The J2EE™ Tutorial
Second Edition

Stephanie Bodoff
Eric Armstrong
Jennifer Ball
Debbie Bode Carson
Ian Evans
Dale Green
Kim Haase
Eric Jendrock

Addison-Wesley
Copyright © 2004 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, U.S.A. All rights reserved. U.S. Government Rights - Commercial software. Government users are subject to the Sun Microsystems, Inc. standard license agreement and applicable provisions of the FAR and its supplements.

This distribution may include materials developed by third parties.

Sun, Sun Microsystems, the Sun logo, Java, JavaBeans, JavaServer, JavaServer Pages, Enterprise JavaBeans, Java Naming and Directory Interface, JavaMail, JDBC, EJB, JSP, J2EE, J2SE, “Write Once, Run Anywhere,” and the Java Coffee Cup logo are trademarks or registered trademarks of Sun Microsystems, Inc., in the U.S. and other countries.

Unless otherwise licensed, software code in all technical materials herein (including articles, FAQs, samples) is provided under this License.

Products covered by and information contained in this service manual are controlled by U.S. Export Control laws and may be subject to the export or import laws in other countries. Nuclear, missile, chemical, biological weapons or nuclear maritime end uses or end users, whether direct or indirect, are strictly prohibited. Export or reexport to countries subject to U.S. embargo or to entities identified on U.S. export exclusion lists, including, but not limited to, the denied persons and specially designated nationals lists is strictly prohibited.

DOCUMENTATION IS PROVIDED “AS IS” AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

Copyright © 2004 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, États-Unis. Tous droits réservés.

Droits du gouvernement américain, utilisateurs gouvernementaux - logiciel commercial. Les utilisateurs gouvernementaux sont soumis au contrat de licence standard de Sun Microsystems, Inc., ainsi qu’aux dispositions en vigueur de la FAR (Federal Acquisition Regulations) et des suppléments à celles-ci.

Cette distribution peut comprendre des composants développés par des tiers.

Sun, Sun Microsystems, le logo Sun, Java, JavaBeans, JavaServer, JavaServer Pages, Enterprise JavaBeans, Java Naming and Directory Interface, JavaMail, JDBC, EJB, JSP, J2EE, J2SE, “Write Once, Run Anywhere,” et le logo Java Coffee Cup sont des marques de fabrique ou des marques déposées de Sun Microsystems, Inc., aux États-Unis et dans d’autres pays.

A moins qu’autrement autorisé, le code de logiciel en tous les matériaux techniques dans le présent (articles y compris, FAQs, échantillons) est fourni sous ce permis.

Les produits qui font l’objet de ce manuel d’entretien et les informations qu’il contient sont régis par la législation américaine en matière de contrôle des exportations et peuvent être soumis au droit d’autres pays dans le domaine des exportations et importations. Les utilisations finales, ou utilisateurs finaux, pour des armes nucléaires, des missiles, des armes biologiques et chimiques, ou du nucléaire maritime, directement ou indirectement, sont strictement interdites. Les exportations ou réexportations vers des pays sous embargo des États-Unis, ou vers des entités figurant sur les listes d’exclusion d’exportation américaines, y compris, mais de manière non exclusive, la liste de personnes qui font objet d’un ordre de ne pas participer, d’une façon directe ou indirecte, aux exportations des produits ou des services qui sont régis par la législation américaine en matière de contrôle des exportations ("U.S. Commerce Department’s Table of Denial Orders" et la liste de ressortissants spécifiquement désignés ("U.S. Treasury Department of Specially Designated Nationals and Blocked Persons"), sont rigoureusement interdites.

LA DOCUMENTATION EST FOURNIE “EN L’ÉTAT” ET TOUTES AUTRES CONDITIONS, DECLARATIONS ET GARANTIES EXPRESSES OU TACITES SONT FORMELLEMENT EXCLUES, DANS LA MESURE AUTORISEE PAR LA LOI APPLICABLE, Y COMPRISS NOTAMMENT TOUTE GARANTIE IMPLICITE RELATIVE À LA QUALITE MARCHANDE, À L’APTITUDE À UNE UTILISATION PARTICULIERE OU À L’ABSENCE DE CONTREFAÇON.
# Contents

Foreword ................................................................. xxxv

About This Tutorial .................................................... xxxvii

Chapter 1  Overview ................................................... 1

- Distributed Multitiered Applications ...................... 2
  - J2EE Components ............................................. 3
  - J2EE Clients ................................................... 4
  - Web Components .............................................. 6
  - Business Components ....................................... 6
  - Enterprise Information System Tier ....................... 8

- J2EE Containers ................................................... 8
  - Container Services .......................................... 8
  - Container Types ............................................. 9

- Web Services Support ........................................... 10
  - XML .................................................................. 11
  - SOAP Transport Protocol ................................... 11
  - WSDL Standard Format ....................................... 12
  - UDDI and ebXML Standard Formats ....................... 12

- Packaging Applications .......................................... 12

- Development Roles ................................................. 14
  - J2EE Product Provider ....................................... 14
  - Tool Provider .................................................. 14
  - Application Component Provider ......................... 15
  - Application Assembler ....................................... 15
  - Application Deployer and Administrator ................. 16

- J2EE 1.4 Platform APIs ........................................... 16
  - Enterprise JavaBeans Technology ......................... 17
  - Java Servlet Technology ..................................... 18
  - JavaServer Pages Technology ............................... 18

vii
Java Message Service API 18
Java Transaction API 18
JavaMail API 18
JavaBeans Activation Framework 19
Java API for XML Processing 19
Java API for XML-Based RPC 19
SOAP with Attachments API for Java 20
Java API for XML Registries 20
J2EE Connector Architecture 20
JDBC API 21
Java Naming and Directory Interface 21
Java Authentication and Authorization Service 22
Simplified Systems Integration 22
Sun Java System Application Server Platform Edition 8
Technologies 24
Tools 25
Starting and Stopping the Application Server 25
Starting the Admin Console 27
Starting the deploytool Utility 27
Starting and Stopping the PointBase Database Server 28
Debugging J2EE Applications 28

Chapter 2 Understanding XML .................. 31

Introduction to XML 31
  What Is XML? 31
  Why Is XML Important? 36
  How Can You Use XML? 38
Generating XML Data 41
  Writing a Simple XML File 41
  Defining the Root Element 42
  Writing Processing Instructions 46
  Introducing an Error 48
  Substituting and Inserting Text 49
  Creating a Document Type Definition 52
  Documents and Data 57
  Defining Attributes and Entities in the DTD 57
  Referencing Binary Entities 64
  Defining Parameter Entities and Conditional Sections 66
  Resolving a Naming Conflict 70
  Using Namespaces 71
CONTENTS

Designing an XML Data Structure 74
  Saving Yourself Some Work 74
  Attributes and Elements 75
  Normalizing Data 77
  Normalizing DTDs 78
Summary 79

Chapter 3  Getting Started with Web Applications . . . . . . . . . . 81
  Web Application Life Cycle 84
  Web Modules 86
    Packaging Web Modules 88
    Deploying Web Modules 89
    Listing Deployed Web Modules 92
    Updating Web Modules 92
    Undeploying Web Modules 95
  Configuring Web Applications 95
    Mapping URLs to Web Components 96
    Declaring Welcome Files 97
    Setting Initialization Parameters 98
    Mapping Errors to Error Screens 99
    Declaring Resource References 99
  Duke’s Bookstore Examples 100
  Accessing Databases from Web Applications 100
    Populating the Example Database 101
    Creating a Data Source in the Application Server 102
    Specifying a Web Application’s Resource Reference 103
    Mapping the Resource Reference to a Data Source 104
Further Information 104

Chapter 4  Java API for XML Processing. . . . . . . . . . . . . . . . . . 105
  The JAXP APIs 105
  An Overview of the Packages 106
  The Simple API for XML APIs 107
    The SAX Packages 109
  The Document Object Model APIs 110
    The DOM Packages 111
  The Extensible Stylesheet Language Transformations APIs 111
    The XSLT Packages 112
  Using the JAXP Libraries 113
  Where Do You Go from Here? 113
## Chapter 5  Simple API for XML  ........................................... 115

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>When to Use SAX</td>
<td>116</td>
</tr>
<tr>
<td>Echoing an XML File with the SAX Parser</td>
<td>117</td>
</tr>
<tr>
<td>Creating the Skeleton</td>
<td>118</td>
</tr>
<tr>
<td>Importing Classes</td>
<td>118</td>
</tr>
<tr>
<td>Setting Up for I/O</td>
<td>119</td>
</tr>
<tr>
<td>Implementing the ContentHandler Interface</td>
<td>119</td>
</tr>
<tr>
<td>Setting up the Parser</td>
<td>120</td>
</tr>
<tr>
<td>Writing the Output</td>
<td>121</td>
</tr>
<tr>
<td>Spacing the Output</td>
<td>122</td>
</tr>
<tr>
<td>Handling Content Events</td>
<td>123</td>
</tr>
<tr>
<td>Compiling and Running the Program</td>
<td>128</td>
</tr>
<tr>
<td>Checking the Output</td>
<td>129</td>
</tr>
<tr>
<td>Identifying the Events</td>
<td>129</td>
</tr>
<tr>
<td>Compressing the Output</td>
<td>132</td>
</tr>
<tr>
<td>Inspecting the Output</td>
<td>134</td>
</tr>
<tr>
<td>Documents and Data</td>
<td>135</td>
</tr>
<tr>
<td>Adding Additional Event Handlers</td>
<td>135</td>
</tr>
<tr>
<td>Identifying the Document’s Location</td>
<td>136</td>
</tr>
<tr>
<td>Handling Processing Instructions</td>
<td>137</td>
</tr>
<tr>
<td>Summary</td>
<td>139</td>
</tr>
<tr>
<td>Handling Errors with the Nonvalidating Parser</td>
<td>139</td>
</tr>
<tr>
<td>Displaying Special Characters and CDATA</td>
<td>147</td>
</tr>
<tr>
<td>Handling Special Characters</td>
<td>147</td>
</tr>
<tr>
<td>Handling Text with XML-Style Syntax</td>
<td>148</td>
</tr>
<tr>
<td>Handling CDATA and Other Characters</td>
<td>149</td>
</tr>
<tr>
<td>Parsing with a DTD</td>
<td>149</td>
</tr>
<tr>
<td>DTD’s Effect on the Nonvalidating Parser</td>
<td>149</td>
</tr>
<tr>
<td>Tracking Ignorable Whitespace</td>
<td>151</td>
</tr>
<tr>
<td>Cleanup</td>
<td>152</td>
</tr>
<tr>
<td>Empty Elements, Revisited</td>
<td>153</td>
</tr>
<tr>
<td>Echoing Entity References</td>
<td>153</td>
</tr>
<tr>
<td>Echoing the External Entity</td>
<td>154</td>
</tr>
<tr>
<td>Summarizing Entities</td>
<td>154</td>
</tr>
<tr>
<td>Choosing Your Parser Implementation</td>
<td>155</td>
</tr>
<tr>
<td>Using the Validating Parser</td>
<td>155</td>
</tr>
<tr>
<td>Configuring the Factory</td>
<td>155</td>
</tr>
<tr>
<td>Validating with XML Schema</td>
<td>156</td>
</tr>
<tr>
<td>Experimenting with Validation Errors</td>
<td>159</td>
</tr>
<tr>
<td>Error Handling in the Validating Parser</td>
<td>161</td>
</tr>
</tbody>
</table>
## CONTENTS

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parsing a Parameterized DTD</td>
<td>162</td>
</tr>
<tr>
<td>DTD Warnings</td>
<td>163</td>
</tr>
<tr>
<td>Handling Lexical Events</td>
<td>164</td>
</tr>
<tr>
<td>How the LexicalHandler Works</td>
<td>164</td>
</tr>
<tr>
<td>Working with a LexicalHandler</td>
<td>165</td>
</tr>
<tr>
<td>Using the DTDHandler and EntityResolver</td>
<td>171</td>
</tr>
<tr>
<td>The DTDHandler API</td>
<td>171</td>
</tr>
<tr>
<td>The EntityResolver API</td>
<td>172</td>
</tr>
<tr>
<td>Further Information</td>
<td>173</td>
</tr>
</tbody>
</table>

### Chapter 6 Document Object Model

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>When to Use DOM</td>
<td>176</td>
</tr>
<tr>
<td>Documents Versus Data</td>
<td>176</td>
</tr>
<tr>
<td>Mixed-Content Model</td>
<td>177</td>
</tr>
<tr>
<td>A Simpler Model</td>
<td>178</td>
</tr>
<tr>
<td>Increasing the Complexity</td>
<td>179</td>
</tr>
<tr>
<td>Choosing Your Model</td>
<td>181</td>
</tr>
<tr>
<td>Reading XML Data into a DOM</td>
<td>182</td>
</tr>
<tr>
<td>Creating the Program</td>
<td>182</td>
</tr>
<tr>
<td>Additional Information</td>
<td>186</td>
</tr>
<tr>
<td>Looking Ahead</td>
<td>188</td>
</tr>
<tr>
<td>Displaying a DOM Hierarchy</td>
<td>189</td>
</tr>
<tr>
<td>Convert DomEcho to a GUI App</td>
<td>189</td>
</tr>
<tr>
<td>Create Adapters to Display the DOM in a JTree</td>
<td>195</td>
</tr>
<tr>
<td>Finishing Up</td>
<td>204</td>
</tr>
<tr>
<td>Examining the Structure of a DOM</td>
<td>205</td>
</tr>
<tr>
<td>Displaying a Simple Tree</td>
<td>205</td>
</tr>
<tr>
<td>Displaying a More Complex Tree</td>
<td>207</td>
</tr>
<tr>
<td>Finishing Up</td>
<td>213</td>
</tr>
<tr>
<td>Constructing a User-Friendly JTree from a DOM</td>
<td>213</td>
</tr>
<tr>
<td>Compressing the Tree View</td>
<td>213</td>
</tr>
<tr>
<td>Acting on Tree Selections</td>
<td>219</td>
</tr>
<tr>
<td>Handling Modifications</td>
<td>228</td>
</tr>
<tr>
<td>Finishing Up</td>
<td>228</td>
</tr>
<tr>
<td>Creating and Manipulating a DOM</td>
<td>228</td>
</tr>
<tr>
<td>Obtaining a DOM from the Factory</td>
<td>228</td>
</tr>
<tr>
<td>Normalizing the DOM</td>
<td>231</td>
</tr>
<tr>
<td>Other Operations</td>
<td>233</td>
</tr>
<tr>
<td>Finishing Up</td>
<td>237</td>
</tr>
</tbody>
</table>
CONTENTS

Validating with XML Schema 237
  Overview of the Validation Process 237
  Configuring the DocumentBuilder Factory 238
  Validating with Multiple Namespaces 239

Further Information 243

Chapter 7  Extensible Stylesheet Language Transformations 245

Introducing XSL, XSLT and XPath 246
  The JAXP Transformation Packages 246

How XPath Works 247
  XPath Expressions 247
  The XSLT/XPath Data Model 248
  Templates and Contexts 249
  Basic XPath Addressing 249
  Basic XPath Expressions 250
  Combining Index Addresses 251
  Wildcards 251
  Extended-Path Addressing 252
  XPath Data Types and Operators 252
  String-Value of an Element 253
  XPath Functions 253
  Summary 257

Writing Out a DOM as an XML File 257
  Reading the XML 258
  Creating a Transformer 259
  Writing the XML 262
  Writing Out a Subtree of the DOM 263
  Summary 264

Generating XML from an Arbitrary Data Structure 264
  Creating a Simple File 265
  Creating a Simple Parser 267
  Modifying the Parser to Generate SAX Events 269
  Using the Parser as a SAXSource 275
  Doing the Conversion 278

Transforming XML Data with XSLT 278
  Defining a Simple <article> Document Type 279
  Creating a Test Document 281
  Writing an XSLT Transform 282
  Processing the Basic Structure Elements 283
  Writing the Basic Program 287
CONTENTS

Trimming the Whitespace 289
Processing the Remaining Structure Elements 291
Process Inline (Content) Elements 296
Printing the HTML 300
What Else Can XSLT Do? 301
Transforming from the Command Line with Xalan 303
Concatenating Transformations with a Filter Chain 303
Writing the Program 303
Understanding How the Filter Chain Works 307
Testing the Program 308
Further Information 311

Chapter 8 Building Web Services with JAX-RPC . . . . . . 313
Setting the Port 314
Creating a Simple Web Service and Client with JAX-RPC 314
Coding the Service Endpoint Interface and Implementation Class 316
Building the Service 317
Packaging the Service 318
Specifying the Endpoint Address 320
Deploying the Service 320
Static Stub Client 321
Types Supported by JAX-RPC 324
J2SE SDK Classes 324
Primitives 325
Arrays 325
Value Types 325
JavaBeans Components 326
Web Service Clients 326
Dynamic Proxy Client 326
Dynamic Invocation Interface Client 329
Application Client 333
More JAX-RPC Clients 336
Web Services Interoperability and JAX-RPC 337
Further Information 337

Chapter 9 SOAP with Attachments API for Java . . . . . . 339
Overview of SAAJ 340
Messages 340
Connections 344
CONTENTS

Creating JAXR Resources 429
Creating and Packaging the Application 430
Deploying the Application 433
Running the Application Client 433

Further Information 434

Chapter 11  Java Servlet Technology. . . . . . . . . . . . . . . . . . . . . 435

What Is a Servlet? 435
The Example Servlets 436
Troubleshooting 440
Servlet Life Cycle 441
Handling Servlet Life-Cycle Events 441
Handling Errors 443
Sharing Information 444
Using Scope Objects 444
Controlling Concurrent Access to Shared Resources 445
Accessing Databases 446
Initializing a Servlet 447
Writing Service Methods 448
Getting Information from Requests 449
Constructing Responses 451
Filtering Requests and Responses 454
Programming Filters 454
Programming Customized Requests and Responses 456
Specifying Filter Mappings 458
Invoking Other Web Resources 460
Including Other Resources in the Response 461
Transferring Control to Another Web Component 462
Accessing the Web Context 463
Maintaining Client State 464
Accessing a Session 464
Associating Objects with a Session 465
Session Management 466
Session Tracking 466
Finalizing a Servlet 467
Tracking Service Requests 468
Notifying Methods to Shut Down 469
Creating Polite Long-Running Methods 470

Further Information 470
Chapter 12 JavaServer Pages Technology .......................... 471

What Is a JSP Page? 471
  Example 472
The Example JSP Pages 476
The Life Cycle of a JSP Page 482
  Translation and Compilation 483
  Execution 484
Creating Static Content 486
  Response and Page Encoding 487
Creating Dynamic Content 487
  Using Objects within JSP Pages 488
Expression Language 489
  Deactivating Expression Evaluation 490
  Using Expressions 490
  Variables 491
  Implicit Objects 492
  Literals 493
  Operators 493
  Reserved Words 494
  Examples 494
  Functions 496
JavaBeans Components 497
  JavaBeans Component Design Conventions 497
  Creating and Using a JavaBeans Component 499
  Setting JavaBeans Component Properties 500
  Retrieving JavaBeans Component Properties 502
Using Custom Tags 502
  Declaring Tag Libraries 503
  Including the Tag Library Implementation 505
Reusing Content in JSP Pages 506
Transferring Control to Another Web Component 507
  jsp:param Element 508
Including an Applet 508
Setting Properties for Groups of JSP Pages 510
Further Information 513

Chapter 13 JavaServer Pages Documents ......................... 515

The Example JSP Document 516
Creating a JSP Document 521
  Declaring Tag Libraries 523
CONTENTS

Including Directives in a JSP Document 525
Creating Static and Dynamic Content 527
Using the jsp:root Element 530
Using the jsp:output Element 531
Identifying the JSP Document to the Container 535

Chapter 14  JavaServer Pages Standard Tag Library . . . . . . . . 537

The Example JSP Pages 538
Using JSTL 541
    Tag Collaboration 542
Core Tag Library 543
    Variable Support Tags 543
    Flow Control Tags 545
    URL Tags 548
    Miscellaneous Tags 549
XML Tag Library 550
    Core Tags 551
    Flow Control Tags 552
    Transformation Tags 553
Internationalization Tag Library 554
    Setting the Locale 555
    Messaging Tags 555
    Formatting Tags 556
SQL Tag Library 556
    query Tag Result Interface 559
Functions 561
Further Information 562

Chapter 15  Custom Tags in JSP Pages. . . . . . . . . . . . . . . . . . . . 563

What Is a Custom Tag? 564
The Example JSP Pages 564
Types of Tags 568
    Tags with Attributes 569
    Tags with Bodies 572
    Tags That Define Variables 572
    Communication between Tags 573
Encapsulating Reusable Content Using Tag Files 573
    Tag File Location 576
    Tag File Directives 576
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Evaluating Fragments Passed to Tag Files</td>
<td>584</td>
</tr>
<tr>
<td>Examples</td>
<td>585</td>
</tr>
<tr>
<td><strong>Tag Library Descriptors</strong></td>
<td>589</td>
</tr>
<tr>
<td>Top-Level Tag Library Descriptor Elements</td>
<td>590</td>
</tr>
<tr>
<td>Declaring Tag Files</td>
<td>591</td>
</tr>
<tr>
<td>Declaring Tag Handlers</td>
<td>594</td>
</tr>
<tr>
<td>Declaring Tag Attributes for Tag Handlers</td>
<td>595</td>
</tr>
<tr>
<td>Declaring Tag Variables for Tag Handlers</td>
<td>596</td>
</tr>
<tr>
<td><strong>Programming Simple Tag Handlers</strong></td>
<td>598</td>
</tr>
<tr>
<td>Including Tag Handlers in Web Applications</td>
<td>599</td>
</tr>
<tr>
<td>How Is a Simple Tag Handler Invoked?</td>
<td>599</td>
</tr>
<tr>
<td>Tag Handlers for Basic Tags</td>
<td>600</td>
</tr>
<tr>
<td>Tag Handlers for Tags with Attributes</td>
<td>600</td>
</tr>
<tr>
<td>Tag Handlers for Tags with Bodies</td>
<td>602</td>
</tr>
<tr>
<td>Tag Handlers for Tags That Define Variables</td>
<td>603</td>
</tr>
<tr>
<td>Cooperating Tags</td>
<td>606</td>
</tr>
<tr>
<td>Examples</td>
<td>608</td>
</tr>
</tbody>
</table>

### Chapter 16  Scripting in JSP Pages ................................. 617

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Example JSP Pages</td>
<td>618</td>
</tr>
<tr>
<td>Using Scripting</td>
<td>619</td>
</tr>
<tr>
<td>Disabling Scripting</td>
<td>620</td>
</tr>
<tr>
<td>Declarations</td>
<td>620</td>
</tr>
<tr>
<td>Initializing and Finalizing a JSP Page</td>
<td>620</td>
</tr>
<tr>
<td>Scriptlets</td>
<td>621</td>
</tr>
<tr>
<td>Expressions</td>
<td>622</td>
</tr>
<tr>
<td>Programming Tags That Accept Scripting Elements</td>
<td>623</td>
</tr>
<tr>
<td>TLD Elements</td>
<td>623</td>
</tr>
<tr>
<td>Tag Handlers</td>
<td>623</td>
</tr>
<tr>
<td>Tags with Bodies</td>
<td>626</td>
</tr>
<tr>
<td>Cooperating Tags</td>
<td>627</td>
</tr>
<tr>
<td>Tags That Define Variables</td>
<td>629</td>
</tr>
</tbody>
</table>

### Chapter 17  JavaServer Faces Technology .............................. 631

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>JavaServer Faces Technology Benefits</td>
<td>633</td>
</tr>
<tr>
<td>What Is a JavaServer Faces Application?</td>
<td>633</td>
</tr>
<tr>
<td>Framework Roles</td>
<td>634</td>
</tr>
<tr>
<td>A Simple JavaServer Faces Application</td>
<td>635</td>
</tr>
<tr>
<td>Steps in the Development Process</td>
<td>635</td>
</tr>
<tr>
<td>Creating the Pages</td>
<td>638</td>
</tr>
</tbody>
</table>
CONTENTS xix

Defining Page Navigation 640
Developing the Beans 642
Adding Managed Bean Declarations 643

User Interface Component Model 644
User Interface Component Classes 645
Component Rendering Model 647
Conversion Model 651
Event and Listener Model 652
Validation Model 654

Navigation Model 654

Backing Bean Management 656

How the Pieces Fit Together 659

The Life Cycle of a JavaServer Faces Page 662
Request Processing Life Cycle Scenarios 663
Standard Request Processing Life Cycle 664

Further Information 669

Chapter 18 Using JavaServer Faces Technology in JSP Pages 671

The Example JavaServer Faces Application 672
Setting Up a Page 676
Using the Core Tags 678
Using the HTML Component Tags 680
UI Component Tag Attributes 681
The UIForm Component 683
The UIColumn Component 684
The UIPanel Component 684
The UIData Component 686
The UIGraphic Component 690
The UIInput and UIOutput Components 690
The UIPanel Component 694
The UISelectBoolean Component 697
The UISelectMany Component 697
The UIMessage and UIMessages Components 698
The UISelectOne Component 699
The UISelectItem, UISelectItems, and UISelectItemGroup Components 700

Using Localized Messages 703
Referencing a ResourceBundle from a Page 704
Referencing a Localized Message 705

...
CONTENTS

Using the Standard Converters 705
  Using DateTimeConverter 707
  Using NumberConverter 709

Registering Listeners on Components 710
  Registering a ValueChangeListener on a Component 711
  Registering an ActionListener on a Component 711

Using the Standard Validators 712
  Requiring a Value 713
  Using the LongRangeValidator 714

Binding Component Values and Instances to External Data Sources 714
  Binding a Component Value to a Property 715
  Binding a Component Value to an Implicit Object 717
  Binding a Component Instance to a Bean Property 718

Referencing a Backing Bean Method 719
  Referencing a Method That Performs Navigation 720
  Referencing a Method That Handles an ActionEvent 721
  Referencing a Method That Performs Validation 722
  Referencing a Method That Handles a ValueChangeEvent 722

Using Custom Objects 723
  Using a Custom Converter 724
  Using a Custom Validator 725
  Using a Custom Component 726

Chapter 19  Developing with JavaServer Faces Technology .................................. 729

Writing Component Properties 730
  Writing Properties Bound to Component Values 730
  Writing Properties Bound to Component Instances 739

Performing Localization 741
  Creating a Resource Bundle 741
  Localizing Dynamic Data 741
  Localizing Messages 742

Creating a Custom Converter 744

Implementing an Event Listener 747
  Implementing Value-Change Listeners 748
  Implementing Action Listeners 749

Creating a Custom Validator 750
  Implementing the Validator Interface 751
  Creating a Custom Tag 753

Writing Backing Bean Methods 755
  Writing a Method to Handle Navigation 755
CONTENTS

Chapter 20 Creating Custom UI Components ..................... 761

Determining Whether You Need a Custom Component or Renderer 762
  When to Use a Custom Component 762
  When to Use a Custom Renderer 763
  Component, Renderer, and Tag Combinations 764
Understanding the Image Map Example 765
  Why Use JavaServer Faces Technology to Implement an Image Map? 765
  Understanding the Rendered HTML 766
  Understanding the JSP Page 767
  Configuring Model Data 769
  Summary of the Application Classes 770
Steps for Creating a Custom Component 771
Creating the Component Tag Handler 772
Defining the Custom Component Tag in a Tag Library Descriptor 777
Creating Custom Component Classes 778
  Performing Encoding 780
  Performing Decoding 783
  Enabling Value-Binding of Component Properties 783
  Saving and Restoring State 785
Delegating Rendering to a Renderer 786
  Creating the Renderer Class 786
  Identifying the Renderer Type 788
Handling Events for Custom Components 788

Chapter 21 Configuring JavaServer Faces Applications ....... 791

Application Configuration Resource File 792
Configuring Beans 793
  Using the managed-bean Element 793
  Initializing Properties using the managed-property Element 795
  Initializing Maps and Lists 801
Registering Messages 802
Registering a Custom Validator 803
Registering a Custom Converter 804
Configuring Navigation Rules 805
CONTENTS

Registering a Custom Renderer with a Render Kit 808
Registering a Custom Component 810
Basic Requirements of a JavaServer Faces Application 811
  Configuring an Application Using deploytool 812
  Including the Required JAR Files 816
  Including the Classes, Pages, and Other Resources 817

Chapter 22  Internationalizing and Localizing Web Applications  819

Java Platform Localization Classes 819
Providing Localized Messages and Labels 820
  Establishing the Locale 821
  Setting the Resource Bundle 821
  Retrieving Localized Messages 822
Date and Number Formatting 823
Character Sets and Encodings 823
  Character Sets 823
  Character Encoding 824
Further Information 827

Chapter 23  Enterprise Beans  829

What Is an Enterprise Bean? 829
  Benefits of Enterprise Beans 830
  When to Use Enterprise Beans 830
  Types of Enterprise Beans 830
What Is a Session Bean? 831
  State Management Modes 831
  When to Use Session Beans 832
What Is an Entity Bean? 833
  What Makes Entity Beans Different from Session Beans? 833
  Container-Managed Persistence 835
  When to Use Entity Beans 838
What Is a Message-Driven Bean? 838
  What Makes Message-Driven Beans Different from Session and Entity Beans? 838
  When to Use Message-Driven Beans 840
Defining Client Access with Interfaces 840
  Remote Clients 840
  Local Clients 841
  Local Interfaces and Container-Managed Relationships 842
  Deciding on Remote or Local Access 842
# Contents

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web Service Clients</td>
<td>843</td>
</tr>
<tr>
<td>Method Parameters and Access</td>
<td>844</td>
</tr>
<tr>
<td>The Contents of an Enterprise Bean</td>
<td>845</td>
</tr>
<tr>
<td>Naming Conventions for Enterprise Beans</td>
<td>846</td>
</tr>
<tr>
<td>The Life Cycles of Enterprise Beans</td>
<td>847</td>
</tr>
<tr>
<td>The Life Cycle of a Stateful Session Bean</td>
<td>847</td>
</tr>
<tr>
<td>The Life Cycle of a Stateless Session Bean</td>
<td>848</td>
</tr>
<tr>
<td>The Life Cycle of an Entity Bean</td>
<td>848</td>
</tr>
<tr>
<td>The Life Cycle of a Message-Driven Bean</td>
<td>850</td>
</tr>
<tr>
<td>Further Information</td>
<td>851</td>
</tr>
</tbody>
</table>

## Chapter 24  Getting Started with Enterprise Beans

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creating the J2EE Application</td>
<td>854</td>
</tr>
<tr>
<td>Creating the Enterprise Bean</td>
<td>854</td>
</tr>
<tr>
<td>Coding the Enterprise Bean</td>
<td>854</td>
</tr>
<tr>
<td>Compiling the Source Files</td>
<td>856</td>
</tr>
<tr>
<td>Packaging the Enterprise Bean</td>
<td>857</td>
</tr>
<tr>
<td>Creating the Application Client</td>
<td>858</td>
</tr>
<tr>
<td>Coding the Application Client</td>
<td>859</td>
</tr>
<tr>
<td>Compiling the Application Client</td>
<td>861</td>
</tr>
<tr>
<td>Packaging the Application Client</td>
<td>861</td>
</tr>
<tr>
<td>Specifying the Application Client’s Enterprise Bean Reference</td>
<td>862</td>
</tr>
<tr>
<td>Creating the Web Client</td>
<td>863</td>
</tr>
<tr>
<td>Coding the Web Client</td>
<td>863</td>
</tr>
<tr>
<td>Compiling the Web Client</td>
<td>865</td>
</tr>
<tr>
<td>Packaging the Web Client</td>
<td>865</td>
</tr>
<tr>
<td>Specifying the Web Client’s Enterprise Bean Reference</td>
<td>866</td>
</tr>
<tr>
<td>Mapping the Enterprise Bean References</td>
<td>866</td>
</tr>
<tr>
<td>Specifying the Web Client’s Context Root</td>
<td>868</td>
</tr>
<tr>
<td>Deploying the J2EE Application</td>
<td>868</td>
</tr>
<tr>
<td>Running the Application Client</td>
<td>869</td>
</tr>
<tr>
<td>Running the Web Client</td>
<td>869</td>
</tr>
<tr>
<td>Modifying the J2EE Application</td>
<td>870</td>
</tr>
<tr>
<td>Modifying a Class File</td>
<td>870</td>
</tr>
<tr>
<td>Adding a File</td>
<td>871</td>
</tr>
<tr>
<td>Modifying a Deployment Setting</td>
<td>872</td>
</tr>
</tbody>
</table>

## Chapter 25  Session Bean Examples

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>The CartBean Example</td>
<td>873</td>
</tr>
<tr>
<td>Session Bean Class</td>
<td>874</td>
</tr>
<tr>
<td>Home Interface</td>
<td>878</td>
</tr>
</tbody>
</table>
## CONTENTS

Remote Interface 879  
Helper Classes 880  
Building the CartBean Example 880  
Creating the Application 880  
Packaging the Enterprise Bean 881  
Packaging the Application Client 882  

**A Web Service Example: HelloServiceBean** 884  
  Web Service Endpoint Interface 884  
  Stateless Session Bean Implementation Class 885  
  Building HelloServiceBean 886  
  Building the Web Service Client 888  
  Running the Web Service Client 889  

**Other Enterprise Bean Features** 889  
  Accessing Environment Entries 890  
  Comparing Enterprise Beans 891  
  Passing an Enterprise Bean’s Object Reference 891  

**Using the Timer Service** 892  
  Creating Timers 893  
  Canceling and Saving Timers 893  
  Getting Timer Information 894  
  Transactions and Timers 894  
  The TimerSessionBean Example 895  
  Building TimerSessionBean 896  

**Handling Exceptions** 901  

---  

**Chapter 26  Bean-Managed Persistence Examples** 903  

**The SavingsAccountBean Example** 903  
  Entity Bean Class 904  
  Home Interface 914  
  Remote Interface 916  
  Running the SavingsAccountBean Example 917  

**Mapping Table Relationships for Bean-Managed Persistence** 919  
  One-to-One Relationships 919  
  One-to-Many Relationships 922  
  Many-to-Many Relationships 930  

**Primary Keys for Bean-Managed Persistence** 933  
  The Primary Key Class 934  
  Primary Keys in the Entity Bean Class 935  
  Getting the Primary Key 936  

**deploytool Tips for Entity Beans with Bean-Managed Persistence** 937
### Chapter 27  Container-Managed Persistence Examples

#### Overview of the RosterApp Application .......................... 939

#### The PlayerBean Code

- Entity Bean Class .......................... 941
- Local Home Interface ..................... 946
- Local Interface .......................... 947

#### Method Invocations in RosterApp ......................... 947

- Creating a Player .......................... 948
- Adding a Player to a Team ................. 949
- Removing a Player .......................... 950
- Dropping a Player from a Team ......... 951
- Getting the Players of a Team .......... 952
- Getting a Copy of a Team’s Players ..... 953
- Finding the Players by Position ........ 955
- Getting the Sports of a Player .......... 956

#### Building and Running the RosterApp Example ............ 958

- Creating the Database Tables ........... 958
- Creating the Data Source ............... 959
- Capturing the Table Schema .......... 959
- Building the Enterprise Beans .......... 960
- Creating the Enterprise Application ... 960
- Packaging the Enterprise Beans ........ 961
- Packaging the Enterprise Application Client 970
- Deploying the Enterprise Application .... 971
- Running the Client Application ........ 971

#### A Guided Tour of the RosterApp Settings .................. 972

- RosterApp .................................. 972
- RosterClient ................................ 974
- RosterJAR .................................. 974
- TeamJAR .................................. 975

#### Primary Keys for Container-Managed Persistence ........ 980

- The Primary Key Class .................... 982

#### Advanced CMP Topics: The OrderApp Example ........... 984

- Structure of OrderApp .................... 984
- Bean Relationships in OrderApp ....... 985
- Primary Keys in OrderApp’s Entity Beans 987
- Entity Bean Mapped to More Than One Database Table .... 989
- Finder and Selector Methods .......... 990
- Using Home Methods ..................... 990
CONTENTS

Cascade Deletes in OrderApp 991
BLOB and CLOB Database Types in OrderApp 991
Building and Running the OrderApp Example 992
deploytool Tips for Entity Beans with Container-Managed Persistence 1000
  Selecting the Persistent Fields and Abstract Schema Name 1001
  Defining EJB QL Queries for Finder and Select Methods 1001
  Defining Relationships 1002
  Creating the Database Tables at Deploy Time in deploytool 1002

Chapter 28   A Message-Driven Bean Example  . . . . . . . . . . . . . . . . . . . . . . . . 1003

  Example Application Overview 1003
  The Application Client 1004
  The Message-Driven Bean Class 1005
    The onMessage Method 1005
    The ejbCreate and ejbRemove Methods 1007
  Deploying and Running SimpleMessageApp 1007
    Creating the Administered Objects 1007
    Deploying the Application 1009
    Running the Client 1009
    Removing the Administered Objects 1009
deploytool Tips for Message-Driven Beans 1010
  Specifying the Bean’s Type 1010
  Setting the Message-Driven Bean’s Characteristics 1010
deploytool Tips for Components That Send Messages 1011
  Setting the Resource References 1012
  Setting the Message Destination References 1012
  Setting the Message Destinations 1013

Chapter 29   Enterprise JavaBeans Query Language  . . . . . . . . . . . . . . . . . . . 1015

  Terminology 1016
  Simplified Syntax 1016
  Example Queries 1017
    Simple Finder Queries 1017
    Finder Queries That Navigate to Related Beans 1019
    Finder Queries with Other Conditional Expressions 1020
    Select Queries 1022
  Full Syntax 1022
    BNF Symbols 1023
    BNF Grammar of EJB QL 1023
## Contents

FROM Clause ........................................ 1027  
Path Expressions .................................. 1030  
WHERE Clause ...................................... 1032  
SELECT Clause ...................................... 1040  
ORDER BY Clause ................................... 1043  
EJB QL Restrictions ................................ 1044  

### Chapter 30 Transactions ................................ 1045  

What Is a Transaction? ................................ 1045  

**Container-Managed Transactions**  
Transaction Attributes ................................ 1046  
Rolling Back a Container-Managed Transaction ........ 1050  
Synchronizing a Session Bean’s Instance Variables .... 1052  
Compiling the BankBean Example ...................... 1053  
Packaging the BankBean Example ...................... 1053  
Methods Not Allowed in Container-Managed Transactions ... 1057  

**Bean-Managed Transactions**  
JDBC Transactions .................................... 1058  
Deploying and Running the WarehouseBean Example .... 1059  
Compiling the WarehouseBean Example .................. 1059  
Packaging the WarehouseBean Example .................. 1059  
JTA Transactions ..................................... 1063  
Deploying and Running the TellerBean Example ........ 1064  
Compiling the TellerBean Example ...................... 1064  
Packaging the TellerBean Example ...................... 1065  
Returning without Committing .......................... 1068  
Methods Not Allowed in Bean-Managed Transactions .... 1068  

**Summary of Transaction Options for Enterprise Beans**  
Transaction Timeouts .................................. 1070  
Isolation Levels ...................................... 1070  
Updating Multiple Databases ............................ 1071  
Transactions in Web Components ....................... 1073  

### Chapter 31 Resource Connections ....................... 1075  

JNDI Naming ........................................... 1075  
DataSource Objects and Connection Pools ............... 1077  
Database Connections ................................ 1078  
Coding a Database Connection .......................... 1078  
Specifying a Resource Reference ....................... 1079  
Creating a Data Source ................................ 1079
CONTENTS

Mail Session Connections 1080
  Running the ConfirmerBean Example 1082
URL Connections 1083
  Running the HTMLReaderBean Example 1085
Further Information 1086

Chapter 32 Security ................................. 1087

Overview 1087
Realms, Users, Groups, and Roles 1088
  Managing Users 1089
  Setting Up Security Roles 1090
  Mapping Roles to Users and Groups 1090
Web-Tier Security 1092
  Protecting Web Resources 1094
  Setting Security Requirements Using deploytool 1095
  Specifying a Secure Connection 1097
  Using Programmatic Security in the Web Tier 1098
Understanding Login Authentication 1100
  Using HTTP Basic Authentication 1100
  Using Form-Based Authentication 1101
  Using Client-Certificate Authentication 1102
  Using Mutual Authentication 1103
  Using Digest Authentication 1105
  Configuring Authentication 1105
  Example: Using Form-Based Authentication 1105
Installing and Configuring SSL Support 1114
  Understanding Digital Certificates 1115
  Configuring the SSL Connector 1122
XML and Web Services Security 1125
  Example: Basic Authentication with JAX-RPC 1126
  Example: Client-Certificate Authentication over HTTP/SSL with JAX-RPC 1133
EJB-Tier Security 1142
  Declaring Method Permissions 1143
  Configuring IOR Security 1143
  Using Programmatic Security in the EJB Tier 1145
  Unauthenticated User Name 1145
Application Client-Tier Security 1145
EIS-Tier Security 1147
  Container-Managed Sign-On 1147
  Component-Managed Sign-On 1148
## CONTENTS

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuring Resource Adapter Security</td>
<td>1148</td>
</tr>
<tr>
<td><strong>Propagating Security Identity</strong></td>
<td>1149</td>
</tr>
<tr>
<td>Configuring a Component’s Propagated Security Identity</td>
<td>1150</td>
</tr>
<tr>
<td>Configuring Client Authentication</td>
<td>1151</td>
</tr>
<tr>
<td><strong>What Is Java Authorization Contract for Containers?</strong></td>
<td>1151</td>
</tr>
<tr>
<td>Further Information</td>
<td>1152</td>
</tr>
</tbody>
</table>

### Chapter 33  The Java Message Service API  

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Overview</strong></td>
<td>1154</td>
</tr>
<tr>
<td>What Is Messaging?</td>
<td>1154</td>
</tr>
<tr>
<td>What Is the JMS API?</td>
<td>1154</td>
</tr>
<tr>
<td>When Can You Use the JMS API?</td>
<td>1155</td>
</tr>
<tr>
<td>How Does the JMS API Work with the J2EE Platform?</td>
<td>1156</td>
</tr>
<tr>
<td><strong>Basic JMS API Concepts</strong></td>
<td>1157</td>
</tr>
<tr>
<td>JMS API Architecture</td>
<td>1158</td>
</tr>
<tr>
<td>Messaging Domains</td>
<td>1159</td>
</tr>
<tr>
<td>Message Consumption</td>
<td>1161</td>
</tr>
<tr>
<td><strong>The JMS API Programming Model</strong></td>
<td>1162</td>
</tr>
<tr>
<td>Administered Objects</td>
<td>1163</td>
</tr>
<tr>
<td>Connections</td>
<td>1165</td>
</tr>
<tr>
<td>Sessions</td>
<td>1165</td>
</tr>
<tr>
<td>Message Producers</td>
<td>1166</td>
</tr>
<tr>
<td>Message Consumers</td>
<td>1167</td>
</tr>
<tr>
<td>Messages</td>
<td>1169</td>
</tr>
<tr>
<td>Exception Handling</td>
<td>1172</td>
</tr>
<tr>
<td><strong>Writing Simple JMS Client Applications</strong></td>
<td>1173</td>
</tr>
<tr>
<td>A Simple Example of Synchronous Message Receives</td>
<td>1174</td>
</tr>
<tr>
<td>A Simple Example of Asynchronous Message Consumption</td>
<td>1183</td>
</tr>
<tr>
<td>Running JMS Client Programs on Multiple Systems</td>
<td>1187</td>
</tr>
<tr>
<td><strong>Creating Robust JMS Applications</strong></td>
<td>1191</td>
</tr>
<tr>
<td>Using Basic Reliability Mechanisms</td>
<td>1193</td>
</tr>
<tr>
<td>Using Advanced Reliability Mechanisms</td>
<td>1200</td>
</tr>
<tr>
<td><strong>Using the JMS API in a J2EE Application</strong></td>
<td>1212</td>
</tr>
<tr>
<td>Using Session and Entity Beans to Produce and to Synchronously Receive Messages</td>
<td>1212</td>
</tr>
<tr>
<td>Using Message-Driven Beans</td>
<td>1214</td>
</tr>
<tr>
<td>Managing Distributed Transactions</td>
<td>1216</td>
</tr>
<tr>
<td>Using the JMS API with Application Clients and Web Components</td>
<td>1219</td>
</tr>
<tr>
<td>Further Information</td>
<td>1220</td>
</tr>
</tbody>
</table>
Chapter 34  J2EE Examples Using the JMS API . . . . . . . . . . . . 1221

A J2EE Application That Uses the JMS API with a Session Bean  1222
  Writing the Application Components  1223
  Creating and Packaging the Application  1225
  Deploying the Application  1229
  Running the Application Client  1230

A J2EE Application That Uses the JMS API with an Entity Bean  1231
  Overview of the Human Resources Application  1231
  Writing the Application Components  1232
  Creating and Packaging the Application  1235
  Deploying the Application  1237
  Running the Application Client  1238

An Application Example That Consumes Messages from a Remote J2EE Server  1239
  Overview of the Applications  1240
  Writing the Application Components  1241
  Creating and Packaging the Applications  1244
  Deploying the Applications  1244
  Running the Application Client  1245

An Application Example That Deploys a Message-Driven Bean on Two J2EE Servers  1246
  Overview of the Applications  1246
  Writing the Application Components  1248
  Creating and Packaging the Applications  1249
  Deploying the Applications  1251
  Running the Application Client  1252

Chapter 35  The Coffee Break Application . . . . . . . . . . . . . 1255

Common Code  1256

JAX-RPC Coffee Supplier Service  1257
  Service Interface  1257
  Service Implementation  1258
  Publishing the Service in the Registry  1259
  Deleting the Service from the Registry  1263

SAAJ Coffee Supplier Service  1265
  SAAJ Client  1266
  SAAJ Service  1274

Coffee Break Server  1280
  JSP Pages  1281
  JavaBeans Components  1282
  RetailPriceListServlet  1284
CONTENTS

JavaServer Faces Version of Coffee Break Server 1284
  JSP Pages 1285
  JavaBeans Components 1288
  Resource Configuration 1289
Building, Packaging, Deploying, and Running the Application 1290
  Setting the Port 1291
  Setting Up the Registry Server 1291
  Using the Provided WARs 1292
  Building the Common Classes 1292
  Building, Packaging, and Deploying the JAX-RPC Service 1292
  Building, Packaging, and Deploying the SAAJ Service 1294
  Building, Packaging, and Deploying the Coffee Break Server 1295
  Building, Packaging, and Deploying the JavaServer Faces Technology Coffee Break Server 1297
  Running the Coffee Break Client 1298
  Removing the Coffee Break Application 1300

Chapter 36 The Duke’s Bank Application ...................... 1301

Enterprise Beans 1302
  Session Beans 1303
  Entity Beans 1306
  Helper Classes 1307
  Database Tables 1308
  Protecting the Enterprise Beans 1309
Application Client 1310
  The Classes and Their Relationships 1311
  BankAdmin Class 1312
  EventHandle Class 1314
  DataModel Class 1315
Web Client 1318
  Design Strategies 1319
  Client Components 1320
  Request Processing 1323
  Protecting the Web Client Resources 1325
Internationalization 1327
Building, Packaging, Deploying, and Running the Application 1328
  Setting Up the Servers 1328
  Compiling the Duke’s Bank Application Code 1330
  Packaging and Deploying the Duke’s Bank Application 1330
  Reviewing JNDI Names 1335
CONTENTS

Running the Clients 1338
  Running the Application Client 1338
  Running the Web Client 1339

Appendix A Java Encoding Schemes ......................... 1341
  Further Information 1342

Appendix B XML and Related Specs: Digesting the Alphabet Soup 1343
  Basic Standards 1344
    SAX 1344
    StAX 1345
    DOM 1345
    JDOM and dom4j 1345
    DTD 1346
    Namespaces 1347
    XSL 1347
    XSLT (+XPath) 1347
  Schema Standards 1348
    XML Schema 1349
    RELAX NG 1349
    SOX 1349
    Schematron 1350
  Linking and Presentation Standards 1350
    XML Linking 1350
    XHTML 1351
  Knowledge Standards 1351
    RDF 1351
    RDF Schema 1352
    XTM 1352
  Standards That Build on XML 1352
    Extended Document Standards 1353
    e-Commerce Standards 1353
  Summary 1354

Appendix C HTTP Overview ................................. 1355
  HTTP Requests 1356
  HTTP Responses 1356
## CONTENTS

Appendix D  J2EE Connector Architecture  ................. 1357

- About Resource Adapters ............................... 1357
- Resource Adapter Contracts ........................... 1359
  - Management Contracts ............................... 1359
  - Outbound Contracts ................................ 1360
  - Inbound Contracts .................................. 1361
- Common Client Interface ............................... 1362
- Further Information ................................... 1363

Glossary ..................................................... 1365

About the Authors ......................................... 1403

Index ......................................................... 1405
Foreword

When the first edition of *The J2EE™ Tutorial* was released, the Java™ 2 Platform, Enterprise Edition (J2EE) was the new kid on the block. Modeled after its forerunner, the Java 2 Platform, Standard Edition (J2SE™), the J2EE platform brought the benefits of “Write Once, Run Anywhere™” API compatibility to enterprise application servers. Now at version 1.4 and with widespread conformance in the application server marketplace, the J2EE platform has firmly established its position as the standard for enterprise application servers.

*The J2EE™ Tutorial, Second Edition* covers the J2EE 1.4 platform and more. If you have used the first edition of *The J2EE™ Tutorial*, you may notice that the second edition is triple the size. This reflects a major expansion in the J2EE platform and the availability of two upcoming J2EE technologies in the Sun Java System Application Server Platform Edition 8, the software on which the tutorial is based.

One of the most important additions to the J2EE 1.4 platform is substantial support for Web services with the JAX-RPC 1.1 API, which enables Web service endpoints based on servlets and enterprise beans. The platform also contains Web services support APIs for handling XML data streams directly (SAAJ) and for accessing Web services registries (JAXR). In addition, the J2EE 1.4 platform requires WS-I Basic Profile 1.0. This means that in addition to platform independence and complete Web services support, the J2EE 1.4 platform offers Web services interoperability.

The J2EE 1.4 platform contains major enhancements to the Java servlet and JavaServer Pages (JSP) technologies that are the foundation of the Web tier. The tutorial also showcases two exciting new technologies, not required by the J2EE 1.4 platform, that simplify the task of building J2EE application user interfaces: JavaServer Pages Standard Tag Library (JSTL) and JavaServer Faces. These new
technologies are available in the Sun Java System Application Server. They will soon be featured in new developer tools and are strong candidates for inclusion in the next version of the J2EE platform.

Readers conversant with the core J2EE platform enterprise bean technology will notice major upgrades with the addition of the previously mentioned Web service endpoints, as well as a timer service, and enhancements to EJB QL and message-driven beans.

With all of these new features, I believe that you will find it well worth your time and energy to take on the J2EE 1.4 platform. You can increase the scope of the J2EE applications you develop, and your applications will run on the widest possible range of application server products.

To help you to learn all about the J2EE 1.4 platform, *The J2EE™ Tutorial, Second Edition* follows the familiar Java Series tutorial model of concise descriptions of the essential features of each technology with code examples that you can deploy and run on the Sun Java System Application Server. Read this tutorial and you will become part of the next wave of J2EE application developers.

Jeff Jackson  
Vice President, J2EE Platform and Application Servers  
Sun Microsystems  
Santa Clara, CA  
Thursday, June 3, 2004
About This Tutorial

The J2EE™ Tutorial, Second Edition is a guide to developing enterprise applications for the Java 2 Platform, Enterprise Edition (J2EE) version 1.4. Here we cover all the things you need to know to make the best use of this tutorial.

Who Should Use This Tutorial

This tutorial is intended for programmers who are interested in developing and deploying J2EE 1.4 applications on the Sun Java System Application Server Platform Edition 8.

Prerequisites

Before proceeding with this tutorial you should have a good knowledge of the Java programming language. A good way to get to that point is to work through all the basic and some of the specialized trails in The Java™ Tutorial, Mary Campione et al. (Addison-Wesley, 2000). In particular, you should be familiar with relational database and security features described in the trails listed in Table 1.

Table 1  Prerequisite Trails in The Java™ Tutorial

<table>
<thead>
<tr>
<th>Trail</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>JDBC</td>
<td><a href="http://java.sun.com/docs/books/tutorial/jdbc">http://java.sun.com/docs/books/tutorial/jdbc</a></td>
</tr>
<tr>
<td>Security</td>
<td><a href="http://java.sun.com/docs/books/tutorial/security1.2">http://java.sun.com/docs/books/tutorial/security1.2</a></td>
</tr>
</tbody>
</table>
How to Read This Tutorial

The J2EE 1.4 platform is quite large, and this tutorial reflects this. However, you don’t have to digest everything in it at once.

This tutorial opens with three introductory chapters, which you should read before proceeding to any specific technology area. Chapter 1 covers the J2EE 1.4 platform architecture and APIs along with the Sun Java System Application Server Platform Edition 8. Chapters 2 and 3 cover XML basics and getting started with Web applications.

When you have digested the basics, you can delve into one or more of the four main technology areas listed next. Because there are dependencies between some of the chapters, Figure 1 contains a roadmap for navigating through the tutorial.

- The Java XML chapters cover the technologies for developing applications that process XML documents and implement Web services components:
  - The Java API for XML Processing (JAXP)
  - The Java API for XML-based RPC (JAX-RPC)
  - SOAP with Attachments API for Java (SAAJ)
  - The Java API for XML Registries (JAXR)

- The Web-tier technology chapters cover the components used in developing the presentation layer of a J2EE or stand-alone Web application:
  - Java Servlet
  - JavaServer Pages (JSP)
  - JavaServer Pages Standard Tag Library (JSTL)
  - JavaServer Faces
  - Web application internationalization and localization

- The Enterprise JavaBeans (EJB) technology chapters cover the components used in developing the business logic of a J2EE application:
  - Session beans
  - Entity beans
  - Message-driven beans
  - Enterprise JavaBeans Query Language
The platform services chapters cover the system services used by all the J2EE component technologies:

- Transactions
- Resource connections
- Security
- Java Message Service
After you have become familiar with some of the technology areas, you are ready to tackle the case studies, which tie together several of the technologies discussed in the tutorial. The Coffee Break Application (Chapter 35) describes an application that uses the Web application and Web services APIs. The Duke’s Bank Application (Chapter 36) describes an application that employs Web application technologies and enterprise beans.

Finally, the appendixes contain auxiliary information helpful to the J2EE application developer along with a brief summary of the J2EE Connector architecture:

- Java Encoding Schemes (Appendix A)
- XML standards (Appendix B)
- HTTP Overview (Appendix C)
- J2EE Connector Architecture (Appendix D)

**About the Examples**

This section tells you everything you need to know to install, build, and run the examples.

**Required Software**

**Tutorial Bundle**

The tutorial example source is contained in the tutorial bundle, which is distributed on the accompanying CD-ROM.

After you have installed the tutorial bundle, the example source code is in the `<INSTALL>/j2eetutorial14/examples/` directory, with subdirectories for each of the technologies discussed in the tutorial.

**Application Server**

The Sun Java System Application Server Platform Edition 8 is targeted as the build and runtime environment for the tutorial examples. To build, deploy, and run the examples, you need a copy of the Application Server and the Java 2 Software Development Kit, Standard Edition (J2SE SDK) 1.4.2_04 or higher.
The Application Server and J2SE SDK are contained in the J2EE 1.4 SDK, which is distributed on the CD-ROM accompanying the tutorial.

**Application Server Installation Tips**

In the Admin configuration pane of the Application Server installer,

- Select the Don’t Prompt for Admin User Name radio button. This will save the user name and password so that you won’t need to provide them when performing administrative operations with asadmin and deploytool. You will still have to provide the user name and password to log in to the Admin Console.

- Note the HTTP port at which the server is installed. This tutorial assumes that you are accepting the default port of 8080. If 8080 is in use during installation and the installer chooses another port, or if you decide to change it yourself, you will need to update the common build properties file (described in the next section) and the configuration files for some of the tutorial examples to reflect the correct port.

In the Installation Options pane, check the Add Bin Directory to PATH checkbox so that Application Server scripts (asadmin, asant, deploytool, and wscompile) override other installations.

**Registry Server**

You need a registry server to run the examples discussed in Chapters 10 and 35. Directions for obtaining and setting up a registry server are provided in those chapters.

**Building the Examples**

Most of the tutorial examples are distributed with a configuration file for asant, a portable build tool contained in the Application Server. This tool is an extension of the Ant tool developed by the Apache Software Foundation (http://ant.apache.org). The asant utility contains additional tasks that invoke the Application Server administration utility asadmin. Directions for building the examples are provided in each chapter.

Build properties and targets common to all the examples are specified in the files `<INSTALL>/j2eetutorial14/examples/common/build.properties` and
ABOUT THIS TUTORIAL

<INSTALL>/j2eetutorial14/examples/common/targets.xml. Build properties and targets common to a particular technology are specified in the files <INSTALL>/j2eetutorial14/examples/tech/common/build.properties and <INSTALL>/j2eetutorial14/examples/tech/common/targets.xml.

To run the asant scripts, you must set two common build properties as follows:

- Set the j2ee.home property in the file <INSTALL>/j2eetutorial14/examples/common/build.properties to the location of your Application Server installation. The build process uses the j2ee.home property to include the libraries in <J2EE_HOME>/lib/ in the classpath. All examples that run on the Application Server include the J2EE library archive— <J2EE_HOME>/lib/j2ee.jar—in the build classpath. Some examples use additional libraries in <J2EE_HOME>/lib/ and <J2EE_HOME>/lib/endorsed/; the required libraries are enumerated in the individual technology chapters. <J2EE_HOME> refers to the directory where you have installed the Application Server or the J2EE 1.4 SDK.

**Note:** On Windows, you must escape any backslashes in the j2ee.home property with another backslash or use forward slashes as a path separator. So, if your Application Server installation is C:\Sun\AppServer, you must set j2ee.home as follows:

j2ee.home = C:\Sun\AppServer

or

j2ee.home=C:/Sun/AppServer

- If you did not use port 8080 when you installed the Application Server, set the value of the domain.resources.port property in <INSTALL>/j2eetutorial14/examples/common/build.properties to the correct port.

**Tutorial Example Directory Structure**

To facilitate iterative development and keep application source separate from compiled files, the source code for the tutorial examples is stored in the following structure under each application directory:
Typographical Conventions

Table 2 lists the typographical conventions used in this tutorial.

<table>
<thead>
<tr>
<th>Font Style</th>
<th>Uses</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>italic</em></td>
<td>Emphasis, titles, first occurrence of terms</td>
</tr>
<tr>
<td><em>monospace</em></td>
<td>URLs; code examples; file names; path names; tool names; application names; programming language keywords; tag, interface, class, method, and field names; properties</td>
</tr>
<tr>
<td><em>italic monospace</em></td>
<td>Variables in code, file paths, and URLs</td>
</tr>
<tr>
<td>&lt;italic monospace&gt;</td>
<td>User-selected file path components</td>
</tr>
</tbody>
</table>
Menu selections indicated with the right-arrow character →, for example, First→Second, should be interpreted as: select the First menu, then choose Second from the First submenu.

Acknowledgments

The J2EE tutorial team would like to thank the J2EE specification leads: Bill Shannon, Pierre Delisle, Mark Roth, Yutaka Yoshida, Farrukh Najmi, Phil Goodwin, Joseph Fialli, Kate Stout, and Ron Monzillo and the J2EE 1.4 SDK team members: Vivek Nagar, Tony Ng, Qingqing Ouyang, Ken Saks, Jean-Francois Arcand, Jan Luehe, Ryan Lubke, Kathy Walsh, Binod P G, Alejandro Murillo, and Manveen Kaur.

The chapters on custom tags and the Coffee Break and Duke’s Bank applications use a template tag library that first appeared in Designing Enterprise Applications with the J2EE™ Platform, Second Edition, Inderjeet Singh et al., (Addison-Wesley, 2002).

The JavaServer Faces technology and JSP Documents chapters benefited greatly from the invaluable documentation reviews and example code contributions of these engineers: Ed Burns, Justyna Horwat, Roger Kitain, Jan Luehe, Craig McClanahan, Raj Premkumar, Mark Roth, and especially Jayashri Visvanathan.

The OrderApp example application described in the Container-Managed Persistence chapter was coded by Marina Vatkina with contributions from Markus Fuchs, Rochelle Raccah, and Deepa Singh. Ms. Vatkina’s JDO/CMP team provided extensive feedback on the tutorial’s discussion of CMP.

The security chapter writers are indebted to Raja Perumal, who was a key contributor both to the chapter and to the examples.

Monica Pawlan and Beth Stearns wrote the Overview and J2EE Connector chapters in the first edition of The J2EE Tutorial and much of that content has been carried forward to the current edition.

We are extremely grateful to the many internal and external reviewers who provided feedback on the tutorial. Their feedback helped improve the technical accuracy and presentation of the chapters and eliminate bugs from the examples.

We would like to thank our manager, Alan Sommerer, for his support and steadying influence.
We also thank Duarte Design, Inc., and Zana Vartanian for developing the illustrations in record time. Thanks are also due to our copy editor, Betsy Hardinger, for helping this multi-author project achieve a common style.

Finally, we would like to express our profound appreciation to Ann Sellers, Elizabeth Ryan, and the production team at Addison-Wesley for graciously seeing our large, complicated manuscript to publication.

Feedback

To send comments, broken link reports, errors, suggestions, and questions about this tutorial to the tutorial team, please use the feedback form at http://java.sun.com/j2ee/1.4/docs/tutorial/information/sendusmail.html.
Using JavaServer Faces Technology in JSP Pages

The page author’s responsibility is to design the pages of a JavaServer Faces application. This includes laying out the components on the page and wiring them to backing beans, validators, converters, and other back-end objects associated with the page. This chapter uses the Duke’s Bookstore application and the Coffee Break application (see Chapter 35) to describe how page authors use the JavaServer Faces tags to

- Layout standard UI components on a page
- Reference localized messages
- Register converters, validators, and listeners on components
- Bind components and their values to back-end objects
- Reference backing bean methods that perform navigation processing, handle events, and perform validation

This chapter also describes how to include custom objects created by application developers and component writers on a JSP page.
The Example JavaServer Faces Application

The JavaServer Faces technology chapters of this tutorial primarily use a rewritten version of the Duke’s Bookstore example to illustrate the basic concepts of JavaServer Faces technology. This version of the Duke’s Bookstore example includes several JavaServer Faces technology features:

- The JavaServer Faces implementation provides FacesServlet, whose instances accept incoming requests and pass them to the implementation for processing. Therefore, the application does not need to include a servlet (such as the Dispatcher servlet) that processes request parameters and dispatches to application logic, as do the other versions of Duke’s Bookstore.
- A custom image map component that allows you to select the locale for the application.
- Navigation configured in a centralized application configuration resource file. This eliminates the need to calculate URLs, as other versions of the Duke’s Bookstore application must do.
- Backing beans associated with the pages. These beans hold the component data and perform other processing associated with the components. This processing includes handling the event generated when a user clicks a button or hyperlink.
- Tables that display the books from the database and the shopping cart are rendered with the dataTable tag, which is used to dynamically render data in a table. The dataTable tag on bookshowcart.jsp also includes input components.
- A custom validator and a custom converter are registered on the credit card field of the bookcashier.jsp page.
- A value-change listener is registered on the Name field of bookcashier.jsp. This listener saves the name in a parameter so that bookreceipt.jsp can access it.

This version of Duke’s Bookstore includes the same pages listed in Table 12–1. It also includes the chooselocale.jsp page, which displays the custom image map that allows you to select the locale of the application. This page is displayed first and advances directly to the bookstore.jsp page after the locale is selected.
The packages of the Duke’s Bookstore application are:

- backing: Includes the backing bean classes
- components: Includes the custom UI component classes
- converters: Includes the custom converter class
- listeners: Includes the event handler and event listener classes
- model: Includes a model bean class
- renderers: Includes the custom renderers
- resources: Includes custom error messages for the custom converter and validator
- taglib: Includes custom tag handler classes
- util: Includes a message factory class
- validators: Includes a custom validator class

Chapter 19 describes how to program backing beans, custom converters and validators, and event listeners. Chapter 20 describes how to program event handlers, custom components, renderers, and tag handlers.

The source code for the application is located in the `<INSTALL>/j2ee-tutorial14/examples/web/bookstore6/` directory. A sample bookstore6.war is provided in `<INSTALL>/j2ee-tutorial14/examples/web/provided-wars/`

To build, package, deploy, and run the example, follow these steps:

1. Build and package the bookstore common files as described in Duke’s Bookstore Examples (page 100).
2. Go to `<INSTALL>/j2ee-tutorial14/examples/web/bookstore6/` and run `asant build`.
4. Perform all the operations described in Accessing Databases from Web Applications, page 100.
5. Start `deploytool`.
7. In the New Web Component wizard:
   a. Select the Create New Stand-Alone WAR Module radio button.
   b. In the WAR Location field, enter `<INSTALL>/j2ee-tutorial14/examples/web/bookstore6.war`.
   c. In the WAR Name field, enter `bookstore6`. 
d. In the Context Root field, enter /bookstore6.
e. Click Edit Contents.
f. In the Edit Contents dialog box, navigate to <INSTALL>/j2ee-tutorial14/examples/web/bookstore6/build/. Select everything in the build directory and click Add.
g. In the Contents tree, drag the resources package to the WEB-INF/classes directory.
h. In the Contents tree, drag faces-config.xml to the WEB-INF directory.
i. In the Edit Contents dialog box, navigate to <INSTALL>/j2ee-tutorial14/examples/web/bookstore/dist/. Select bookstore.jar and click Add.
j. In the Edit Contents dialog box, navigate to <J2EE_HOME>/lib/ and select the jsf-api.jar. Click Add, and then Click OK.
k. Click Next.
l. Select the Servlet radio button.
m. Click Next.

n. Select javax.faces.webapp.FacesServlet from the Servlet Class combo box.
o. In the Startup Load Sequence Position combo box, enter 1.
p. Click Finish.

8. Provide a mapping for the FacesServlet Web component.
a. Select the FacesServlet Web component that is contained in the bookstore6 Web application from the tree.
b. Select the Aliases tab.
c. Click Add and enter *.faces in the Aliases field.

9. Specify where state is saved.
a. Select the bookstore6 WAR from the tree.
b. Select the Context tabbed pane and click Add.
c. Enter javax.faces.STATE_SAVING_METHOD in the Coded Parameter field.
d. Enter client in the Value field.

10. Set preludes and codas for all JSP pages.
a. Select the JSP Properties tab.
b. Click Add.
c. Enter bookstore6 in the Name field.
d. Click Add URL.
e. Enter *.jsp in the URL Patterns field.
f. Click Edit Preludes.
g. Click Add.
h. Enter /template/prelude.jspf.
i. Click OK.
j. Click Edit Codas.
k. Click Add.
l. Enter /template/coda.jspf.
m. Click OK.

11. Add the listener class listeners.ContextListener (described in Handling Servlet Life-Cycle Events, page 441).
a. Select the Event Listeners tab.
b. Click Add.
c. Select the listeners.ContextListener class from the drop-down menu in the Event Listener Classes pane.

12. Add a resource reference for the database.
a. Select the Resource Ref's tab.
b. Click Add.
c. Enter jdbc/BookDB in the Coded Name field.
d. Accept the default type javax.sql.DataSource.
e. Accept the default authorization Container.
f. Accept the default selected Shareable.
g. Enter jdbc/BookDB in the JNDI Name field of the Sun-specific Settings frame.

13. Select File→Save.

14. Deploy the application.

15. Select Tools→Deploy.

16. In the Connection Settings frame, enter the user name and password you specified when you installed the Application Server.

17. Click OK.

18. A pop-up dialog box will display the results of the deployment. Click Close.

Setting Up a Page

To use the JavaServer Faces UI components in your JSP page, you need to give the page access to the two tag libraries: the JavaServer Faces standard HTML render kit tag library and the JavaServer Faces core tag library. The JavaServer Faces standard HTML render kit tag library defines tags that represent common HTML user interface components. The JavaServer Faces core tag library defines tags that perform core actions and are independent of a particular render kit.

Using these tag libraries is similar to using any other custom tag library. This chapter assumes that you are familiar with the basics of using custom tags in JSP pages (see Using Custom Tags, page 502).

As is the case with any tag library, each JavaServer Faces tag library must have a TLD that describes it. The html_basic TLD describes the The JavaServer Faces standard HTML render kit tag library. The jsf_core TLD describes the JavaServer Faces core tag library.

Please refer to the TLD documentation at http://java.sun.com/j2ee/javaserverfaces/1.0/docs/tlddocs/index.html for a complete list of the JavaServer Faces tags and their attributes.

Your application needs access to these TLDs in order for your pages to use them. The Application Server includes these TLDs in jsf-impl.jar, located in <J2EE_HOME>/lib.

To use any of the JavaServer Faces tags, you need to include these taglib directives at the top of each page containing the tags defined by these tag libraries:

```jsp
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
```

The uri attribute value uniquely identifies the TLD. The prefix attribute value is used to distinguish tags belonging to the tag library. You can use other prefixes rather than the h or f prefixes. However, you must use the prefix you have chosen when including the tag in the page. For example, the form tag must be referenced in the page via the h prefix because the preceding tag library directive uses the h prefix to distinguish the tags defined in html_basic.tld:

```jsp
<h:form ...>
```

A page containing JavaServer Faces tags is represented by a tree of components. At the root of the tree is the UIViewRoot component. The view tag represents
this component on the page. Thus, all component tags on the page must be enclosed in the view tag, which is defined in the jsf_core TLD:

```
<f:view>
    ... other faces tags, possibly mixed with other content ...
</f:view>
```

You can enclose other content, including HTML and other JSP tags, within the view tag, but all JavaServer Faces tags must be enclosed within the view tag.

The view tag has an optional locale attribute. If this attribute is present, its value overrides the Locale stored in the UIViewRoot. This value is specified as a String and must be of this form:

```
:language:[:country:][:variant]
```

The :language:, :country:, and :variant: parts of the expression are as specified in java.util.Locale.

A typical JSP page includes a form, which is submitted when a button or hyperlink on the page is clicked. For the data of other components on the page to be submitted with the form, the tags representing the components must be nested inside the form tag. See The UIForm Component (page 683) for more details on using the form tag.

If you want to include a page containing JavaServer Faces tags within another JSP page (which could also contain JavaServer Faces tags), you must enclose the entire nested page in a subview tag. You can add the subview tag on the parent page and nest a jsp:include inside it to include the page:

```
<f:subview id="myNestedPage">
    <jsp:include page="theNestedPage.jsp"/>
</f:subview>
```

You can also include the subview tag inside the nested page, but it must enclose all the JavaServer Faces tags on the nested page.

In summary, a typical JSP page that uses JavaServer Faces tags will look something like this:

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>

<f:view>
```
<h:form>
    other JavaServer Faces tags and core tags, including one or more button or hyperlink components for submitting the form
</h:form>
</f:view>

The sections Using the Core Tags (page 678) and Using the HTML Component Tags (page 680) describe how to use the core tags from the JavaServer Faces core tag library and the component tags from the JavaServer Faces standard HTML render kit tag library.

## Using the Core Tags

The tags included in the JavaServer Faces core tag library are used to perform core actions that are independent of a particular render kit. These tags are listed in Table 18–1.

### Table 18–1  The jsf_core Tags

<table>
<thead>
<tr>
<th>Tag Categories</th>
<th>Tags</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event-handling tags</td>
<td>actionListener</td>
<td>Registers an action listener on a parent component</td>
</tr>
<tr>
<td></td>
<td>valueChangeListener</td>
<td>Registers a value-change listener on a parent component</td>
</tr>
<tr>
<td>Attribute configuration tag</td>
<td>attribute</td>
<td>Adds configurable attributes to a parent component</td>
</tr>
<tr>
<td>Data conversion tags</td>
<td>converter</td>
<td>Registers an arbitrary converter on the parent component</td>
</tr>
<tr>
<td></td>
<td>convertDateTime</td>
<td>Registers a DateTime converter instance on the parent component</td>
</tr>
<tr>
<td></td>
<td>convertNumber</td>
<td>Registers a Number converter instance on the parent component</td>
</tr>
<tr>
<td>Facet tag</td>
<td>facet</td>
<td>Signifies a nested component that has a special relationship to its enclosing tag</td>
</tr>
</tbody>
</table>
These tags are used in conjunction with component tags and are therefore explained in other sections of this tutorial. Table 18–2 lists the sections that explain how to use specific jsf_core tags.
Using the HTML Component Tags

The tags defined by the JavaServer Faces standard HTML render kit tag library represent HTML form controls and other basic HTML elements. These controls display data or accept data from the user. This data is collected as part of a form and is submitted to the server, usually when the user clicks a button. This section explains how to use each of the component tags shown in Table 17–2, and is organized according to the `UIComponent` classes from which the tags are derived.

The next section explains the more important tag attributes that are common to most component tags. Please refer to the TLD documentation at http://java.sun.com/j2ee/javaserverfaces/1.0/docs/tlddocs/index.html for a complete list of tags and their attributes.

For each of the components discussed in the following sections, Writing Component Properties (page 730) explains how to write a bean property bound to a particular UI component or its value.
UI Component Tag Attributes

In general, most of the component tags support these attributes:

- **id**: Uniquely identifies the component.
- **immediate**: If set to `true`, indicates that any events, validation, and conversion associated with the component should happen in the apply request values phase rather than a later phase.
- **rendered**: Specifies a condition in which the component should be rendered. If the condition is not satisfied, the component is not rendered.
- **style**: Specifies a Cascading Style Sheet (CSS) style for the tag.
- **styleClass**: Specifies a CSS stylesheet class that contains definitions of the styles.
- **value**: Identifies an external data source and binds the component’s value to it.
- **binding**: Identifies a bean property and binds the component instance to it.

All of the UI component tag attributes (except `id` and `var`) are value-binding-enabled, which means that they accept JavaServer Faces EL expressions. These expressions allow you to use mixed literals and JSP 2.0 expression language syntax and operators. See Expression Language (page 489) for more information about the JSP 2.0 expression language.

The id Attribute

The `id` attribute is not required for a component tag except in these situations:

- Another component or a server-side class must refer to the component.
- The component tag is impacted by a JSTL conditional or iterator tag (for more information, see Flow Control Tags, page 545).

If you don’t include an `id` attribute, the JavaServer Faces implementation automatically generates a component ID.

The immediate Attribute

UIInput components and command components (those that implement Action-Source, such as buttons and hyperlinks) can set the `immediate` attribute to `true` to force events, validations, and conversions to be processed during the apply request values phase of the life cycle. Page authors need to carefully consider
how the combination of an input component’s immediate value and a command component’s immediate value determines what happens when the command component is activated.

Assume that you have a page with a button and a field for entering the quantity of a book in a shopping cart. If both the button’s and the field’s immediate attributes are set to true, the new value of the field will be available for any processing associated with the event that is generated when the button is clicked. The event associated with the button and the event, validation, and conversion associated with the field are all handled during the apply request values phase.

If the button’s immediate attribute is set to true but the field’s immediate attribute is set to false, the event associated with the button is processed without updating the field’s local value to the model layer. This is because any events, conversion, or validation associated with the field occurs during its usual phases of the life cycle, which come after the apply request values phase.

The bookshowcart.jsp page of the Duke’s Bookstore application has examples of components using the immediate attribute to control which component’s data is updated when certain buttons are clicked. The quantity field for each book has its immediate attribute set to false. (The quantity fields are generated by the UIData component. See The UIData Component, page 686, for more information.) The immediate attribute of the Continue Shopping hyperlink is set to true. The immediate attribute of the Update Quantities hyperlink is set to false.

If you click the Continue Shopping hyperlink, none of the changes entered into the quantity input fields will be processed. If you click the Update Quantities hyperlink, the values in the quantity fields will be updated in the shopping cart.

The rendered Attribute

A component tag uses a Boolean JavaServer Faces (EL) expression, along with the rendered attribute, to determine whether or not the component will be rendered. For example, the check commandLink component on the bookcatalog.jsp page is not rendered if the cart contains no items:

```xml
<h:commandLink id="check"
    rendered="#{cart.numberOfItems > 0}"
    <h:outputText
        value="#{bundle.CartCheck}"/>
</h:commandLink>
```
The style and styleClass Attributes

The style and styleClass attributes allow you to specify Cascading Style Sheets (CSS) styles for the rendered output of your component tags. The UIMessage and UIMessages Components (page 698) describes an example of using the style attribute to specify styles directly in the attribute. A component tag can instead refer to a CSS stylesheet class. The dataTable tag on the bookcatalog.jsp page of the Duke’s Bookstore application references the style class list-background:

```xml
<h:dataTable id="books"
... styleClass="list-background"
value="#{bookDBAO.books}"
var="book">
```

The stylesheet that defines this class is stylesheet.css, which is included in the application. For more information on defining styles, please see the Cascading Style Sheets Specification at http://www.w3.org/Style/CSS/.

The value and binding Attributes

A tag representing a component defined by UIOutput or a subclass of UIOutput uses value and binding attributes to bind its component’s value or instance to an external data source. Binding Component Values and Instances to External Data Sources (page 714) explains how to use these attributes.

The UIForm Component

A UIForm component is an input form that has child components representing data that is either presented to the user or submitted with the form. The form tag encloses all the controls that display or collect data from the user. Here is an example:

```xml
<h:form>
... other faces tags and other content...
</h:form>
```

The form tag can also include HTML markup to lay out the controls on the page. The form tag itself does not perform any layout; its purpose is to collect data and to declare attributes that can be used by other components in the form. A page
can include multiple form tags, but only the values from the form that the user submits will be included in the postback.

The **UIColumn Component**

The UIColumn component represents a column of data in a UIData component. While the UIData component is iterating over the rows of data, it processes the UIColumn for each row. UIColumn has no renderer associated with it and is represented on the page with a column tag. Here is an example column tag from the bookshowcart.jsp page of the Duke’s Bookstore example:

```html
<h:dataTable id="items"
...  value="#{cart.items}"
  var="item">
...  
    <h:column>
      <f:facet name="header">
        <h:outputText value="#{bundle.ItemQuantity}"/>
      </f:facet>
      <h:inputText
        ...  value="#{item.quantity}">
        <f:validateLongRange minimum="1"/>
      </h:inputText>
    </h:column>
...  
</h:dataTable>
```

The UIData component in this example iterates through the list of books (cart.items) in the shopping cart and displays their titles, authors, and prices. The column tag shown in the example renders the column that displays text fields that allow customers to change the quantity of each book in the shopping cart. Each time UIData iterates through the list of books, it renders one cell in each column.

The **UICommand Component**

The UICommand component performs an action when it is activated. The most common example of such a component is the button. This release supports Button and Link as UICommand component renderers.
In addition to the tag attributes listed in Using the HTML Component Tags (page 680), the commandButton and commandLink tags can use these attributes:

- **action**, which is either a logical outcome String or a method-binding expression that points to a bean method that returns a logical outcome String. In either case, the logical outcome String is used by the default NavigationHandler instance to determine what page to access when the UICommand component is activated.

- **actionListener**, which is a method-binding expression that points to a bean method that processes an ActionEvent fired by the UICommand component.

See Referencing a Method That Performs Navigation (page 720) for more information on using the action attribute.

See Referencing a Method That Handles an ActionEvent (page 721) for details on using the actionListener attribute.

### Using the commandButton Tag

The bookcashier.jsp page of the Duke’s Bookstore application includes a commandButton tag. When a user clicks the button, the data from the current page is processed, and the next page is opened. Here is the commandButton tag from bookcashier.jsp:

```xml
<h:commandButton value="#{bundle.Submit}"
    action="#{cashier.submit}"/>
```

Clicking the button will cause the submit method of CashierBean to be invoked because the action attribute references the submit method of the CashierBean backing bean. The submit method performs some processing and returns a logical outcome. This is passed to the default NavigationHandler, which matches the outcome against a set of navigation rules defined in the application configuration resource file.

The value attribute of the preceding example commandButton tag references the localized message for the button’s label. The bundle part of the expression refers to the ResourceBundle that contains a set of localized messages. The Submit part of the expression is the key that corresponds to the message that is displayed on the button. For more information on referencing localized messages, see Using Localized Messages (page 703). See Referencing a Method That Performs Navigation (page 720) for information on how to use the action attribute.
Using the commandLink Tag

The commandLink tag represents an HTML hyperlink and is rendered as an HTML `<a>` element. The commandLink tag is used to submit an action event to the application. See Implementing Action Listeners (page 749) for more information on action events.

A commandLink tag must include a nested outputText tag, which represents the text the user clicks to generate the event. The following tag is from the chooseLocale.jsp page from the Duke’s Bookstore application.

```xml
<h:commandLink id="NAmerica" action="bookstore"
    actionListener="#{localeBean.chooseLocaleFromLink}">
    <h:outputText value="#{bundle.English}" />
</h:commandLink>
```

This tag will render the following HTML:

```html
<a id="_id3:NAmerica" href="#"
    onclick="document.forms["_id3"][_id3:NAmerica].value='_id3:NAmerica';
    document.forms["_id3"].submit();
    return false;"">English</a>
```

**Note:** Notice that the commandLink tag will render JavaScript. If you use this tag, make sure your browser is JavaScript-enabled.

The UIData Component

The UIData component supports data binding to a collection of data objects. It does the work of iterating over each record in the data source. The standard Table renderer displays the data as an HTML table. The UIColumn component represents a column of data within the table. Here is a portion of the dataTable tag used by the bookshowcart.jsp page of the Duke’s Bookstore example:

```xml
<h:dataTable id="items"
    columnClasses="list-column-center, list-column-left,
    list-column-right, list-column-center"
    footerClass="list-footer"
    headerClass="list-header"
    rowClasses="list-row-even, list-row-odd"
    styleClass="list-background"
    value="#{cart.items}"
    var="item">
```
Figure 18–1 shows a data grid that this dataTable tag can display.

<table>
<thead>
<tr>
<th>Quantity</th>
<th>Title</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Web Servers for Fun and Profit</td>
<td>$40.75</td>
</tr>
<tr>
<td>1</td>
<td>The Green Project Programming for Consumer Devices</td>
<td>$30.00</td>
</tr>
<tr>
<td>1</td>
<td>Java Intermediate Bytecodes</td>
<td>$30.95</td>
</tr>
<tr>
<td>3</td>
<td>My Early Years: Growing up on *7</td>
<td>$30.75</td>
</tr>
<tr>
<td>4</td>
<td>Web Components for Web Developers</td>
<td>$27.75</td>
</tr>
<tr>
<td>2</td>
<td>Duke: A Biography of the Java Evangelist</td>
<td>$45.00</td>
</tr>
<tr>
<td>2</td>
<td>From Oak to Java: The Revolution of a Language</td>
<td>$10.75</td>
</tr>
</tbody>
</table>

**Subtotal:** $416.45

*Update Quantities*
The example `dataTable` tag displays the books in the shopping cart as well as the number of each book in the shopping cart, the prices, and a set of buttons, which the user can click to remove books from the shopping cart.

The `facet` tag inside the first `column` tag renders a header for that column. The other column tags also contain `facet` tags. Facets can have only one child, and so a `panelGroup` tag is needed if you want to group more than one component within a `facet`. Because the facet tag representing the footer includes more than one tag, the `panelGroup` is needed to group those tags.

A `facet` tag is usually used to represent headers and footers. In general, a `facet` is used to represent a component that is independent of the parent-child relationship of the page’s component tree. In the case of a data grid, header and footer data is not repeated like the other rows in the table, and therefore, the elements representing headers and footers are not updated as are the other components in the tree.

This example is a classic use case for a UIData component because the number of books might not be known to the application developer or the page author at the time the application is developed. The UIData component can dynamically adjust the number of rows of the table to accommodate the underlying data.

The `value` attribute of a `dataTable` tag references the data to be included in the table. This data can take the form of:

- A list of beans
- An array of beans
- A single bean
- A `javax.faces.model.DataModel`
- A `java.sql.ResultSet`
- A `javax.servlet.jsp.jstl.sql.ResultSet`
- A `javax.sql.RowSet`

All data sources for UIData components have a `DataModel` wrapper. Unless you explicitly construct a `DataModel`, the JavaServer Faces implementation will create a `DataModel` wrapper around data of any of the other acceptable types. See Writing Component Properties (page 730) for more information on how to write properties for use with a UIData component.

The `var` attribute specifies a name that is used by the components within the `dataTable` tag as an alias to the data referenced in the `value` attribute of `dataTable`.
In the `dataTable` tag from the `bookshowcart.jsp` page, the `value` attribute points to a list of books. The `var` attribute points to a single book in that list. As the `UIData` component iterates through the list, each reference to `item` points to the current book in the list.

The `UIData` component also has the ability to display only a subset of the underlying data. This is not shown in the preceding example. To display a subset of the data, you use the optional `first` and `rows` attributes.

The `first` attribute specifies the first row to be displayed. The `rows` attribute specifies the number of rows—starting with the first row—to be displayed. By default, both `first` and `rows` are set to zero, and this causes all the rows of the underlying data to display. For example, if you wanted to display records 2 through 10 of the underlying data, you would set `first` to 2 and `rows` to 9.

When you display a subset of the data in your pages, you might want to consider including a link or button that causes subsequent rows to display when clicked.

The `dataTable` tag also has a set of optional attributes for adding styles to the table:

- `columnClasses`: Defines styles for all the columns
- `footerClass`: Defines styles for the footer
- `headerClass`: Defines styles for the header
- `rowClasses`: Defines styles for the rows
- `styleClass`: Defines styles for the entire table

Each of these attributes can specify more than one style. If `columnClasses` or `rowClasses` specifies more than one style, the styles are applied to the columns or rows in the order that the styles are listed in the attribute. For example, if `columnClasses` specifies styles `list-column-center` and `list-column-right` and if there are two columns in the table, the first column will have style `list-column-center`, and the second column will have style `list-column-right`.

If the `style` attribute specifies more styles than there are columns or rows, the remaining styles will be assigned to columns or rows starting from the first column or row. Similarly, if the `style` attribute specifies fewer styles than there are columns or rows, the remaining columns or rows will be assigned styles starting from the first style.
The UIGraphic Component

The UIGraphic component displays an image. The Duke's Bookstore application uses a graphicImage tag to display the map image on the chooselocale.jsp page:

```html
<h:graphicImage id="mapImage" url="/template/world.jpg"
    alt="#{bundle.chooseLocale}" usemap="#worldMap" />
```

The url attribute specifies the path to the image. It also corresponds to the local value of the UIGraphic component so that the URL can be retrieved, possibly from a backing bean. The URL of the example tag begins with a /, which adds the relative context path of the Web application to the beginning of the path to the image.

The alt attribute specifies the alternative text displayed when the user mouses over the image. In this example, the alt attribute refers to a localized message. See Performing Localization (page 741) for details on how to localize your JavaServer Faces application.

The usemap attribute refers to the image map defined by the custom component, MapComponent, which is on the same page. See Chapter 20 for more information on the image map.

The UIInput and UIOutput Components

The UIInput component displays a value to the user and allows the user to modify this data. The most common example is a text field. The UIOutput component displays data that cannot be modified. The most common example is a label.

The UIInput and UIOutput components can each be rendered in four ways. Table 18–3 lists the renderers of UIInput and UIOutput. Recall from Component Rendering Model (page 647) that the tags are composed of the component and the renderer. For example, the inputText tag refers to a UIInput component that is rendered with the Text renderer.

The UIInput component supports the following tag attributes in addition to the tag attributes described at the beginning of Using the HTML Component Tags (page 680). The UIOutput component supports the first of the following tag attributes in addition to those listed in Using the HTML Component Tags (page 680).

- converter: Identifies a converter that will be used to convert the component's local data. See Using the Standard Converters (page 705) for more information on how to use this attribute.
• **validator**: Identifies a method-binding expression pointing to a backing bean method that performs validation on the component's data. See Referencing a Method That Performs Validation (page 722) for an example of using the **validator** tag.

• **valueChangeListener**: Identifies a method-binding expression pointing to a backing bean method that handles the event of entering a value in this component. See Referencing a Method That Handles a ValueChangeEvent (page 722) for an example of using **valueChangeListener**.

The rest of this section explains how to use selected tags listed in Table 18–3. The other tags are written in a similar way.

### Using the outputText and inputText Tags

The **Text** renderer can render both **UIInput** and **UIOutput** components. The **inputText** tag displays and accepts a single-line string. The **outputText** tag displays a single-line string. This section shows you how to use the **inputText** tag. The **outputText** tag is written in a similar way.

#### Table 18–3 **UIInput** and **UIOutput** Renderers

<table>
<thead>
<tr>
<th>Component</th>
<th>Renderer</th>
<th>Tag</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>UIInput</strong></td>
<td>Hidden</td>
<td>inputHidden</td>
<td>Allows a page author to include a hidden variable in a page</td>
</tr>
<tr>
<td></td>
<td>Secret</td>
<td>inputSecret</td>
<td>Accepts one line of text with no spaces and displays it as a set of asterisks as it is typed</td>
</tr>
<tr>
<td></td>
<td>Text</td>
<td>inputText</td>
<td>Accepts a text string of one line</td>
</tr>
<tr>
<td></td>
<td>TextArea</td>
<td>inputTextarea</td>
<td>Accepts multiple lines of text</td>
</tr>
<tr>
<td><strong>UIOutput</strong></td>
<td>Label</td>
<td>outputLabel</td>
<td>Displays a nested component as a label for a specified input field</td>
</tr>
<tr>
<td></td>
<td>Link</td>
<td>outputLink</td>
<td>Displays an <code>&lt;a href&gt;</code> tag that links to another page without generating an <code>ActionEvent</code></td>
</tr>
<tr>
<td></td>
<td>OutputMessage</td>
<td>outputFormat</td>
<td>Displays a localized message</td>
</tr>
<tr>
<td></td>
<td>Text</td>
<td>outputText</td>
<td>Displays a text string of one line</td>
</tr>
</tbody>
</table>
Here is an example of an `inputText` tag from the bookcashier.jsp page:

```xml
<h:inputText id="name" size="50"
    value="#{cashier.name}"
    required="true">
    <f:valueChangeListener type="listeners.NameChanged" />
</h:inputText>
```

The `value` attribute refers to the `name` property of `CashierBean`. This property holds the data for the `name` component. After the user submits the form, the value of the `name` property in `CashierBean` will be set to the text entered in the field corresponding to this tag.

The `required` attribute causes the page to reload with errors displayed if the user does not enter a value in the `name` text field. See Requiring a Value (page 713) for more information on requiring input for a component.

### Using the `outputLabel` Tag

The `outputLabel` tag is used to attach a label to a specified input field for accessibility purposes. The bookcashier.jsp page uses an `outputLabel` tag to render the label of a checkbox:

```xml
<h:selectBooleanCheckbox
    id="fanClub"
    rendered="false"
    binding="#{cashier.specialOffer}" />
<h:outputLabel for="fanClubLabel"
    rendered="false"
    binding="#{cashier.specialOfferText}" >
    <h:outputText id="fanClubLabel"
        value="#{bundle.DukeFanClub}" />
</h:outputLabel>
```

The `for` attribute of the `outputLabel` tag maps to the id of the input field to which the label is attached. The `outputText` tag nested inside the `outputLabel` tag represents the actual label component. The `value` attribute on the `outputText` tag indicates the text that is displayed next to the input field.

### Using the `outputLink` Tag

The `outputLink` tag is used to render a hyperlink that, when clicked, loads another page but does not generate an action event. You should use this tag...
instead of the commandLink tag if you always want the URL—specified by the outputLink tag’s value attribute—to open and do not have to perform any processing when the user clicks on the link. The Duke’s Bookstore application does not utilize this tag, but here is an example of it:

```xml
<h:outputLink value="javadocs">
  <f:verbatim>Documentation for this demo</f:verbatim>
</h:outputLink>
```

As shown in this example, the outputLink tag requires a nested verbatim tag, which identifies the text the user clicks to get to the next page.

You can use the verbatim tag on its own when you want to simply output some text on the page.

### Using the outputFormat Tag

The outputFormat tag allows a page author to display concatenated messages as a MessageFormat pattern, as described in the API documentation for java.text.MessageFormat (see http://java.sun.com/j2se/1.4.2/docs/api/java/text/MessageFormat.html). Here is an example of an outputFormat tag from the bookshowcart.jsp page of the Duke’s Bookstore application:

```xml
<h:outputFormat value="#{bundle.CartItemCount}"
  <f:param value="#{cart.numberOfItems}"/>
</h:outputFormat>
```

The value attribute specifies the MessageFormat pattern. The param tag specifies the substitution parameters for the message.

In the example outputFormat tag, the value for the parameter maps to the number of items in the shopping cart. When the message is displayed in the page, the number of items in the cart replaces the {0} in the message corresponding to the CartItemCount key in the bundle resource bundle:

```
Your shopping cart contains " + "{0,choice,0#no items|1#one item|1< {0} items
```

This message represents three possibilities:

- Your shopping cart contains no items.
- Your shopping cart contains one item.
- Your shopping cart contains {0} items.
The value of the parameter replaces the \{0\} from the message in the sentence in the third bullet. This is an example of a value-binding-enabled tag attribute accepting a complex JSP 2.0 EL expression.

An outputFormat tag can include more than one param tag for those messages that have more than one parameter that must be concatenated into the message. If you have more than one parameter for one message, make sure that you put the param tags in the proper order so that the data is inserted in the correct place in the message.

A page author can also hardcode the data to be substituted in the message by using a literal value with the value attribute on the param tag.

**Using the inputSecret Tag**

The inputSecret tag renders an `<input type="password">` HTML tag. When the user types a string into this field, a row of asterisks is displayed instead of the text the user types. The Duke’s Bookstore application does not include this tag, but here is an example of one:

```xml
<h:inputSecret redisplay="false"
value="#{LoginBean.password}" />
```

In this example, the redisplay attribute is set to false. This will prevent the password from being displayed in a query string or in the source file of the resulting HTML page.

**The UIPanel Component**

The UIPanel component is used as a layout container for its children. When you use the renderers from the HTML render kit, UIPanel is rendered as an HTML table. This component differs from UIData in that UIData can dynamically add or delete rows to accommodate the underlying data source, whereas UIPanel must have the number of rows predetermined. Table 18–4 lists all the renderers and tags corresponding to the UIPanel component.

The panelGrid tag is used to represent an entire table. The panelGroup tag is used to represent rows in a table. Other UI component tags are used to represent individual cells in the rows.
The panelGrid tag has a set of attributes that specify CSS stylesheet classes:
columnClasses, footerClass, headerClass, panelClass, and rowClasses. These styleshet attributes are not required. It also has a columns attribute. The columns attribute is required if you want your table to have more than one column because the columns attribute tells the renderer how to group the data in the table.

If a headerClass is specified, the panelGrid must have a header as its first child. Similarly, if a footerClass is specified, the panelGrid must have a footer as its last child.

The Duke’s Bookstore application includes three panelGrid tags on the book-cashier.jsp page. Here is a portion of one of them:

```xml
<h:panelGrid columns="3" headerClass="list-header"
rowClasses="list-row-even, list-row-odd"
styleClass="list-background"
title="#{bundle.Checkout}"
<f:facet name="header">
  <h:outputText value="#{bundle.Checkout}"/>
</f:facet>
<h:outputText value="#{bundle.Name}"/>
<h:inputText id="name" size="50"
  value="#{cashier.name}"
  required="true">
  <f:valueChangeListener
type="listeners.NameChanged" />
</h:inputText>
<h:message styleClass="validationMessage" for="name"/>
<h:outputText value="#{bundle.CCNumber}"/>
```

### Table 18–4  UIPanel Renderers and Tags

<table>
<thead>
<tr>
<th>Renderer</th>
<th>Tag</th>
<th>Renderer Attributes</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grid</td>
<td>panelGrid</td>
<td>columnClasses, columns, footerClass, headerClass, panelClass, rowClasses</td>
<td>Displays a table</td>
</tr>
<tr>
<td>Group</td>
<td>panelGroup</td>
<td></td>
<td>Groups a set of components under one parent</td>
</tr>
</tbody>
</table>
<h:inputText id="ccno" size="19" 
   converter="CreditCardConverter" required="true">
   <bookstore:formatValidator 
       formatPatterns="9999999999999999|9999 9999 9999 9999|9999-9999-9999-9999"/>
</h:inputText>
<h:message styleClass="validationMessage" for="ccno"/>
...
</h:panelGrid>

This panelGrid tag is rendered to a table that contains controls for the customer of the bookstore to input personal information. This panelGrid uses stylesheet tags classes to format the table. The CSS classes are defined in the stylesheet.css file in the <INSTALL>/j2eetutorial14/examples/web/ bookstore6/web/ directory. The list-header definition is

```html
.list-header {
    background-color: #ffffff;
    color: #000000;
    text-align: center;
}
```

Because the panelGrid tag specifies a headerClass, the panelGrid must contain a header. The example panelGrid tag uses a facet tag for the header. Facets can have only one child, and so a panelGroup tag is needed if you want to group more than one component within a facet. Because the example panelGrid tag has only one cell of data, a panelGroup tag is not needed.

A panelGroup tag can also be used to encapsulate a nested tree of components so that the tree of components appears as a single component to the parent component.

The data represented by the nested component tags is grouped into rows according to the value of the columns attribute of the panelGrid tag. The columns attribute in the example is set to "3", and therefore the table will have three columns. In which column each component is displayed is determined by the order that the component is listed on the page modulo 3. So if a component is the fifth one in the list of components, that component will be in the 5 modulo 3 column, or column 2.
The UISelectBoolean Component

The UISelectBoolean class defines components that have a boolean value. The selectBooleanCheckbox tag is the only tag that JavaServer Faces technology provides for representing boolean state. The Duke's Bookstore application includes a selectBooleanCheckbox tag on the bookcashier.jsp page:

```html
<h:selectBooleanCheckbox
    id="fanClub"
    rendered="false"
    binding="#{cashier.specialOffer}" />
<h:outputLabel
    for="fanClubLabel"
    rendered="false"
    binding="#{cashier.specialOfferText}">
    <h:outputText
        id="fanClubLabel"
        value="#{bundle.DukeFanClub}" />
</h:outputLabel>
```

This example tag displays a checkbox to allow users to indicate whether they want to join the Duke Fan Club. The label for the checkbox is rendered by the outputLabel tag. The actual text is represented by the nested outputText tag. Binding a Component Instance to a Bean Property (page 718) discusses this example in more detail.

The UISelectMany Component

The UISelectMany class defines a component that allows the user to select zero or more values from a set of values. This component can be rendered as a set of checkboxes, a list box, or a menu. This section explains the selectManyCheckbox tag. The selectManyListbox tag and selectManyMenu tag are written in a similar way.

A list box differs from a menu in that it displays a subset of items in a box, whereas a menu displays only one item at a time until you select the menu. The size attribute of the selectManyListbox tag determines the number of items displayed at one time. The list box includes a scrollbar for scrolling through any remaining items in the list.
Using the selectManyCheckbox Tag

The selectManyCheckbox tag renders a set of checkboxes, with each checkbox representing one value that can be selected. Duke’s Bookstore uses a selectManyCheckbox tag on the bookcashier.jsp page to allow the user to subscribe to one or more newsletters:

```xml
<h:selectManyCheckbox
    id="newsletters"
    layout="pageDirection"
    value="#{cashier.newsletters}"
    <f:selectItems
        value="#{newsletters}"/>
</h:selectManyCheckbox>
```

The value attribute of the selectManyCheckbox tag identifies the CashierBean backing bean property, newsletters, for the current set of newsletters. This property holds the values of the currently selected items from the set of checkboxes.

The layout attribute indicates how the set of checkboxes are arranged on the page. Because layout is set to pageDirection, the checkboxes are arranged vertically. The default is lineDirection, which aligns the checkboxes horizontally.

The selectManyCheckbox tag must also contain a tag or set of tags representing the set of checkboxes. To represent a set of items, you use the selectItems tag. To represent each item individually, you use a selectItem tag for each item. The UISelectItem, UISelectItems, and UISelectItemGroup Components (page 700) explains these two tags in more detail.

The UIMessage and UIMessages Components

The UIMessage and UIMessages components are used to display error messages. Here is an example message tag from the guessNumber application, discussed in Steps in the Development Process (page 635):

```xml
<h:inputText id="userNo" value="#{UserNumberBean.userNumber}"
    <f:validateLongRange minimum="0" maximum="10" />
...
</h:message>
Using the HTML Component Tags

The `for` attribute refers to the ID of the component that generated the error message. The `message` tag will display the error message wherever it appears on the page.

The `style` attribute allows you to specify the style of the text of the message. In the example in this section, the text will be red, New Century Schoolbook, serif font family, and oblique style, and a line will appear over the text.

If you use the `messages` tag instead of the `message` tag, all error messages will display.

The UISelectOne Component

A UISelectOne component allows the user to select one value from a set of values. This component can be rendered as a list box, a radio button, or a menu. This section explains the `selectOneMenu` tag. The `selectOneRadio` and `selectOneListbox` tags are written in a similar way. The `selectOneListbox` tag is similar to the `selectOneMenu` tag except that `selectOneListbox` defines a `size` attribute that determines how many of the items are displayed at once.

Using the `selectOneMenu` Tag

The `selectOneMenu` tag represents a component that contains a list of items, from which a user can choose one item. The menu is also commonly known as a drop-down list or a combo box. The following code example shows the `selectOneMenu` tag from the `bookcashier.jsp` page of the Duke’s Bookstore application. This tag allows the user to select a shipping method:

```xml
<h:selectOneMenu id="shippingOption" required="true" value="#{cashier.shippingOption}"/>
```

```xml
<f:selectItem itemValue="2" itemLabel="#{bundle.QuickShip}"/>
```

```xml
<f:selectItem itemValue="5" itemLabel="#{bundle.NormalShip}"/>
```
<f:selectItem
  itemValue="7"
  itemLabel="#{bundle.SaverShip}"/>
</h:selectOneMenu>

The value attribute of the selectOneMenu tag maps to the property that holds the currently selected item’s value.

Like the selectOneRadio tag, the selectOneMenu tag must contain either a selectItems tag or a set of selectItem tags for representing the items in the list. The next section explains these two tags.

**The UISelectItem, UISelectItems, and UISelectItemGroup Components**

UISelectItem and UISelectItems represent components that can be nested inside a UISelectOne or a UISelectMany component. UISelectItem is associated with a SelectItem instance, which contains the value, label, and description of a single item in the UISelectOne or UISelectMany component.

The UISelectItems instance represents either of the following:

- A set of SelectItem instances, containing the values, labels, and descriptions of the entire list of items
- A set of SelectItemGroup instances, each of which represents a set of SelectItem instances

Figure 18–2 shows an example of a list box constructed with a SelectItems component representing two SelectItemGroup instances, each of which represents two categories of beans. Each category is an array of SelectItem instances.

The selectItem tag represents a UISelectItem component. The selectItems tag represents a UISelectItems component. You can use either a set of selectItem tags or a single selectItems tag within your selectOne or selectMany tag.

The advantages of using the selectItems tag are as follows:

- You can represent the items using different data structures, including Array, Map, and Collection. The data structure is composed of SelectItem instances or SelectItemGroup instances.
You can concatenate different lists together into a single UISelectMany or UISelectOne component and group the lists within the component, as shown in Figure 18–2.

You can dynamically generate values at runtime.

The advantages of using selectItem are as follows:

- The page author can define the items in the list from the page.
- You have less code to write in the bean for the selectItem properties.

For more information on writing component properties for the UISelectItems components, see Writing Component Properties (page 730). The rest of this section shows you how to use the selectItems and selectItem tags.

**Using the selectItems Tag**

Here is the selectManyCheckbox tag from the section The UISelectMany Component (page 697):

```xml
<h:selectManyCheckbox id="newsletters" layout="pageDirection" value="#{cashier.newsletters}"
   <f:selectItems value="#{newsletters}"/>
</h:selectManyCheckbox>
```
The value attribute of the selectItems tag is bound to the newsletters managed bean, which is configured in the application configuration resource file. The newsletters managed bean is configured as a list:

```xml
<managed-bean>
  <managed-bean-name>newsletters</managed-bean-name>
  <managed-bean-class>java.util.ArrayList</managed-bean-class>
  <managed-bean-scope>application</managed-bean-scope>
  <list-entries>
    <value-class>javax.faces.model.SelectItem</value-class>
    <value>#{newsletter0}</value>
    <value>#{newsletter1}</value>
    <value>#{newsletter2}</value>
    <value>#{newsletter3}</value>
  </list-entries>
</managed-bean>
<managed-bean>
  <managed-bean-name>newsletter0</managed-bean-name>
  <managed-bean-class>javax.faces.model.SelectItem</managed-bean-class>
  <managed-bean-scope>none</managed-bean-scope>
  <managed-property>
    <property-name>label</property-name>
    <value>Duke's Quarterly</value>
  </managed-property>
  <managed-property>
    <property-name>value</property-name>
    <value>200</value>
  </managed-property>
</managed-bean>
...
```

As shown in the managed-bean element, the UISelectItems component is a collection of SelectItem instances. See Initializing Array and List Properties (page 799) for more information on configuring collections as beans.

You can also create the list corresponding to a UISelectMany or UISelectOne component programmatically in the backing bean. See Writing Component Properties (page 730) for information on how to write a backing bean property corresponding to a UISelectMany or UISelectOne component.
The arguments to the SelectItem constructor are:

- An Object representing the value of the item
- A String representing the label that displays in the UISelectMany component on the page
- A String representing the description of the item

UISelectItems Properties (page 737) describes in more detail how to write a backing bean property for a UISelectItems component.

Using the selectItem Tag

The selectItem tag represents a single item in a list of items. Here is the example from Using the selectOneMenu Tag (page 699):

```xml
<h:selectOneMenu
   id="shippingOption" required="true"
   value="#{cashier.shippingOption}"
   <f:selectItem
      itemValue="2"
      itemLabel="#{bundle.QuickShip}"/>
   <f:selectItem
      itemValue="5"
      itemLabel="#{bundle.NormalShip}"/>
   <f:selectItem
      itemValue="7"
      itemLabel="#{bundle.SaverShip}"/>
</h:selectOneMenu>
```

The itemValue attribute represents the default value of the SelectItem instance. The itemLabel attribute represents the String that appears in the drop-down menu component on the page.

The itemValue and itemLabel attributes are value-binding-enabled, meaning that they can use value-binding expressions to refer to values in external objects. They can also define literal values, as shown in the example selectOneMenu tag.

Using Localized Messages

All data and messages in the Duke’s Bookstore application have been completely localized for Spanish, French, German, and American English. Performing
Localization (page 741) explains how to produce the localized messages as well as how to localize dynamic data and messages.

The image map on the first page allows you to select your preferred locale. See Chapter 20 for information on how the image map custom component was created.

This section explains how to use localized static data and messages for JavaServer Faces applications. If you are not familiar with the basics of localizing Web applications, see Chapter 22. Localized static data can be included in a page by using the `loadBundle` tag, defined in `jsf_core.tld`. Follow these steps:

1. Reference a `ResourceBundle` from the page.
2. Reference the localized message located within the bundle.

A `ResourceBundle` contains a set of localized messages. For more information about resource bundles, see

   http://java.sun.com/docs/books/tutorial/i18n/index.html

After the application developer has produced a `ResourceBundle`, the application architect puts it in the same directory as the application classes. Much of the data for the Duke’s Bookstore application is stored in a `ResourceBundle` called `BookstoreMessages`.

**Referencing a ResourceBundle from a Page**

For a page with JavaServer Faces tags to use the localized messages contained in a `ResourceBundle`, the page must reference the `ResourceBundle` using a `loadBundle` tag.

The `loadBundle` tag from `bookstore.jsp` is

```xml
<f:loadBundle var="bundle"
    basename="messages.BookstoreMessages" />
```

The `basename` attribute value refers to the `ResourceBundle`, located in the messages package of the `bookstore` application. Make sure that the `basename` attribute specifies the fully qualified class name of the file.
The `var` attribute is an alias to the `ResourceBundle`. This alias can be used by other tags in the page in order to access the localized messages.

### Referencing a Localized Message

To reference a localized message from a `ResourceBundle`, you use a value-binding expression from an attribute of the component tag that will display the localized data. You can reference the message from any component tag attribute that is value-binding-enabled.

The value-binding expression has the notation "var.message", in which `var` matches the `var` attribute of the `loadBundle` tag, and `message` matches the key of the message contained in the `ResourceBundle` referred to by the `var` attribute. Here is an example from `bookstore.jsp`:

```xml
<h:outputText value="#{bundle.Talk}"/>
```

Notice that `bundle` matches the `var` attribute from the `loadBundle` tag and that `Talk` matches the key in the `ResourceBundle`.

Another example is the `graphicImage` tag from `chooseLocale.jsp`:

```xml
<h:graphicImage id="mapImage" url="/template/world.jpg" alt="#{bundle.ChooseLocale}" usemap="#worldMap"/>
```

The `alt` attribute is value-binding-enabled, and this means that it can use value-binding expressions. In this case, the `alt` attribute refers to localized text, which will be included in the alternative text of the image rendered by this tag.

See Creating the Component Tag Handler (page 772) and Enabling Value-Binding of Component Properties (page 783) for information on how to enable value binding on your custom component’s attributes.

### Using the Standard Converters

The JavaServer Faces implementation provides a set of `Converter` implementations that you can use to convert component data. For more information on the conceptual details of the conversion model, see Conversion Model (page 651).
The standard Converter implementations, located in the javax.faces.convert package, are as follows:

- BigDecimalConverter
- BigIntegerConverter
- BooleanConverter
- ByteConverter
- CharacterConverter
- DateTimeConverter
- DoubleConverter
- FloatConverter
- IntegerConverter
- LongConverter
- NumberConverter
- ShortConverter

Two of these standard converters (DateTimeConverter and NumberConverter) have their own tags, which allow you to configure the format of the component data by configuring the tag attributes. Using DateTimeConverter (page 707) discusses using DateTimeConverter. Using NumberConverter (page 709) discusses using NumberConverter.

You can use the other standard converters in one of three ways:

- You can make sure that the component that uses the converter has its value bound to a backing bean property of the same type as the converter.
- You can refer to the converter by class or by its ID using the component tag’s converter attribute. The ID is defined in the application configuration resource file (see Application Configuration Resource File, page 792).
- You can refer to the converter by its ID using the converterId attribute of the converter tag.

The latter two will convert the component’s local value. The first method will convert the model value of the component. For example, if you want a component’s data to be converted to an Integer, you can bind the component to a property similar to this:

```java
Integer age = 0;
public Integer getAge() { return age; }
public void setAge(Integer age) { this.age = age; }
```
Alternatively, if the component is not bound to a bean property, you can use the converter attribute on the component tag:

```html
<h:inputText value="#{LoginBean.Age}"
    converter="javax.faces.convert.IntegerConverter" />
```

The data corresponding to this tag will be converted to a `java.lang.Integer`. Notice that the Integer type is already a supported type of the NumberConverter. If you don't need to specify any formatting instructions using the convertNumber tag attributes, and if one of the other converters will suffice, you can simply reference that converter using the component tag's converter attribute.

Finally, you can nest a converter tag within the component tag and refer to the converter's ID via the converter tag's converterId attribute. If the tag is referring to a custom converter, the value of converterID must match the ID in the application configuration resource file. Here is an example:

```html
<h:inputText value="#{LoginBean.Age}" />
    <f:converter converterId="Integer" />
</h:inputText>
```

### Using DateTimeConverter

You can convert a component's data to a `java.util.Date` by nesting the convertDateTime tag inside the component tag. The convertDateTime tag has several attributes that allow you to specify the format and type of the data. Table 18–5 lists the attributes.

Here is a simple example of a convertDateTime tag from the bookreceipt.jsp page:

```html
<h:outputText value="#{cashier.shipDate}">
    <f:convertDateTime dateStyle="full" />
</h:outputText>
```

Here is an example of a date and time that this tag can display:

Saturday, Feb 22, 2003

You can also display the same date and time using this tag:

```html
<h:outputText value="#{cashier.shipDate}">
    <f:convertDateTime
        pattern="EEEEEEE, MMM dd, yyyy" />
</h:outputText>
```
If you want to display the example date in Spanish, you can use the `parseLocale` attribute:

```xml
<h:inputText value="#{cashier.shipDate}">
  <f:convertDateTime dateStyle="full"
    locale="Locale.SPAIN"
    timeStyle="long" type="both" />
</h:inputText>
```

This tag would display

Sabado, Feb 22, 2003
Please refer to the Customizing Formats lesson of the Java Tutorial at http://java.sun.com/docs/books/tutorial/i18n/format/simpleDateFormat.html for more information on how to format the output using the pattern attribute of the convertDateTime tag.

Using NumberConverter

You can convert a component’s data to a java.lang.Number by nesting the convertNumber tag inside the component tag. The convertNumber tag has several attributes that allow you to specify the format and type of the data. Table 18–6 lists the attributes.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>currencyCode</td>
<td>String</td>
<td>ISO4217 currency code, used only when formatting currencies.</td>
</tr>
<tr>
<td>currencySymbol</td>
<td>String</td>
<td>Currency symbol, applied only when formatting currencies.</td>
</tr>
<tr>
<td>groupingUsed</td>
<td>boolean</td>
<td>Specifies whether formatted output contains grouping separators.</td>
</tr>
<tr>
<td>integerOnly</td>
<td>boolean</td>
<td>Specifies whether only the integer part of the value will be parsed.</td>
</tr>
<tr>
<td>maxFractionDigits</td>
<td>int</td>
<td>Maximum number of digits formatted in the fractional part of the output.</td>
</tr>
<tr>
<td>maxIntegerDigits</td>
<td>int</td>
<td>Maximum number of digits formatted in the integer part of the output.</td>
</tr>
<tr>
<td>minFractionDigits</td>
<td>int</td>
<td>Minimum number of digits formatted in the fractional part of the output.</td>
</tr>
<tr>
<td>minIntegerDigits</td>
<td>int</td>
<td>Minimum number of digits formatted in the integer part of the output.</td>
</tr>
</tbody>
</table>

Continues
The bookcashier.jsp page of Duke's Bookstore uses a convertNumber tag to display the total prices of the books in the shopping cart:

```jsp
<h:outputText value="#{cart.total}"
  <f:convertNumber type="currency"/>
</h:outputText>
```

Here is an example of a number this tag can display:

$934

This number can also be displayed using this tag:

```jsp
<h:outputText id="cartTotal"
  value="#{cart.Total}">
  <f:convertNumber pattern="$####" />
</h:outputText>
```

Please refer to the Customizing Formats lesson of the Java Tutorial at http://java.sun.com/docs/books/tutorial/i18n/format/decimalFormat.html for more information on how to format the output using the pattern attribute of the convertNumber tag.

### Registering Listeners on Components

A page author can register a listener implementation class on a component by nesting either a valuechangeListener tag or an actionListener tag within the component's tag on the page.
An application developer can instead implement these listeners as backing bean methods. To reference these methods, a page author uses the component tag’s valueChangeListener and actionListener attributes, as described in Referencing a Method That Handles an ActionEvent (page 721) and Referencing a Method That Handles a ValueChangeEvent (page 722).

The Duke's Bookstore application includes a value-change listener implementation class but does not use an action listener implementation class. This section explains how to register the NameChanged ValueChangeListener and a hypothetical LocaleChange ActionListener implementation on components. Implementing Value-Change Listeners (page 748) explains how to implement NameChanged. Implementing Action Listeners (page 749) explains how to implement the hypothetical LocaleChange.

**Registering a ValueChangeListener on a Component**

A page author can register a ValueChangeListener implementation on a UIInput component or a component represented by one of the subclasses of UIInput by nesting a valueChangeListener tag within the component’s tag on the page. Here is the tag corresponding to the name component from the bookcashier.jsp page:

```xml
<h:inputText id="name" size="50" value="#{cashier.name}" required="true">
  <f:valueChangeListener type="listeners.NameChanged" />
</h:inputText>
```

The type attribute of the valueChangeListener tag specifies the fully qualified class name of the ValueChangeListener implementation.

After this component tag is processed and local values have been validated, its corresponding component instance will queue the ValueChangeEvent associated with the specified ValueChangeListener to the component.

**Registering an ActionListener on a Component**

A page author can register an ActionListener implementation on a UICommand component by nesting an actionListener tag within the component’s tag on
the page. Duke’s Bookstore does not use any ActionListener implementations. Here is one of the commandLink tags on the choose_locale.jsp page, changed to reference an ActionListener implementation rather than a backing bean method:

```html
<h:commandLink id="NAmerica" action="bookstore">
    <f:actionListener type="listeners.LocaleChange" />
</h:commandLink>
```

The type attribute of the actionListener tag specifies the fully qualified class name of the ActionListener implementation.

When this tag’s component is activated, the component’s decode method (or its associated Renderer) automatically queues the ActionEvent implementation associated with the specified ActionListener implementation to the component.

## Using the Standard Validators

JavaServer Faces technology provides a set of standard classes and associated tags that page authors and application developers can use to validate a component’s data. Table 18–7 lists all the standard validator classes and the tags that allow you to use the validators from the page.

### Table 18–7  The Validator Classes

<table>
<thead>
<tr>
<th>Validator Class</th>
<th>Tag</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>DoubleRangeValidator</td>
<td>validateDoubleRange</td>
<td>Checks whether the local value of a component is within a certain range. The value must be floating-point or convertible to floating-point.</td>
</tr>
<tr>
<td>LengthValidator</td>
<td>validateLength</td>
<td>Checks whether the length of a component’s local value is within a certain range. The value must be a java.lang.String.</td>
</tr>
<tr>
<td>LongRangeValidator</td>
<td>validateLongRange</td>
<td>Checks whether the local value of a component is within a certain range. The value must be any numeric type or String that can be converted to a long.</td>
</tr>
</tbody>
</table>
All these validator classes implement the `Validator` interface. Component writers and application developers can also implement this interface to define their own set of constraints for a component’s value.

When using the standard `Validator` implementations, you don’t need to write any code to perform validation. You simply nest the standard validator tag of your choice inside a tag that represents a component of type `UIInput` (or a subclass of `UIInput`) and provide the necessary constraints, if the tag requires it. Validation can be performed only on `UIInput` components or components whose classes extend `UIInput` because these components accept values that can be validated.

This section shows you how to use the standard `Validator` implementations.

See The `UIMessage` and `UIMessages` Components (page 698) for information on how to display validation error messages on the page.

**Requiring a Value**

The `name inputText` tag on the `bookcashier.jsp` page has a `required` attribute, which is set to `true`. Because of this, the JavaServer Faces implementation checks whether the value of the component is `null` or is an empty `String`.

If your component must have a non-null value or a `String` value at least one character in length, you should add a `required` attribute to your component tag and set it to `true`. If your tag does have a `required` attribute that is set to `true` and the value is `null` or a zero-length string, no other validators registered on the tag are called. If your tag does not have a `required` attribute set to `true`, other validators registered on the tag are called, but those validators must handle the possibility of a `null` or zero-length string.

Here is the `name inputText` tag:

```xml
<h:inputText id="name" size="50"
    value="#{cashier.name}" required="true">
    ...
</h:inputText>
```
Using the LongRangeValidator

The Duke's Bookstore application uses a validateLongRange tag on the quantity input field of the bookshowcart.jsp page:

```html
<h:inputText id="quantity" size="4"
    value="#{item.quantity}" >
    <f:validateLongRange minimum="1"/>
</h:inputText>
<h:message for="quantity"/>
```

This tag requires that the user enter a number that is at least 1. The size attribute specifies that the number can have no more than four digits. The validateLongRange tag also has a maximum attribute, with which you can set a maximum value of the input.

Binding Component Values and Instances to External Data Sources

As explained in Backing Bean Management (page 656), a component tag can wire its component's data to a back-end data object by doing one of the following:

- Binding its component's value to a bean property or other external data source
- Binding its component's instance to a bean property

A component tag’s `value` attribute uses a value-binding expression to bind a component’s value to an external data source, such as a bean property. A component tag’s `binding` attribute uses a value-binding expression to bind a component instance to a bean property.

When referencing the property using the component tag’s `value` attribute, you need to use the proper syntax. For example, suppose a backing bean called MyBean has this int property:

```java
int currentOption = null;
int getCurrentOption(){...}
void setCurrentOption(int option){...}
```

The `value` attribute that references this property must have this value-binding expression:

"#{MyBean.currentOption}"
In addition to binding a component’s value to a bean property, the `value` attribute can specify a literal value or can map the component’s data to any primitive (such as `int`), structure (such as an array), or collection (such as a list), independent of a JavaBeans component. Table 18–8 lists some example value-binding expressions that you can use with the `value` attribute.

The next two sections explain in more detail how to use the `value` attribute to bind a component’s value to a bean property or other external data sources and how to use the `binding` attribute to bind a component instance to a bean property.

### Binding a Component Value to a Property

To bind a component’s value to a bean property, you specify the name of the bean and the property using the `value` attribute. As explained in Backing Bean Management (page 656), the value-binding expression of the component tag’s `value` attribute must match the corresponding managed bean declaration in the application configuration resource file.

This means that the name of the bean in the value-binding expression must match the `managed-bean-name` element of the managed bean declaration up to the first `. in the expression. Similarly, the part of the value-binding expression after the `. must match the name specified in the corresponding `property-name` element in the application configuration resource file.
For example, consider this managed bean configuration, which configures the ImageArea bean corresponding to the North America part of the image map on the chooselocale.jsp page of the Duke’s Bookstore application:

```xml
<managed-bean>
    <managed-bean-name> NA </managed-bean-name>
    <managed-bean-class> model.ImageArea </managed-bean-class>
    <managed-bean-scope> application </managed-bean-scope>
    <managed-property>
        <property-name>shape</property-name>
        <value>poly</value>
    </managed-property>
    <managed-property>
        <property-name>alt</property-name>
        <value>NAmerica</value>
    </managed-property>
...
</managed-bean>
```

This example configures a bean called NA, which has several properties, one of which is called shape.

Although the area tags on the chooselocale.jsp page do not bind to an ImageArea property (they bind to the bean itself), to do this, you refer to the property using a value-binding expression from the value attribute of the component’s tag:

```xml
<h:outputText value="#{NA.shape}" />
```

Much of the time you will not include definitions for a managed bean’s properties when configuring it. You need to define a property and its value only when you want the property to be initialized with a value when the bean is initialized.

If a component tag’s value attribute must refer to a property that is not initialized in the managed-bean configuration, the part of the value-binding expression after the . must match the property name as it is defined in the backing bean.

See Application Configuration Resource File (page 792) for information on how to configure beans in the application configuration resource file.

Writing Component Properties (page 730) explains in more detail how to write the backing bean properties for each of the component types.
Binding a Component Value to an Implicit Object

One external data source that a value attribute can refer to is an implicit object. The bookreceipt.jsp page of the Duke’s Bookstore application includes a reference to an implicit object from a parameter substitution tag:

```html
<h:outputFormat title="thanks" value="#{bundle.ThankYouParm}"
    <f:param value="#{sessionScope.name}"/>
</h:outputFormat>
```

This tag gets the name of the customer from the session scope and inserts it into the parameterized message at the key ThankYouParm from the resource bundle. For example, if the name of the customer is Gwen Canigetit, this tag will render:

Thank you, Gwen Canigetit, for purchasing your books from us.

The name tag on the bookcashier.jsp page has the NameChanged listener implementation registered on it. This listener saves the customer's name in the session scope when the bookcashier.jsp page is submitted. See Implementing Value-Change Listeners (page 748) for more information on how this listener works. See Registering a ValueChangeListener on a Component (page 711) to learn how the listener is registered on the tag.

Retrieving values from other implicit objects is done in a similar way to the example shown in this section. Table 18–9 lists the implicit objects that a value attribute can refer to. All of the implicit objects except for the scope objects are read-only and therefore should not be used as a value for a UIInput component.

<table>
<thead>
<tr>
<th>Implicit Object</th>
<th>What It Is</th>
</tr>
</thead>
<tbody>
<tr>
<td>applicationScope</td>
<td>A Map of the application scope attribute values, keyed by attribute name</td>
</tr>
<tr>
<td>cookie</td>
<td>A Map of the cookie values for the current request, keyed by cookie name</td>
</tr>
<tr>
<td>facesContext</td>
<td>The FacesContext instance for the current request</td>
</tr>
</tbody>
</table>

Continues
Binding a Component Instance to a Bean Property

A component instance can be bound to a bean property using a value-binding expression with the binding attribute of the component’s tag. You usually bind a component instance rather than its value to a bean property if the bean must dynamically change the component’s attributes.

Here are two tags from the bookcashier.jsp page that bind components to bean properties:

```<h:selectBooleanCheckbox
    id="fanClub"
    rendered="false"
    binding="#{cashier.specialOffer}" />
<h:outputLabel for="fanClubLabel"
               rendered="false"
               binding="#{cashier.specialOfferText}" />
```
Referencing a Backing Bean Method

A component tag has a set of attributes for referencing backing bean methods that can perform certain functions for the component associated with the tag. These attributes are summarized in Table 18–10.

Only components that implement ActionSource can use the action and actionListener attributes. Only UIInput components or components that extend UIInput can use the validator or valueChangeListener attributes.
The component tag refers to a backing bean method using method-binding expression as a value of one of the attributes. The following four sections give examples of how to use the four different attributes.

### Referencing a Method That Performs Navigation

If your page includes a component (such as a button or hyperlink) that causes the application to navigate to another page when the component is activated, the tag corresponding to this component must include an `action` attribute. This attribute does one of the following:

- Specifies a logical outcome String that tells the application which page to access next
- References a backing bean method that performs some processing and returns a logical outcome String

The `bookcashier.jsp` page of the Duke's Bookstore application has a `commandButton` tag that refers to a backing bean method that calculates the shipping date. If the customer has ordered more than $100 (or 100 euros) worth of books, this method also sets the rendered properties of some of the components to true and returns `null`; otherwise it returns `receipt`, which causes the `bookreceipt.jsp` page to display. Here is the `commandButton` tag from the `bookcashier.jsp` page:

---

**Table 18–10  Component Tag Attributes that Reference Backing Bean Methods**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>action</td>
<td>Refers to a backing bean method that performs navigation processing for the component and returns a logical outcome String</td>
</tr>
<tr>
<td>actionListener</td>
<td>Refers to a backing bean method that handles ActionEvents</td>
</tr>
<tr>
<td>validator</td>
<td>Refers to a backing bean method that performs validation on the component's value</td>
</tr>
<tr>
<td>valueChangeListener</td>
<td>Refers to a backing bean method that handles ValueChangeEvents</td>
</tr>
</tbody>
</table>

---
REFERENCING A BACKING BEAN METHOD

The action attribute uses a method-binding expression to refer to the submit method of CashierBean. This method will process the event fired by the component corresponding to this tag.

Writing a Method to Handle Navigation (page 755) describes how to implement the submit method of CashierBean.

The application architect must configure a navigation rule that determines which page to access given the current page and the logical outcome, which is either returned from the backing bean method or specified in the tag. See Configuring Navigation Rules (page 805) for information on how to define navigation rules in the application configuration resource file.

Referencing a Method That Handles an ActionEvent

If a component on your page generates an ActionEvent, and if that event is handled by a backing bean method, you refer to the method by using the component's actionListener attribute.

The chooselocale.jsp page of the Duke’s Bookstore application includes some components that generate action events. One of them is the NAmerica component:

```html
<h:commandLink id="NAmerica" action="bookstore"
    actionListener="#{localeBean.chooseLocaleFromLink}"
/>```

The actionListener attribute of this component tag references the chooseLocaleFromLink method using a method-binding expression. The chooseLocaleFromLink method handles the event of a user clicking on the hyperlink rendered by this component.

The actionListener attribute can be used only with the tags of components that implement ActionSource. These include UICommand components.

Writing a Method to Handle an ActionEvent (page 757) describes how to implement a method that handles an action event.
Referencing a Method That Performs Validation

If the input of one of the components on your page is validated by a backing bean method, you refer to the method from the component’s tag using the validator attribute.

The Coffee Break application includes a method that performs validation of the email input component on the checkoutForm.jsp page. Here is the tag corresponding to this component:

```html
<h:inputText id="email" value="#{checkoutFormBean.email}" size="25" maxlength="125" validator="#{checkoutFormBean.validateEmail}"/>
```

This tag references the validateEmail method described in Writing a Method to Perform Validation (page 757) using a method-binding expression.

The validator attribute can be used only with UIInput components or those components whose classes extend UIInput.

Writing a Method to Perform Validation (page 757) describes how to implement a method that performs validation.

Referencing a Method That Handles a ValueChangeEvent

If you want a component on your page to generate a ValueChangeEvent and you want that event to be handled by a backing bean method, you refer to the method using the component’s valueChangeListener attribute.

The name component on the bookcashier.jsp page of the Duke’s Bookstore application references a ValueChangeListener implementation that handles the event of a user entering a name in the name input field:

```html
<h:inputText
   id="name"
   size="50"
   value="#{cashier.name}"
   required="true">
   <f:valueChangeListener type="listeners.NameChanged" />
</h:inputText>
```
For illustration, Writing a Method to Handle a Value-Change Event (page 758) describes how to implement this listener with a backing bean method instead of a listener implementation class. To refer to this backing bean method, the tag uses the valueChangeListener attribute:

```html
<h:inputText
   id="name"
   size="50"
   value="#{cashier.name}"
   required="true"
   valueChangeListener="#{cashier.processValueChangeEvent}" />
</h:inputText>
```

The valueChangeListener attribute of this component tag references the processValueChange method of CashierBean using a method-binding expression. The processValueChange method handles the event of a user entering his name in the input field rendered by this component.

The valueChangeListener attribute can be used only with the tags of UIInput components and components whose classes extend UIInput.

Writing a Method to Handle a Value-Change Event (page 758) describes how to implement a method that handles a ValueChangeEvent.

### Using Custom Objects

As a page author, you might need to use custom converters, validators, or components packaged with the application on your JSP pages.

A custom converter is applied to a component either by using the component tag’s converter attribute or by nesting a converter tag inside the component’s tag.

A custom validator is applied to a component by nesting either a validator tag or the validator’s custom tag inside the component’s tag.

To use a custom component, you use the custom tag associated with the component.

As explained in Setting Up a Page (page 676), you must ensure that the TLD that defines the custom tags is packaged in the application. TLD files are stored in the WEB-INF directory or subdirectory of the WAR file or in the META-INF/ directory or subdirectory of a tag library packaged in a JAR.
Next, you include a taglib declaration so that the page has access to the tags. All custom objects for the Duke’s Bookstore application are defined in bookstore.tld. Here is the taglib declaration that you would include on your page so that you can use the tags from this TLD:

```html
<%@ taglib uri="/WEB-INF/bookstore.tld" prefix="bookstore" %>
```

When including the custom tag in the page, you can consult the TLD to determine which attributes the tag supports and how they are used.

The next three sections describe how to use the custom converter, validator, and UI components included in the Duke’s Bookstore application.

### Using a Custom Converter

To apply the data conversion performed by a custom converter to a particular component’s value, you must either set the converter attribute of the component’s tag to the Converter implementation’s identifier or set the nested converter tag’s converterId attribute to the Converter implementation’s identifier. The application architect provides this identifier when registering the Converter with the application, as explained in Registering a Custom Converter (page 804). Creating a Custom Converter (page 744) explains how a custom converter is implemented.

The identifier for the CreditCardConverter is creditCardConverter. The CreditCardConverter is registered on the ccno component, as shown in this tag from the bookcashier.jsp page:

```html
<h:inputText id="ccno"
  size="19"
  converter="CreditCardConverter"
  required="true">
  ...
</h:inputText>
```

By setting the converter attribute of a component’s tag to the converter’s identifier, you cause that component’s local value to be automatically converted according to the rules specified in the Converter implementation.

A page author can use the same custom converter with any similar component by simply supplying the Converter implementation’s identifier to the converter attribute of the component’s tag or to the converId attribute of the nested converter tag.
Using a Custom Validator

To use a custom validator in a JSP page, you must nest the validator’s custom tag inside the tag of the component whose value you want to be validated by the custom validator.

Here is the `formatValidator` tag from the `ccno` field on the `bookcashier.jsp` page of the Duke’s Bookstore application:

```xml
<h:inputText id="ccno" size="19"
  ... required="true">
  <bookstore:formatValidator
    formatPatterns="9999999999999999|9999 9999 9999 9999|
    9999-9999-9999-9999" />
</h:inputText>
<h:message styleClass="validationMessage" for="ccno"/>
```

This tag validates the input of the `ccno` field against the patterns defined by the page author in the `formatPatterns` attribute.

You can use the same custom validator for any similar component by simply nesting the custom validator tag within the component tag.

Creating a Custom Validator (page 750) describes how to create the custom validator and its custom tag.

If the application developer who created the custom validator prefers to configure the attributes in the `Validator` implementation rather than allow the page author to configure the attributes from the page, the developer will not create a custom tag for use with the validator.

Instead, the page author must follow these steps:

1. Nest the `validator` tag inside the tag of the component whose data needs to be validated.
2. Set the `validator` tag’s `validatorId` attribute to the ID of the validator that is defined in the application configuration resource file. Registering a Custom Validator (page 803) explains how to define the validator in the application configuration resource file.
The following tag registers a hypothetical validator on a component using a validator tag and referencing the ID of the validator:

```html
<h:inputText id="name" value="#{CustomerBean.name}" size="10" ...
  <f:validator validatorId="customValidator" />
...  
</h:inputText>
```

### Using a Custom Component

Using a custom component on a page is similar to using a custom validator, except that custom validator tags must be nested inside component tags. In order to use the custom component in the page, you need to declare the tag library that defines the custom tag that renders the custom component. This is explained in Using Custom Objects (page 723).

The Duke's Bookstore application includes a custom image map component on the chooselocale.jsp page. This component allows you to select the locale for the application by clicking on a region of the image map:

```html
...  
<h:graphicImage id="mapImage" url="/template/world.jpg"
  alt="#{bundle.chooseLocale}"
  usemap="#worldMap" />
<bookstore:map id="worldMap" current="NAmericas"
  immediate="true"
  action="bookstore"
  actionListener="#{localeBean.chooseLocaleFromMap}">
<bookstore:area id="NAmerica" value="#{NA}" onmouseover="/template/world_namer.jpg" onmouseout="/template/world.jpg"
  targetImage="mapImage" />
...  
<bookstore:area id="France" value="#{fraA}" onmouseover="/template/world_france.jpg" onmouseout="/template/world.jpg"
  targetImage="mapImage" />
</bookstore:map>
```

The graphicImage tag associates an image (world.jpg) with an image map that is referenced in the usemap attribute value.
The custom map tag represents the custom component, MapComponent, specifies the image map, and contains a set of custom area tags. Each area tag represents a custom AreaComponent and specifies a region of the image map.

On the page, the onmouseover and onmouseout attributes define the image that is displayed when the user performs the actions described by the attributes. The page author defines what these images are. The custom renderer also renders an onclick attribute.

In the rendered HTML page, the onmouseover, onmouseout, and onclick attributes define which JavaScript code is executed when these events occur. When the user moves the mouse over a region, the onmouseover function associated with the region displays the map with that region highlighted. When the user moves the mouse out of a region, the onmouseout function redisplays the original image. When the user clicks a region, the onclick function sets the value of a hidden input tag to the ID of the selected area and submits the page.

When the custom renderer renders these attributes in HTML, it also renders the JavaScript code. The custom renderer also renders the entire onclick attribute rather than let the page author set it.

The custom renderer that renders the map tag also renders a hidden input component that holds the current area. The server-side objects retrieve the value of the hidden input field and set the locale in the FacesContext according to which region was selected.

Chapter 20 describes the custom tags in more detail and also explains how to create the custom image map components, renderers, and tags.