `getAttributes()` method, 65
- `modifyAttributes()` method, 73
- needed to add entries to an LDAP server, 66
- `search()` method, 59

direction

- relationships, entity beans, 837

directives, JSP

- tag category, 444
- use, syntax and types, 445
- XML syntax equivalent, 476

directories

- web applications, 410
- private directories, 410

directory services, 41

- component of data layer in three-tier system, 15
- diagrammatic representations of, 45, 46
- examples of general-purpose services, 41
- J2EE implementation, 1153
- JNDI prevents direct connection to servers, 47
- role of JNDI, 31

dirty reads

- predefined isolation levels, 878

disableHttp property, java.rmi.server package, 137, 150**Dispatcher View Pattern, 1084**

- Sun Java Center discussion, 1092
- use as macro-patterns, 1108

displayAllRegisteredUsers.jsp page, 623

- uses both jr and jx tag libraries, 625

DisplayCount servlet, 387**displayDetails.jsp page, 612****displayMessage() method**

- JavaMail POP3 access example, 675

displaySingleMessage() method

- JavaMail POP3 access example, 675

disposition field, 653

- checking in javamail_pop class, 678

DISTINCT keyword

- SELECT clause, EJB QL, 805

distributable applications, 433

- maintaining session and context, 435

distributable containers

- need for serializable objects, 334

distributable web containers, 433**<distributable> element**

- deployment descriptors, 435

distributed computing

- access to EJB components, 895

factors to consider in distributed designs, 1069**interfaces between distributed components, 1070****Java architectures and LDAP support, 52**

- traditional, compared with J2EE, 18

using RMI, 83**distributed events**

- design considerations, 1106

distributed object services

- J2EE implementation, 1153

distributed transactions, 870

- need XA interface, 871

transaction management, 870**distributed transactions, JDBC, 219**

- contrasted with local transactions, 213

definition and overview, 214**java.sql package does not support, 199****JDBC 2.0 drivers supporting, 871****JDBC 2.0 extension API, 871****JNDI lookup, 221****role of JTA (Java Transaction API), 213****steps required for programmatic demarcation, 221****support for in javax.sql package, 200****distribution mapping**

- CORBA IDL to EJB, 896

DIT (Directory Information Trees)

- base, specifying in a search, 57

LDAP data organization, 43

DN (Distinguished Names)

DN (Distinguished Names)
adding entries to LDAP servers, [66](#)
LDAP data organization, [43](#)

DNS (Domain Naming Service)
compared to LDAP, [78](#)
internet use, returning IP addresses for FQDNs, [41](#)
JNDI service provider, [78](#)
using JNDI without LDAP for, [77](#)

doAfterBody() method
javax.servlet.jsp.tagext.BodyTag interface, [520](#)
javax.servlet.jsp.tagext.IterationTag interface, [518, 540](#)
overriding, [522](#)

doCatch() method
javax.servlet.jsp.tagext.TryCatchFinally interface, [551](#)

DOCTYPE declaration
deployment descriptors, [1177](#)

doEndTag() method
javax.servlet.jsp.tagext.Tag interface, [512, 516, 536](#)
overriding, [522](#)

doFilter() method
javax.servlet.Filter interface, [387, 391, 392](#)
javax.servlet.FilterChain interface, [393](#)
MessageModeratorFilter, [399, 401](#)

doFinally() method
javax.servlet.jsp.tagext.TryCatchFinally interface, [552](#)

doGet() method
ChatAdminServlet, [353](#)
ChatAdminServlet class, [354, 356](#)
ChatRoomServlet class, [364](#)
ListRoomsServlet class
generating an HTML page, [359](#)
ResponseServlet, [380](#)
ShowErrorServlet, [433](#)

doInitBody() method
javax.servlet.jsp.tagext.BodyTag interface, [520, 544](#)
overriding, [522](#)

dollar character
distinguishing expressions from string literals, [571](#)

DOM (Document Object Model)
JAXP (Java API for XML Processing), [1122](#)
UsersToXMLConverter class uses DOM [2.0](#), [628](#)

domains and sub-domains, [40](#)

doPost() method
ChatAdminServlet class, [353, 354, 356](#)
ChatRoomServlet class, [364](#)
GreetingServlet class, [252](#)
TechSupportServlet class, [301](#)

doStartTag() method
javax.servlet.jsp.tagext.BodyTag interface, [519](#)
javax.servlet.jsp.tagext.Tag interface, [511, 516](#)
overriding, [522](#)

double angle brackets
representing a UML stereotype, [725](#)

Double class, java.lang package
valueOf() method, [459](#)

doXXX() methods
HttpServlet class, [269](#)

Drink.java class
JNDI tutorial, [67](#)

Driver interface, java.sql package, [163](#)
connect() method, [166, 169](#)
methods, [170](#)

DriverManager class, java.sql package, [163](#)
getConnection() method, [169, 173](#)
getLoginTimeout() method, [170](#)
getLogWriter() method, [170](#)

javax.sql.DataSource equivalent, [200](#)
limitations addressed by javax.sql, [199](#)
logging methods, [170](#)
methods for managing database drivers, [168](#)
methods to obtain connections, [169](#)
println() method, [170](#)
registerDriver() method, [167](#)
registering database drivers with, [166](#)
setLoginTimeout() method, [170](#)
setLogWriter() method, [170](#)

DriverPropertyInfo class, java.sql package, [163](#)

DriverTest class, [171](#)

DSML (Directory Services Markup Language)
XML specification for LDAP, [48](#)

DSNs (Data Source Names)
ODBC configuration and, [160](#)

DTD (Document Type Definitions)
compared to XML Schemas, [1123](#)
deployment descriptors, [426](#)
differ between JSP [1.1](#) and [1.2 TLDs](#), [510, 523](#)
GreetingServlet class, [255](#)

DTP (Distributed Transaction Processing) model, [214](#)

DuplicateKeyException class, javax.ejb package,
predefined in EJB specification, [892](#)

durable subscriptions
client ID, [986](#)
JMS pub/sub messages, [972](#)
example, [984](#)

durable transactions
see also ACID properties.
reflected in data store, [863](#)

dynamic client-side configuration
class distribution in RMI, [105](#)

dynamic content management
possible use of filters, [385](#)

dynamic interposition
services based on deployment descriptors, [26](#)

dynamic loading
RMI classes, [104](#)

dynamic server-side configuration
class distribution in RMI, [105](#)

dynamic web resources
filters can be used with, [385](#)

DynamicClient class, [106](#)

DynamicServer class, [106](#)

E

EAI (Enterprise Application Integration), [969](#)
JCA (Java Connector Architecture), [1055](#)

EAR (enterprise archive) files
class loading schemes and, [1171](#)
before EJB [2.0](#), [1171](#)
EJB [2.0](#) specification, [1173](#)
local interfaces, [1173](#)
creating, [1176](#)
jar tool, [1176](#)

deployment descriptors, [1177](#)
enterprise applications package structure, [1175](#)
example, [1178](#)
deploying application, [1181](#)
deployment descriptors, [1180](#)
deploytool, [1181](#)
running application, [1181](#)
servlets, [1178, 1179](#)
stateless session beans, [1178](#)
bean class, 1178

EAR (enterprise archive) files (continued)

J2EE modules, packaging, [35](#), 1166, 1168
 limitations of EAR files, 1170
 ordering modules, 1182

ebXML (electronic business XML)

registry, 1121
 web services, 1121

e-commerce applications

effects on value of information assets, [11](#)
 requirements contrasted with those of banks and ATMs, 1071
 requirements of design process, [1060](#)
 tear-off applications and, 1066

e-commerce purchasing system see **purchasing system J2EE design example.****EIS (Enterprise Information Systems)**

description, 1010
 JCA (Java Connector Architecture) and, 1010
 Resource Adapter, 1011

EIS sign-on

application-managed sign-on, 1016
 container-managed sign-on, 1016
 system-level contracts, 1016

EJB (Enterprise JavaBeans), 685

see also entity beans; manufacturing EJB example; message-driven beans; session beans.

.NET and COM as alternatives to, 687
 access control in EJB specification, 882
 access following system exceptions, 894
 application assembler role, 916-917, 923, 926
 banking example, 883, 903, 909, 926, 933
 bean provider role
 Account entity bean example, 911
 AccountManager session bean example, 909
 bean info content fields, deployment descriptor, 913
 deployment descriptors, 912
 <ejb-local-ref> element, [908](#)
 <ejb-ref> element, 907
 overview, 902
 reference types, providing, 906
 <resource-ref> element, 907
 security role reference, [908](#)
 specifying entity bean's primary key class, 914
 caching state, avoiding ejbLoad() and ejbStore()
 methods, 874

client developers, [698](#)

client-tier access, [717](#)

communication, 895, 897

compared to web applications, 1171

container/application server vendor role, 938

containers see **EJB containers**.

CORBA to EJB mapping, 896

creation and Service Locator Patterns, 1079

data access and the Fast Lane Reader Pattern, 1087

data access and the Session Façade Pattern, 1078

database connectivity, 783

defined, 686

deployer role, 927, 929, 931-933

deployment descriptors, 913

 application assembler role, 916

 banking example, 914, [917](#), 923

 manufacturing EJB example, 961

design process, 718

development and deployment roles, 901

 application assembler, 916

 container/application server vendor, 938

 deployer, 927

 overview, 902, 964

 system administrator, 936

direct access from a servlet, [717](#)

direct access via RMI compared to web access, [717](#)

EAR class loading schemes, 1171

EJB 2.0 specification, [1173](#)

EJB containers implement EJB API, [22](#)

encapsulating data from, with Value Object Patterns, [1080](#)

environment variables, 783

executing client in a container, 926

finder methods, 761

handle serialization, 762

heavyweight components, 720

home, local and remote interfaces, [698](#)

implementing MVC model components as, 715

indirection, developer - product relationship, 906
 application-level, [908](#)

instances, discarded because of system exceptions, 894

interface design, [1104](#)

interfaces inherently RMI-IIOP, [149](#)

isolation level management, 877

J2EE component technology, [29](#)

J2EE platforms must support, [20](#)

JavaBeans compared to, 687

JNDI importance to, [45](#)

limitations, [711](#)

listing behavior, 762

one of three Java distributed architectures, [53](#)

overview, 905

packaging dependency libraries

 Class-Path: manifest entry, 1185

 using extension mechanism, 1184

pooling, 738

portability, implementation environment independence, 905

recovery from system exceptions, 893

resource recycling, 1103

restrictions on use, 712-714

scalability improvement over JavaBeans-based model, 1093

security, understanding, 882

specification

 predefined exceptions in, [889](#)

 transactional attributes in, 871

specification supports declarative demarcation, [216](#)

specification, predefined exceptions in, 892

state checking in locking strategies, 880

suitable areas for using EJBs, 689

system administrator role, 936-938

three varieties of beans, [688](#)

trade-offs in implementations, [731](#)

troubleshooting tips, 963

 JAR deployment failure, 964

types introduced, [29](#)

using JNDI without LDAP for, [77](#)

viewed as business objects, 690

EJB 2.0 new features

IIOP communication transport mandatory, [717](#)

local interfaces, 1173

message-driven beans, 995

summarized, 685

EJB 2.0 specification

bean provider view, 700

client views, [698](#)

EAR class loading schemes, [1173](#)

J2EE implementation, 1150

prohibited activities, [711](#)

rules for EJB use, 694

three-layer model, 694

EJB Application JAR files

packaging and deployment, J2EE, 1168

EJB clients

three categories, 22

EJB components

using in web applications, 714

EJB container services

declarative transactions, 692

optional services, 695

separation from java files, 693

EJB containers

see also containers; web containers.

application servers, 1148

contracts with beans, 694

declarative semantics, 695

deployment descriptors, 1148

enterprise applications, 1148

executing clients in, 926

interposition, 696

isolation level management, 877

limits on support for transactions, 863

managing transactions, 870

mechanisms of service provision, 693

one of four classes of container, 22

packaging and deployment, J2EE, 1167

principals, translating, 886

runtime incompatibility, 895

security responsibilities of, 882

support for error handling, 889

support for system-level services, 861

system-level services, 690

EJB modules

make-up and packaging, 35

EJB QL

CMP 2.0, 804

compared to SQL, 804

EJB 2.0 introduced, 685

finder methods, 825, 839

FROM clause, 805

- SELECT clause, 805

WHERE clause, 806

ejbActivate() method

entity beans, 828

not called with stateless session beans, 744

stateful session beans, 751

EJBContext interface, javax.ejb package, 886

getCallerPrincipal() method, 886

getRollbackOnly() method, 891

isCallerInRole() Method, 886

setRollbackOnly() method, 891

ejbCreate() method

bean class, 704

BMP (bean-managed persistence), 812

CMP (Container-Managed Persistence), 811

Create function, 811

OrderManagement example, 211

session beans, 742

stateful session beans, 751

stateless session beans, 744

EJBHome interface, javax.ejb package

home interfaces must extend, 703

ejb-jar.xml files

see also deployment descriptors.

EJB deployment descriptor, 35, 701

ejbLoad() method

BMP (bean-managed persistence), 815, 816

caching data

problems with, 874

when to use, 874

CMP (Container-Managed Persistence), 815

Read function, 814

EJBLocalObject interface, javax.ejb package

local interfaces must extend, 702

<ejb-local-ref> element

bean provider role, 908

EJBObject interface, javax.ejb package

remote interfaces must extend, 701

ejbPassivate() method

entity beans, 828

not called with stateless session beans, 744

stateful session beans, 751

ejbPostCreate() method

bean class, 704

BMP (bean-managed persistence), 812

CMP (Container-Managed Persistence), 811

Create function, 811

<ejb-ref> element

bean provider role, 907

ejbRemove() method

BMP (bean-managed persistence), 818

Container-Managed Persistence, 818

Delete function, 818

javax.jms.MessageDrivenBean interface, 996

stateful session beans, 751

stateless session beans, 744

transitioning, 746

EJBs *see* EJB.

ejbStore() method

BMP (bean-managed persistence), 817

CMP (Container-Managed Persistence), 817

Update function, 817

exprvalues

attribute values accepted by jx tag library, 571

e-mail

see also mail.

address check on need for customer registration, 369

technologies introduced, 635

viewed as a loosely-coupled messaging technique, 968

viewed as an asynchronous communication mechanism, 1100

e-mail addresses

creation and handling, 656

e-mail attachments

example of sending, 669

need for non-ASCII led to MIME, 638

saving to disk, 677

e-mail messages

addressing, 646

addressing within folders, 661

components, 644

content, handling in JavaMail, 652

copying and moving, 664

deleting, 676

delivery, 666

fields searchable with javax.mail.search package, 665

header field delimiters, 648

header fields, 645, 648

storing SMTP server name, 680

headers, common fields in, 645

loading and sending stored messages, 680

- e-mail messages (continued)**
- retrieval, 662
 - advanced message fetching, 663
 - example retrieving from a POP folder, 662
 - saving to disk, 678
 - searching message stores, 664
 - status flags, 654
- encapsulation**
- java code in View Helper Patterns, 1083
 - ordering workflow, in purchase order example, 1096
- encode tag**
- foundation tags, used in `displayDetails.jsp` page, 616
- encodeURL() method**
- `HttpServletResponse` interface, 332
- encryption** see **SSL; TLS.**
- Enhydra**
- open source application servers, 1152
- enterName.jsp, 500**
- output, 502
- enterprise application development process, 1174**
- deployment descriptors, 1175
 - description, 1174
- enterprise applications**
- application servers, 1147
 - deployment descriptors, 1177
 - EJB containers, 1148
 - examples and special features of, 10
 - forces constraining the architecture of, 1071
 - J2EE suitability for, 10, 1147
 - resources giving help on development, 1061
 - web containers, 1148
- enterprise applications package structure, 1175**
- enterprise architecture**
- distinction from simple n-tier, 16
- enterprise data**
- JSP tag extensions for concealing, 563
- enterprise portals**
- JCA (Java Connector Architecture), 1055
- entity beans**
- Account class
 - purchasing system J2EE design example, 1103
 - activation, 828
 - `ejbActivate()` method, 828
 - advantages, 799
 - avoiding loopback calls, 714
 - banking example, 903
 - bean and container-managed persistence, 689, 799
 - bean class derived from `javax.ejb.EntityBean`, 704
 - caching, 822
 - clients, 831
 - compared to session beans, 794
 - stateful session beans, 794
 - stateless session beans, 797
 - configuring WebLogic Server 6.1, 835
 - `config.xml` file, 835
 - CRUD callbacks, 809
 - Create function, 810
 - Delete function, 818
 - Read function, 814
 - Update function, 817
 - DemoConnector Adapter example, 1030
 - bean class, 1031
 - deploying bean, 1037
 - deployment descriptors, 1036
 - home interfaces, 1030
 - ProcessingException class, 1036
 - remote interfaces, 1031
- deployment descriptors**, 819, 820
- DemoConnector Adapter example, 1036
 - distinction from other EJB varieties, 688
 - EJB 2.0 improvements, 685
 - finder methods, 824
 - `findByPrimaryKey()` method, 825
 - home and remote interfaces, 698
 - introduced, 30
 - introduction, 793
 - lifecycle, 828
 - manufacturing EJB example, 838
 - must use container-managed transaction demarcation, 864
 - passivation, 828
 - `ejbPassivate()` method, 828
 - pooled state, 828
 - primary keys, 807
 - ready state, 828
 - reentrancy, 829
 - relationships, 837
 - remote interfaces, 830
 - representation, 793
 - SportBean laboratory example, 807
- entity objects, 734**
- aggregation during implementation, 721
 - analysis model views of control objects and, 728
 - manufacturing EJB example, 726
 - stereotype icons, 726
 - UML analysis objects, 719
- EntityBean interface, javax.ejb package**
- bean class for entity beans derived from, 704
- EntityContext interface, javax.ejb package**
- Read function, 815
 - `getEJBObject()` method
 - alternative to passing 'this' references, 714
- environment variables**
- Enterprise Java Beans, 783
- Error class, java.lang package, 894**
- system exceptions can derive from, 894
- error codes**
- compared with throwing exceptions, 432
- error handling**
- EJB container service, 692
 - EJB container services and, 862
 - EJB container support for, 889
 - `HttpServletResponse` interface methods, 295
 - JSP (JavaServer Pages), 551
 - `ServletException` class, 431
- error messages**
- JSP tag extensions, 514
 - providing user-friendly messages, 622
- error pages**
- deployment descriptors, 430
 - displaying for `UnavailableException`, 283
 - `error.html` page, 424
 - `error.jsp`, 487
- <error-page> element**
- deployment descriptors, 432
- escaping characters**
- JSP attributes, 444
- escaping content**
- scriptlets in XML-based JSP, as CDATA, 477
- evaluating J2EE implementations, 1158**
- BEA, 1158
 - IBM, 1158
 - Oracle, 1158

event handling

distributed events, 1106
ServletContext lifecycle, 349
session attributes, 339
session lifecycle, 337

example applications
see also JSP, examples and individual examples.
account transfer example, 864
banking example (EJB), 883
bookstore CCI example, 1045
Chatroom example application, 350
CounterFilter page request example, 386
DemoConnectorAdapter example, 1023
Financial Aid calculator bean example, 739
FreakServlet example, 275
GreetingServlet web application, 242
Manufacturing EJB example, 722, 760, 939
movie catalog example, 175
OrderManagementBean example, 699
Purchasing System J2EE example, 1064
Registration and authentication JSP example, 590
shopping cart example application, 341
SportBean laboratory example, 807
StockQuote web service example, 1130
TechSupportServlet example, 297

<example> sub-element
TLD <tag> elements, new in JSP 1.2, 525

ExamplesTLDServletContextListener class, 560

exception and warning classes
CCI (Common Client Interface) API, 1040
Exception class, java.lang package, 889
application exceptions derivable from, 889
exception classes, javax.servlet package, 272

exception handling
JSP tag extensions, 523
message-driven beans, 996
nesting SQLExceptions, 177
purchasing system J2EE design example, 1097
RMI (Remote Method Invocation), 91
specifying in deployment descriptors, 432

ExceptionListener interface, javax.jms package, 990

exceptions
see also application exceptions; system exceptions.
predefined exceptions, EJB specification, 889, 892
servlets, 272
throwing, compared with using HTTP error codes, 432
unexpected exceptions, 889
warnings and, java.sql package, 165

executables
web clients should not download, 411

execute() method
java.sql.Statement interface, 180
sun.jdbc.rowset.CachedRowSet class, 228

executeBatch() method
java.sql.Statement interface, 198

executeQuery() method
java.sql.Statement interface, 180

executeUpdate() method
java.sql.Statement interface, 176, 177

existing systems *see legacy systems.*

explicit invocation
declarative invocation and, 26

exportObject() methods
java.rmi.activation.Activatable class, 116, 120
java.rmi.server.UnicastRemoteObject class, 95, 109, 142

expr tag
JSPTL basic tag, 577

expression language

JSPTL support expected to grow, 631
expression.jsp, 454
expression-language support, JSPTL, 571
expressions, JSP, 454
tag extension body content, 533
XML equivalent syntax, 476

Extensible Markup Language *see XML.*

Extensible Stylesheet Language Transformations *see XSLT.*

extension mechanism
limited support for, 1184
packaging dependency libraries, 1184
Class-Path: manifest entry, 1185
EJBs (Enterprise JavaBeans), 1184

F

façades *see also Session Façade Pattern.*
advantages of using, 736
using session beans as façades, 735

failure-recovery
distributed web containers, 435

Fast Lane Reader Pattern, 1087
use in middle tier of purchase order example, 1095

fat clients
application clients viewed as, 233

federation, 49

feedback
iteration and, in design processes, 1072

fetch direction
methods of the ResultSet interface, 194

fetch() method
javax.mail.Folder class, 663

FetchProfile class, javax.mail package, 663
field groupings, 664
JavaMail POP3 access example, 675

FIFO (First In First Out) rule
point-to-point message queues, 970

file systems
EJB 2.0 specification restrictions, 713

FileDataSource class, JAF
JavaMail file attachments example, 671

filter API
classes and interfaces, 390
example application to illustrate, 386

filter chaining, 394
introduced, 384

<filter> and <filter-mapping> elements
deployment descriptor, CounterFilter class, 388
deployment descriptor, filters, 393

Filter interface, javax.servlet package, 387, 391

FilterChain interface, javax.servlet package, 393

FilterConfig interface, javax.servlet package, 392

filtering
chatroom example application, 395
classes, as category of servlet API, 264
filters compared to request dispatching, 386

filters compared to servlets
lifecycle and action, 384
performing specific tasks, 385

introduced in servlet API version 2.3, 383

JSP tag extension body content, 544

LDAP search filters, 57

filter lifecycle stages, 391

order of filter invocation set by deployment descriptor, 394

possible use of wrapper classes, 297

finalize() method
 avoiding using, 744

Financial Aid Calculator bean example, 739

- stateful and stateless session beans, 755
 - access control layer, 755
 - bean class, 756
 - clients, 755
 - deployment descriptors, 758
 - home interfaces, 755
 - remote interfaces, 755
 - services layer, 755
- stateful session beans, 748
 - bean class, 749
 - clients, 752
 - deployment descriptors, 753
 - home interfaces, 748
 - remote interfaces, 748
 - setSessionContext() method, 751
- stateless session beans, 740
 - bean class, 742
 - clients, 741
 - deployment descriptors, 746
 - home interfaces, 742
 - remote interfaces, 740
 - setSessionContext() method, 744

findAncestorWithClass() method
`javax.servlet.jsp.tagext.TagSupport` class, 547, 548, 577

findByPrimaryKey() method
 entity beans, 825
 implementation

- BMP (bean-managed persistence), 826

 primary keys, 825

finder methods
 BMP (bean-managed persistence), 825, 826
 CMP (Container-Managed Persistence), 825
 EJB QL, 825, 839
 Enterprise JavaBeans, 761
 entity beans, 824
 findByPrimaryKey() method, 825
`FinderException`, 825
 manufacturing EJB example, 839
 naming convention, 824
 Order EJB, 839
 SELECT clause, EJB QL, 805
 SportBean laboratory example, 825, 826

FinderException class, javax.ejb package, 892
 finder methods, 825
 predefined in EJB specification, 892

firewalls
 bypassing using SOCKS protocol, 138
 bypassing with downloaded socket factories, 138
 RMI and bypassing, 136
 support

- RMI-IIOP advantage over RMI/JRMP, 148

first property
`IteratorTagStatus` object, 586

first() method
`java.sql.ResultSet` interface, 194

Flags class, javax.mail package, 654

flushBuffer() method
`ServletResponse` interface, 294

Folder class, javax.mail package, 661

- copyMessages() method, 664
- fetch() method, 663
- getPermanentFlags() method, 654
- search() method, 664

folders
 access methods for, 658
 addressing e-mail messages within, 661
 opening in READ_WRITE mode, 676

forEach tag
 JSPTL iteration tag, 580

forEachColor.jsp page, 572

form tag
 HTML tag library, registration example, 600, 603
`loginForm.jsp` page, 612

form-based authentication, 422
 adding users, 425
 example, 423

forTokens tag
 JSPTL iteration tag, 581

forward() method
`RequestDispatcher` interface, 369, 380

forward.html page, 468

forward.jsp, 469

foundation tags, 615

FQDN (Fully Qualified Domain Names), 40
 DNS returns an IP address for, 41
 specifying to `java.rmi.cgi` script, 137

frame refresh
 using `writeMessages()` method, 364

FreakServlet class
`destroy()` method, 285
`getInitParameter()` method, 282
`getServletConfig()` method, 282

FreakServlet example, 275
 initialization, 282, 427
 instantiation, 282
 service state, 283
 threading, 285

FROM clause, EJB QL, 805

From field
 e-mail message headers, 646

Front Controller Pattern, 1075
 Dispatcher View Pattern combines View Helper and, 1084
 JSP design approach, 491

- benefits, 503
- compared to page-centric designs, 493
- implementation, 492, 494

 Service to Worker Pattern combines View Helper and, 1085

FTP (File Transfer Protocol)
 early messaging technique, 968
 example of a stateful protocol, 318

functional groups of classes, java.sql package, 162

functionality
 delegation to a common component, 25

fundamental tags, 615

G

garbage collection see DGC.

gatekeeper role

- JSP tag extensions
 - alternatives, 562

GC see DGC.

generic controller frameworks

- JSP design and, 504

generic user accounts, 207

GenericServlet class, javax.servlet package

GenericServlet class, javax.servlet package, 266
implementing Servlet interface by extending, 264
init() method, 266
log() method, 266
GET requests, HTTP
compared to POST requests, 235
conditional GET requests in HTTP 1.1, 269
parameter appended to URL as query string, 289
getAll() method
javax.naming.directory.Attribute interface, 62
getAllRecipients() method
javax.mail.Message class, 646
getAttribute() method
Catalog servlet, called by, 342
HttpSession interface, 333
ServletContext interface, 349, 359
ServletRequest interface, 288
getAttributeNames() method
HttpSession interface, 333
ServletContext interface, 349
ServletRequest interface, 288
getAttributes() method
javax.naming.directory.DirContext interface, 65
javax.naming.directory.SearchResult class, 62
getAuthType() method
HttpServletRequest interface, 422
getBodyPart() method
javax.mail.Multipart class, 651
getBufferSize() method
ServletResponse interface, 294
getCallerPrincipal() method
Session Context interface, 745
getCallerPrincipal() Method
EJBContext interface, 886
getCharacterEncoding() method
ServletRequest interface, 289
getColumnCount() method
java.sql.ResultSetMetaData interface, 182
getConcurrency() method
java.sql.ResultSet interface, 197
getConnection() method
ConnectionFactory interface, 1041, 1043
java.sql.DriverManager class, 169, 173
javax.sql.DataSource interface, 201
javax.sql.PooledConnection interface, 211
ManagedConnection interface, 1044
getContent() method
javax.activation.DataHandler class, 649, 650
javax.mail.Part interface, 650
getContentType() method
javax.activation.DataHandler class, 649
javax.mail.Multipart class, 651
javax.mail.Part interface, 652
getContext() method
ServletContext interface, 348
getCookies() method
HttpServletResponse interface, 324
getCount() method
javax.mail.Multipart class, 651
getCreationTime() method
HttpSession interface, 327, 330
getDefault() method
java.util.Locale class, 605
getDefaultFolder() method
javax.mail.Folder class, 658
getDefaultInstance() method
javax.mail.Session class, 641, 643

getDefaultValueIsolation() method
DatabaseMetaData interface, 877
getDescription() method
javax.mail.Part interface, 653
getDisposition() method
javax.mail.Part interface, 653
getDriver() method
java.sql.DriverManager class, 168
getDrivers() method
java.sql.DriverManager class, 168
getEJBHome() method
Session Context interface, 744
getEJBObject() method
Session Context interface, 744
Entity Context interface
alternative to passing 'this' references, 714
getEnclosingWriter() method
javax.servlet.jsp.tagext.BodyContent class, 521
getFetchDirection() method
java.sql.ResultSet interface, 195
getFetchSize() method
java.sql.ResultSet interface, 195
getFileName() method
javax.mail.Part interface, 653
getFilterName() method
javax.servlet.FilterConfig interface, 392
getFlags() method
javax.mail.Message class, 655
getFolder() method
javax.mail.Folder class, 658
getFrom() method
javax.mail.Message class, 646
getHeader() method
HttpServletRequest interface, 292
javax.mail.Message class, 648
getHeaderNames() method
HttpServletRequest interface, 292
getHeaders() method
HttpServletRequest interface, 292
getId() method
HttpSession interface, 327, 330
getID() method
javax.naming.directory.Attribute interface, 62
getInitParameter() method
FreakServlet class, 282
javax.servlet.FilterConfig interface, 392
ServletConfig interface, 270
ServletContext interface, 348, 426
getInitParameterNames() method
javax.servlet.FilterConfig interface, 392
ServletConfig interface, 271
ServletContext interface, 348, 426
getInputStream() method
javax.activation.DataHandler class, 649
ServletRequest interface, 288
getInstance() method
javax.mail.Session class, 641, 643
getInvalidAddresses() method
javax.mail.Transport class, 667
getLastAccessedTime() method
HttpSession interface, 327, 330
getLineCount() method
javax.mail.Part interface, 653
getLoginTimeout() method
java.sql.DriverManager class, 170
javax.sql.DataSource interface, 201

getLogWriter() method
 DataSource interface, 202
 java.sql.DriverManager class, [170](#)

getMaxInactiveInterval() method
 HttpSession interface, 327, [330](#)

getMessage() method
 javax.mail.Folder class, 662
 javax.mail.Message class, 647

getMessageCount() method
 javax.mail.Folder class, 662

getMessages() method
 javax.mail.Folder class, 662, 664

getMetaData() method
 Connection interface, 877
 java.sql.Connection interface, 173

getMethod() method
 HttpServletRequest interface, 292

getMimeType() method
 ServletContext interface, 347

getName() method
 HttpSessionBindingEvent class, 340
 Principal interface, 886

getNamedDispatcher() method
 ServletContext interface, 347, 369, 380

getNewMessageCount() method
 javax.mail.Folder class, 662
 overhead associated with, 663

getNextException() method
 java.sql.SQLException class, 177

getOriginal() method
 sun.jdbc.rowset.CachedRowSet class, 228

getOriginalRow() method
 sun.jdbc.rowset.CachedRowSet class, 228

getOutputStream() method
 javax.activation.DataHandler class, 649
 ServletResponse interface, 293

getParameter() method
 chatroom example application, 400
 HttpServletRequest interface, 253
 javax.servlet.http.HttpServletRequestWrapper class,
 401
 ServletRequest interface, 287
 TechSupportServlet class, 302

getParameterNames() method
 ServletRequest interface, 287

getParameterValues() method
 ServletRequest interface, 287
 ShoppingCart servlet, 344

getParent() method
 javax.mail.Multipart class, 651, 652
 javax.servlet.jsp.tagext.Tag interface, 517, 547, 548

getPasswordAuthentication() method
 javax.mail.PasswordAuthentication class, 643

getPathInfo() method
 HttpServletRequest interface, 291

getPathTranslated() method
 HttpServletRequest interface, 291

getPermanentFlags() method
 javax.mail.Folder class, 654

getPersonalNamespaces() method
 javax.mail.Folder class, 658

getPooledConnection() method
 javax.sql.ConnectionPooledDataSource interface, 211

getProtocol() method
 ServletRequest interface, 269

getQueryString() method
 HttpServletRequest interface, 291

getQueryTimeout() method
 java.sql.Statement interface, [182](#)

getReader() method
 ServletRequest interface, [288](#)

getRealPath() method
 ServletContext interface, 347

getRecipients() method
 javax.mail.Message class, 646

getRecordFactory() method
 ConnectionFactory interface, 1041

getRegistry() method
 java.rmi.registry.LocateRegistry class, [90](#)

getRemoteUser() method
 HttpServletRequest interface, 422

getReply() method
 javax.mail.Message class, 647

getRequestDispatcher() method
 javax.servlet.ServletContext interface, 619
 ServletContext interface, 347, 369

getRequestURI() method
 HttpServletRequest interface, 291

getRequestURL() method
 HttpServletRequest interface, 291

getResource() method
 ServletContext interface, 347

getResourceAsStream() method
 java.lang.Class class, 713
 ServletContext interface, 347

getResourcePaths() method
 ServletContext interface, 348

getRollbackOnly() method
 javax.ejb.EJBContext interface, 891
 SessionContext interface, 745

getRootCause() method
 ServletException class, 272

getSentDate() method
 javax.mail.Message class, 647

getServerInfo() method
 ServletContext interface, 348

getServletConfig() method
 FreakServlet class, 282
 Servlet interface, [266](#)
 accessing the ServletConfig object, 271

getServletContext() method
 javax.servlet.FilterConfig interface, 393
 javax.servlet.ServletContextEvent class, 559
 ServletConfig interface, 271

getServletContextName() method
 ServletContext interface, 348

getServletInfo() method
 Servlet interface, [266](#)

getServletName() method
 ServletConfig interface, 271

getServletPath() method
 HttpServletRequest interface, 291
 javax.servlet.http.HttpServletRequest interface, [495](#)

getSession() methods
 Catalog servlet, [342](#)
 HttpServletRequest interface, 324-325, [330](#)
 HttpSessionBindingEvent class, 341
 HttpSessionEvent class, 339

getSharedNamespaces() method
 javax.mail.Folder class, 658

getSize() method
 javax.mail.Part interface, 653

getStatus() method

getStatus() method
javax.transaction.UserTransaction interface, 221
Order EJB, 844
UserTransaction interface, 875, 891

getStockPrice() method
StockQuoteService web service example, 1130

getStore() method
javax.mail.Store class, 657

getString() method
java.sql.ResultSet interface, 181
javax.servlet.jsp.tagext.BodyContent class, 521

getTransactionIsolation() method
Connection interface, 877

getType() method
javax.mail.Folder class, 659, 661

getUnreadMessageCount() method
javax.mail.Folder class, 662

getUnsentAddresses() method
javax.mail.Transport class, 667

getURLName() method
javax.mail.Store class, 660

getUserNamespaces() method
javax.mail.Folder class, 658

getUserPrincipal() method
HttpServletRequest interface, 422

getUserTransaction() method
SessionContext interface, 745

getValidSentAddresses() method
javax.mail.Transport class, 667

getValue() method
HttpSessionBindingEvent class, 341

getVariableInfo() class
javax.servlet.jsp.tagext.TagExtraInfo class, 538

getWarnings() method
java.sql.Connection interface, 178

getWriter() method
HttpServletResponse interface
warning against using when forwarding requests, 380
ServletResponse interface, 293

getXAConnection() method
javax.sql.XADataSource interface, 219

getXAResource() method
javax.sql.XAConnection interface, 219

GoNative security provider, 127

granularity
J2EE components, 13

greetings example web application, 242

GreetingServlet class, 243
components of, 252
deployment descriptor, 254
doPost() method, 252
packages imported by, 252

GregorianCalendar object, java.util package, 253

group loading
abstract accessors, CMP 2.0, 803

hashing algorithms
digest authentication, 421

HashMap class, java.util package
chatroom example application, 351

hashmaps
chatroom example application, 351

hasNewMessages() method
javax.mail.Folder class, 662

HEAD requests, HTTP, 235

heavyweight objects
EJBs as, 720, 730
optimizing calls to, 730

hello tag, 510

Hello.tld file, 512
modified to use a servlet context listener, 560

helloAttribute.jsp page, 530

HelloClient class, 95
activatable objects version, 118
modified to connect to the server when instantiated, 105
RMI-IIOP version, 141

HelloInterface, 93
modified to return a remote object, 101

HelloInterface.idl file, 143
using Java IDL, 144

helloIterator.jsp page, 542

helloIteratorWithNesting.jsp page, 549

HelloServant class, CORBA, 145

HelloServer class, 95
activatable objects version, 117
export methods version, 120
Java IDL version, 145
remote object version, 101
RMI-IIOP and RMI/JRMP version, 142
RMI-IIOP version, 140
secure version, 128

HelloTag class, 511, 523

helloTagContent.jsp page, 533

helloWithVariables.jsp page, 537

Hidden form fields
not used by servet API, 322
session tracking using, 320, 322

hierarchies
directories as hierarchical databases, 39, 41
JavaMail folders, 659
LDAP data organization, 43

home interfaces
banking example
entity beans, 904
session beans, 904
Bookstore CCI example, 1046
casting with RMI-IIOP, 700
DemoConnector Adapter example, 1030
EJBs, 698
extend EJBHome interface, 703
Financial Aid Calculator bean example, 742, 748, 755
ManageOrders EJB, 764
Manufacture EJB, 766
Order EJB, 839
Product EJB, 845

hostname property, java.rmi.server package, 150

HTML (Hypertext Markup Language)
browser-specific, using <jsp:plugin>, 470
error page generation, 284
example application counting page requests, 386

H

handle serialization
Enterprise JavaBeans, 762

handleRequest() method
RegistrationControllerServlet, 596
RequestHandler interface
Front Controller implementation, 494

HTML (Hypertext Markup Language) (Continued)

form controls, table, 289
 forms, TechSupportServlet example, 298
 generating with JSP tag extensions
 whether advisable, 562
 JSP generation of, in purchase order example, 1088
 JSP tag extensions generate, 509
 register.html page, 374

HTML (Hypertext Markup Language)

static pages unsuitable for URL rewriting, 322
 techsupp.html page revised, 370
 web application example, 243

HTML forms

limitations of the <input> tag, 599
 using JSP pages, 599
HTTP (Hypertext Transfer Protocol), 234
see also GET requests; HTTP version 1.1; POST requests.
 HttpServletRequest methods for reading headers, 292
 J2EE-supported communication technology, 32
 javax.servlet.http classes are specific to, 260
 key features of the protocol, 236
 request methods, 234
 requests
 extracting messages from, 400
 interception can be altered by filters, but not servlets, 383
 mapping to servlets, 415
 Tech Support servlet example, 297, 301
 requests and responses, classes for servlet accessss, 286
 responses, 235
 error status codes, 431
 SOAP messages, 1114
 stateless protocol, 318
 status codes, 431
 suitability for web browsing but not web applications, 319
 versions and HttpServlet.doXXX() methods, 269

HTTP environments

customization, as possible use of filters, 385

HTTP session events

application lifecycle events, 557

HTTP tunneling

disadvantages, 137
 RMI technique for bypassing firewalls, 136

HTTP version 1.1

additional request methods, 234
 conditional GET requests, 269
 persistent connections, 319

HttpJspClass, javax.servlet.jsp package

JSP classes must extend, 24

HTTPS protocol

use of SSL, 321
HttpServlet class, javax.servlet.http package, 268
 doXXX() methods, 269
 implementing Servlet interface by extending, 264
 service() methods, 268

HttpServletRequest interface, javax.servlet.http package, 286, 289, 325

accessing request parameters, 289
 getAuthType() method, 422
 getHeader() method, 292
 getHeaderNames() method, 292
 getHeaders() method, 292
 getMethod() method, 292

getParameter() method, 253
 getPathInfo() method, 291
 getPathTranslated() method, 291
 getQueryString() method, 291
 getRemoteUser() method, 422
 getRequestURI() method, 291
 getRequestURL() method, 291
 getServletPath() method, 291, 495
 getSession() method, 324, 330
 getSession() methods, 325
 getUserPrincipal() method, 422
 isUserInRole() method, 422
 methods, 290, 325
 setAttribute() method, 380, 432

HttpServletRequestWrapper class, javax.servlet.http package, 292, 296, 401

HttpServletResponse interface, javax.servlet.http package, 286, 325
 encodeURL() method, 332
 error handling methods, 295
 getCookies() method, 324
 getWriter() method
 warning against using when forwarding requests, 380
 methods, 295
 sendError() method, 295, 430
 sendRedirect() method, 296
 setContentType() method, 429
 setStatus() method, 295
 URL encoding, methods for, 325

HttpServletResponseWrapper class, javax.servlet.http package, 296

HttpSession interface, javax.servlet.http package, 326
 AttributeServlet illustrating methods of, 337
 getAttribute() method, 333
 getAttributeNames() method, 333
 getCreationTime() method, 327, 330
 getId() method, 327, 330
 getLastAccessedTime() method, 327, 330
 getMaxInactiveInterval() method, 327, 330
 invalidate() method, 328, 332
 isNew() method, 328
 methods of, 326
 removeAttribute() method, 334
 session tracking with the servlet API, 324
 setAttribute() method, 333
 setMaxInactiveInterval() method, 327, 428

HttpSessionActivationListener interface, javax.servlet.http package
 methods, 338

HttpSessionAttributesListener interface, javax.servlet.http package
 methods, 340

HttpSessionBindingEvent class, javax.servlet.http package, 325
 methods, 340
 session tracking, 325

HttpSessionBindingListener interface, javax.servlet.http package, 325
 methods, 339
 session tracking, 325

HttpSessionContextListener interface, javax.servlet.http package, 557

HttpSessionEvent class, javax.servlet.http package
 getSession() method, 339

HttpSessionListener interface, javax.servlet.http package, 561
 methods, 338

HTTP-to-CGI tunneling, 137

HTTP-to-port tunneling, [136](#)

HttpUtils class, javax.servlet.http package, [264](#)
providing helper methods in servlet API, [264](#)

I

i18n *see* **internationalization**.

IBM

alternative service provider to those in JNDI SDK, [50](#)
evaluating J2EE implementations, [1158](#)
WebSphere application server, [1149](#)

IBM web services toolkit

proxy generation utility, [1137](#)
StockQuoteService web service example, [1130](#)
WebSphere server, [1137](#)
WSDL Generation Tool, [1132](#)

Identification

services needed for a secure system, [881](#)

Identity and Policy block

Sun ONE (Sun Open Net Environment), [1142](#)

IDEs (Integrated Development Environments)

integrating J2EE with different IDEs
application servers, additional functionality, [1155](#)

IDL (Interface Definition Language)

compared to RMI-IIOP, [144](#)
interfaces in OTS model, [215](#)
J2SE API which J2EE platforms must support, [20](#)
supported remote object protocol, [33](#)

idlj compiler, [144](#)

inheritance, delegation and Tie models, [147](#)

IETF (Internet Engineering Task Force)

LDAP maintained by, [48](#)
TLS as encryption standard of, [126](#)

if tag

JSPTL conditional tag, [587](#)
displayDetails.jsp page, [615](#)

IIOP (Internet Inter-ORB Protocol)

see also **RMI-IIOP**.
EJB 2.0 specification requires, [717](#)
OTS distributed transaction model and, [215](#)
servers supporting both JRMP and, [142](#)

IllegalArgumentException, [435](#)

IMAP (Interactive Mail Access Protocol), [637](#)

implementation class *see* **bean class**.

implementation helper methods

manufacturing EJB example, [781](#), [788](#)

implementation-independent APIs

for accessing services, [20](#)

implementations

SOAP with XMLP (XML Protocol), [1115](#)
UDDI, [1120](#)

implicit objects, JSP

based on servlet API, [473](#)

import element

location attribute, [1117](#)
namespace attribute, [1117](#)
WSDL document, [1117](#)

inactive() method

java.rmi.activation.Activable class, [123](#)

INBOX keyword, [658](#)

include directive, JSP, [619](#)

contrasted with <jsp:include> standard action, [465](#)
example, [467](#)
example using both, [465](#)

example, [448](#), [449](#)

syntax and attributes, [447](#)

include() method

javax.servlet.RequestDispatcher() interface, [619](#)
RequestDispatcher interface, [369](#), [380](#)

includeaction.jsp

contrasting include action and include directive, [465](#)

included.jsp, [449](#)

 servlet code, [449](#)

included2.html page, [465](#)

included2.jsp, [466](#)

includeDirective1.jsp, [448](#)

includeDirective2.jsp, [449](#)

index property

 IteratorTagStatus object, [586](#)

index.html file

 chatroom example application, [404](#)

 default welcome file, [429](#)

 registration and authentication example, [591](#)

 RMI client applet, [133](#)

indirection

 level of, in security references, [887](#)

inetOrgPerson class, [67](#)

information assets

 increased value of, [11](#)

infrastructure services

 component of n-tier architecture, [16](#)

inheritance hierarchies

 EJB interfaces, [1105](#)

inheritance model

 idlj compiler, [147](#)

init() method

 ChatAdminServlet class, [356](#)

 ControllerServlet class, [495](#)

 FreakServlet class, [282](#)

 GenericServlet class, [266](#)

 overriding, [271](#)

 javax.servlet.Filter interface, [391](#), [392](#)

 MessageModeratorFilter, [398](#)

 Servlet interface, [265](#), [274](#)

initial context

 JNDI lookup and, [204](#), [699](#)

initialization

 FreakServlet example, [282](#)

initialize() method

 CreateMovieTables class, [180](#), [206](#)

Inprise application server, [242](#)

input controls, HTML

 Hidden controls in session tracking, [322](#)

input stream

 accessing with ServletRequest methods, [288](#)

<input> tags

 values returned for different control types, [289](#)

insertBatchData() method

 CreateMovieTables class, [198](#)

insertData() method

 CreateMovieTables class, [176](#), [180](#)

insertPreparedStatement() method

 CreateMovieTables class, [184](#)

insertRow() method

 java.sql.ResultSet interface, [198](#)

instance pooling

 JSP tag extensions, [554](#)

 servlet instances, to ensure single thread invocation,

[267](#)

 tag handler class, [515](#)

 tag handlers, [557](#)

instantiate() method

 java.beans.Bean class, [457](#)

- instantiation**
 FreakServlet example, 282
- integerInput tag**
 HTML tag library, registration example, 601
- integration platform**
 application servers, future prospects, 1157
- integration with enterprise systems**
 application servers, 1148
- Interaction interfaces, javax.resource.cci package**
 CCI (Common Client Interface) API, 1040-1041
- InteractionSpec interface, javax.resource.cci package, 1041**
 CCI (Common Client Interface) API, 1041
- interception process**
 filters, unlike servlets, can alter, 383
- Interface Definition Language** see IDL.
- interface design**
 EJBs, 1104
 role-specific interfaces, 1105
- interface objects, 734**
 advantages over simpler object models, 719
 aggregation during implementation, 721
 analysis model views of control objects and, 728
 analysis model views of UML use cases and, 727
 manufacturing EJB example, 729
 state management, 722
 stereotype icons, 725
 UML analysis objects, 718
- interfaces**
 application interfaces seen as contracts, 17
 business logic, 762
 making generic where possible, 17
 web service interface, 1129
- intermediate database access servers**
 Type 3 database drivers, 161
- internationalization**
 JSP code, 605
 registration and authentication example, 590
 tag library, 606
 thankyou.jsp page, 604
- internet**
 effects on value of information assets, 11
 protocols supported by J2EE, 32
- internet applications**
 connection pooling and, 207
 requirements of design process, 1060
- internet downtime**
 importance in business transactions, 12
- InternetAddress class, javax.mail.internet package, 656**
- interoperability**
 CORBA CSI, in CORBA EJB mapping, 896
 RMI-IIOP and CORBA, 139, 148
 SOAP and, 1115
 Value Object Patterns and, 1095
- interposition of services**
 containers, advantages of, 27
 EJB container services into component calls, 693
 EJB containers and, 696
 'this' references and, 213
- inter-servlet communication** see request dispatching.
- invalidate() method**
 HttpSession interface, 328, 332
- In-VM method calls**
 EJB communication, 897
 local interfaces, 897
- IP addresses**
 DNS returns, for FQDNs, 41
- iPlanet server, 42, 242**
- isAfterLast() method**
 java.sql.ResultSet interface, 193
- isBeforeFirst() method**
 java.sql.ResultSet interface, 193
- isCallerInRole() method**
 EJBContext interface, 886
 SessionContext interface, 745
- isCommitted() method**
 ServletResponse interface, 294
- isExpunged() method**
 javax.mail.Message class, 655
- isFirst() method**
 java.sql.ResultSet interface, 194
- isLast() method**
 java.sql.ResultSet interface, 194
- isMimeType() method**
 javax.mail.Part interface, 652
- isNew() method**
 HttpSession interface, 328
- isolated transactions**
 see also ACID properties.
 business logic independent of other activities, 863
 performance trade-offs, 877
- isolation levels**
 available through JDBC API, 879
 predefined by Connection interface, 878
 specifying programmatically, 877
- isOpen() method**
 javax.mail.Folder class, 662
- isRegistered() method**
 techSupportBean class, 481
- isSet() method**
 javax.mail.Message class, 655
- isUserInRole() method**
 HttpServletRequest interface, 422
- iteration**
 feedback and, in design processes, 1072
 iteration status tag
 jsptl-examples.war, 584
 iteration tags, JSPTL, 580
 extensibility, 586
 JSP tag extensions, 563
 JSPTL example, 572
- IterationTag interface, javax.servlet.jsp.tagext package**
 doAfterBody() method, 518, 540
 JSP 1.2 new feature, 510
- IteratorTag interface, javax.servlet.jsp.tagext package, 586**
- IteratorTagStatus interface, javax.servlet.jsp.tagext package, 586**
- IteratorTagSupport interface, javax.servlet.jsp.tagext package, 587**

J

J2EE

see also containers; J2EE implementations.

advantages for enterprise applications, 12

APIs, application servers, 1153

APIs, required to be supported, 19

characteristics as a design framework, 1060

choices favoring, 1069

common deployment configuration approach, 436

compared to J2SE and J2ME editions, 9

component technologies, 28

maturity model for web-based applications, 1093

container architecture, 23

deploying resource adapter as J2EE application, 1022

design considerations, 1059

J2EE (continued)

distributed application server environment, 18
document relating the technologies of J2EE, 689
enterprise applications, 1147
Microsoft .NET as alternative to, 10, 12
packaging and deployment, 1165
 class loading schemes, 1170
 configuring packages, 1174
 containers, 1166
 dependency libraries, 1183, 1188
 EAR files, 1166
 introduction, 1166
 roles and responsibilities, 1168
providers must implement both messaging domains, 970
reconciling long- and short-term demands, 18
SDK (Software Developer's Kit)
 J2EE implementation, 1150
 roles performed by, 1150
server-side application requirements met by, 237
SPI (Service Provider Interface) layer
 implementing future standards, 1156
suitability of javax.sql package for, 200
suitability for enterprise applications, 13
support for RMI-IIOP required, 149
web services, 1111, 1121
 JAX pack, 1121

J2EE 1.2 and 1.2.1
table comparing to J2EE 1.3, 1159

J2EE 1.3
table comparing to J2EE 1.2, 1159

J2EE 1.3 new features
EJB local interfaces, 22
filters, 383
Java Connector Architecture, 32
summarized, 1060

J2EE Blueprints, 689, 1060
MVC example, 216
source of J2EE patterns, 1074
variant names for J2EE patterns, 1087

J2EE CTS (Compatibility Test Suite)
J2EE implementation, 1150
supplements JCK, 1150

J2EE implementations, 1147
see also J2EE RI.
application servers, 1147
 additional functionality, 1154
 architecture, 1149
 bundling with operating systems, 1161
 competition among vendors, 1151
 component behavior, 1148
 deployment descriptors, 1154
 EJB containers, 1148
 future prospects, 1156
 integration with enterprise systems, 1148
 J2EE APIs, 1153
 network protocols, 1154
 open source products, 1152
 scalability, 1148
 SPIs (Service Provider Interfaces), 1153
 web containers, 1148
 WebLogic application server, 1149
 WebSphere application server, 1149
availability services, 1153
choosing implementation, 1147
database connection pooling, 1153
development communities, 1161
directory services, 1153
distributed object services, 1153
EJB 2.0 specification, 1150

evaluation parameters, 1158
 BEA, 1158
 IBM, 1158
 Oracle, 1158
implementing future standards, 1155
 Deployment APIs, 1156
 J2EE SPI, 1156
 JDBC RowSet, 1156
 JNLP (Java Network Launching Protocol), 1156
 Management APIs, 1156
 Security APIs, 1156
 SQLJ Part 0, 1156
 web services, 1155
 XML data binding APIs, 1156

J2EE CTS (Compatibility Test Suite), 1150
J2EE SDK, 1150
Java Community Process, 1150
JSP specification, 1150
minimum facilities required by specification, 1151
permitted extensions, 1151
presentation services, 1152
product provider role, 1169
security services, 1153
servlet engines, 1147, 1149
 open source products, 1152
 WebLogic Express, 1149
Servlet specification, 1150
transaction services, 1153

J2EE patterns
see also individual pattern names.
Business Delegate Pattern, 1082
Composite View Pattern, 1076
Data Access Object Pattern, 1082
Dispatcher View Pattern, 1084
Fast Lane Reader Pattern, 1087
Front Controller Pattern, 1075
Service Locator Pattern, 1079
Service to Worker Pattern, 1085
Session Façade Pattern, 1078
sources of further information, 1074-1075
tabulated with descriptions, 1074
Value List Handler Pattern, 1085
Value Object Pattern, 1080
View Helper Pattern, 1083

J2EE RI (Reference Implementation)
Cloudscape database ships with, 299
deploying the FreakServlet example, 278
deploying the TechSupportServlet example, 303
j2eeadmin command-line tool, 975
Java Pet Store Application, 1151
javax.sql package examples, 200
message-driven bean example, 999
requires variable-class sub-element in JSP, 535, 536
running the JSP tag extension example, 513
using as a web container, 243

j2eeadmin tool
J2EE RI provides, 972, 975
javax.jms.Connection.setClientID() method and, 986
use in publish/subscribe example, 984

J2ME (Java 2 Platform, Micro Edition), 9

J2SE (Java 2 Platform, Standard Edition), 9
includes JDBC-ODBC bridge support, 160
resource bundle class, 605
version 1.3 includes new idlj compiler, 144
version 1.4 will include JSSE, 127

JAAS (Java Authentication and Authorization Service)
configuring for security, 882
J2EE platforms must support, 20
service technology, 32

- JAF (JavaBeans Activation Framework),** 649
 J2EE platforms must support, 20
JAXM (Java API for XML Messaging), 1126
 required for using JavaMail, 639
Jakarta Struts, 504, 592
Jakarta Taglibs
 commercially available tag libraries, 569
Jakarta Tomcat *see* Tomcat servlet engine.
JAR (Java archive) files
 creating using the JDK tool, 701
 distributing tag libraries packaged into, 528
 packaging EJB and J2EE modules, 35
 packaging EJBs into, 701
 shipping with JavaMail API, 639
 StockQuoteService web service example, 1132
jar tool
 EAR (Enterprise Archive) files, creating, 1176
 packaging an application as a WAR file, 413
Java
 built-in networking support, 39
 component technologies chart, 688
 Java 2 platforms, editions and version status, 9
Java 1.2 release
 custom sockets as new feature, 124
java beans
 rowset objects as java bean components, 224
Java Community Process *see* JCP.
Java DataBase Connectivity *see* JDBC.
Java IDL *see* IDL.
Java modules
 as client classes, 35
Java Naming and Directory Interface *see* JNDI.
Java objects
 storing and retrieving in LDAP, 75
Java Pet Store Application, 1060, 1068, 1151
 tracking updates via Model Manager, 1092
Java references
 storing in LDAP, 76, 77
Java resource bundles
 locating, diagram, 606
Java SDK 1.4 release
 log handling expected, 618
java servlets *see* servlets.
Java WebStart, 599
java. packages
for methods, see under individual class and interfaces or method names.
java.beans package
 Bean class
java.io package
 EJB 2.0 specification prohibitions, 712, 713
 PrintWriter class, 254, 285
java.lang package
 Class class
 ClassLoader object, 167
 Double class
 Error class, 894
 Exception class, 889
 RuntimeException class, 894
 Thread class
java.lang.reflect package, 87
java.net package
 ServerSocket class, 123
 Socket class, 123, 124
java.rmi package
 AccessException class, 91
 AlreadyBoundException class, 91
 ConnectException class, 91
 ConnectIOException class, 91
 MarshalException class, 91
 Naming class
 methods, 89
 NoSuchObjectException class, 91
 NotBoundException class, 92
 PortableRemoteObject class, 140
 Remote interface, 93
 RemoteException class, 92, 894
 subclasses, 91
 RMISecurityManager class, 98, 105
 RMISocketFactory class, 124
 ServerError class, 92
 ServerException class, 92
 StubNotFoundException class, 92
 UnexpectedException class, 92
 UnknownException class, 92
 UnmarshalException class, 92
java.rmi.activation package
 Activatable class, 94, 116, 117
 specifying socket factories, 126
 ActivationDesc class, 115
 ActivationGroup class, 113
 ActivationGroupDesc class, 114
 Activator interface, 151
 RMI object activation model implementations, 113
java.rmi.dgc package
 leaseValue property, 100, 150
java.rmi.registry package
 LocateRegistry class, 90
 Registry interface, 88
 methods, 89
java.rmi.server package
 codebase property, 104, 150
 disableHttp property, 137, 150
 hostname property, 150
 logCalls property, 150
 randomIDs property, 150
 RemoteObject class, 93
 RemoteServer class, 94
 methods of, and of subclasses, 94
 RMIClassLoader class, 105
 RMIServerSocketFactory interface, 129
 RMISocketFactory interface
 UnicastRemoteObject class, 94
 specifying socket factories, 126
 Unreferenced interface, 100
 useCodebaseOnly property, 104, 150
 useLocalHostName property, 150
java.sql package
 Array interface, 164
 BatchUpdateException class, 165
 Blob interface, 164
 CallableStatement interface, 175
 invoking stored procedures with, 185
 CallableStatement interface, 163
 Clob interface, 164
 Connection interface, 163, 877, 1020
 full list of public methods, 170
 implementation of connection pooling, 213
 loading a database driver, 166
 connection management classes, 163
 data types, 164
 database access classes, 163
 database metadata classes, 165
 DatabaseMetadata interface, 165, 877
 DataTruncation class, 165
 date and time representation, 185

java.sql package (continued)

java.sql package (continued)

Date class, [164](#), 185
Driver interface, [163](#)
 methods, [170](#)
DriverManager class, [163](#)
 javax.sql.DataSource equivalent, 200
 limitations addressed by javax.sql, 199
 methods for managing database drivers, 168
DriverPropertyInfo class, [163](#)
exceptions and warnings, 165
functional groups of classes, [162](#)
ParameterMetadata interface, 165
PreparedStatement interface, [163](#), 175
 Tech Support servlet example, 302
Ref class, [164](#)
ResultSet interface
 concurrency types and updateable resultsets, 197
 methods for retrieving data, 181
 methods for scrolling, [194](#)
 methods related to cursor position, 193
 three resultset types, 192
ResultSet interface, [163](#)
ResultSetMetaData interface, 165
 methods, [182](#)
role in JDBC [3.0](#), [158](#)
Savepoint interface, [190](#)
SQLException class, 165
SQLWarning class, 165, [431](#)
Statement interface, [163](#)
 full list of methods, [174](#)
Struct interface, [164](#)
Time class, [164](#), 185
Timestamp class, [164](#), 185
Types class, [164](#)
 specifies JDBC SQL data types, [186](#)

java.util package

- ArrayList object
 - shopping cart example application, 343
- Date class, 185, 977, 979
- GregorianCalendar object, 253
- HashMap class
 - chatroom example application, [351](#)
- Locale class
- ResourceBundle class, 605
- Stack class
 - ChatRoom class extends, 352
- Vector class, 285, 651

JavaBeans

- EJBs compared to, 687
- JavaBean proxies as MVC controller components, 716
- JSP tag handlers as, 508, 514
- rowset creation follows the JavaBeans model, [226](#)
- separating code from JSP pages, [463](#)
- setting properties with <jsp:setProperty>, 458

JavaIDL see **IDL**.

JavaMail, [635](#)

- block structure of messages, 648
- compared to JMS, [32](#)
- environment properties, 641
- examples, 667
- installation and configuration, 639
- J2EE platforms must support, [20](#)
- protocols bundled with, 636
- reading mail, [672](#)
- sending mail, 667
- version [1.2](#) as latest release, 639
- version [1.2](#) POP3 support added, 636

JavaMail API

- DataHandlers for MIME types, 650

javax.mail.Address class, 655
javax.mail.Authenticator class, 642
javax.mail.Folder class, 661
javax.mail.internet.MimeMessage class, 645
javax.mail.Message class, 644
javax.mail.Part interface, 648
javax.mail.Session class, 641
javax.mail.Store class, 657
javax.mail.Transport class, [666](#)
overview, 638
resources, 683
role in e-mail connectivity, 635
serialization support limited, 678

javamail_pop class, [672](#)

- extending to allow deleting messages, 676
- extending to allow saving attachments, 677

javamail_save class, [679](#)

javamail_send class, [667](#)

javamail_send_attachment class, [669](#)

javamail_sendAll class, [680](#)

java-rmi.cgi script, [137](#)

Java-to-IDL mapping specification

- CORBA [2.3](#) enhancement for RMI-IIOP interoperability, [140](#)

javax packages

- for methods, see under individual class and interfaces or method names.*
- part of J2EE SDK, 252

javax.activation package

- DataHandler class
 - methods, 649

javax.ejb package

- CreateException class, 892
- DuplicateKeyException class, 892
- EJBCancel interface, 886
- EJBHome interface
 - home interfaces must extend, 703
- EJBLocalObject interface
 - local interfaces must extend, [702](#)
- EJBObject interface
 - remote interfaces must extend, 701
- EntityBean interface
 - bean class for entity beans derived from, 704
- EntityContext interface, 714
- FinderException class, 892
- MessageDrivenBean class
 - bean class for message-driven beans derived from, 704
- ObjectNotFoundException class, 892
- RemoveException class, 892
- SessionBean interface, 734
 - bean class for session beans derived from, 704
- SessionContext interface, 744-745

javax.jms package

- BytesMessage interface, 989
- Connection interface, 989
- ConnectionFactory interface, 990
- ConnectionMetaData interface, 990
- DeliveryMode interface, 990
- Destination interface, 990
- ExceptionListener interface, 990
 - interfaces, 989
- MapMessage interface, 990
- Message interface, 990
- MessageConsumer interface, 990
- MessageDrivenBean interface, 996
- MessageListener interface, 990

javax.jms package (continued)

- message-driven beans must use, 996
- MessageProducer interface**, 991
- ObjectMessage interface**, 991
- Queue interface**, 991
- QueueBrowser interface**, 991
- QueueConnection interface**, 991
 - JMS architecture component, 972
- QueueConnectionFactory**
 - JNDI lookup, 974
- QueueConnectionFactory interface**, 991
- QueueReceiver interface**, 991
- QueueSender interface**, 991
- QueueSession interface**, 992
- Session interface**, 992
- StreamMessage interface**, 992
- TemporaryQueue interface**, 992
- TemporaryTopic interface**, 992
- TextMessage interface**, 992
- Topic interface**, 992
- TopicConnection interface**, 992
 - JMS architecture component, 972
- TopicConnectionFactory interface**, 992
- TopicPublisher interface**, 992
- TopicSession interface**, 992
- TopicSubscriber interface**, 993

javax.lang package

- StringBuffer class**
 - reverse() method, 544

javax.mail package

- Address class**, 655
- Authenticator class**, 642
- BodyPart class**, 648, 652
- FetchProfile class**, 663
 - field groupings, 664
- Flags class**, 654
- Folder class**, 661
- Message class**, 644
 - checking message flag status, 655
- Multipart class**, 648, 650
- Part interface**, 648
 - implemented by Message class, 644
- PasswordAuthentication class**
- Session class**, 641
- Store class**, 657
- Transport class**, 666
- URLName class**, 660

javax.mail.internet package

- InternetAddress class**, 656
- MimeMessage class**, 645
- NewsAddress class**, 656

javax.mail.search package, 665

- ComparisonTerm class**, 665
- SearchTerm class**, 665

javax.naming package

- Context interface**
 - service providers must implement, 49
- NamingEnumeration interface**, 61

javax.naming.directory package

- Attribute interface**
- BasicAttributes class**, 67, 72
- DirContext interface**
 - needed to add entries to an LDAP server, 66
- ModificationItem class**, 72, 73
- SearchResult class**

javax.net.ssl package

- trustStore property**, 132

javax.resource.cci package

- Connection interface**, 1041
- ConnectionFactory interface**, 1040
- ConnectionSpec interface**, 1041
- Interaction interface**, 1041
- InteractionSpec interface**, 1041
- LocalTransaction interface**, 1041
- Record interface**, 1041
- RecordFactory interface**, 1041

javax.resource.spi package

- ManagedConnection interface**, 1044
- ManagedConnectionFactory interface**, 1019

javax.rmi package

- PortableRemoteObject class**

javax.security package

- Principal interface**, 886

javax.servlet package

- class diagram**, 260
- exception classes**, 272
- Filter interface**, 387, 391
- FilterChain interface**, 393
- FilterConfig interface**, 392
- GenericServlet class**, 266
- GreetingServlet imports**, 252
- RequestDispatcher interface**, 368
- RequestDispatcher() interface**
- Servlet Context interface**
- Servlet interface**, 264
- ServletConfig interface**, 270
- ServletContextAttributesListener interface**, 349
- ServletContextEvent class**
- ServletContextListener interface**, 557, 558
- ServletException class**, 272, 431
- ServletRequest interface**, 286
- ServletRequestWrapper class**, 289
- ServletResponse interface**, 292
- ServletResponseWrapper class**, 294
- UnavailableException class**, 272

javax.servlet.http package

- class diagram**, 260
- Cookie class**, 264, 324
- GreetingServlet imports**, 252
- HttpServlet class**, 268
- HttpServletRequest interface**, 286, 289, 325
- HttpServletRequestWrapper class**, 292, 410
- HttpServletResponse interface**, 286, 325
- HttpServletResponseWrapper class**, 296
- HttpSession interface**, 326
- HttpSessionActivationListener interface**, 338
- HttpSessionAttributesListener interface**, 340
- HttpSessionBindingEvent class**, 325
- HttpSessionBindingListener interface**, 325
- HttpSessionContextListener interface**, 557
- HttpSessionListener interface**, 561
- HttpUtils class**, 264

javax.servlet.jsp package

- HttpJspClass**
 - JSP classes must extend, 24
- JspPage interface**, implemented by JSP-derived servlets, 443
- JSPTagException class**, 523
- PageContext class**
 - objects available to tag handlers, 522

javax.servlet.jsp.tagext package

- BodyContent class**, 521
- BodyTag interface**, 518

[javax.servlet.jsp.tagext package \(continued\)](#)

[javax.servlet.jsp.tagext package \(continued\)](#)

BodyTagSupport class, 521
convenience classes, 521
interfaces implemented by tag handlers, 514
IterationTag interface
Tag interface, 515
tag handler classes must implement, 524
TagData class, 551
TagExtraInfo class, 538
TagLibraryValidator class
TagSupport class, 521
TryCatchFinally interface, 551
VariableInfo class, 539

[javax.servlet.jsp.tagext package](#)

IteratorTag interface, [586](#)
IteratorTagStatus interface, [586](#)
IteratorTagSupport interface, 587

[javax.sql package](#)

classes and interfaces for connection pooling, [211](#)
compared with java.sql, 199
connection pooling in, [210](#)
ConnectionEvent class, 212
ConnectionEventListener interface
ConnectionPooledDataSource interface
DataSource interface, 201, [1019](#), 1027
equivalent to java.sql.DriverManager, 200
implementation of connection pooling, 213
responsibilities under the API, 203
few classes for developers, 200
JNDI database lookup, 199
PooledConnection interface
role in JDBC [3.0](#), [158](#)
RowSet interface
no interfaces specified, [226](#)
properties and events, 225
RowSetListener interface
rowsets a feature of, 223
XAConnection interface
XADatasource interface, 221

[javax.transaction package](#)

UserTransaction interface, 220, [875](#), 997
programmatic transactions only, 220

[javax.transaction.xa package](#)

XAResource interface, 220

[JAX pack \(Java APIs for XML\), 1121](#)

JAXB (Java Architecture for XML Binding), 1122
JAXM (Java API for XML Messaging), 1125
JAXP (Java API for XML Processing), 1122
JAXR (Java API for XML Registries), 1127
JAX-RPC (Java API for XML-based RPC), 1126
web services, 1121

[JAXB \(Java Architecture for XML Binding\), 1100, 1122](#)

data binding, XML, 1123
design goals, 1123
XML Schemas, 1123
mapping schema to Java classes, 1124

[JAXM \(Java API for XML Messaging\), 1101, 1125](#)

JAF (JavaBeans Activation Framework), 1126
messaging profiles, 1126
SOAP (Simple Object Access Protocol), 1125

[JAXP \(Java API for XML Processing\), 1122](#)

DOM (Document Object Model), 1122
J2EE platforms must support, [20](#)
SAX (Simple API for XML), 1122
TrAX (Transformation API for XML), 1122
UsersToXMLConverter class uses JAXP [1.1](#), 628
XSLT (Extensible Stylesheet Language Transformations), 1122

[JAXR \(Java API for XML Registries\), 1127](#)

architecture, 1127
design goals, 1128
locating web services, 1129
JAX-RPC (Java API for XML-based RPC), 1126
mapping RPC calls to Java classes, 1126

JBoss

open source application servers, [1152](#)

[JCA \(Java Connector Architecture\), 1009](#)

benefits, 1055
B2B (Business-to-Business) integration, 1056
EAI (Enterprise Application Integration), 1055
web-enabled enterprise portals, 1055
CCI (Common Client Interface) API, 1011, [1012](#), 1039
CCI Black Box Adapter, 1045
connection interfaces, 1040
connection with managed-application, 1042
connection with non-managed application, 1043
data representation interfaces, 1040
exception and warning classes, 1040
interaction interfaces, 1040
metadata interfaces, 1040
compared to JDBC, 1013
EIS (Enterprise Information Systems) and, 1010
introduction, 1009
J2EE platforms must support, [20](#)
limitations, 1056
managed environment, 1012
non-managed environment, 1012
packaging and deployment interfaces, 1012
Resource Adapter, 1016

[Resource Adapter, 1011](#)

Black Box Resource Adapters, 1023
DemoConnector Adapter example, 1023
contracts, 1014
deployment descriptors, 1019
deployment options, 1022
packaging and deployment interfaces, 1016
service technology for legacy access, [32](#)
should simplify database connectivity, [31](#)
system-level contracts, 1011
system-level services, 1011

[JCK \(Java Conformance Kit\)](#)

supplemented by J2EE CTS, 1150

[JCP \(Java Community Process\)](#)

involved in JMS development, 969
J2EE implementation, 1150
JSPTL developed through, 576

[JDBC, 157](#)

API, isolation levels available through, 879
availability of database drivers for, [162](#)
compared to JCA (Java Connector Architecture), 1013
configuring drivers
EJB system administrator role, 937
constraints in technical support application, 302
creating and executing SQL statements, [174](#)
data types, mapping to Java and SQL, 186
database insertion using, 398
drivers, support for XA interface, 871, 881
establishing a connection to a database, [170](#)
interface for isolation performance tradeoffs, 877
J2SE API which J2EE platforms must support, [20](#)
java.sql package model
unsuitable for distributed applications, 199
JavaMail API similarity, 635, 638
JNDI similarity, [45](#), [47](#)
movie catalog example, 175
no interfaces specified for javax.sql.RowSet, [226](#)
prepared statements, 183

- JDBC (continued)**
- providing database connectivity, 31
 - URLs, identifying database drivers, 166
 - version 2.1**
 - introduced batch updates, 198
 - introduced scrollable resultsets, 180, 191
 - version 3.0 API, 157**
 - version 3.0, new features**
 - save points, 158, 190
 - JDBC 2.0 Extension** *see also* javax.sql package.
 - XA interface for distributed transactions, 871
- JDBC 2.0 Optional Package**
- J2EE platforms must support, 20
- JDBC RowSet**
- implementing future standards, 1156
- JDBCHelper class, 481**
- JDBC-ODBC bridge, 159**
- JDK (Java Development Kit)**
- DNS service provider in JDK 1.4, 78
 - J2SE commonly described as, 9
 - JNDI ships with different versions of, 49
 - policy-based access control from version 1.3 new features, 98
- Jini, 155**
- JMS (Java Message Service)**
- architecture components diagram, 973
 - architecture components listed, 972
 - asynchronous communication framework, 969
 - asynchronous communication using purchasing system example, 1100
 - commercial implementations, 967, 994
 - compared to JavaMail, 32
 - component parts of a message, 973
 - distributed event handling and, 1106
 - history and status, 969
 - interfaces of the javax.jms package, 989
 - J2EE platforms must support, 20
 - legacy messaging system accessible via SpiritWave, 995
 - message-driven beans and, 30, 995
 - point-to-point messaging, 970
 - questions to ask when choosing an implementation, 994
 - RMI alternative, 83
 - service technology for asynchronous communication, 32
 - SMTP e-mail bridge, 994
 - using transactions with, 993
- JMSTimestamp property, 978**
- JMX (Java Management Extensions)**
- application and network management
 - EJB system administrator role, 938
- JNDI (Java Naming and Directory Interface), 39, 44**
- see also* service providers.
 - access to remote services, 25
 - access to services from within containers, 25
 - adding entries to LDAP servers, 66
 - connecting to LDAP servers, 54
 - context
 - distinguished from java.rmi.Naming class, 88
 - initial context and EJB authentication, 882
 - initial context and JNDI lookup, 699
 - J2SE API which J2EE platforms must support, 21
 - LDAP functions not all supported by, 47
 - lookup
 - based on JNDI, in javax.sql package, 199
 - distributed transactions, 221
 - lookup centralized, for DataSource objects, 201
- mapping beans and references into JNDI namespace
- EJBs banking example, 933
 - namespace population by containers, 27
 - Netscape Directory SDK for Java as lower-level alternative, 47
 - RMI, locating remote objects, 88
 - role distinguished from that of LDAP, 42
 - security and access control, 204
 - software requirements and installation, 48
 - tutorial, 67
 - twofold role within J2EE, 31
 - use with LDAP, 46
 - using with LDAP
 - efficiency issues, 66
 - using without LDAP, 48, 77
 - version 1.2 supports federation, 49
- JNDIAdd class, 69**
- JNDIDel class, 74**
- JNDIMod class, 72**
- JNDISearch class, 59**
- JNDISearchAttrs class, 64**
- JNDISearchAuth class, 62**
- JNDISearchDNS class, 78**
- JNLP (Java Network Launching Protocol)**
- implementing future standards, 1156
- JonAS**
- open source application servers, 1152
- Jr tag library**
- accepts rtxprvalues attribute values, 571
 - displayAllRegisteredUsers.jsp page, 625
- JRMP (Java Remote Method Protocol)**
- development with, compared to RMI-IIOP, 140
 - servers supporting both IIOP and, 142
 - used with TCP/IP by RMI before SDK 1.3, 87
 - version of RMI for distributed computing, 895
- JRun Application Server**
- JRun 3.0** JSP container
 - compilation strategy for JSP static includes, 467
 - JRun Enterprise Server**
 - commercially available tag libraries, 569
 - JRun web container, 242**
 - translation units, 441
- jsessionid**
- cookie, set by web container, 323
 - path parameter, URL rewriting technique, 322
- JSP (JavaServer Pages)**
- authoring tools, 569
 - banners easier to generate than with servlets, 487
 - classes must extend javax.servlet.jsp.HttpJspClass, 24
 - compared with ASP, 240
 - debugging difficulties, 504
 - definitions and mappings in deployment descriptors, 241
 - design strategies, 488, 489
 - Front Controller Pattern, 491
 - generic controller frameworks, 504
 - error handling, 551
 - examples
 - applet.jsp, 471
 - beans.jsp page, 461
 - declaration.jsp, 451
 - enterName.jsp, 500
 - error.jsp, 487
 - expression.jsp, 454
 - forward.jsp, 469
 - includeaction.jsp, 465
 - included.jsp, 449

JSP (JavaServer Pages) (continued)

examples (continued)
 included2.jsp, 466
 includeDirective1.jsp, 448
 includeDirective2.jsp, 449
 pageDirective.jsp, 447
 regform.jsp, 485
 register.jsp, 485
 resonse.jsp and banner.jsp, 486
 scriptlet.jsp, 452
 showInfo.jsp, 500
 simpleJSP.jsp, 441
 sorryNotFound.jsp, 501
 techSupport.jsp, 481
 XML-based syntax, 477
implementing MVC components using, 715
implicit objects, 473
incorporating server-side includes, 620
intelligibility and tag extensions, 563
internationalization, 605
introduced, 439
J2EE platforms must support, 20
J2EE web component technology, 29
lifecycle events, 443
looping solutions, 543
manipulating as XML, 475
page structure, 444
possible use in chatroom example application, 352
purchasing system J2EE design example, 1089
servlets compared with, 29, 239, 440
 strengths and weaknesses of each, 489
standard actions, 455
suitability for HTML generation, 600, 1088
technical support application converted to use, 477
understandable without java skills, 441
View Helper Patterns and, 1083
viewed as web components, 237
web application, limitations, 488
web interface, creating
 manufacturing EJB example, 940-960
XML-based syntax for, 475
 example, 477
 scriptlets need escaping as CDATA, 477

JSP 1.1
scripting variables required programmatic definition, 538

JSP 1.2
required for using JSPTL, 571

JSP 1.2 new features, 440
application lifecycle events, 557
error handling improvements, 551
includes allowed without flushing buffers, 464
javax.servlet.jsp.tagext.IterationTag interface, 518
second method for introducing scripting variables, 534
tag extension enhancements, 509
validation improvements, 550

JSP containers, 442

JSP lifecycle
managed by web containers, 237

JSP Model 1 designs, 489

JSP Model 2 designs, 489

JSP specification
J2EE implementation, 1150

JSP tag extensions, 507
attributes must be in quotation marks, 526
calling tags in a library, 526
compared to JavaBeans, 509
distributing third-party packaged into JARs, 528

enhancements in JSP 1.2, 509

freeing resources, 517
implementation requirements, 514
intelligibility of JSP pages and, 563
introducing
 scripting variables, 534
must be explicitly imported, 526
nesting, 546
parametrizing with XML attributes, 529
possible uses reviewed, 562
tag prefixes like XML namespaces, 526
typical uses, 509
using in JSP pages, 526
validation, 550
writing tag extensions, 529

JSP views

in Front Controller implementation, 494, 500

<jsp:fallback> tag
support tag for <jsp:plugin>, 470

<jsp:forward> standard action
use, syntax and example, 468

<jsp:getProperty> standard action
example using, 460
JSP page-view with bean architectures, 491
use, syntax and attributes, 460

<jsp:include> standard action
contrasted with include Directive, JSP, 465, 619
 example, 465, 467
use, syntax and attributes, 463

<jsp:param> standard action
use, syntax and attributes, 463

<jsp:params> tag
support tag for <jsp:plugin>, 470

<jsp:plugin> standard action
use, syntax and attributes, 470

<jsp:setProperty> standard action
example using, 460
use in techSupport.jsp, 481
use, syntax and attributes, 458

<jsp:useBean> standard action
example using, 460
Front Controller architecture example, 498, 499
registration and authentication application, 598, 615
syntax and attributes, 456

jspDestroy() method
javax.servlet.jsp.JspPage interface, 443

jspInit() method
javax.servlet.jsp.JspPage interface, 443

JspPage interface, javax.servlet.jsp package, implemented by
JSP-derived servlets, 443

jspTagApps.war application, 631

JSPTagException class, javax.servlet.jsp package, 523

JspTagExceptions, 551, 557

JSPTL (Standard Tag Library), 569
basic tags, 577
benefits and status, 570
conditional tags, 587
Early Access 1.1 version, contents, 570
future provision of generic custom tags, 631
introduced, 568
iteration tags, 580
likely future developments, 570
two versions, 571

JSPTL with expression language support see jx tag library.

JSPTL with request time expression values see jr tag library.

jspt-examples.war
conditional tags, 588

jsptl-examples.war (continued)
 iteration status tag, 584
 Mutually Exclusive Conditional Execution tag, 589
 sample web application, 581
JSPTLEExamplesInit class, 573
Jsp-version
 <taglib> sub-elements in TLDs, 524
JSPWriter, 519
JSSE (Java Secure Sockets Extension), 126
 installing the JSSE provider, 131
JTA (Java Transaction API)
 distributed transactions, role in, 213
 example of a service-based architecture, 1063
 J2EE platforms must support, 20
 javax.transaction package interfaces and, 220
 not aimed at application programmers, 692
 service technology for distributed transactions, 31
JTS (Java Transaction Service)
 java mapping of OTS, 215
 JMS transactions used with, 989
 not aimed at application programmers, 692
 requirements for applets as clients, 132
 service technology for distributed transactions, 31
 transacted JMS sessions and, 993
JVM (Java Virtual Machines)
 classloading, 1170
 objects with the same ActivationGroupID are activated
 on the same JVM, 120
 objects with the same ActivationGroupID are activated
 on the same JVM, 113
 RMI allows object-to-object communication between,
 83
 starting multiple, 120
 version 1.3 plug-ins, 134
jx tag library
 accepts exprvalues attribute values, 571
 displayAllRegisteredUsers.jsp page, 625
 iteration example, 572

K

Kerberos
 one mechanism used by SASL, 56
keyboard input
 EJB 2.0 specification prohibitions, 712
keytool utility, 127
keywords
 display keyword, 675
 INBOX keyword, reserved in POP3, 658

L

LanguageBean class, 461
language-neutrality
 objects stored in LDAP, 76
last property
 IteratorTagStatus object, 586
last() method
 java.sql.ResultSet interface, 194
layered architectures
 distinguished from tiered, 1061
 RMI, 85
layers of indirection, 160
lazy activation, RMI, 111, 121

lazy loading
 abstract accessors, CMP 2.0, 802
 views shouldn't access lazily loaded data, 500
LDAP (Lightweight Directory Access Protocol), 42
 can't store EJBs or DNS entries, 77
 compared to DNS, 78
 functions not all supported by JNDI, 47
 IETF maintains as open standard, 48
 role distinguished from that of JNDI, 42
 schemas, LDAP servers, 44
 service provider for in JNDI SDK, 50
 sources of further information, 75
 standard operations, 54
 storing and retrieving Java objects, 75
 use with JNDI, 46
 efficiency issues, 66
 uses in Java applications, 50
 using without JNDI, 47
 version 3.0 security support, 55
 authentication, compared with version 2, 57
LDAP servers
 adding entries using JNDI, 66
 connecting to, using JNDI, 54
 deleting entries using JNDI, 74
 modifying entries using JNDI, 72
 search operation, 57
LDAP_SCOPE_BASE
 illustration of a search using, 59
LDAP_SCOPE_ONELEVEL
 illustration of a search using, 58
LDAP_SCOPE_SUBTREE
 illustration of a search using, 58
LDIF (LDAP Data Interchange Format), 43
 used for examples, 48
leaseValue property, java.rmi.dgc package, 100, 150
leasing, RMI, 100
 lease value property, 103, 150

legacy systems
 accessible with JCA, 32
 enterprise architecture and, 17
 integrating into enterprise applications, 12
 messaging, SpiritWave drivers for, 995
 need to adapt new technologies to work with, 11

lifecycle and framework methods
 manufacturing EJB example, 783-784
 Order EJB, 842
 Product EJB, 847
lifecycle stages see also *servlet lifecycle*.
 applets, 24
 components, 24
 entity beans, 828
 filters, 384, 391
 JSP pages, 237, 443
 message-driven beans, 997
 sessions, 328, 330, 337

list iteration
 JSP pages, 543
list tag
 HTML tag library, registration example, 601, 603
list() method
 javax.mail.Folder class, 659
 java.rmi.Naming class, 89
<listener> element
 <taglib> sub-elements in TLDs, 524
 TLD (Tag Library Descriptors), 558
listening sockets, 123
 monitoring with netstat, 124

listing behavior
Enterprise JavaBeans, 762
ListRoomsServlet
chatroom example application, 357
ListRoomsServlet class
doGet() method, 359
listSubscribed() method
javax.mail.Folder class, 660
load balancing
application servers, additional functionality, 1155
distributable containers, 28
distributed web containers, 435
JMS queue consumption as, 971
loadClass() method
java.rmi.server.RMIClassLoader class, 105
<load-on-startup> element
deployment descriptors, 422
local interfaces
accessible to EJBs in same deployment unit, 698
advantages, 803
CMP 2.0, 803
creating, 837
EAR class loading schemes, 1173
EJB 2.0 introduced, 685, 1173
extend EJBLocalObject interface, 702
In-VM method calls, 897
Product EJB, 845
relationships, entity beans, 837
local transactions
DemoConnector Adapter example, 1024
transaction management, 1015
locale
changing on servers, 609
Locale class, java.util package
getDefault() method, 605
LocalTransaction interface, javax.resource.cci package
CCI (Common Client Interface) API, 1041
LocateRegistry class, java.rmi.registry package, 90
createRegistry() method, 127
location attribute
import element, WSDL, 1117
<location> tag
deployment descriptors, 432
locking strategies
optimistic and pessimistic, 880
log tag
foundation tags, used in displayDetails.jsp page, 616
log() method
GenericServlet class, 266
ServletContext interface, 348
logCalls property, java.rmi.server package, 150
logging
methods in DriverManager class, 170
system exceptions, 893
logging requests
possible use of filters, 385
logical names
assigning to data sources, 204
login forms
example using <jsp:forward> standard action, 468
security in web applications, 419
login servlets
preventing direct servlet access, 420
login.html page, 424
<login-config> tag
deployment descriptors, 423
loginForm.jsp page, 611
loginServlet, 420
long transactions
managing, 879
lookup() method
java.rmi.Naming class, 89
java.rmi.registry.Registry interface, 89
loopback calls
EJB 2.0 specification limitations, 714
looping
JSP pages, 543

M

mail protocols
bundled with JavaMail, 636
mail servers
illustrating application logic in request-response process, 238
IMAP, responsible for e-mail storage, 637
storing SMTP server name in message headers, 680
supporting IMAP, 638
mail storage and retrieval
JavaMail API, 639
mail.debug property, 642
mail.from property, 642
mail.host property, 642
mail.protocol.host property, 642
mail.protocol.user property, 642
mail.store.protocol property, 642
mail.transport.protocol property, 642
mail.user property, 642
main() method
CreateMovieTables class, 180
maintenance
problems with two-tiered systems, 14
manageability
EJB container services and, 693
managed environment
CCI (Common Client Interface) API
connection with managed-application, 1042
JCA (Java Connector Architecture), 1012
provided by J2EE containers, 13
ManagedConnection interface, javax.resources.spi package, 1044
ManagedConnectionFactory interface, javax.resources.spi package, 1019
getConnection() method, 1044
Resource Adapter, 1019
Management APIs
implementing future standards, 1156
ManageOrders EJB
business methods, 775
home interfaces, 764
implementation helper methods, 781
lifecycle and framework methods, 783
manufacturing EJB example, 763, 775
remote interfaces, 763
ManageSampleOrders client
manufacturing EJB example, 773
Mandatory transactional attribute, 872
manufacturing EJB example
BeginManufacture client, 768
business logic, 762

manufacturing EJB example (continued)

clients, 766
CompleteManufacture client, 770
ContextPropertiesFactory class, 767
CreateProducts client, 774
deployment descriptors, 848
entity beans, 838
implementing, 760, 838
ManageOrders EJB, 763, 775
business methods, 775
implementation helper methods, 781
lifecycle and framework methods, 783
ManageSampleOrders client, 773
Manufacture EJB, 765, 783
business methods, 785
home interfaces, 766
implementation helper methods, 788
lifecycle and framework methods, 784
remote interfaces, 765
Order EJB, 839
business methods, 840
finder methods, 839
lifecycle and framework methods, 842
primary keys, 843
PlaceSampleOrders client, 771
primary keys, 843
Product EJB, 844
business methods, 846
lifecycle and framework methods, 847
requirements, 722
running application, 855
session beans, 760
stateful session beans, 783
stateless session beans, 775
use case steps, 724
web interface, creating, 939
application.xml J2EE file, adding web archive to, 963
constants used, class source code, 945
creating product view, 951
manufacture routing steps view, 959
model and request processor, class source code, 941
model manager, class source code, 946
MVC design, implementing, 940
open order view, 954
order manufacture choice view, 957
order placing view, 953
overdue order view, 955
product routing steps view, 952
shipping info view, 960
thank-you message view, 954
user login view, 958
view container, class source code, 949
web archive, deployment descriptor, 961
web archive, directory structure, 962
wellcome view, 960

many-to-many relationship, CMP 2.0, 804**MapMessage interface, javax.jms package, 990****mappings**

actual onto logical security identity, 882
default mapping, TLDs, 528
CORBA to EJB, 896
EJB to CORBA, 149
HTTP requests to servlets, 414
JSP tag extension attributes
uses reflection, 529
JSP tag library TLD files, 513
object/relational mapping and design decisions, 721
requests to servlets
advantages, 418
conflicts, 418
SQL data types to Java, 186

marker interfaces

SingleThreadModel, 267

MarshalException class, java.rmi package, 91

marshaling and unmarshaling, 85

matching rules

LDAP attributes, 44

maturity model

J2EE web applications, 1093

MDS (Message Digest)

one mechanism used by SASL, 56

password authentication, 51

MDB see **message-driven beans**.

memory overhead

involved in using JNDI, 47

use with LDAP, 66

message body, JMS

sub-types, 973

Message class, javax.mail package, 644

checking message flag status, 655

message consumers

JMS architecture component, 972

JMS point-to-point messages have single consumers, 970

JMS publish/subscribe message can have multiple consumers, 971

message element

WSDL document, 1118

Message ID field

e-mail message headers, 647

Message interface, javax.jms package, 990**message logging**

chatroom example application, 395, 396

message manipulation

JavaMail API, 639

message moderation

chatroom example application, 395, 398

message numbers, 661**message producers**

JMS architecture component, 972

message selectors, 989

JMS pub/sub example, 984

message status flags, 654**message tag**

internationalization tag library, 607

message transportation

JavaMail API, 639

messageArg tag

internationalization tag library, 608

message-broker middleware, 968**MessageConsumer interface, javax.jms package, 990**

receive() method, 978

message-driven beans, 30, 995

bean class derived from javax.ejb.MessageDrivenBean, 704

deploying, 998, 1003

distinction from other EJB varieties, 689

EJB 2.0 new feature, 685

example using, 998

lifecycle, 997

restrictions on use, 996

transaction management, 997

use as adapters

Service Activator Pattern discussion, 1102

MessageDrivenBean class, javax.ejb package

bean class for message-driven beans derived from, 704

MessageDrivenBean interface, javax.jms package, 996

ejbRemove() method, 996

setMessageDrivenContext() method, 996

- MessageDrivenApressBean class, 998**
- MessageListener interface, javax.jms package, 990**
 - message-driven beans must use, 996
 - onMessage() method, 981
- MessageLogFilter, 396**
- MessageModeratorFilter, 398, 401**
 - deployment descriptor, 403
- MessageProducer interface, javax.jms package, 991**
- messages**
 - see also e-mail messages; JMS; SOAP messages.*
 - history of messaging, 968
 - two messaging system domains, 970
- messaging profiles**
 - JAXM (Java API for XML Messaging), 1126
- messaging systems** *see JMS.*
- metadata interfaces**
 - CCI (Common Client Interface) API, 1040
- META-INF subdirectory, 413**
 - case sensitivity of name, 701
 - holds EJB deployment descriptor, 701
- method permissions**
 - banking example, 885
 - defined in deployment descriptor, 884
 - information type, EJB security, 883
 - programmatic access control alternative, 886
- method signatures**
 - detecting incorrect signatures, 1104
- Microsoft Corporation**
 - integrating .NET with Java classes
 - application servers, additional functionality, 1155
 - .NET and COM as alternative to EJBs, 687
 - .NET as alternative to J2EE, 10, 12
 - .NET similarity to J2EE, 1060
 - SpiritWave JMS adapters for MS COM applications, 995
- Microsoft Transaction Server**
 - supports declarative demarcation, 216
- MIME (Multipurpose Internet Mail Extensions), 638**
 - introduced, 236
 - JavaMail API DataHandlers for MIME types, 650
 - mapping in deployment descriptors, 428
 - role in e-mail content interpretation, 635
 - ServletResponse interface methods for setting MIME type, 293
 - setting, for Greeting Servlet example, 254
 - use for JavaMail multipart messages, 649
- MimeMessage class, javax.mail.internet package, 645**
- mnemonics**
 - LDAP attributes use, 43
- model components. MVC**
 - implementing using EJBs, 715
- Model Manager**
 - use in Java Pet Store example application, 1092
- modeling requirements**
 - chatroom example application, 350
- modeling tools**
 - complexity of UML models favors use, 728
- moderated_words.txt file, 404**
- ModeratedRequest class, 400**
- ModificationItem class, javax.naming.directory package, 72, 73**
- modifyAttributes() method**
 - javax.naming.directory.DirContext interface, 73
- modifying entries**
 - dangers of REPLACE, 72
 - LDAP server, using JNDI, 72
- Modula-3, 100**
- <module> tag**
 - deployment descriptors, 1177
- modules**
 - compared to containers, 1166
 - deployment descriptors, 1177
 - EJB, makeup and packaging, 35
 - enterprise applications package structure, 1175
 - Java modules as client classes, 35
 - packaging from application components, 33
 - packaging into applications, 34, 35, 1166
 - ordering modules, 1182
 - use of JAR and WAR files, 35
 - web modules, makeup and packaging, 35
- MoleculeViewer applet, 471**
- MOM (Message-Oriented Middleware), 968**
- moveToInsertRow() method**
 - java.sql.ResultSet interface, 198
- movie catalog example**
 - batch updating, 198
 - JDBC, 175
 - modified to use datasources, 205
- MsgServer class, 101**
- Multipart class, javax.mail package, 648, 650**
 - addBodyPart() method, 671
- multiple inheritance**
 - EJB interfaces, 1105
 - prohibited in java, 119
- Mutually Exclusive Conditional Execution tag**
 - jsptl-examples.war, 589
- MVC (Model View Controller) pattern, 714**
 - comparison of JSP and servlets, 489
 - controller servlets, 592
 - divisions of functionality, 715
 - Front Controller architecture example, 503
 - implementation diagram, 716
 - manufacturing EJB example web interface
 - implementing, diagram and class source-code, 940
 - n-tier architecture conforms to, 16
 - purchasing system J2EE design example, 1091
 - treatment of distributed events, 1106
 - web applications and, 492
- MyClientSocketFactory class, 130**
- MyServerSocketFactory class, 129**

N

- name attribute**
 - definitions element, WSDL, 1117
- name-from-attribute element**
 - sub-element of <variable>, 534
- name-given element**
 - sub-element of <variable>, 534
- namespace attribute**
 - import element, WSDL, 1117
- namespaces**
 - URL path prefixes, 415
- name-value pairs**
 - Cookies, 322
- Naming class, java.rmi package**
 - bind() and rebind() methods, 97
 - bind() method, 104
 - distinguished from JNDI context, 88
 - methods, 89
- naming conventions**
 - JSP 1.2 hyphenation, 510
 - tag library prefixes, 526

- naming mapping**
 OMG Cos-Naming service and EJB, 896
- naming services**, 40
- NamingEnumeration interface**, javax.naming package, 61
- narrow() method**
 java.rmi.PortableRemoteObject class, 141
 javax.rmi.PortableRemoteObject class, 700
- native library loading**
 EJB 2.0 specification prohibitions, 712
 EJB 2.0 specification restrictions, 713
- NDS (Novell Directory Services)**, 41, 42
 service provider alternatives to those in JNDI SDK, 50
- nesting**
 doFilter() method calls, 394
 JSP tag extensions, 508, 517, 546
 scripting variables with nested scope, 539
- .NET framework** *see Microsoft Corporation.*
- Netscape Directory SDK for Java**
 lower-level alternative to JNDI, 47
- Netscape Directory Server**
 arius.com sample data, 57
- netstat**
 monitoring listening sockets, 124
- network operations**
 EJB 2.0 specification prohibitions, 712, 713
 minimizing network traffic as design principle, 1107
 minimizing with Session Façade Patterns, 1079
- network performance**
 client/server systems, 14
 holding connected JDBC resultsets, 227
 trade off with isolation requirements, 877
- network protocols**
 application servers, 1154
- networking support**
 built in to Java, 39
- Never transactional attribute**, 872
- new features**
 see also EJB 2.0 new features; HTTP version 1.1;
 J2EE 1.3 new features; JSP 1.2 new features;
 servlet API version 2.3.
 JavaMail 1.3, POP3 support, 636
 JDBC 3.0, save points, 158, 190
 JDK 1.2, policy-based access control, 98
 JNDI 1.2, federation in, 49
- new technologies**
 importance of adapting to work with legacy systems, 11
- New Web Component Wizard**
 deploying the TechSupportServlet example, 303
- NewsAddress class**, javax.mail.internet package, 656
- next() method**
 java.sql.ResultSet interface, 182
- NIS (Network Information Services)/NIS+**, 41, 42
 service provider for in JNDI SDK, 50
- NNTP (Network News Transfer Protocol)**, 647
- non-managed environment**
 CCI (Common Client Interface) API
 connection with non-managed application, 1043
 JCA (Java Connector Architecture), 1012
- non-repeatable reads**
 predefined isolation levels, 878
- non-XA transactions** *see local transactions.*
- NoSuchObjectException class**, java.rmi package, 91
- NoSuchRoutingInstruction**
 Product EJB, 848
- noSuchUser.jsp page**, 618
- NotBoundException class**, java.rmi package, 92
- NotSupportedException** *transactional attribute*, 871
 when to use, 873
- Novell** *see NDS; Netscape.*
- n-tier architecture**
 characteristics, 15
 enterprise architecture distinguished from, 16
- number tag**
 foundation tags, used in displayDetails.jsp page, 615
- O**
- object activation**, RMI, 111
- object models**
 advantages of using interface and control objects, 719
 aggregation of objects and functionality, 720
 complexity of UML, favors use of tools, 728
- object pooling**
 connection pooling as commonest form of, 207
 technique for managing and reusing objects, 207
- object serialization**, 85
 RMI object parameters, passed by value, 99
- object/relational mapping**
 effects on final system design, 721
- objectclass attribute**, LDAP, 43
- ObjectMessage interface**, javax.jms package, 991
- ObjectNotFoundException class**, javax.ejb package, 892
 predefined in EJB specification, 892
- object-orientation**
 code re-use via, 13
- Objects by Value specification**
 CORBA 2.3 enhancement for RMI-IIOP
 interoperability, 140
- ODBC (Open DataBase Connectivity)**
 programming model unsuited to Internet-based
 applications, 199
- offensive language**
 message moderation to remove, 398
- one-to-many relationship**, CMP 2.0, 804
- one-to-one relationship**, CMP 2.0, 804
- onMessage() method**
 javax.jms.MessageListener interface, 981, 996
- open source products**
 application servers, 1152
 Enhydra, 1152
 JBoss, 1152
 JonAS, 1152
 servlet engines, 1152
 Tomcat, 1152
- open() method**
 javax.mail.Folder class, 661
- operating systems**
 bundling with application servers, 1161
- optimistic locking**
 compared to pessimistic locking, 880
- Oracle**
 evaluating J2EE implementations, 1158
- ORB (Object Request Brokers)**
 see also CORBA; IIOP.
 Java IDL as, 144
- order creation scenario**
 illustrating SQL transactions, 189
- Order EJB**
 bean class, 840

Order EJB (continued)

business methods, 840
cancelOrder() method, [844](#)
finder methods, 839
getStatus() method, [844](#)
home interfaces, 839
lifecycle and framework methods, 842
manufacturing EJB example, 839
OrderNotCancelableException, [844](#)
primary keys, 843
remote interfaces, 840

ordering workflow

encapsulating, in purchase order example, 1096

OrderManagement bean example, 699

compiling and deploying, 707

OrderManagementBean class, 704

OrderNotCancelableException

Order EJB, [844](#)

OSI (Open System Interconnections)

RMI abstraction layer diagram, [85](#)

othersDeletesAreVisible() method

java.sql.DatabaseMetadata interface, 193

othersInsertsAreVisible() method

java.sql.DatabaseMetadata interface, 193

othersUpdatesAreVisible() method

java.sql.DatabaseMetadata interface, 193

otherwise tag

JSP conditional tag, 588

displayAllRegisteredUsers.jsp page, 625

OTS (Object Transaction Service)

and CORBA EJB transaction mapping, 896

OMG distributed transaction processing model, 215

out object, JSP, 474

out-of-process connections

web containers, 433

output generation

ServletResponse interface methods for, 293

ownDeletesAreVisible() method

java.sql.DatabaseMetadata interface, 193

ownInsertsAreVisible() method

java.sql.DatabaseMetadata interface, 193

ownUpdatesAreVisible() method

java.sql.DatabaseMetadata interface, 193

P

packaging

application components into modules, 35

components into applications

three-level scheme for, 34

EAR files as, 35

EJBs into JAR files, 701

web applications, 237, 412

packaging and deployment, 1165

applications, 1166

class loading schemes, [1170](#)

configuring packages, [1174](#)

deployment descriptors, 1177

optional tags, [1181](#)

enterprise application development process, 1174

enterprise applications package structure, 1175

ordering modules, 1182

containers, [1166](#)

applet containers, 1168

application client containers, 1167

EJB containers, 1167

web containers, 1167

dependency libraries, 1183

example, 1185

impact of dependency on packaging, 1188

problems with, 1183

using Class-Path: manifest entry, 1185

using extension mechanism, 1184

using single unified package, 1183

WEB-INF\lib directory, 1183

EAR (Enterprise Archive) files, 1166

Application Client JAR files, 1168

EJB Application JAR files, 1168

example, 1178

limitations of EAR files for packaging, 1170

Resource Adapter RAR files, 1168

Web Application WAR files, 1168

introduction, 1166

roles and responsibilities, 1168

application component provider role, 1169

application assembler role, 1169

deployer role, 1169

system administrator role, 1170

tool provider role, 1169

separation from design process, 409

packaging and deployment interfaces

JCA (Java Connector Architecture), 1012

Resource Adapter, 1012, 1016

deploying a resource adapter, 1018

packaging a resource adapter, 1017

resource adapter module, 1016

page beans

in Front Controller implementation, 494

page Directive, JSP

example, 447

syntax and attributes, 445

page object, JSP, 474

page requests counter example application, 386

page scope

JSP objects, 474

Page-by-Page Iterator Pattern see Value List Handler Pattern.

page-centric designs, JSP, 489

Front Controller Pattern compared to, 493

PageContext class, javax.servlet.jsp package

objects available to tag handlers, 522

popbody() and pushbody() methods, 520

pageContext object, JSP, 473

pageDirective.jsp, 447

page-view architecture

JSP design, 490

page-view with bean architecture

JSP design, [491](#)

technical support application uses, [491](#)

ParameterMetadata interface, java.sql package, 165

parameters

initialization by deployment descriptors, 244

JDBC prepared statements, 183

passing in RMI, [98](#)

parent classloaders, 1170

parse() method

javax.mail.internet.InternetAddress class, 656

Part interface, javax.mail package, 648

implemented by Message class, 644

partitioning

web applications, 412

passivation

conditions likely to cause, 337

ejbPassivate() method, 828

entity beans, 828

inactive servlet sessions, 334

session beans, 738

PasswordAuthentication class, javax.mail package
 getPasswordAuthentication() method, 643

passwordInput tag
 HTML tag library, registration example, 601, [603](#)
 loginForm.jsp page, 612

passwords
 authentication by, 51
 authentication alternatives to, 881
 security in web applications, [419](#)

pattern languages, 1089, 1108

patterns
see also J2EE patterns.
 software, systems and languages, 1073

performance
 advantages of stored procedures over entity beans, 1107
 advantages of type 4 database drivers, [162](#)
 connection pooling and, 207
 degradation with Http tunneling, [137](#)
 degradation with server-side includes, 620
 improvement with JDBC batch updates, [198](#)
 overhead of encryption, [127](#)
 passing object graphs over networks, [99](#)
 problems, thread synchronization, 267
 tuning, RMI, [149](#)

persistence management
see also BMP; CMP.
 BMP (bean-managed persistence), 799
 choice of implementation technology and, 721
 CMP (Container-Managed Persistence), 799
 EJB container service, 691

persistent connections
 HTTP 1.1, don't carry over session state, [319](#)
 J2EE destination queues, 979

persistent storage
 session beans, 739

PersonalInfo class, 548

personalization
 application servers, future prospects, 1157

pessimistic locking
 compared to optimistic locking, 880

Pet Store Application *see Java Pet Store Application.*

phantom reads
 predefined isolation levels, 878

PlaceSampleOrders client
 manufacturing EJB example, 771

Platform block
 Sun ONE (Sun Open Net Environment), 1142

platform independence
 servlets, [238](#)

platform requirements
 for modern application developments, 12

plug-ins
 JVM 1.3, [134](#)

point-to-point messaging
 characteristics, 970
 example, 973
 with a synchronous consumer, 977
 with an asynchronous consumer, 979
 message domain based on queues, 970

policy files, RMI, [90](#)
 system-wide security policies and, [97](#)

pooled state
 entity beans, 828

PooledConnection interface, javax.sql package
 addConnectionEventListener() method, 212
 close() method, 211

getConnection() method, 211
removeConnectionEventListener() method, 212

pooling
see also connection pooling; instance pooling; object pooling.
 EJBs (Enterprise JavaBeans), 738

POP3 (Post Office Protocol), [637](#)
 asynchronous messaging technique, 968
 JavaMail 1.2 added support for, 636

popbody() method
 javax.servlet.jsp.PageContext class, 520

popup() method
 ClientInterface interface, [109](#)

port element
 WSDL document, 1119

port redirectors
 Http tunneling
 alternative to java-rmi.cgi script, [137](#)

portability
 EJB container contracts and, 694
 EJB containers services and, 693
 JMS approach, 969
 SQL data types and, 188

PortableRemoteObject class, java.rmi package, [140](#)
 narrow() method, [141](#)

PortableRemoteObject class, javax.rmi package
 narrow() method, 700

portal capabilities
 application servers, future prospects, 1157

portType element
 WSDL document, 1118

POST requests, HTTP
 compared to GET requests, 235
 parameter sent within body of request, 289
 web application example, 243

prefix attribute
 taglib directive, JSP, 526

prepareCall() method
 java.sql.Connection interface, [174](#), 185

prepared statements, 183
 avoid need to recompile, 184

PreparedStatement interface, java.sql package, [163](#), 175
 setXXX() methods, mapping SQL data types, 186
 Tech Support servlet example, 302

prepareStatement() method
 java.sql.Connection interface, [174](#), 184

presentation logic layer
see also user-interface tier.
 component of n-tier architecture, 15
 three-tier architecture, 14

presentation services
 J2EE implementation, 1152

previous() method
 java.sql.ResultSet interface, [194](#)

primary keys, 807
 BMP (bean-managed persistence), 807
 CMP (Container-Managed Persistence), 808
 deployment descriptors, 809
 entity beans, 807
 specifying primary key class, 914
 findByPrimaryKey() method, 825
 manufacturing EJB example, 843
 Order EJB, 843
 SportBean laboratory example, 808

primitive data types
 not allowed as scripting variables, 538
 RMI parameters, passed by value, [98](#)

Principal interface, javax.security package

Principal interface, javax.security package

getName() Method, 886
principals
identifiers in Java security APIs, 881
translation between by EJB containers, 886
printers
storing a PrinterFactory as a Java object, 77
println() method
java.sql.DriverManager class, 170
java.io.PrintWriter class, 285
PrintWriter class, java.io package, 254, 285
println() method, 285
private resources, web applications
see also WEB-INF directory.
include Directive, JSP can access, 447
privileges, users
maintained in access control lists, 881
problem domain models, 1067
ProcessingException class
DemoConnector Adapter example, 1036
Product EJB
bean class, 846
business methods, 846
home interfaces, 845
lifecycle and framework methods, 847
local interfaces, 845
manufacturing EJB example, 844
NoSuchRoutingInstruction, 848
remote interfaces, 846
routing instructions, 844
programmatic demarcation, 215
special precautions, 223
steps required, 221
programmatic security, 421, 422
programming models
J2EE and technology integration, 13
specified by J2EE, 12
programming productivity
importance in modern application development, 11
protocols
JDBC URLs, 167
network protocols, 1154
stateless and stateful protocols defined, 318
stateless, suitability for fast information retrieval, 319
prototyping
JSP page-view architectures, 491
proxy class, generating
StockQuoteService web service example, 1137
proxy generation utility, 1137
Proxy design pattern, RMI, 85
proxy generation utility
proxy class, generating
StockQuoteService web service example, 1137
public directory
web applications, 241, 410
JSP include Directive not restricted to content of, 447
public key certificates, 52
HTTPS client authentication, 421
publish() method
javax.jms.TopicPublisher interface, 983
publish/subscribe messaging
example, 982
message domain based on topics, 971
purchasing system J2EE design example
choosing and refining an architecture, 1071
classes, deriving from problem domain model, 1068

creating a purchase order, 1096
design issues beyond the example, 1104
design principles, 1088
displaying product data, 1088
elaborating the context, 1068
elaborating the requirements, 1067
evolution of the system, 1092
functional requirements, 1065
introduced, 1064
JSP use, 1089
non-functional requirements, 1066
order approval, 1102
order processing, 1099
transactions in, 1103
pushbody() method
javax.servlet.jsp.PageContext class, 520

Q

queryAll() method
QueryMovieTables class, 181
QueryMovieTables class
modifying to use datasources, 205
queryAll() method, 181
question marks
parameterizing JDBC prepared statements, 183
Queue interface, javax.jms package, 991
QueueBrowser interface, javax.jms package, 991
QueueConnection interface, javax.jms package, 991
JMS architecture component, 972
QueueConnectionFactory, javax.jms package
JNDI lookup, 974
QueueConnectionFactory interface, javax.jms package, 991
QueueReceiver interface, javax.jms package, 991
setMessageListener() method, 980, 985
queues
point-to-point message domain based on, 970
QueueSender interface, javax.jms package, 991
send() method, 975, 979
QueueSession interface, javax.jms package, 992
createSender() method, 974
quotation marks
required round JSP custom tag attributes, 526
strings passed to JSP custom tag attributes, 532

R

radioButton tag
HTML tag library, registration example, 602
randomIDs property, java.rmi.server package, 150
Rational Unified Process, 1072
RDNs (Relative Distinguished Names), 43
reach of applications, 1071
asynchronous communication mechanisms compared, 1100, 1101
Read function
ejbLoad() method, 814
entity beans, 814
EntityContext interface, 815
SportBean laboratory example, 816
readLine() method
JavaMail POP3 access example, 674
ready state
entity beans, 828

- realInput tag**
HTML tag library, registration example, 601
- realms, 419**
- rebind() method**
java.rmi.Naming class, 89, 97
- receipt acknowledgement**
JMS point-to-point messages, 971
- receive() method**
javax.jms.MessageConsumer interface, 978
- Record interface, javax.resource.cci package, 1041**
CCI (Common Client Interface) API, 1041
- RecordFactory interface, javax.resource.cci package, 1041**
CCI (Common Client Interface) API, 1041
createIndexedRecord() method, 1043, 1045
createMappedRecord() method, 1043, 1045
createResultSet() method, 1043, 1045
- recovery**
system exceptions, appropriateness, 893
- reentrancy**
definition, 829
entity beans, 829
- Ref class, java.sql package, 164**
- references**
Java, storing in LDAP, 76, 77
- reflection**
mapping JSP tag extension attributes, 529
replaces skeletons in JRMP from SDK 1.3, 87
- regform.jsp page, 485**
- regForm.jsp page, 596**
use of HTML tag library, 602
- register() method**
java.rmi.activation.Activatable class, 116
- register.html page, 374**
forwarding requests to, 380
JSP equivalent is regform.jsp, 479
- register.jsp, 485**
- registerCustomer() method**
techSupportBean class, 485
- RegisterCustomerServlet, 375**
JSP equivalent is register.jsp, 479
- registerDriver() method**
java.sql.DriverManager class, 167, 168
- RegisterIt class, 96, 102**
activatable objects version, 117
RMI-IIOP and RMI/JRMP version, 143
starting multiple JVMs, 121
- registerSupport() method**
TechSupportBean class, 484
- registerSupportRequest() method**
techSupportBean class, 481
TechSupportBean class, 484
- registration and authentication example, JSP**
administration
displaying all registered users, 623
XML-based web service, 626
deploying, 629
HTML tag library, 600
introduced, 590
login error page, 618
login form page, 611
registration controller servlet, 592
registration form page, 596
registration error page, 610
viewing the user's favorite website, 619
viewing your user profile, 612
- RegistrationControllerServlet, 593**
web service created by, 627
- Registry interface, java.rmi.registry package, 88**
methods, 89
- registry services**
Java distributed architectures, 53
- registry, ebXML, 1121**
- registry, RMI**
use of sockets, 125
- registry, UDDI, 1120**
SOAP (Simple Object Access Protocol), 1120
XML (Extensible Markup Language), 1120
- relational database systems** *see databases*.
- relationships, entity beans, 837**
cardinality, 837
CMP 2.0, 283
many-to-many, 804
one-to-many, 804
one-to-one, 804
CMR fields, 837
defining relationships, 837
deployment descriptors, 838
direction, 837
local interfaces, 837
- relative() method**
java.sql.ResultSet interface, 194
- relay servers, SMTP**
SendSMTP example, 640
spamming and, 636
- release() method**
javax.servlet.jsp.tagext.Tag interface, 517
- releaseSavePoint() method**
java.sql.Connection interface, 191
- reliability**
JMS message delivery, 970
- remote callbacks, 108**
- Remote interface, java.rmi package, 93**
- remote interfaces**
banking example
entity beans, 903
session beans, 904
Bookstore CCI example, 1046
DemoConnector Adapter example, 1031
EJBs, 698
entity beans, 830
extend EJBObject interface, 701
Financial Aid Calculator bean example, 740, 748, 755
ManageOrders EJB, 763
Manufacture EJB, 765
Order EJB, 840
OrderManagement EJB example, 703
Product EJB, 846
RMI, defining, 93
RMI, implementing, 93
SportBean laboratory example, 830
- RemoteObject class, java.rmi.server package, 93**
- remote object protocols**
J2EE communications technologies, 33
- remote objects**
RMI, exporting and registering, 104
RMI, locating, 88
RMI, making activatable, 116
RMI, registering, 96
- remote parameters**
passing objects as, 99
- remote reference layer, 86**
RMI abstraction layer, 84

RemoteException class, java.rmi package

- RemoteException class, java.rmi package, 92, 894**
 subclasses, 91
 system exceptions can derive from, 894
- RemoteServer class, java.rmi.server package, 94**
 methods of, and of subclasses, 94
- remove() method**
 EJB remote interfaces, 700
- removeAttribute() method**
 HttpSession interface, 334
 ServletContext interface, 349
 ServletRequest interface, 288
- removeBodyPart() method**
 javax.mail.Multipart class, 651
- removeConnectionEventListener() method**
 javax.sql.PooledConnection interface, 212
- RemoveException class, javax.ejb package, 892**
 predefined in EJB specification, 892
- removeRowSetListener() method**
 javax.sql.RowSet interface, 226
- REPLACE modifications**
 LDAP, dangers of, 72
- replication**
 maintaining session and context, 435
- reply() method**
 javax.mail.Message class, 645
- Reply-to field**
 e-mail message headers, 647
- request attributes**
 methods for managing in servlet API specification, 287
- request dispatching, 368**
 approach to servlet collaboration, 367
 filters compared with, 386
 TechSupportServlet revised version uses, 373, 379
- request logging**
 possible use of filters, 385
- request object, JSP, 473**
- request parameters**
 access methods of the HttpServletRequest interface, 289
 access methods of the ServletRequest interface, 287
- request paths**
 HttpServletRequest methods for accessing, 291
- request scope**
 JSP objects, 475
- request serialization**
 servlet instances, to ensure single thread invocation, 267
- request validation**
 possible use of filters, 385
- RequestDispatcher API**
 possible use of wrapper classes, 297
- RequestDispatcher interface, javax.servlet package, 368**
 forward() method, 380
 include() method, 380, 619
 methods, 369
- RequestDispatcher objects**
 three ways of obtaining, 369
- RequestHandler interface**
 in Front Controller implementation, 493
 handleRequest() Method, 494
- requests and responses category**
 servlet API, introduced, 263
- required**
 <attribute> sub-tags, TLD <tag> element, 525
- Required transactional attribute, 872**
 avoiding use of UserTransaction interface with, 876
 database modification and, 873
- RequiresNew transactional attribute, 872**
- reserved keywords, 658**
- reset() method**
 ServletResponse interface, 294
- resetBuffer() method**
 ServletResponse interface, 294
- Resource Adapter**
 application component provider role, 1022
 application server vendor role, 1022
 Black Box Resource Adapters, 1023
 CCI Black Box Adapter, 1045
 DemoConnector Adapter example, 1023
- Connection interface, 1020**
- contracts, 1014**
 application contracts, 1014
 system-level contracts, 1014
 connection management, 1014
 EIS sign-on, 1016
 security management, 1015
 transaction management, 1015
- DataSource interface, 1019**
- deployer role, 1022**
 deploying resource adapter as J2EE application, 1022
- deployment descriptors, 1019**
- deployment options, 1022**
- EIS (Enterprise Information Systems), 1011**
- JCA (Java Connector Architecture), 1011**
- ManagedConnectionFactory interface, 1019**
- packaging and deployment interfaces, 1012, 1016**
 deploying a resource adapter, 1018
 packaging a resource adapter, 1017
 resource adapter module, 1016
- resource adapter provider role, 1021**
- resource adapter module**
 packaging and deployment of resource adapter, 1016
- Resource Adapter RAR files**
 packaging and deployment, J2EE, 1168
- resource bundles**
 locating, diagram, 606
- resource enlistment**
 in distributed transactions, 216, 223
- resource managers**
 calls to from EJB container, 871
 responsibilities for transaction processing, 218
- resource pooling**
 implemented by containers, 27
- resource recycling**
 see also reusability.
- EJBs, 1103**
- ResourceBundle class, java.util package, 605**
- resourceBundle tag**
 internationalization tag library, 606
 registration and authentication example, JSP, 608
- <resource-ref> element**
 bean provider role, 907
- resources, access**
 EJB in unspecified transaction context, 871
- response headers, HTTP, 235**
- response object, JSP, 473**
- response.jsp, 486**
- ResponseServlet, 376**
 doGet() method, 380
 forwarding requests to, 379
 JSP equivalent is response.jsp, 479
- ResultSet interface, java.sql package, 163**
 absolute() method, 194
 afterLast() method, 194
 beforeFirst() method, 194

ResultSet interface, java.sql package (continued)

- cancelRowUpdates() method, 197
- concurrency types and updateable resultsets, 197
- deleteRow() method, 198
- first() method, 194
- getConcurrency() method, 197
- getFetchDirection() method, 195
- getFetchSize() method, 195
- getString() method, 181
- getXXX() methods, mapping SQL data types, 186
- insertRow() method, 198
- isAfterLast() method, 193
- isBeforeFirst() method, 193
- isFirst() method, 194
- isLast() method, 194
- last() method, 194
- methods for retrieving data, 181
- methods for scrolling, 194
- methods related to cursor position, 193
- moveToInsertRow() method, 198
- next() method, 182
- previous() method, 194
- relative() method, 194
- rowDeleted() method, 198
- rowsUpdated() method, 197
- setFetchDirection() method, 194
- setFetchSize() method, 195
- three resultset types, 192
- updateXXX() methods, 197

ResultSetMetaData interface, java.sql package, 165

- getColumnCount() method, 182
- methods, 182

resultsets *see also scrollable resultsets.*

- updateable resultsets new feature in JDBC 2.1, 192, 197

reusability

- EJBs as reusable components, 687
- JSP tag extensions, 509
 - principles for, 563
 - value of using attributes, 532
 - web applications, 558

reusability of code

- object-orientation and distributed components in, 13

reusability of components, 13

- J2EE application structure facilitates, 36

reverse() method

- javax.lang.StringBuffer class, 544

reverse.jsp page, 545

RMI (Remote Method Invocation), 83

- see also RMI-IIOP.*
- developing applications, 92
- dynamically loading classes, 104
- EJB interfaces defined as RMI interfaces, 895
- exception handling, 91
- firewalls and HTTP, 136
- illustrating principle of interposition, 696
- J2EE communications technology, 33
- locating remote objects, 88
- making objects activatable, 116
- object activation, 111
- one of three Java distributed architectures, 53
- over JRMP, disadvantages for EJBs, 149
- parameter passing, 98
- performance tuning and debugging, 149
- properties specific to Sun JDK implementation, 151
- service provider for RMI registry in JNDI SDK, 50
- sockets and JMS compared to, 83
- writing the client to use remote objects, 95

RMI object serialization, 85

- object parameters passed by value, 99

rmic tool, 86

- developing in RMI-IIOP, example, 141
- generating a CORBA IDL, 140
- generating stubs and skeletons with, 96

RMIClassLoader class, java.rmi.server package, 105

rmid utility, 114, 115, 119

RMI-IIOP, 138, 895

- compared to Java IDL, 144
- development, compared to RMI/JRMP, 140
 - benefits to the developer, 148
- interoperability with CORBA, 139
- J2SE API which J2EE platforms must support, 21
- replaces JRMP in SDK 1.3, 87
- support required in J2EE containers, 149
- supported remote object protocol, 33
- weaknesses, 148
- web clients access to EJB container, 22

rmiregistry tool, 88, 90, 107

RMISecurityManager

- use before and after JDK 1.3, 98

RMISecurityManager class, java.rmi package, 98, 105

RMIserverSocketFactory interface, java.rmi.server package, 129

- createServerSocket() method, 126

RMISocketFactory class, java.rmi package, 124

RMISocketFactory class, java.rmi.server package

- createSocket() method, 126
- setSocketFactory() Method, 127

roles and responsibilities

- see also bean provider role; system administrator role.*
- application component provider role, 1022, 1169
- application assembler role, 916, 1169
- application development, 34
- application server vendor role, 1022
- benefits of using tag libraries, 568
- container/application server vendor role, 938
- deployer role, 927, 1022, 1169
- EJB client programmer and bean provider differences, 698
- J2EE product provider role, 1169
- resource adapter provider role, 1021
- separation offered by EJB use, 690
- tool provider role, 1169

roles, security, 421

- banking example, 883
- chatroom example application, 423
- description, 883
- EJB specific, defined in deployment descriptor, 883
- information type, EJB security, 883
- J2EE authentication based on, 421

role-specific interfaces, 1105

rollback

- commitment or, as transaction endpoint, 863
- EJB recovery from system exceptions, 893
- need not follow an application exception, 890

rollback() method

- java.sql.Connection interface, 188, 213
- javax.jms.Session interface, 974
- javax.transaction.UserTransaction interface, 220
- UserTransaction interface, 875
- warning about, in distributed transactions, 223

root-cause exceptions

- ServletException class, 272

routing instructions

- Product EJB, 844

rowChanged() method

rowChanged() method
javax.sql.RowSetListener interface, 225

rowDeleted() method
java.sql.ResultSet interface, 198

RowSet interface, javax.sql package
addRowSetListener() method, 226
no interfaces specified, 226
properties and events, 225
removeRowSetListener() method, 226

rowSetChanged() method
javax.sql.RowSetListener interface, 225

RowSetListener interface, javax.sql package
cursorMoved() method, 225
rowChanged() method, 225
rowSetChanged() method, 225

rowsets
command execution and results, 226
encapsulate database access, 223
JavaBean compliant object, 200
JavaBean-compliant objects, 223
possible implementations, 226
support for as feature of javax.sql package, 200
vendors supporting the features of, 227
weaknesses in terms of type safety, 1095
web rowsets, intended use, 229

rowsUpdated() method
java.sql.ResultSet interface, 197

RPC (Remote Procedure Calls)
JAX-RPC (Java API for XML-based RPC), 1126
mapping RPC calls to Java classes, 1126

RS232
early messaging technique, 968

RSA encryption algorithm, 128

rtextrvalues
<attribute> sub-tags, TLD <tag> element, 525
attribute values accepted by jr tag library, 571
helloAttrib TLD, 530

run time
JSPs run as servlets at, 442

runtime infrastructure
provided by the J2EE environment, 18

RuntimeException class, java.lang package, 894
system exceptions can derive from, 894

schemas
see also XML schemas.
LDAP servers, 44

scope
JSP implicit objects, 474
request attributes and, 287

scope attribute
<jsp:useBean> tag, 458

scope element
sub-element of <variable>, 535

scope, LDAP, 57
setting and defaults, 61

scoped attributes
JSPTL explicit collaboration, 577

scripting elements, JSP
allow code insertion, 450
tag category, 444
XML equivalent syntax, 476

scripting variables
declared in iteration example, 542
declared in TLD from JSP 1.2, 510
declaring using foundation tags, 616
JSP tag extensions, 508
example, 535
power of, 550
usefulness, 562
JSP tag extensions introducing, 534
methods of defining should not be mixed, 539
programmatic definition, 538

scriptlet.jsp, 452

scriptlets, JSP, 452
compared to declarations, 454
control flow example using, 578
tag extension body content, 533
thread safety, 454
use with nested tags, 546
XML equivalent syntax, 476

scroll sensitivity
resultsets, introduced with JDBC 2.1, 192

scrollable resultsets, 174
driver support for, 196
introduced in JDBC 2.1, 180, 191

SDK (Software Developer's Kit)
J2EE implementation, 1150
roles performed by, 1150

search filters, LDAP, 57

search() method
javax.mail.Folder class, 664
javax.naming.directory.DirContext interface, 59

searching
LDAP servers, 57
authenticated searching, 62
restricting the attributes displayed, 64

SearchResult class, javax.naming.directory package
getAttributes() method, 62

SearchTerm class, javax.mail.search package, 665

security
see also JAAS.
access control, 881
application design and, 888
authentication, 881
concerns over cookies, 324
configuring security files
EJB system administrator role, 937
control below business method level, 886
data confidentiality, 881
declarative and programmatic security, 421
declarative security as an EJB container service, 692

S

safe stores
MOM, MessageListeners as, 980

SASL (Simple Authentication and Security Layer)
LDAP supports, 44, 52, 55
uses MD5 or Kerberos mechanisms, 56

save points, 190

saveAttachment() method
javamail.pop class, 677

Savepoint interface, java.sql package, 190

SAX (Simple API for XML)
JAXP (Java API for XML Processing), 1122

scalability
advantages of asynchronous communications, 1101, 1108
application servers, 1148
choice of component type and, 722
EJB container services and, 693
importance in modern application development, 12
limitation JavaBeans-based model, 1092
RMI-IIOP advantage over RMI/JRMP, 149
stateless session bean advantages, 1096

security (continued)

EJBs
 mapping actual onto logical security identity, 882
 specifying requirements, 883
 enterprise level, 862
 identification, 881
 importance in modern application development, 12
 internet applications, using web rowsets, 229
 LDAP, 44
 LDAP version 3.0 support, 55
 notifying exceptions, 887
 programmatic security, 422
 RMI policy files, 90, 97
 security role reference
 bean provider role, 908
 services needed to protect data and processes, 881
 web applications, 419

Security APIs
 implementing future standards, 1156

security attributes
 EJB interfaces, 1105

security contexts
 RMI-IIOP advantage over RMI/JRMP, 148-149

security management
 system-level contracts, 1015

security mapping
 via SSL and RMI-IIOP, 896

security realms, 419, 425

security roles, 421
 banking example, 883
 chatroom example application, 423
 description, 883
 EJB specific, defined in deployment descriptor, 883
 information type, EJB security, 883
 J2EE authentication based on, 421

security services
 J2EE implementation, 1153

<security-constraints> tag
 deployment descriptors, 423

<security-role> tag
 deployment descriptors, 1181

SELECT clause, EJB QL, 805
 DISTINCT keyword, 805
 finder methods, 805
 select methods, 805

select methods
 SELECT clause, EJB QL, 805

semicolon terminator
 not used by JSP expressions, 455

send() method
 javax.jms.QueueSender interface, 975, 979
 javax.mail.Transport class, 666, 669

sendError() method
 HttpServletResponse interface, 295, 430

sendMail() method
 javamail_sendAll class, 682

sendRedirect() method
 HttpServletResponse interface, 296

SendSMTP class, 639
 setting mail content, 650

separating code from JSP pages
 using JavaBeans, 463
 using tag extensions, 508

separating presentation from business logic
 design principle, 1089
 View Helper Patterns, 1083

separating presentation from content

Front Controller Pattern, 492
 JSP custom tags, 569

separation of roles
 designer and developer, offered by EJB use, 690

separation of services and business logic
 using EJB technology, 861

sequence diagrams, UML
 Composite View Pattern, 1077
 Front Controller Pattern, 1076
 JSP invoking the BodyTag interface, 518
 JSP tag handler calls from compiled servlet, 515
 Service Locator Patterns, 1079
 servlet lifecycle, 274
 Value List Handler Pattern, 1087
 Value Object Patterns, 1081
 View Helper Patterns, 1083

serializable data
 replication requires, 435

serializable objects
 distributed containers and passivation require, 334
 serialized JavaBeans
 design pattern for networked applications, 1070
 storing in LDAP, 76, 77

serialization
 JavaMail API support limited, 678

servant classes, CORBA, 145

server administrator
 javax.sql connection pooling and, 210

server sockets, 123
 monitoring with netstat, 124

server.xml files
 <context> tag, 414

ServerError class, java.rmi package, 92

ServerException class, java.rmi package, 92

servers
see also web servers.
 changing locale, 609
 ClassFileServer, 107
 examples implementing servlet classes, 263
 heterogeneous, communication between, 895
 J2EE commercial application servers, 242
 JNDI prevent direct connection, 47
 LDAP, connecting to using JNDI, 54
 LDAP, schemas, 44
 resources required by client/server systems, 19
 RMI meaning of clients and, 84
 RMI, running the client and, 97
 RMI, use of sockets, 125
 servants and, CORBA terminology, 145
 supporting both IIOP and JRMP clients, 142

server-side applications
 requirements of, 236
 scarcity of resources and J2EE, 19

server-side includes see SSI.

serverSideIncludePeek.jsp page, 619, 622

ServerSocket class, java.net package, 123

server-to-server communication
 via XML, 31

Service Activator Pattern
 Sun Java Center discussion, 1102

service APIs
 single standard for access to technologies, 26

Service Container block
 Sun ONE (Sun Open Net Environment), 1142

Service Creation and Assembly component

Sun ONE (Sun Open Net Environment), 1142

Service Delivery block

Sun ONE (Sun Open Net Environment), 1143

service element

WSDL document, 1120

Service Integration block

Sun ONE (Sun Open Net Environment), 1143

service invocation

declarative and explicit invocation, 26

Service Locator Pattern, 1079

using to abstract services, 1108

service providers, JNDI, 46, 49

alternatives to those in the SDK, 50

obtaining and calling, 50

using with DNS, 78

service state

FreakServlet lifecycle, 283

service technologies

J2EE, 31

Service to Worker Pattern, 1085

Sun Java Center discussion, 1092

use as macro-patterns, 1108

service() methods

HttpServlet class, 268

overriding to be avoided, 268

Servlet interface, 265

ensuring method is thread-safe, 267, 274

service-based architectures, 1063

services layer, 734

Financial Aid Calculator bean example, 755

servlet API

authentication mechanisms, 421

classes categorized, 263

custom authentication not provided, 425

event handlers, 337

JSP implicit objects based on, 473

methods and classes

validity in course of servlet lifecycle, 285

methods for managing request attributes, 287

online documentation, 260

overview of classes and interfaces, 261

requests and responses category introduced, 263

requires Cookie support by web containers, 323

servlet configuration classes introduced, 263

servlet context and, 318, 346

servlet exceptions introduced, 263

servlet implementation classes introduced, 263, 264

session tracking in, 318, 319, 324

state maintenance in, 319

version 2.3 deployment descriptors, 426

version 2.3 new features

filters, 383

version 2.3 required for using JSPTL, 571

versions 2.1, servlet context interface revision, 346

web container implements, 22, 24

servlet API, version 2.2

deployment descriptor configuration of servlets, 270

servlet API, version 2.3

current version, 259

new features

additional classes and interfaces, 261

filtering introduced, 264

wrapper classes, 296

wrapper classes introduced, 263

servlet chaining

approach to servlet collaboration, 367

not supported by current web containers, 368

servlet collaboration

introduced, 318

reasons for and solutions, 367

technical support example, 373

servlet configuration classes, 270

introduced, 263

servlet context, 346

chatroom example application using, 350

classes, as category of servlet API, 264

clustering and, 436

introduced, 318

J2EE web applications, 239

web applications associated with, 414

servlet context events

application lifecycle events, 557

servlet controllers see controller servlets

servlet engines

see also Tomcat servlet engine; web containers

J2EE implementation, 1147, 1149

open source products, 1152

Tomcat, 1152

WebLogic Express, 1149

servlet exception classes

introduced, 263

servlet implementation classes

introduced, 263, 264

servlet initialization parameters, 427

Servlet interface, javax.servlet package, 264

destroy() method, 265

getServletConfig() method, 266

getServletInfo() method, 266

init() method, 265

methods, 264

service() method, 265

servlet lifecycles

FreakServlet example, 275

instantiation stage, 282

management by containers, 237, 273

stages shown by FreakServlet example, 281

validity of methods and objects during, 285

servlet programming, 259

technical support example, 297

servlet session lifetime, 326

Servlet specification

J2EE implementation, 1150

ServletConfig interface, javax.servlet package, 270

getInitParameter() method, 270

getInitParameterNames() method, 271

getServletContext() method, 271

getServletName() method, 271

ServletConfig objects

obtaining a reference to, 271

ServletContext interface, javax.servlet package

getAttribute() method, 359

getInitParameter() method, 426

getInitParameterNames() method, 426

getNamedDispatcher() method, 369, 380

getRequestDispatcher() method, 369, 619

methods, 346

ServletContext lifecycle

event handling, 349

ServletContextAttributesListener interface

methods, 349

ServletContextEvent class, javax.servlet package

getServletContext() method, 559

ServletContextListener interface, javax.servlet package, 557, 558

methods, 349

BOOKS FOR PROFESSIONALS BY PROFESSIONALS®

APRESS, THE AUTHOR'S PRESS® IS A NEW KIND OF PUBLISHING COMPANY, founded by authors and focused on authors. Apress publishes books for you, the IT professional, programmer, and enthusiast. Apress goes back to the foundations of what computer book publishing is all about; providing high-quality information from expert authors directly to you, the reader. We give our writers the freedom to explore their ideas and write the book they think is needed, not the book the publisher dictates. Apress is the publisher that doesn't get between you and the authors who write for you.



Professional Java Server Programming J2EE 1.3 Edition

Apress is pleased to bring you this reprint of a book that was originally published by the former Wrox Press Limited, an imprint of Peer Information Group. When Peer became insolvent, Apress stepped in and bought the publishing rights to more than 90 percent of Peer's titles. Apress is dedicated to ensuring the valuable information these books contain is not lost, so we are reprinting as many of the classics as possible.

On a personal note, I am very happy with the acquisition. Apress is unique among computer book publishing companies because the editors and I have continued our careers as programmers and writers. I must confess that as a publisher, programmer, and writer, I have long admired many of the books whose publishing rights Apress acquired. One of them is the book you hold in your hands right now.

Sincerely,
Gary Cornell
Apress author, cofounder, and publisher

The release of the 1.3 version of the Java 2 Platform, Enterprise Edition (2EE) represents the evolution of Sun Microsystems' server-side development platform into a more mature and sophisticated specification/ Servlets 2.3 gain events and filtering; JavaServer Pages (JSP) 1.2 gain a new XML syntax and enhancements to the custom tag mechanisms; and Enterprise JavaBeans (EJB) 2.0 has some significant changes to its container-managed persistence model, as well as support for asynchronous processing with the new message-driven beans.

This book demonstrates how to design and construct secure and scalable n-tier J2EE applications using JSP and servlets for the web tier and EJBs for the business logic. It also covers J2EE Connector Architecture that allows you to easily integrate your J2EE applications to enterprise information systems.

For sale in the Indian Subcontinent (India, Pakistan, Bangladesh, Nepal, Bhutan, Sri Lanka,) only. Illegal for sale outside of these countries.

Shelve in:
Java

User level:
Intermediate—Advanced

Rs. 675.00

Distributed by:

dreamtech
PRESS
19-A, Ansari Road, Daryaganj,
New Delhi-110002

www.apress.com

a!
Apress

ISBN 978-81-8128-757-1



9 788181 287571

Copyrighted material