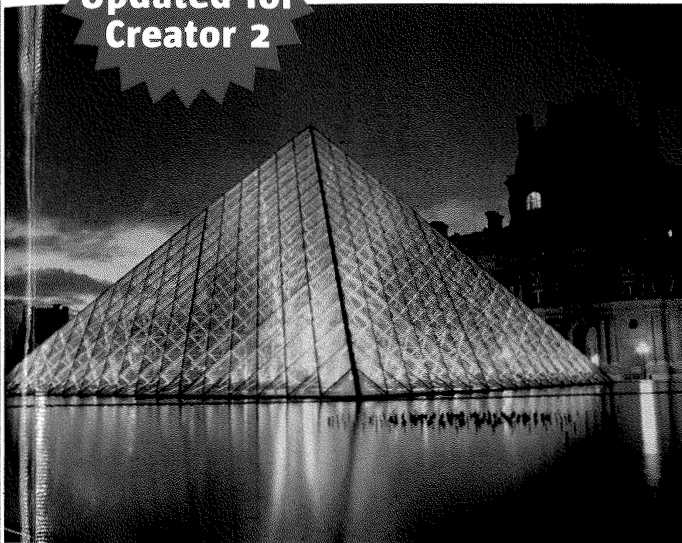


2ND EDITION

PRENTICE  
HALL

# JAVA™ STUDIO CREATOR FIELD GUIDE

Revised  
and  
Updated for  
Creator 2



- How to drag and drop components, specify page navigation, and access Web services and databases
- Creator Tips provide snippets of advice on leveraging the IDE
- Component Catalog lists each component, validator, and data converter for easy lookup
- Includes complete Web services case study using Google Web Service APIs

JAVAONE<sup>SM</sup> EDITION



GAIL ANDERSON · PAUL ANDERSON

Foreword by James Gosling

Facebook's Exhibit No. 1003

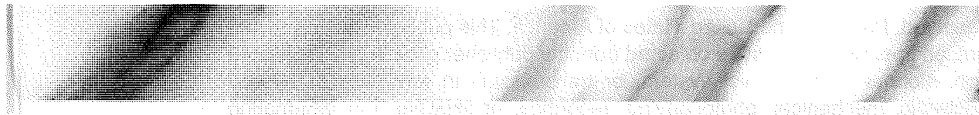
Page 001



Facebook's Exhibit No. 1003  
Page 002

# JAVA™ STUDIO CREATOR FIELD GUIDE

SECOND EDITION



GAIL ANDERSON • PAUL ANDERSON

*Sun Microsystems Press*  
*A Prentice Hall Title*



Prentice Hall PTR, Upper Saddle River, NJ 07458  
[www.phptr.com](http://www.phptr.com)

Facebook's Exhibit No. 1003  
Page 003

DA76  
173  
J38A 473  
2006  
copy

Sun Microsystems, Inc. has intellectual property rights relating to implementations of the technology described in this publication. In particular, and without limitation, these intellectual property rights may include one or more U.S. patents, foreign patents, or pending applications.

Sun, Sun Microsystems, the Sun logo, J2ME, J2EE, Java Card, and all Sun and Java based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc., in the United States and other countries. UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd.

THIS PUBLICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION. SUN MICROSYSTEMS, INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS PUBLICATION AT ANY TIME.

Visit us on the Web: [www.prenhallprofessional.com](http://www.prenhallprofessional.com)

Copyright © 2006 Gail Anderson and Paul Anderson

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc.  
Rights and Contracts Department  
One Lake Street  
Upper Saddle River, NJ 07458  
Fax: (201) 236-3290

ISBN 0-13-225460-3  
Text printed in the United States on recycled paper at Courier in Stoughton, Massachusetts  
First printing, May 2006

TX 6-525-204



# Contents

FOREWORD xxiii

PREFACE xxv

How This Book Is Organized xxv  
About the Examples xxvii  
Notational Conventions xxvii  
About the Front Cover xxviii

ACKNOWLEDGMENTS xxix

## CHAPTER 1 JAVA TECHNOLOGY OVERVIEW 2

1.1 Introduction 3  
1.2 The Java Programming Language 5  
    Object-Oriented Programming 5  
    Creating Objects 6  
    Classes 7  
    Packages 8  
    Exceptions 9

**Contents**

- Inheritance 10
- Interfaces 12
- 1.3 JavaBeans Components 13
- 1.4 NetBeans Software 13
- 1.5 The XML Language 14
- 1.6 The J2EE Architecture 15
- 1.7 Java Servlet Technology 16
- 1.8 JavaServer Pages Technology 16
- 1.9 JDBC API and Database Access 17
- 1.10 JavaServer Faces Technology 17
- 1.11 Ant Build Tool 18
- 1.12 Web Services 18
- 1.13 Enterprise JavaBeans (EJB) 18
- 1.14 Portlets 19
- 1.15 Key Point Summary 19

**CHAPTER 2 CREATOR BASICS 22**

- 2.1 Examples Installation 23
  - Download Examples 24
- 2.2 Creator Views 24
  - Welcome Window 24
  - Design Editor 26
  - Properties 28
  - Palette 30
  - Outline 32
  - Projects 33
  - Files 35
  - JSP Editor 36
  - Java Source Editor 37

- Code Clips Palette 41
- Page Navigation Editor 43
- Output Window 44
- Servers 46
- Debugging Windows 49
- Creator Help System 50
- 2.3 Sample Application 51
  - Create a Project 51
  - Add Components to the Page 52
  - Deploy and Run 55
- 2.4 Key Point Summary 55

## CHAPTER 3 CREATOR COMPONENTS 58

- 3.1 JSF Overview 59
  - JSF Architecture 60
  - The JSP Page 60
  - JSF Expression Language (EL) 61
  - Converters and Validators 62
  - Event Handling 62
  - Java Page Bean 63
- 3.2 Components 65
  - Components Palette 65
  - Component Properties 66
  - Input Components 69
  - Virtual Forms 70
  - Data-Aware Components 71
- 3.3 Basic Components 72
  - Anchor 72
  - Button 72
  - Calendar 74
  - Checkbox 75
  - Checkbox Group 76
  - Drop Down List 77
  - File Upload 79
  - Hidden Field 81
  - Hyperlink 82
  - Image 83

	Image Hyperlink	84
	Label	85
	Listbox	86
	Message	88
	Message Group	90
	Password Field	91
	Radio Button	92
	Radio Button Group	93
	Static Text	94
	Table	95
	Text Area	100
	Text Field	101
	Tree	103
3.4	Layout Components	105
	Form	105
	Grid Panel	106
	Group Panel	108
	Layout Panel	108
	Page Alert	109
	Page Fragment Box	110
	Page Separator	112
	Property Sheet	112
	Tab Set	114
3.5	Composite Components	116
	Add Remove List	116
	Alert	118
	Breadcrumbs	121
	Inline Help	122
3.6	Validators	124
	Validation Model	124
3.7	Converters	125
	Conversion Model	126
3.8	AJAX Components	129
	Importing a Component Library	129
3.9	Key Point Summary	132



## CHAPTER 4 SOFTWARE DEVELOPMENT 134

- 4.1 Using the Java Source Editor 136
  - Finding What You Need 136
  - Formatting Code 137
  - Fonts and Colors 138
  - Code Completion 139
  - Show Line Numbers 140
  - Code Folding 140
  - Handling Imports 141
  - Using Javadoc 141
  - Abbreviations 143
  - Generating Methods 143
  - Generating Properties 145
  - Searching and Replacing 146
  - Navigating Files 149
  - Task Lists 150
- 4.2 Refactoring 152
  - What is Refactoring? 152
  - Refactoring Window 153
  - Payment Project 154
  - Copy Project 154
  - Find Usages 154
  - Renaming Classes 155
  - Undo and Redo 156
  - Renaming Fields and Methods 158
  - Encapsulating Fields 158
  - Changing Method Signatures 160
  - Moving Classes to Different Packages 164
- 4.3 Source Code Control with CVS 168
  - Copy Project 168
  - Setting up CVS 168
  - Importing Files 169
  - Checking Out Files 171
  - Updating Source Files 172
  - Comparing File Revisions 175
  - Viewing History 176
  - Adding and Removing Files 178
  - Configuring CVS Settings 181
  - Advanced CVS Features 183

- 4.4 Creating Non-Web Projects 183
  - Create a General Project 183
  - Add a Java Package 184
  - Add a Java Class File 184
  - Add a JAR/Folder 184
  - Build and Run Project 186
- 4.5 Key Point Summary 186

## CHAPTER 5 PAGE NAVIGATION 188

- 5.1 Navigation Model 190
- 5.2 Simple Navigation 191
  - Create a New Project 192
  - Add a Label Component 192
  - Add a Grid Panel Component 193
  - Add Button Components 193
  - Deploy and Test Run 194
  - Add Page Navigation 195
  - New Rules! 195
  - Add Label Components 198
  - Deploy and Run 198
  - Add Event Handler Code 198
  - Deploy and Run 199
  - Draggable Mode 200
- 5.3 Noncommand Components 201
  - Copy the Project 202
  - Delete the Buttons 202
  - Add a Drop Down List 202
  - Value Change Event vs. Action Event 203
  - Match the Navigation Labels 204
  - Add Event Handler Code 204
  - Add Button Components 205
  - Deploy and Run 205
- 5.4 Dynamic Navigation 206
  - Create a New Project 207
  - Add a Label Component 207
  - Create the Form's Input Components 208
  - Add Button Components 209

- tabIndex Property 210
- Deploy and Test Run 210
- Add Event Handler Code 210
- Create New Web Pages 212
- Add Components to Page LoginBad 213
- Add a Component to Page LoginGood 213
- Specify Page Navigation 214
- Deploy and Test 216
- Configure Virtual Forms 216
- Deploy and Run 218

## 5.5 Key Point Summary 219

# CHAPTER 6 ANATOMY OF A CREATOR PROJECT 220

## 6.1 What Is a Bean? 222

- Properties 222
- Setters and Getters 222
- Default Constructor 223
- Property Binding 223
- Scope of Web Applications 224
- Predefined Creator Java Objects 226

## 6.2 LoginBean 229

- LoginBean Outside View 229
- Advantages of JavaBeans Objects 230
- Property Binding with Creator Components 230
- Copy the Project 231
- Add LoginBean to Your Project 232
- Configure LoginBean.java 233
- Add a LoginBean Property to SessionBean1 234
- Bind Input Components 237
- Modify Event Handler 238
- Modify Page LoginGood.jsp 239
- Deploy and Run 241

## 6.3 LoanBean 242

- LoanBean Outside View 242
- Create a New Project 243
- Add a Label Component 244

**Contents**

- Add LoanBean to Your Project 244
- Add a LoanBean Property to SessionBean1 248
- LoanBean.java Code 249
- Create the Form's Input Components 250
- Use Validators and Converters 252
- Specify Property Binding 253
- Place Interest Rate and Term Components 254
- Place Button, Label and Static Text Components 257
- Deploy and Run 258
- 6.4 The Creator-JSF Life Cycle 260
  - JSF Life Cycle 261
  - Creator Life Cycle Callback Methods 265
- 6.5 Key Point Summary 269

**CHAPTER 7 WEB PAGE DESIGN 272**

- 7.1 Using the Visual Design Editor 273
  - Create a Project 274
  - Add Components to the Page 274
  - Working with Components on the Page 275
  - Component Alignment 275
  - Deploy and Run 277
- 7.2 Themes 277
  - Changing the Look with Themes 278
  - Add Components to the Page 279
  - Change the Current Theme 280
  - Modifying the Default Theme 281
- 7.3 About Style 281
  - Copy the Project 281
  - Using the Style Editor 282
  - Deploy and Run 283
- 7.4 Cascading Style Sheets 284
  - Using Property styleClass 284
  - Deploy and Run 287
- 7.5 Page Layout 287
  - Layout Panel 288

- Create a Project 288
- Add Components to the Page 289
- Deploy and Run 293
- More CSS Style Issues 293
- Centering Components on a Page 295
- Deploy and Run 297
- Grid Panel 297
- Create a Project 298
- Add Components to the Page 298
- Deploy and Run 304
- 7.6 Page Fragments 304
  - Create a New Project 306
  - Modify Default Style Sheet 306
  - Use the Gray Theme 308
  - Add SessionBean1 Properties 308
  - Banner Page Fragment 309
  - Navigation Page Fragment 311
  - CactusFooter Page Fragment 313
  - Add Pages 314
  - Page-Specific Content for Page1 315
  - Page-Specific Content 318
  - Page Fragments and Navigation 318
  - Logout Event Handler 321
  - Deploy and Run 321
  - Reuse with Project Templates 321
- 7.7 Introducing Tab Sets 323
  - Using Separate Tab Sets 324
  - Create a New Project 324
  - Add Components to Page1 324
  - Configure Navigation 326
  - Deploy and Run 327
  - Using Tab Sets and Page Fragments 327
  - Add Tab Set and Tabs to CactusBanner 329
  - Setting the Currently Selected Tab 331
  - Configure Action Method for Tabs 332
  - Check Pages with Modified CactusBanner 333
- 7.8 Key Point Summary 333

## CHAPTER 8 INTRODUCING DATA PROVIDERS 336

- 8.1 Data Provider Basics 337
  - Table Data Providers 340
- 8.2 Object Data Provider 345
  - Object Data Provider Methods 345
  - Copy the Project 347
  - Add the Object Data Provider 347
  - Provide Binding to Components 347
  - Modify Event Handler Code 348
  - Modify LoginGood Page 349
  - Deploy and Run 350
  - Other Singleton Object Data Providers 351
- 8.3 Object List Data Provider 351
  - Copy the Project 351
  - Replace LoanBean.java 352
  - Add PaymentVO.java 352
  - Deploy and Run 353
  - LoanBean Bean Patterns 353
  - PaymentVO Bean Patterns 353
  - Add Components to Page1 354
  - Configure the Calendar Component 355
  - Configure Virtual Forms 357
  - Add a New Page 358
  - Add Components to Schedule Page 358
  - Configure the Table 360
  - Configure Page Navigation 361
  - Deploy and Run 362
- 8.4 Cached RowSet Data Provider 362
  - Configuring the Database 363
  - Add Data Source 364
  - Inspect the Data Source 365
  - Copy the Project 366
  - Add the Data Source 367
  - Replace LoginBean.java 367
  - Add a Message Group 368
  - Deploy and Run 369

RS

8.5 Key Point Summary 369

**CHAPTER 9 ACCESSING DATABASES 372****9.1 Database Fundamentals 374**Music Collection Database 374  
JDBC CachedRowSets 376**9.2 Data Sources 376**Configuring the Bundled Database 376  
Add Data Source 377  
Inspect the Data Source 378  
Loading Other Data Sources 381**9.3 Accessing the Music Database 381**Create a New Project 381  
Add Components 382  
Add a Database Table 383  
Add a Message Group Component 386  
Deploy and Run 386  
Query and Table Configuration 387  
Deploy and Run 389**9.4 Master Detail Application - Two Page 390**Copy the Project 390  
Add a RecordingID Request Bean Property 391  
Add a RecordingTitle Request Bean Property 391  
Command Components in a Table Column 391  
Add a New Page 393  
Modify SQL Query 394  
Add Page Navigation 395  
Add Prerender Code 396  
Configure Table Component 397  
Deploy and Run 397**9.5 Master Detail Application - Single Page 398**Create a New Project 398  
Add Components 399  
Add a Listbox Component 400  
Add a Data Source 401  
Deploy and Run 401

- Add a Table Component 402
- Modify the SQL Query 405
- Connect Dropdown List to Query 408
- Deploy and Run 410
- 9.6 Database Updates 410
  - Create a New Project 411
  - Add Components 411
  - Add Buttons and a Table 411
  - Modify the Table Layout 412
  - Add the Button Event Handlers 414
  - Deploy and Run 415
- 9.7 Database Row Inserts 416
  - Virtual Forms 417
  - Copy the Project 417
  - Add Components 417
  - Configure Virtual Forms 420
  - Add ApplicationBean1 Property 423
  - Add Button Event Handler Code 426
  - Deploy and Run 427
- 9.8 Database Deletions 427
  - Copy the Project 429
  - Add Components 429
  - Configure Checkbox Components 430
  - Add the Delete Button Event Handler 433
  - Configure Virtual Forms 436
  - Deploy and Run 436
- 9.9 Handle Cascading Deletes 438
  - Copy the Project 439
  - Include Additional Data Source Tables 439
  - Modify the SQL Queries 440
  - Modify Button Event Handler Code 441
  - Add Method Cascade Delete 442
  - Deploy and Run 444
- 9.10 Key Point Summary 445

## CHAPTER 10 ACCESSING WEB SERVICES 448



- 10.1 Google Web Services 450
  - Create a New Project 450
  - Add the Google Logo 451
  - Add a Text Field Component 451
  - Add a Button Component 452
  - Add the Google Web Services 453
  - Adding a Web Service to the IDE 453
  - Add Search Result Properties to Page1 455
  - Add a Data Provider 456
  - Layout and Grouping with Grid Panel 457
  - Add a Static Text Component 457
  - Using Hyperlink with a Nested Static Text 458
  - Add a Message Group to Display Errors 459
  - Deploy and Run 460
  - Inspect the Web Service 461
  - Testing the Google Web Service 464
  - Configure Web Service Call 465
  - Add Event Handling Code for Button 466
  - Handling Exceptions and Error Messages 466
  - Specify Binding for the Display Components 467
  - Deploy and Run 470
- 10.2 Validation - Project Google2 471
  - Copy the Project 472
  - Add a Validator 472
  - Add a Message Component 474
  - Message and Message Group Components 474
- 10.3 Displaying Multiple Result Elements 475
  - Copy the Project 476
  - Add a Table Component 477
  - Configure the Table 477
  - Deploy and Run 480
- 10.4 Displaying Multiple Pages 480
  - Copy the Project 481
  - Add an Image Hyperlink Component 481
  - A Second Image Hyperlink Component 483
  - Deploy and Run 484
  - Add SessionBean1 Properties 484
  - Specify the Action Code 485
  - Deploy and Run 489
- 10.5 Key Point Summary 489

## CHAPTER 11 USING EJB COMPONENTS 492

- 11.1 Consuming EJBs 493
  - Invoke the EJB Method 495
  - Instantiate a Data Provider 497
- 11.2 EJBs as Business Objects 499
  - Create a Project 500
  - Add an EJB Client 500
  - Add Session Bean Properties 501
  - Add Components to the Page 502
  - Add Event Handling Code 506
  - Deploy and Run 508
  - Copy the Project 509
  - Add EJB Method Data Providers 509
  - Bind Components to Data Object 509
  - Modify Event Handler Code 510
  - Specify Data Provider Initialization 511
  - Deploy and Run 512
- 11.3 Greeting Two Ways 512
  - Create a Project 513
  - Add Components to the Page 513
  - Add EJB Method Data Providers 513
  - Bind Components to Data Object 514
  - Deploy and Run 515
  - Copy the Project 515
  - Delete Unneeded Components 516
  - Add Components to the Page 516
  - Add EJB Method Data Provider 517
  - Add Event Handling Code 517
  - Deploy and Run 519
- 11.4 Implementing a Master-Detail Page with EJBs 519
  - Create a Project 521
  - Add Components to the Page 521
  - Add EJB Method Data Providers 523
  - Configure the Table Components 524
  - Add a Session Bean Property 524
  - Add Event Handling Code 525
  - Specify Initialization 526
  - Deploy and Run 526

- 11.5 Adding EJBs to Creator 527
  - Add LoanEJB 529
  - Consuming the LoanEJB 531
  - Copy the Project 532
  - Add an EJB Method 533
  - Delete Local LoanBean Component 533
  - Add a Session Bean Property 533
  - Provide Property Bindings 534
  - Deploy and Run 535
  - Project Payment2 Alternative 535
- 11.6 Key Point Summary 536

## CHAPTER 12 PORTLETS 538

- 12.1 What Are Portlets? 539
  - Portlet Modes 540
  - Portlet Navigation 540
  - Portlet Real Estate 541
  - Portlet Life Cycle 541
- 12.2 Creating a Portlet Project 543
  - Create a Portlet Project 543
  - Add Components to PortletPage1 544
  - Add a Save Text Session Bean Property 545
  - Specify Property Bindings 546
  - Deploy and Run 547
- 12.3 Database Access with Portlets 548
  - Create a Portlet Project 548
  - Add Session Bean Properties 548
  - Add Components to the Page 549
  - Add a Database Table 550
  - Query and Table Configuration 550
  - Add a New Page 552
  - Modify SQL Query 553
  - Add Page Navigation 554
  - Add Prerender Code 555
  - Configure Table Component 556
  - Deploy and Run 557
- 12.4 Web Services and Portlets 557

- Create a Portlet Project 558
- Add Session Bean Properties 559
- Add Components to the Page 562
- Add the Google Web Service Client 566
- Configure Web Service Call 566
- Add a Table Component 567
- Configure the Table 567
- Add Event Handling Code 569
- Portlet Life Cycle Issues 574
- Deploy and Run 575
- 12.5 Portlet Edit Mode 575
  - Copy the Project 576
  - Add a New Edit Mode Page 576
  - Add SessionBean1 Properties 577
  - Add ApplicationBean1 Data 577
  - Add Components to the Page 579
  - Specify Property Bindings 582
  - Deploy and Run 583
- 12.6 Portlet Help Mode 583
  - Copy the Project 583
  - Add a New Help Mode Page 584
  - Add ApplicationBean1 Property 585
  - Add Static Text Component to the Page 586
  - Specify Property Binding 587
  - Deploy and Run 587
- 12.7 Key Point Summary 587

## CHAPTER 13 CUSTOMIZING APPLICATIONS WITH CREATOR 590

- 13.1 Localizing an Application 591
  - A Word About Locales 593
  - Localize Application Labels and Text 593
  - Copy the Project 593
  - Isolate Labels and Text Messages 594
  - Add the asg.jar Jar File 595
  - Localize the JSF Source 595
  - Using Grid Panel to Improve Page Layout 597

- Modify the Components for Localized Text 599
- Deploy and Run 601
- 13.2 Internationalizing an Application 601
  - Provide Translations 601
  - Specify Supported Locales 603
  - Configure Your Browser 604
  - Deploy and Run 604
- 13.3 Controlling the Locale from the Application 606
  - Copy the Project 606
  - Add a SessionBean1 Property 607
  - Add Components to Page1 607
  - Final Configurations 610
  - Deploy and Run 611
- 13.4 Custom Validation Method 611
  - Create a Project 613
  - Add a JAR File to Your Project 613
  - Add a ColorBean Property to SessionBean1 613
  - ColorBean.java Code 615
  - Isolate Localized Text 616
  - Add a Validation Method 617
  - Adding Components to the Page 619
  - Add Components for Input 620
  - Add a Button and a Static Text Component 624
  - Configure for the Validator Method 624
  - Configure the Components for Localized Text 625
  - Deploy and Run 625
  - Internationalize for Spanish 626
  - Specify Supported Locales 626
  - Configure Your Browser 627
  - Deploy and Run 628
- 13.5 Using AJAX-Enabled Components 628
  - State Codes Auto Complete Example 629
  - Add myStateCode Session Property 630
  - Add stateCodes Application Property 630
  - Add Components to the Page 631
  - Configure the AJAX Component 634
  - Deploy and Run 635
  - Add statesMap Application Property 636
  - Add myStateName Session Property 637
  - Configure Static Text Component 638

H

- Deploy and Run 638
- 13.6 Using AJAX-Enabled Components with Web Services 638
  - Adding the Dictionary Web Service 639
  - Create a New Project 641
  - Add Components to the Page 642
  - Add the Dictionary Web Service to the Page 643
  - Configure the Event Handlers 643
  - Deploy and Run 645
- 13.7 Key Point Summary 646

## CHAPTER 14 DEBUGGING WITH CREATOR 648

- 14.1 Planning for Debugging 650
  - Local Variables 650
  - Assertions 651
  - Displaying Debug Information 651
- 14.2 Debugger Overview 652
  - Debugger Features 652
  - Debugger Windows 652
  - Debugging Commands 653
- 14.3 Running the Debugger 655
  - Open Project and Files 655
  - Run and Deploy in Debug Mode 655
  - Debugging Views 655
- 14.4 Setting Breakpoints 656
- 14.5 Managing Breakpoints 659
- 14.6 Stepping Through the Code 661
- 14.7 Tracking Variables 663
- 14.8 Setting Watches 664
- 14.9 Using the Call Stack 667
- 14.10 Detecting Exceptions 669

rvices

- 14.11 Finish Debugging 672
- 14.12 Debug Methods 672
  - Method info() 672
  - Method log() 673
- 14.13 Using the HTTP Monitor 675
  - Enabling the HTTP Monitor 676
  - The HTTP Monitor Window 676
  - Viewing Record Data 677
  - Editing HTTP Requests 678
  - Saving HTTP Requests 679
  - Replaying HTTP Requests 681
- 14.14 Key Point Summary 682

18

INDEX 685

Facebook's Exhibit No. 1003  
Page 0024



# Foreword

**T**he developers who set out to build Java Studio Creator had a very difficult task: to make the creation of sophisticated enterprise applications easy.

The set of technologies that comprise Java2 Enterprise Edition (J2EE) is huge. The good side is that J2EE is battle hardened and field proven to be an excellent base for large scale mission critical applications. There are many excellent books that cover all of the aspects of J2EE in great detail. But all of this leads to the bad side of J2EE: it can be difficult and time-consuming to learn and use. Tremendous effort goes into making J2EE as simple as possible, but it remains daunting.

Java Studio Creator is a huge leap in the simplification of the process of developing J2EE based web applications. Developers don't need to deal with all of the gory details: Creator just handles them. Instead, developers can focus on what their application does and looks like in a very simple and straightforward way. They weave together data sources through a simple drag-and-drop interface. Very little knowledge of Java or J2EE is required to develop applications in Creator. Creator not only simplifies the process: it also accelerates it.

This book contains all that you need to know to generate enterprise applications using Creator. It doesn't require that you know anything about J2EE or even Java: there are gentle introductory chapters that lead you through what you need to know. This is a great entry point for developers from other platforms (like Visual Basic!) to enter the world of large-scale, mission-critical applications. It's fun. Take the plunge.

**James Gosling**  
Sun Microsystems, Inc.

Facebook's Exhibit No. 1003  
Page 0026

# Preface

**Y**ou're about to embark on a journey that we hope will prove both enjoyable and fruitful. Certainly the aim of any application development tool is to help developers become efficient and allow them to spend time on creative tasks while the tool silently generates the drudgery for them. To that end, we hope this book will teach you the ins and outs of Creator so that you can quickly begin to build web applications.

## How This Book Is Organized

Creator2 has significant enhancements over the first version of Creator and we've completely revamped (and grown!) this text to reflect the "new" Creator. In a nutshell, you'll see an improved design time experience, with an advanced page design editor and component style editor. Creator2 includes a complete set of new (and more numerous) UI components sporting an improved theme-based look and intuitive property names. Data providers give you a consistent interface with the persistence tier and NetBeans provides the underpinnings of the IDE. Building portlet-based projects, consuming EJBs, and using AJAX-enabled components round out the list of improvements. Enjoy!

Chapter 1 introduces the world of Java and its supporting technologies. Creator depends on these well-established Java technologies to do its job. With the Java programming language and XML, JavaServer Faces component system, and NetBeans tool building technology, Creator has tapped the available standards. Here we provide a gentle introduction to these topics, so you'll get the

“big picture” of how Creator fits into the Java world. We also spend time on the Java programming language since you will use Java often with Creator. If you come from another programming environment, we want you to feel comfortable right away with Java-based web applications.

Chapter 2 introduces Creator, with the aim of getting you up to speed with its various windows, the design canvas, and its editors. Knowing how to move around in the IDE will quickly make you productive. You’ll also build your first project from scratch.

Chapter 3 is your “Components Catalog.” This reference chapter lets you choose the best component for your application from Creator’s store of components, validators, and data converters. We’ll also point you to places in the book where these components are used in complete projects.

Chapter 4 is all about working with the Java source editor, managing your code with the Concurrent Versioning System (CVS), and performing project-level tasks, such as refactoring. Creator is built on the NetBeans IDE, which provides the backbone for these software management tasks.

Chapter 5 introduces Page Navigation in Creator. You’ll learn how to specify page flow in a web application and understand which components are suitable for page navigation. We’ll also discuss Creator’s navigation model and illustrate page navigation with several projects.

Chapter 6 discusses the anatomy of Creator projects. JavaBeans components (beans) provide one of the key supporting technologies that Creator uses. Developers who understand the advantages of JavaBeans components can build robust applications with reusable components. The chapter also delves into the JSF/Creator page request life cycle so that you can best leverage Creator’s life cycle callback methods.

Chapter 7 covers features that help you with web page design and layout. Page fragments give your application a consistent look and several layout components help you design your page layout. Creator’s Cascading Style Sheet (CSS) editor lets you build page and component style rules.

Chapter 8 introduces data providers, a standard layer connecting your web application’s components to a persistence tier, such as database, web service, JavaBeans component, or Enterprise JavaBeans (EJB) component. We explore the data provider interface and show you common ways to work with data providers in your projects.

Chapter 9 shows you how to use a database with Creator. You’ll build projects with essential database operations, such as read, update, insert, and delete. Creator’s data-aware components make linking to a database (via data providers) easy and straightforward.

Chapter 10 shows you how to access a web service from a Creator-built application. Creator bundles a selection of web services; you can also add additional web services to the IDE. In this chapter, you’ll build an application that uses the Google Search Web Service.

Enterprise JavaBeans (EJBs) provide a powerful model for distributed systems. Creator lets you build applications that consume EJBs, automatically generating the code to invoke methods on deployed objects. Chapter 11 shows you how to invoke EJB methods and manipulate returned data through standard data providers.

With Creator, you can create a JSF portlet application that conforms to the Java Specification Request (JSR) 168 Portlet specification. Chapter 12 covers portlet application development, including examples on using web services, database access, and portlet Edit and Help modes.

Chapter 13 shows you how to customize a web application with Creator. You'll learn how to localize/internationalize an application. We also show you how to write and install custom validation methods. Anxious to use AJAX technology? We include several examples that use an auto-complete AJAX-enabled text field.

Chapter 14 shows you how to use Creator's debugger in your projects. You'll learn how to set breakpoints, look at the server log file, respond to exceptions, and use the NetBeans HTTP monitor.

## About the Examples

*Java Studio Creator Field Guide, Second Edition* is an example-driven book. All example projects that we describe in these chapters are provided prebuilt in the example download bundle. The database chapter includes a sample Music database, which we also provide in the download bundle. The example code is available on the Sun Microsystems Creator2 web site.

[http://developers.sun.com/prodtech/javatools/  
jscreator/index.jsp](http://developers.sun.com/prodtech/javatools/jscreator/index.jsp)

You can also download the example code from the authors' web site at

<http://www.asgteach.com>

We recommend you visit both web sites to keep current with the latest Creator updates!

## Notational Conventions

We've applied a rather "light hand" with font conventions in an attempt to keep the page uncluttered. Here are the conventions we follow.

---

<i>Element</i>	<i>Font and Example</i>
property name	text property, cellpadding property
code comments	// User event code here . . .
component name	text field loanAmount
Creator components	output text, text field, drop down list
filename	<b>color1.properties, Page1.jsp</b>
Java class	String, Object, Integer
Java code	username.setValue("rave4u")
JavaBeans components	LoginBean, ColorBean, Page1
JSF EL expressions	{Page1.colorBean.redValue}
JSP code, HTML code	<f:loadBundle>, <img>
key combinations	press <Alt+Shift+I>; press <Enter>
menu selections	View Page1 Java Class, Music > Tables
project name	Login1, Echo
user input	Type the text <b>Music1</b> , specify <b>loginBean</b>

---

## About the Front Cover

I.M. Pei's Glass Pyramid is an apt symbol for a book on Java Studio Creator. The pyramid serves as the entrance to the Louvre Museum in Paris. Truly international, it was designed by a Chinese-American, who is both an engineer and an artist. We were drawn to its elegant simplicity of glass and light, its multiple facets, and the blending of a structure that is modern yet simultaneously representative of Ancient Egypt.

Java Studio Creator is built on the same concept of layered architecture. Based on JavaServer Faces technology, Creator leverages the existing Java 2 Platform Enterprise Edition (J2EE) architecture and of course, the solid foundations of Java and its runtime environment. Likewise, the Glass Pyramid is an extraordinary example of layered architecture whose backdrop is a traditional Renaissance palace. The Louvre itself houses artistic treasures dating from antiquity, perhaps none more famous than Leonardo da Vinci's Mona Lisa.

**Gail and Paul Anderson**

Anderson Software Group, Inc.

Facebook's Exhibit No. 1003

Page 0030

# Acknowledgments

Without the help of others, not only would this text be lacking in timeliness and accuracy, but its very existence is questionable. Jim Inscore of Sun Microsystems blazed the path for this project, taking care of both administration and coordinating technical help. Without his desire to see this project a reality, the Second Edition would have remained unrealized.

Many people from the Sun Microsystems Creator Team helped us directly and indirectly by answering technical questions, giving us insights into design and architectural issues, providing updated "bits" so that we could always work with a current system, and providing valuable feedback to early drafts. We'd like to especially acknowledge Winston Prakash. He gave us crucial technical support and answered many questions. He was our go-to guy and we cannot praise his dedication to this project enough!

Octavian Tanase and Sandip Chitale were also on the frontline for us, helping with various technical questions throughout the entire writing process. Octavian also was central dispatch for our questions, feeding them to the right Creator Team member. Thus, we'd also like to thank those who answered when asked for help: David Botterill, David Folk, Chau Nguyen, Tor Norbye, Matt Bohm, Craig McClanahan, Chris Kutler, Edwin Goei, Dongmei Cao, Vaughn Spurlin, and Dusan Pavlica. Valerie Lipman was in on the early proposed content talks. Valuable feedback also came from one of our readers, Les Hawkins.

The Second Edition builds on the initial work of the First Edition. Greg Doench, our editor at Prentice Hall, made that possible. Vicky Hilpert of Vietsbronn, Germany helped with the German translations and Blanca Lazaro helped with the Spanish. Thanks also to our family (especially Sara and Kellen) and friends for love and support.

Lastly, we'd like to thank James Gosling for giving us all Java.

Facebook's Exhibit No. 1003  
Page 0032



# JAVA™ STUDIO CREATOR FIELD GUIDE

Facebook's Exhibit No. 1003  
Page 0033

# JAVA TECHNOLOGY OVERVIEW

## Topics in This Chapter

- The Java Programming Language
- JavaBeans Components
- NetBeans Software
- The XML Language
- The J2EE Architecture
- JavaServer Faces Technology
- JDBC and Databases
- Ant Build Tool
- Web Services
- EJBs and Portlets

# Chapter

# 1

**W**elcome to Creator! Creator is an IDE (Integrated Development Environment) that helps you build web applications. While many IDEs out in the world do that, Creator is unique in that it is built on a layered technology anchored in Java. At the core of this technology is the Java programming language. Java includes a compiler that produces portable bytecode and a Java Virtual Machine (JVM) that runs this byte code on any processor. Java is an important part of Creator because it makes your web applications portable.

But Java is more than just a programming language. It is also a *technology platform*. Many large systems have been developed that use Java as their core. These systems are highly scalable and provide services and structure that address some of the high-volume, distributed computing environments of today.

## 1.1 Introduction

Creator depends on multiple technologies, so it's worthwhile touching on them in this chapter. If you're new to Java, many of its parts and acronyms can be daunting. Java technologies are divided into related packages containing classes and interfaces. To build an application, you might need parts of one system and parts of another. This chapter provides you with a road map of Java

technologies and documentation sources to help you design your web applications with Creator.

We'll begin with an overview of the Java programming language. This will help you get comfortable writing Java code to customize your Creator applications. But before we do that, we show you how to find the documentation for Java classes and methods. This will help you use them with confidence in your programs.

Most of the documentation for a Java Application Program Interface (API) can be accessed through Creator's Help System, located under Help in the main menu. Sometimes all you need is the name of the package or the system to find out what API a class, interface, or method belongs to. Java consists of the basic language (all packages under `java`) and Java extensions (all packages under `javax`). Once you locate a package, you can explore the interfaces and classes and learn about the methods they implement.

You can also access the Java documentation online. Here's a good starting point for the Java API documentation.

<http://java.sun.com/docs/>

This page contains links to the Java 2 Platform Standard Edition, which contains the core APIs. It also has a link to all of the other Java APIs and technologies, found at

<http://java.sun.com/reference/docs/index.html>

Creator is also built on the technology of JavaServer Faces (JSF). You can find the current JSF API documentation at

<http://java.sun.com/j2ee/javaxserverfaces/1.0/docs/api/index.html>

JSF is described as part of the J2EE Tutorial, which can be found at

<http://java.sun.com/j2ee/1.4/docs/tutorial/doc/index.html>

These are all important references for you. We've included them at the beginning of this book so it's easy to find them later (when you're deep in the challenges of web application development). For now, let's begin with Java as a programming language. Then we'll look at some of the other supporting technologies on which Creator is built.

## 1.2 The Java Programming Language

This cursory overview of the Java programming language is for readers who come from a non-Java programming environment. It's not meant to be an in-depth reference, but a starting point. Much of Creator involves manipulating components through the design canvas and the components' property sheets. However, there are times when you must add code to a Java page bean (the supporting Java code for your web application's page) or use a JavaBeans component in your application. You'll want a basic understanding of Java to more easily use Creator.

### ***Object-Oriented Programming***

Languages like C and Basic are procedure-oriented languages, which means data and functions are separated. To write programs, you either pass data as arguments to functions or make your data global to functions. This arrangement can be problematic when you need to hide data like passwords, customer identification codes, and network addresses. Procedure-oriented designs work fine when you write simple programs but are often not suitable to more complex tasks like distributed programming and web applications. Function libraries help, but error handling can be difficult and global variables may introduce side effects during program maintenance.

Object-oriented programming, on the other hand, combines data and functions into units called *objects*. Languages like Java hide private data (*fields*) from user programs and expose only functions (*methods*) as a public interface. This concept of *encapsulation* allows you to control how callers access your objects. It allows you to break up applications into groups of objects that behave in a similar way, a concept called *abstraction*. In Java, you implement an object with a Java class and your object's public interface becomes its *outside view*. Java has inheritance to create new data types as extensions of existing types. Java also has interfaces, which allow objects to implement required behaviors of certain classes of objects. All of these concepts help separate an object's implementation (inside view) from its interface (outside view).

All objects created from the same class have the same data type. Java is a strongly typed language, and all objects are implicitly derived from type `Object` (except the built-in primitive types of `int`, `boolean`, `char`, `double`, `long`, etc.). You can convert an object from one type to another with a converter. Casting to a different type is only allowed if the conversion is known by the compiler. Creator's Java editor helps you create well-formed statements with dynamic syntax analysis and code completion choices. You'll see how this works in Chapter 2.

Error handling has always been a tough problem to solve, but with web applications error handling is even more difficult. Processing errors can occur

on the server but need to propagate in a well-behaved way back to the user. Java implements exception handling to handle errors as objects and recover gracefully. The Java compiler forces programmers to use the built-in exception handling mechanism.

And, Java forbids global variables, a restriction that helps program maintenance.

## Creating Objects

Operator `new` creates objects in Java. You don't have to worry about destroying them, because Java uses a garbage collection mechanism to automatically destroy objects which are no longer used by your program.

```
Point p = new Point();           // create a Point at (0, 0)
Point q = new Point(10, 20);    // create a Point at (10, 20)
```

Operator `new` creates an object at run time and returns its address in memory to the caller. In Java, you use *references* (`p` and `q`) to store the addresses of objects so that you can refer to them later. Every reference has a type (`Point`), and objects can be built with arguments to initialize their data. In this example, we create two `Point` objects with `x` and `y` coordinates, one with a default of `(0, 0)` and the other one with `(10, 20)`.

Once you create an object, you can call its methods with a reference.

```
p.move(30, 30);                 // move object p to (30, 30)
q.up();                          // move object q up in y direction
p.right();                       // move object p right in x direction

int xp = p.getX();              // get x coordinate of object p
int yp = p.getY();              // get y coordinate of object p
q.setX(5);                      // change x coordinate in object q
p.setY(25);                     // change y coordinate in object p
```

As you can see, you can do a lot of things with `Point` objects. It's possible to move a `Point` object to a new location, or make it go up or to the right, all of which affect one or more of a `Point` object's coordinates. We also have getter methods to return the `x` and `y` coordinates separately and setter methods to change them.

Why is this all this worthwhile? Because a `Point` object's data (`x` and `y` coordinates) are *hidden*. The only way you can manipulate a `Point` object is through its public methods. This makes it easier to maintain the integrity of `Point` objects.

## Classes

Java already has a `Point` class in its API, but for the purposes of this discussion, let's roll our own. Here's our Java `Point` class, which describes the functionality we've shown you.

---

### Listing 1.1 Point class

---

```
// Point.java - Point class
class Point {
// Fields
    private double x, y;          // x and y coordinates

// Constructors
    public Point(double x, double y) { move(x, y); }
    public Point() { move(0, 0); }

// Instance Methods
    public void move(double x, double y) {
        this.x = x;  this.y = y;
    }
    public void up() { y++; }
    public void down() { y--; }
    public void right() { x++; }
    public void left() { x--; }

// getters
    public double getX() { return x; }
    public double getY() { return y; }

// setters
    public void setX(double x) { this.x = x; }
    public void setY(double y) { this.y = y; }
}
```

The `Point` class is divided into three sections: Fields, Constructors, and Instance Methods. Fields hold internal data, constructors initialize the fields, and instance methods are called by you with references. Note that the fields for `x` and `y` are *private*. This enforces data encapsulation in object-oriented programming, since users may not access these values directly. Everything else, however, is declared public, making it accessible to all clients.

The `Point` class has two *constructors* to build `Point` objects. The first constructor accepts two double arguments, and the second one is a default constructor with no arguments. Note that both constructors call the `move()` method to initialize the `x` and `y` fields. Method `move()` uses the Java `this` key-

word to distinguish local variable names in the method from class field names in the object. The `setX()` and `setY()` methods use the same technique.<sup>1</sup>

Most of the `Point` methods use `void` for their return type, which means the method does not return anything. The `++` and `--` operators increment or decrement their values by one, respectively. Each method has a *signature*, which is another name for a function's argument list. Note that a signature may be empty.

## Packages

The `Point` class definition lives in a file called `Point.java`. In Java, you must name a file with the same name as your class name. This makes it convenient for the Java run-time interpreter to find class definitions when it's time to instantiate (create) objects. When all classes live in the same directory, it's easy to compile and run Java programs.

In the real world, however, classes have to live in different places, so Java has *packages* that allow you to group related classes. A package in Java is both a directory and a library. This means a one-to-one correspondence exists between a package hierarchy name and a file's pathname in a directory structure. Unique package names are typically formed by reversing Internet domain names (`com.mycompany`). Java also provides access to packages from class paths and JAR (Java Archive) files.

Suppose you want to store the `Point` class in a package called `MyPackage.examples`. Here's how you do it.

```
package MyPackage.examples;
class Point {
    . . .
}
```

Package names with dot (`.`) delimiters map directly to path names, so `Point.java` lives in the `examples` directory under the `MyPackage` directory. A Java `import` statement makes it easy to use class names without fully qualifying their package names. Import statements are also applicable to class names from any Java API.

```
// Another Java program
import java.util.Date;
import javax.faces.context.*;
import MyPackage.examples.Point;
```

---

1. The `this` reference is not necessary if you use different names for the arguments.



The first import statement provides the `Date` class name to our Java program from the `java.util` package. The second import uses a wildcard (\*) to make *all* class definitions available from `javax.faces.context`. The last import brings our `Point` class into scope from package `MyPackage.examples`.

## Exceptions

We mentioned earlier that one of the downfalls of procedure-oriented languages is that subroutine libraries don't handle errors well. This is because libraries can only detect problems, not fix them. Even with libraries that support elaborate error mechanisms, you cannot force someone to check a function's return value or peek at a global error flag. For these and other reasons, it has been difficult to write distributed software that gracefully recovers from errors.

Object-oriented languages like Java have a built-in exception handling mechanism that lets you handle error conditions as objects. When an error occurs inside a try block of critical code, an exception object can be thrown from a library method back to a catch handler. Inside user code, these catch handlers may call methods in the exception object to do a range of different things, like display error messages, retry, or take other actions.

The exception handling mechanism is built around three Java keywords: `throw`, `catch`, and `try`. Here's a simple example to show you how it works.

```
class SomeClass {
    . . .
    public void doSomething(String input) {
        int number;
        try {
            number = Integer.parseInt(input);
        }
        catch (NumberFormatException e) {
            String msg = e.getMessage();
            // do something with msg
        }
        . . .
    }
}
```

Suppose a method called `doSomething()` needs to convert a string of characters (`input`) to an integer value in memory (`number`). In Java, the call to `Integer.parseInt()` performs the necessary conversion for you, but what about malformed string arguments? Fortunately, the `parseInt()` method throws a `NumberFormatException` if the input string has illegal characters. All we do is place this call in a try block and use a catch handler to generate an error message when the exception is caught.

All that's left is to show you how the exception gets thrown. This is often called a *throw point*.

```
class Integer {
    public static int parseInt(String input)
        throws NumberFormatException {
        . . .
        // input string has bad chars
        throw new NumberFormatException("illegal chars");
    }
    . . .
}
```

The static `parseInt()` method<sup>2</sup> illustrates two important points about exceptions. First, the `throws` clause in the method signature announces that `parseInt()` throws an exception object of type `NumberFormatException`. The `throws` clause allows the Java compiler to enforce error handling. To call the `parseInt()` method, you must put the call inside a `try` block or in a method that also has the same `throws` clause. Second, operator `new` calls the `NumberFormatException` constructor to build an exception object. This exception object is built with an error string argument and thrown to a catch handler whose signature *matches* the type of the exception object (`NumberFormatException`).<sup>3</sup> As you have seen, a catch handler calls `getMessage()` with the exception object to access the error message.

Why are Java exceptions important? As you develop web applications with Creator, you'll have to deal with thrown exceptions. Fortunately, Creator has a built-in debugger that helps you monitor exceptions. In the Chapter 14, we show you how to set breakpoints to track exceptions in your web application (see "Detecting Exceptions" on page 669).

## Inheritance

The concept of code reuse is a major goal of object-oriented programming. When designing a new class, you may derive it from an existing one. Inheritance, therefore, implements an "is a" relationship between classes. Inheritance also makes it easy to hook into existing frameworks so that you can take on

- 
2. Inside class `Integer`, the `static` keyword means you don't have to instantiate an `Integer` object to call `parseInt()`. Instead, you call the static method with a class name rather than a reference.
  3. The match doesn't have to be exact. The exception thrown can match the catch handler's object exactly or any exception object derived from it by inheritance. To catch any possible exception, you can use the superclass `Exception`. We discuss inheritance in the next section.

new functionalities. With inheritance, you can retain the existing structure and behavior of an existing class and specialize certain aspects of it to suit your needs.

In Java, inheritance is implemented by *extending* classes. When you extend one class from another, the public methods of the "parent" class become part of the public interface of the "child class." The parent class is called a *superclass* and the child class is called a *subclass*. Here are some examples.

```
class Pixel extends Point {  
    . . .  
}  
  
class NumberFormatException extends IllegalArgumentException {  
    . . .  
}
```

In the first example, `Point` is a superclass and `Pixel` is a subclass. A `Pixel` "is a" `Point` with, say, color. Inside the `Pixel` class, a color field with setter and getter methods can assist in manipulating colors. `Pixel` objects, however, are `Point` objects, so you can move them up, down, left or right, and you can get or set their `x` and `y` coordinates. (You can also invoke any of `Point`'s public methods with a reference to a `Pixel` object.) Note that you don't have to write any code in the `Pixel` class to do these things because they have been inherited from the `Point` class. Likewise, in `NumberFormatException`, you may introduce new methods but inherit the functionality of `IllegalArgumentException`.

Another point about inheritance. You can write your own version of a method in a subclass that has the same name and signature as the method in the superclass. Suppose, for instance, we add a `clear()` method in our `Point` class to reset `Point` objects back to (0, 0). In the `Pixel` class that extends from `Point`, we may *override* the `clear()` method.<sup>4</sup> This new version could move a `Pixel` object to (0, 0) *and* reset its color. Note that `clear()` in class `Point` is called for `Point` objects, but `clear()` in class `Pixel` will be called for `Pixel` objects. With a `Point` reference set to either type of object, different behaviors happen when you call this method.

It's important to understand that these kinds of method calls in Java are resolved at run time. This is called *dynamic binding*. In the object-oriented paradigm, dynamic binding means that the resolution of method calls with objects

---

4. Creator uses this same feature by providing methods that are called at different points in the JSF page request life cycle. You can override any of these methods and thus provide your own code, "hooking" into the page request life cycle. We show you how to do this in Chapter 6 (see "The Creator-JSF Life Cycle" on page 260).

is delayed until you run a program. In web applications and other types of distributed software, dynamic binding plays a key role in how objects call methods from different machines across a network or from different processes in a multitasking system.

## Interfaces

In Java, a method with a signature and no code body is called an *abstract* method. Abstract methods must be overridden in subclasses and help define *interfaces*. A Java interface is like a class but has no fields and only abstract public methods. Interfaces are important because they specify a *contract*. Any new class that implements an interface must provide code for the interface's methods.

Here's an example of an interface.

```
interface Encryptable {
    void encode(String key);
    String decode();
}

class Password implements Encryptable {
    . . .
    void encode(String key) { . . . }
    String decode() { . . . }
}
```

The `Encryptable` interface contains only the abstract public methods `encode()` and `decode()`. Class `Password` implements the `Encryptable` interface and must provide implementations for these methods. Remember, interfaces are types, just like classes. This means you can implement the same interface with other classes and treat them all as `Encryptable` types.

Java prohibits a class from inheriting from more than one superclass, but it does allow classes to implement multiple interfaces. Interfaces, therefore, allow arbitrary classes to "take on" the characteristics of any given interface.

One of the most common interfaces implemented by classes in Java is the `Serializable` interface. When an object implements `Serializable`, you can use it in a networked environment or make it *persistent* (this means the state of an object can be saved and restored by different clients). There are methods to serialize the object (before sending it over the network or storing it) and to deserialize it (after retrieving it from the network or reading it from storage).

## 1.3 JavaBeans Components

A JavaBeans component is a Java class with certain structure requirements. JavaBeans components define and manipulate properties, which are objects of a certain type. A JavaBeans component must have a default constructor so that it can be instantiated when needed. Beans also have getter and setter methods that manipulate a bean property and conform to a specific naming convention. These structural requirements make it possible for development tools and other programs to create JavaBeans components and manipulate their properties.

Here's a simple example of a JavaBeans component.

```
public class Book {
    private String title;
    private String author;
    public Book() { setTitle(""); setAuthor(""); }
    public void setTitle(String t) { title = t; }
    public String getTitle() { return title; }
    public void setAuthor(String a) { author = a; }
    public String getAuthor() { return author; }
}
```

Why are JavaBeans components important? First and most important, they are accessible to Creator. When you write a JavaBeans component that conforms to the specified design convention, you may use it with Creator and bind JSF components to bean properties. Second, JavaBeans components can encapsulate business logic. This helps separate your User Interface (UI) components or presentation from the business data model.

In subsequent chapters, we show you several examples of JavaBeans components. We'll use a `LoginBean` to handle users that login with names and passwords and show you a `LoanBean` that calculates mortgage payments for loans. The `Point` class in Listing 1.1 on page 7 is another example of a JavaBeans component.

## 1.4 NetBeans Software

NetBeans software is an open source IDE written in the Java programming language. It also includes an API that supports building any type of application. The IDE has support for Java, but its architecture is flexible and extensible, making support for other languages possible.

NetBeans is an Open Source project. You can view more information on its history, structure, and relationship with Sun Microsystems at its web site

<http://www.netbeans.org/>

NetBeans and Creator are related because Creator is based on the NetBeans platform. In building Creator, Sun is offering an IDE aimed specifically at creating web-based applications. Thus, the IDE integrates page design with generated JSP source and page bean components. NetBeans provides features such as source code completion, workspace manipulation of windows, expandable tree views of files and components, and debugging facilities. Because NetBeans is extensible, the Creator architects included Java language features such as inheritance to adapt components from NetBeans into Creator applications with the necessary IDE functions.

## 1.5 The XML Language

XML is a metalanguage that dictates how to define custom languages and describe data. The name is an acronym for Extensible Markup Language. XML is not a programming language, however. In fact, it's based on simple character text in which the data are surrounded by text markup that documents data. This means you can use XML to describe almost anything. Since XML is self-describing, it's easy to read with tools and other programs to decide what actions to take. You can transport XML documents easily between systems or across the Internet, and virtually any type of data can be expressed and validated in an XML document. Furthermore, XML is portable because it's language and system independent.

Creator uses XML to define several configuration files as well as the source for the JSP web pages. Here's an example XML file (**managed-beans.xml**) that Creator generates for managing a JavaBeans component in a web application.

```
<faces-config>
  <managed-bean>
    <managed-bean-name>LoanBean</managed-bean-name>
    <managed-bean-class>asg.bean_examples.LoanBean
      </managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
  </managed-bean>
</faces-config>
```

Every XML file has opening tags (<tag>) and closing tags (</tag>) that define self-describing information. Here, we specify a managed-bean element

to tell Creator what it needs to know about the LoanBean component. This includes its name (LoanBean), class name and package (asg.bean-examples.LoanBean), and the scope of the bean (session). When you add your own JavaBeans components to Creator as managed beans, Creator generates this configuration information for you.

Creator maintains and updates its XML files for you, but it's a good idea to be familiar with XML syntax. This will allow you to customize the Creator XML files if necessary.

## 1.6 The J2EE Architecture

The J2EE platform gives you a multitiered application model to develop distributed components. Although any number of tiers is possible, we'll use a three-tier architecture for the applications in this book. Figure 1-1 shows the approach.

The client machine supports web browsers, applets, and stand-alone applications. A client application may be as simple as a command-line program running as an administrator client or a graphical user interface created from Java Swing or Abstract Window Toolkit (AWT) components. Regardless, the J2EE specification encourages *thin clients* in the presentation tier. A thin client is a lightweight interface that does not perform database queries, implement business logic, or connect to legacy code. These types of "heavyweight" operations preferably belong to other tiers.

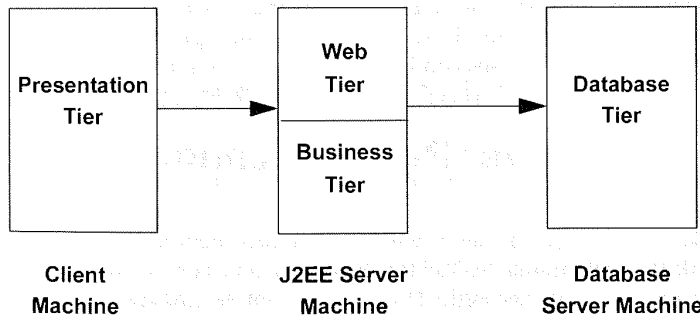


Figure 1-1 Three-tier J2EE architecture

The J2EE server machine is the center of the architecture. This middle tier contains web components and business objects managed by the application server. The web components dynamically process user requests and construct responses to client applications. The business objects implement the logic of a business domain. Both components are managed by a J2EE application server that provides these components with important system services, such as security, transaction management, naming and directory lookups, and remote connectivity. By placing these services under control of the J2EE application server, client components focus on either presentation logic or business logic. And, business objects are easier for developers to write. Furthermore, the architecture *encourages* the separation of business logic from presentation logic (or model from view).

The database server machine handles the database back end. This includes mainframe transactions, databases, Enterprise Resource Planning (ERP) systems, and legacy code. Another advantage of the three-tier architecture is that older systems can take on a whole new “look” by using the J2EE platform. This is the approach many businesses are taking as they integrate legacy systems into a modern distributed computing environment and expose application services and data to the web.

## 1.7 Java Servlet Technology

The Java Servlet component technology presents a request-response programming model in the middle tier. Servlets let you define HTTP-specific servlet classes that accept data from clients and pass them on to business objects for processing. Servlets run under the control of the J2EE application server and often extend applications hosted by web servers. Servlet code is written in Java and compiled. It is particularly suited to server-side processing for web applications since each Servlet session is handled in its own thread.

## 1.8 JavaServer Pages Technology

A JavaServer Pages (JSP) page is a text-based document interspersed with Java code. A JSP engine translates JSP text into Java Servlet code. It is then dynamically compiled and executed. This component technology lets you create dynamic web pages in the middle tier. JSP pages contain static template data (HTML, WML, and XML) and JSP elements that determine how a page constructs dynamic content. The JSP API provides an efficient, thread-based mechanism to create dynamic page content.



Creator uses JavaServer Faces (JSF), which is built on both the servlet and JSP technologies. However, by using Creator, you are shielded from much of the details of not only JSP and servlet programming, but JSF details as well.

## 1.9 JDBC API and Database Access

Java Data Base Connectivity (JDBC) is an API that lets you invoke SQL commands from Java methods in the middle tier. Typically, you use the JDBC API to access a database from servlets or JSP pages. The JDBC API has an application-level interface for database access and a service provider interface to attach JDBC drivers to the J2EE platform. In support of JDBC, J2EE application servers manage a pool of database connections. This pool provides business objects efficient access to database servers.

The JDBC `CachedRowSet` API is a newer technology that makes database access more flexible. Creator accesses configured data sources using a `CachedRowSet` object, a JavaBeans component that is scrollable, updatable, and serializable. These components are disconnected from the database, caching its rows into memory. When web applications modify data in the cached rowset object, the result propagates back to the data source through a subsequent connection. By default, Creator instantiates a cached rowset object in session scope.

The concept of *data providers* is also important because it produces a level of abstraction for data flow within Creator's application environment. Creator's data providers allow you to change the source of data (say, from a database table to a web services call or an EJB method) by hooking the data provider to a different data source.

We introduce data providers in Chapter 8 and show how to use them with databases in Chapter 9.

## 1.10 JavaServer Faces Technology

The JavaServer Faces (JSF) technology helps you develop web applications using a server-side user interface (UI) component framework. The JSF API gives you a rich set of UI components and lets you handle events, validate and convert user input, define page navigation, and support internationalization. JSF has custom tag libraries for connecting components to server-side objects. We show you these components and tag libraries in Chapter 3.

JSF incorporates many of the lower level tasks that JSP developers are used to doing. Unlike JSP applications, however, applications developed with JSF can map HTTP requests to component-specific event handlers and manage UI elements as stateful objects on the server. This means JSF offers a better separa-

tion of model and presentation. The JSF API is also layered directly on top of the Servlet API.

## 1.11 Ant Build Tool

Ant is a tool from the Apache Software Foundation ([www.apache.org](http://www.apache.org)) that helps you manage the “build” of a software application. The name is an acronym for “Another Neat Tool” and is similar in concept to older build tools like `make` under Unix and `gmake` under Linux. However, Ant is XML-based, it’s easier to use, and it’s platform independent.

Ant is written in Java and accepts instructions from XML documents. Ant is well suited for performing complicated and repetitive tasks. Creator uses Ant to compile and deploy your web applications. Ant gets its instructions for building a system from the configuration file, `build.xml`. You won’t have to know too much about Ant to use Creator, but you should be aware that it’s behind the scenes doing a lot of work for you.

## 1.12 Web Services

Web services are software APIs that are accessible over a network in a heterogeneous environment. Network accessibility is achieved by means of a set of XML-based open standards such as the Web Services Description Language (WSDL), the Simple Object Access Protocol (SOAP), and Universal Description, Discovery, and Integration (UDDI). Web service providers and clients use these standards to define, publish, and access web services.

Creator’s application server (J2EE 1.4) provides support for web services. In Creator, you can access methods of a web service by dragging its node onto the design canvas. We show you web services with Creator in Chapter 10.

## 1.13 Enterprise JavaBeans (EJB)

EJB is a component technology that helps developers create business objects in the middle tier. These business objects (enterprise beans) consist of fields and methods that implement business logic. EJBs are server-side components written in Java that serve as building blocks for enterprise systems. They perform specific tasks by themselves, or forward operations to other enterprise beans. EJBs are under control of the J2EE application server. We show you how to access EJBs from Creator applications in Chapter 11.

## 1.14 Portlets

A portlet is an application that runs on a web site managed by a server called a *portal*. A portal server manages multiple portlet applications, displaying them on the web page together. Each portlet consumes a fragment of the page and manages its own information and user interaction. Portlet application developers will typically target portlets to run under portals provided by various portal vendors.

You can use Creator to develop portlets. Creator builds JSF portlets. This means your design-time experience in building portlet web application using the visual, drag-and-drop features of Creator will be familiar. Most of the interaction with the IDE is exactly the same as it is for non-portlet JSF projects. We show you how to create portlets in Chapter 12.

## 1.15 Key Point Summary

- Creator is an IDE built on layered Java technologies that helps you build web applications.
- Procedure-oriented languages separate data and functions, whereas object-oriented languages combine them.
- Encapsulation enforces data hiding and allows you to control access to your objects.
- Java is a strongly typed object-oriented language with a large set of APIs that help you develop portable web applications.
- In Java, operator `new` returns a reference to a newly created object so that you can call methods with the reference.
- Java classes have fields, constructors, and instance methods. The `private` keyword is used for encapsulation, and the `public` keyword grants access to clients.
- Java packages allow you to store class files and retrieve them with `import` statements in Java programs.
- Java uses `try`, `catch`, and `throw` to handle error conditions with a built-in exception handling mechanism.
- Inheritance is a code reuse mechanism that implements an “is a” relationship between classes.
- Dynamically bound method calls are resolved at run time in Java. Dynamic binding is essential with distributed web applications.
- An interface has no fields and only abstract public methods. A class that implements an interface must provide code for the interface’s methods.
- The J2EE architecture is a multitiered application model to develop distributed components.

- Java Servlets let you define HTTP-specific servlet classes that accept data from clients and pass them on to business objects for processing.
- A JSP page is a text-based document interspersed with Java code that allows you to create dynamic web pages.
- JDBC is an API for database access from servlets, JSP pages, or JSF. Creator uses data providers to introduce a level of abstraction between Creator UI components and sources of data.
- JavaServer Faces (JSF) helps you develop web applications using a server-side user interface component framework. Creator generates and manages all of the configuration files required by JSF.
- A JavaBeans component is a Java class with a default constructor and setter and getter methods to manipulate its properties.
- NetBeans is a standards-based IDE and platform written in the Java programming language. Java Studio Creator is based on the NetBeans platform.
- XML is a self-describing, text-based language that documents data and makes it easy to transport between systems.
- Ant is a Java build tool that helps you compile and deploy web applications.
- Web services are software APIs that are accessible over a network in a heterogeneous environment.
- EJBs are server-side components written in Java that implement business logic and serve as building blocks for enterprise systems.
- Portlets are applications that consume a portion of a web page. They run on web sites managed by a portal server and execute along with other portlets on the page.
- Portlets help divide web pages into smaller, more manageable fragments.

cept data

that allows

of F. Creator  
creator UI

a server-  
manages

and setter

iva  
beans

a and

applications.  
in a

business

ey run on  
r portlets

gments.

# CREATOR BASICS

## Topics in This Chapter

- Creator Window Layout
- Visual Design Editor
- Components and Clips Palette
- Source Editors/Code Completion
- Page Navigation Editor
- Outline Window
- Projects Window
- Servers and Resources
- Creator Help System
- Basic Project Building

# Chapter 2

**S**un Java Studio Creator makes it easy to work with web applications from multiple points of view. This chapter explores some of Creator's basic capabilities, the different windows (views) and the way in which you use them to build your application. We show you how to manipulate your application through the drag-and-drop mechanism for placing components, configuring components in the Properties window, controlling page flow with the Page Navigation editor, and selecting services from the Servers window.

## 2.1 Examples Installation

We assume that you've successfully installed Creator. The best source of information for installing Creator is Sun's product information page at the following URL.

<http://developers.sun.com/prodtech/javatools/jscreator/>

Creator runs on a variety of platforms and can be configured with different application servers and JDBC database drivers. However, to run all our examples we've used the bundled application server (Sun Java System Application Server 8.2) and the bundled database server (Derby). Once you've configured

Creator for your system, the examples you build here should run the same on your system.

## Download Examples

You can download the examples for this book at the Sun Creator web site. The examples are packed in a zip file. When you unzip the file, you'll see the **FieldGuide2/Examples** directory and subdirectories for the various chapters and projects. As each chapter references the examples, you will be instructed on how to access the files.

You're now ready to start the tour of Creator.

## 2.2 Creator Views

Figure 2-1 shows Creator's initial window layout in its default configuration. When you first bring it up, no projects are open and Creator displays its Welcome window.

There are other windows besides those shown in the initial window layout. As you'll see, you can hide and display windows, as well as move them around. As we begin this tour of Creator, you'll probably want to run Creator while reading the text.

### Welcome Window

The Welcome window lets you create new projects or work on existing ones. Figure 2-2 shows the Welcome window in more detail. It lists the projects you've worked on recently and offers selection buttons for opening existing projects or creating new projects. If you hover with the mouse over a recently opened project name, Creator displays the full pathname of the project in a tooltip.

To demonstrate Creator, let's open a project that we've already built. The project is included in the book's download bundle, in directory **FieldGuide2/Examples/Navigation/Projects/Login1**.



#### Creator Tip

---

*We show you how to build this project from scratch in Chapter 5 (see "Dynamic Navigation" on page 206). For our tour of the IDE, however, we'll use the pre-built project from the examples download.*

---



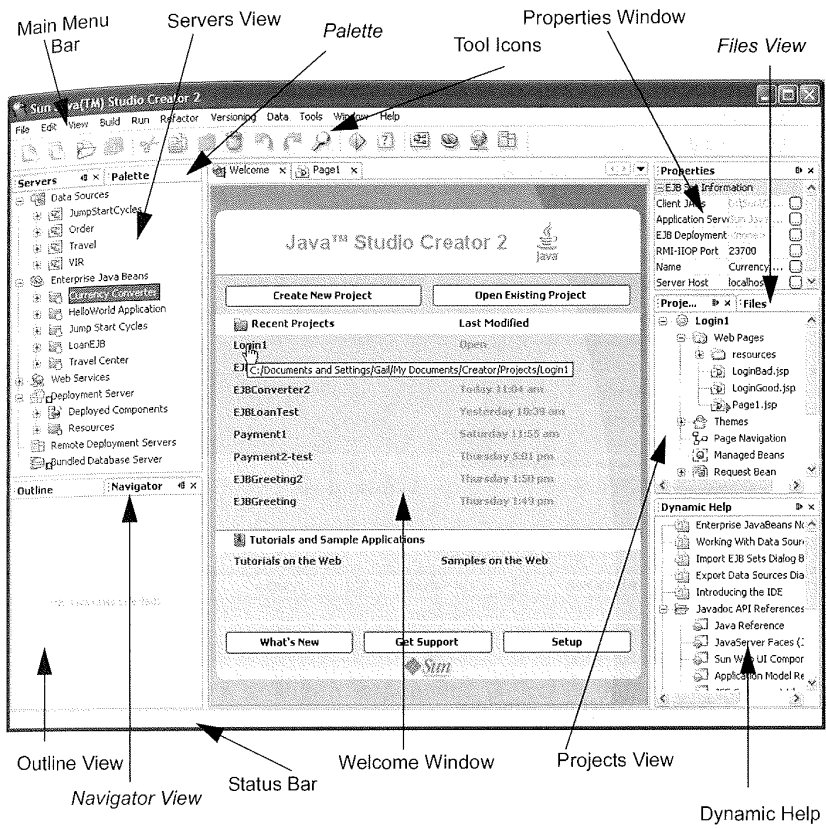


Figure 2-1 Creator's initial window layout

1. Select the Open Existing Project button and browse to the **FieldGuide2/Examples/Navigation/Projects** directory.
2. Select **Login1** (look for the projects icon) and click Open Project Folder. This opens the **Login1** project in the Creator IDE.
3. Page1 should display in the visual editor, as shown in Figure 2-3. If Page1 does not open in the design editor, find the Projects view (its default position is on the right, under the Properties view).
4. In the Projects view, expand node Login1, then Web Pages. Double-click **Page1.jsp**. Page1 should now appear in the design editor.

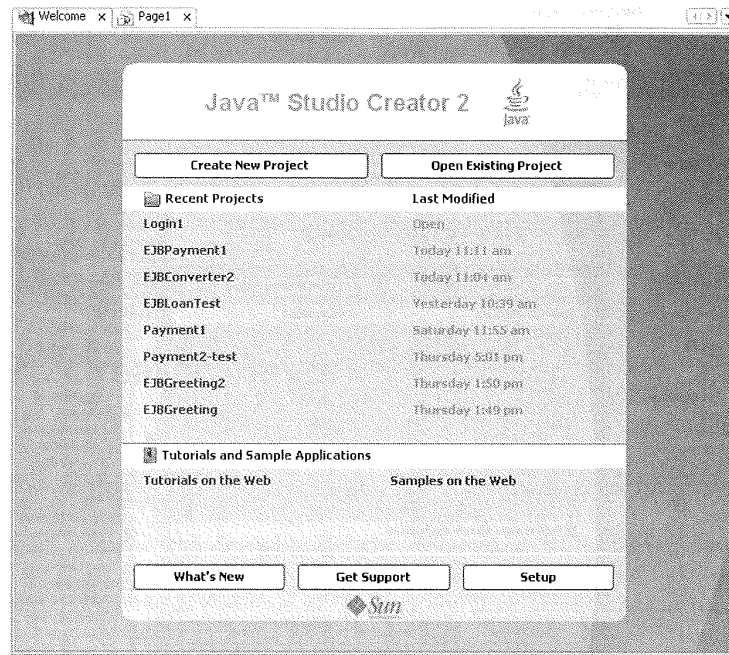


Figure 2-2 Creator's Welcome window

## Design Editor

Figure 2-3 shows a close-up of the design canvas (the visual design editor) of Page1. You see the design grid and the components we've placed on the canvas. The design editor lets you visually populate the page with components.

Page1 contains a "virtual form." Virtual forms are accessible on a page by selecting the Show Virtual Forms icon on the editing toolbar, as shown in Figure 2-3. Virtual forms let you assign different components to different actions on the page. We show you how to use virtual forms in "Configure Virtual Forms" on page 216 (for project Login1 in Chapter 5) and in "Virtual Forms" on page 417 (for project MusicAdd in Chapter 9).

Select the text field component. The design canvas marks the component with selection and resizing handles. Now move the text field component around on the grid. You'll note that it snaps to the grid marks automatically when you release the mouse. You can temporarily disable the snap to grid feature by moving the component and pressing the Shift key at the same time. You can also select more than one component at a time (use <Shift+Click>) and Cre-

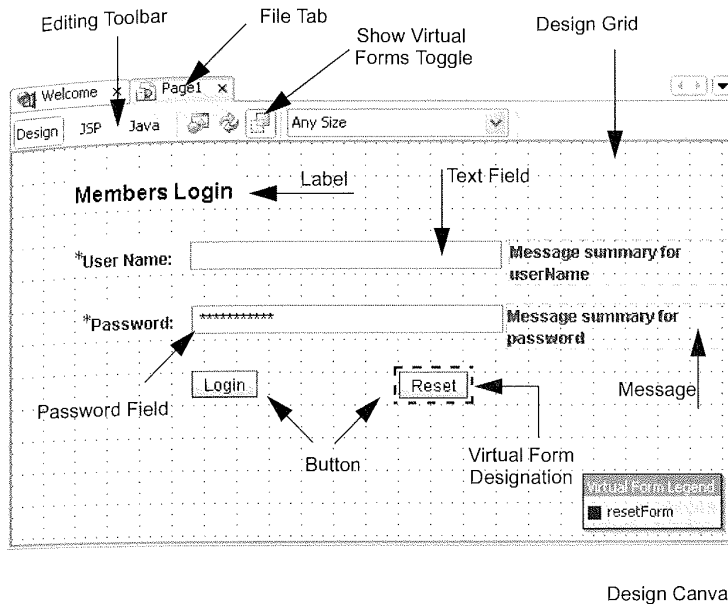


Figure 2-3 Creator's design canvas showing project Login1

ator provides options to align components. We cover the mechanics of page design in Chapter 7 (see "Using the Visual Design Editor" on page 273).

Note that when you make modifications to a page, Creator indicates that changes have been made to the project by appending an asterisk to the file name tab. Once you save your project by clicking the Save All icon in the toolbar (or selecting File > Save All from the main menu), the Save All icon is disabled and the asterisk is cleared from the file name tab.

Typically applications consist of more than one page. You can have more than one of your project's pages open at a time (currently, there's just one page open). When you open other files, a file tab appears at the top of the editor pane. The file tab lets you select other files to display in the editor pane.

Creator lets you configure your display's workspace to suit the tasks you're working on. All the windows can be hidden when not needed (click the small x in a window's title bar to close it) and moved (grab the window's title bar and move it to a new location). To view a hidden window, select View from the menu bar and then the window name. Figure 2-4 shows the View menu with the various windows you can open, along with a key stroke shortcut for opening each window.

You can also dock Creator windows by selecting the pushpin in the window title bar. This action minimizes the window along the left or right side of the

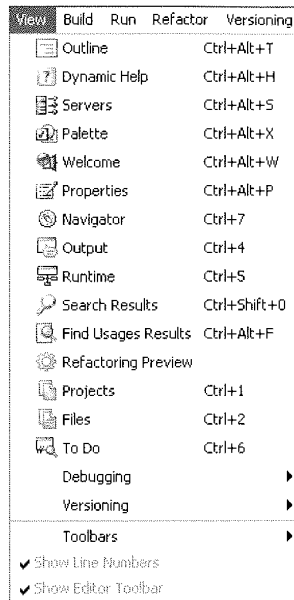


Figure 2-4 Creator's View Menu allows you to select specific views of your project

workspace. Make it visible again by moving the cursor over its docked position. Toggling the pushpin icon undocks the window. Figure 2-5 shows the Properties view with both the Projects and Files windows docked.

## Properties

As you select individual components, their properties appear in the Properties window. Select the text field component on the design canvas. This brings up its Properties window, as shown in Figure 2-5.

Creator lets you configure the components you use by manipulating their properties. When you change a component's properties, Creator automatically updates the underlying JSP source for you. Let's look at several properties of the text field component. If you hold the cursor over any property value, Creator displays its setting in a tooltip.

Components have many properties in common; other properties are unique to the specific component type. The `id` property uniquely identifies the component on the page. Creator generates the name for you, but you can change it (as we have in this example) to more easily work with generated code. The `label` property enables you to specify a textual label associated with the text field. The red asterisk next to the label in the design view indicates that input is

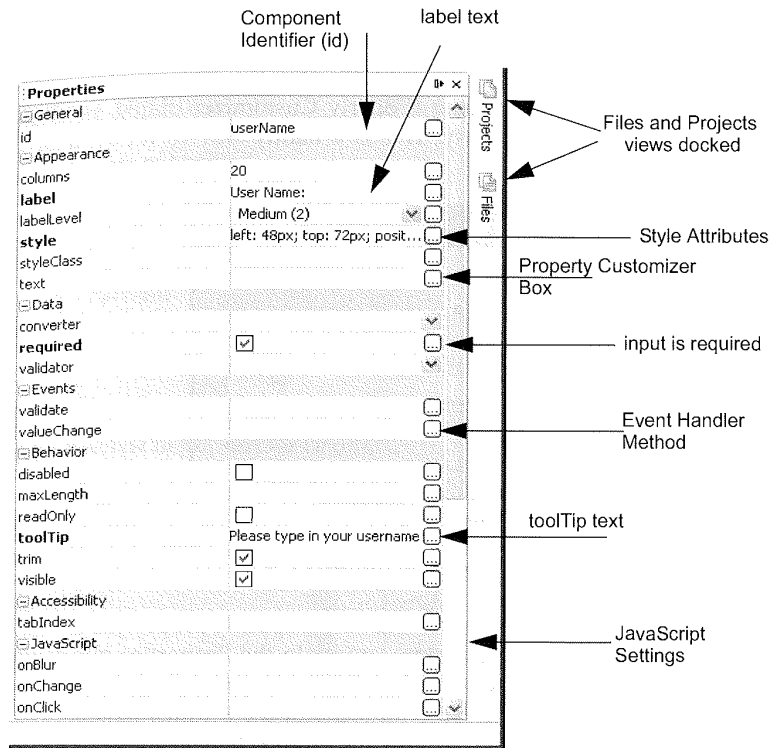


Figure 2-5 Properties window for text field component "userName"

r project  
 cked posi-  
 shows the

Properties  
 brings up  
 uting their  
 omatically  
 roperties of  
 alue, Cre-  
 re unique  
 re compo-  
 ange it (as  
 he label  
 text field.  
 t input is

required for this component. Property text holds the text that the user submits. You can use the style property to change a component's appearance. The style property's position tag reflects the component's position on the page. When you move the component in the design view, Creator updates this for you.

Property styleClass takes previously defined CSS style classes (you can apply more than one). File **stylesheet.css** (under Web Pages > resources in the Projects window) is the default style sheet for your projects. We cover style, styleClass and using Creator's preconfigured Themes in Chapter 7.

Text field components can take a converter (specified in property converter) and a validator (property validator). The validate and valueChange properties (under Events) expect method names and are used to provide custom validation or to process input when the component's text value changes.

Click on the text field component (again) in the design canvas until Creator displays a gray outline around the component. Now type in some text and finish editing with **<Enter>**. The text you type appears opposite property text in the Properties window. To reset the property, click the customizer box opposite property text. Creator pops up a Property Customizer dialog, as shown in Figure 2-6. Select Unset Property. This is a handy way to return a property value to its unset state.

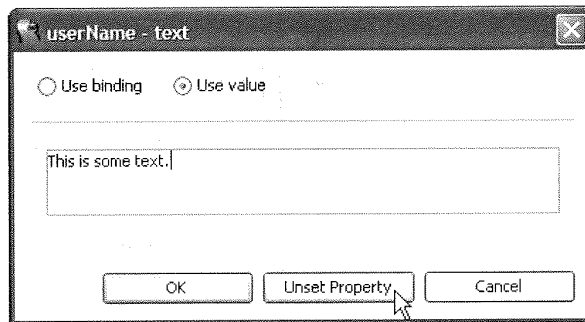


Figure 2-6 Property customizer dialog for property text

Each property's customizer is tailored to the specific property. For example, select the Login button on the design canvas. In the Properties window, click the property customizer box opposite property *style*. Creator pops up an elaborate style editor. Experiment with some of the settings (change the font style or color, for example) and see how the button changes in the design view. You can also preview the look. Right-click inside the design view and select Preview in Browser. Figure 2-7 shows a preview of Login1 with a different appearance for the Login button.

## Palette

Creator provides a rich set of basic components, as well as special-function components such as Calendar, File Upload, Tab Set, and Tree. The palette is divided into sections that can be expanded or collapsed. Figure 2-8 shows the Basic Components palette, which includes all of the components on Page1 of project Login1. In Figure 2-8 you also see the Layout and Composite Components palette.

The palette lets you drag and drop components on the page of your application. Once a component is on a page, you can reposition it with the mouse or configure it with the Properties window.

Figure 2-9 shows the Validators and Converters palette. Creator's converters and validators let you specify how to convert and validate input. Because con-

util Creator  
xt and fin-  
ty text in  
x opposite  
shown in  
a property

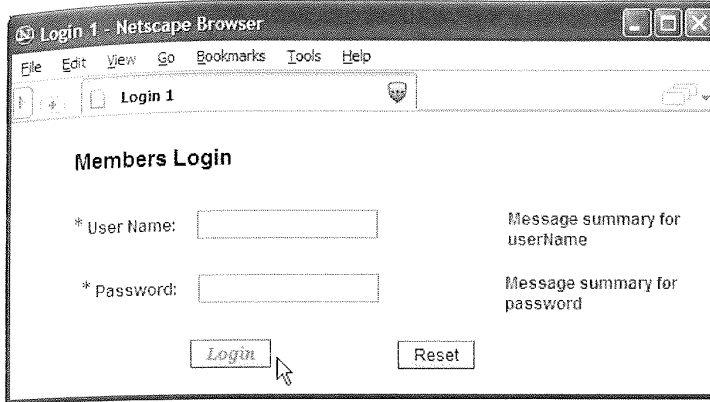


Figure 2-7 Preview in Browser for Login1

r example,  
dow, click  
ps up an  
; the font  
:sign view.  
and select  
a different

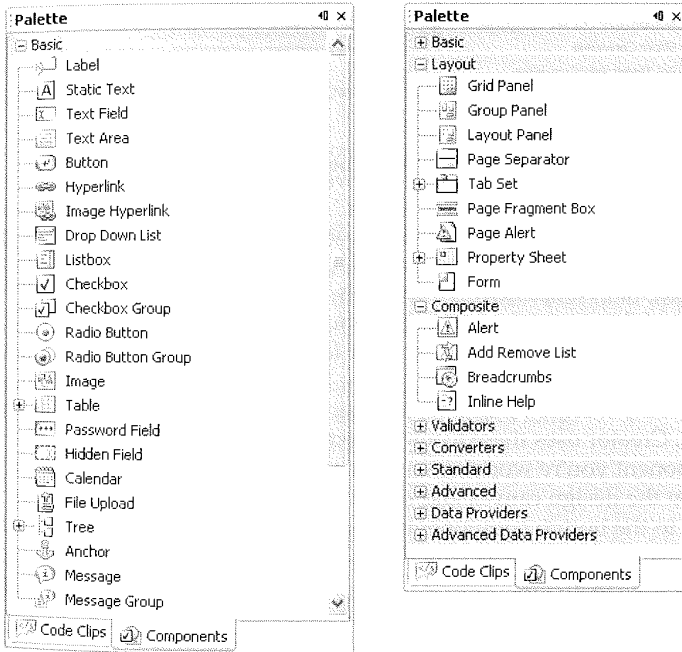


Figure 2-8 Basic, Layout and Composite Components palette

il-function  
palette is  
shows the  
1 Page1 of  
e Compo-

ir applica-  
mouse or

version and validation are built into the JSF application model, developers can concentrate on providing event handling code for valid input.

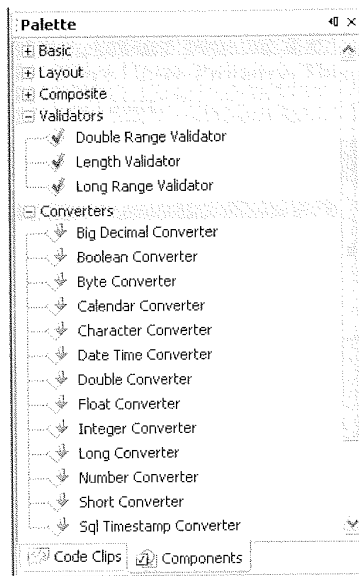


Figure 2-9 Creator Validators and Converters Components palette

You select converters and validators just like the UI components. When you drag one to the canvas and drop it on top of a component, the validator or converter binds to that component. To test this, select the Length Validator and drop it on top of the `userName` text field component. You'll see a length validator `lengthValidator1` defined for the text field's `validator` property in the Properties window.

Note that components, validators, and converters all have associated icons in the palette. Creator uses these icons consistently so you can easily spot what kind of component you're working with. For example, select the Login button component on the design canvas and examine the Outline view. You'll see that the icon next to the button components (Login and Reset) matches component Button in the Basic Components palette.

## Outline

Figure 2-10 is the Page1 Outline view for project **Login1**. (Its default placement is in the lower-left portion of the display.) The Outline window is handy for showing both visual and nonvisual components for the page that's currently



developers can

displayed in the design canvas. You can select the preconfigured managed beans, RequestBean1, SessionBean1 and ApplicationBean1. These JavaBeans components hold your project's data that belong in either request (page), session or application scope, respectively. (We discuss scope issues for web application objects in "Scope of Web Applications" on page 224.)

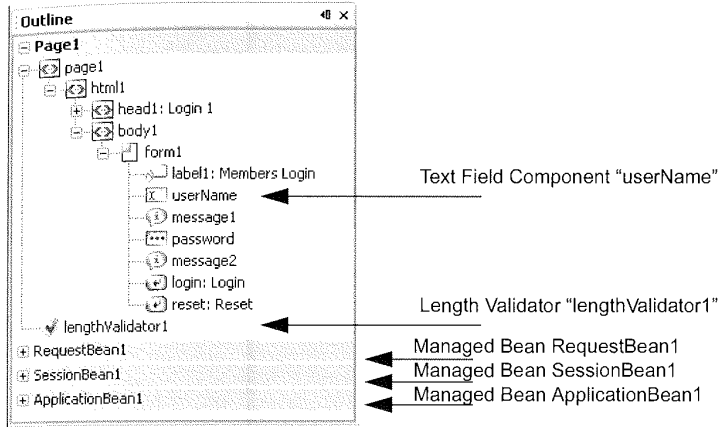


Figure 2-10 Creator's Outline window for project Login1

Some components are composite components (they contain nested elements). The Outline window shows composite components as nodes that you can expand and compress with '+' and '-' as needed. Suppose, for example, you select grid panel for layout. When you add components to this grid panel, they appear nested underneath the panel component in the Outline view.

The length validator component on the `userName` text field appears as component `lengthValidator1` in the Outline view. Select the length validator and examine it in the Properties view. Specify values for properties `maximum` (use 10) and `minimum` (use 3). This limits input for the `userName` text field component to a string that is between 3 and 10 characters long.

Now let's look at the Projects window.

### Projects

Figure 2-11 shows the Projects window for project **Login1**. Its default location is in the lower-right corner. Whereas the Outline view displays components for individual pages and managed beans, the Projects window displays your entire project, organized in a logical hierarchy. (Since Creator lets you open more than one project at a time, the Projects window displays all currently opened projects.) Project **Login1** contains three JSP pages under the Web Pages node: **Page1.jsp**, **LoginGood.jsp**, and **LoginBad.jsp**. Double-click any one of

When you  
tor or con-  
idator and  
gth valida-  
erty in the

iated icons  
spot what  
gin button  
I'll see that  
component

placement  
handy for  
s currently

them to bring it up in the design canvas. When the page opens, Creator displays a file name tab so you can easily switch among different open files in the design canvas.

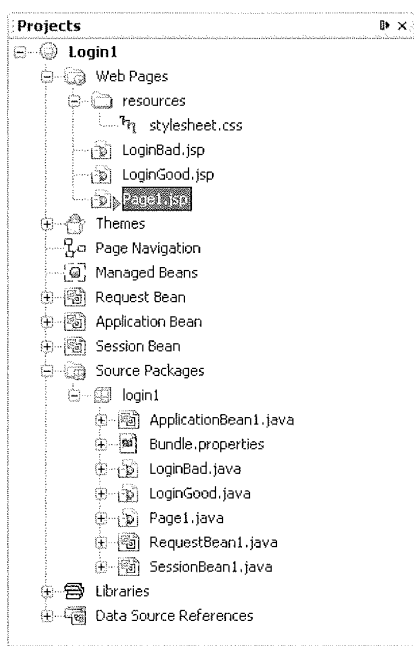


Figure 2-11 Creator's Project Navigator window for project Login1

When you create your own projects, each page has its own Java component "page bean." These are Java classes that conform to the JavaBeans structure we mention in Chapter 1 (see "JavaBeans Components" on page 13). To see the Java files in this project, expand the Source Packages node (click on the '+'), then the **login1** folder. When you double-click on any of the Java files, Creator brings it up in the Java source editor. (We'll examine the Java source editor shortly.) Without going to the editor, you can also see Java classes, fields, constructors, and methods by expanding the '+' next to each level of the Java file.

The Projects view displays Creator's "scoped beans." These are pre-configured JavaBeans components that store data for your project in different scopes. You can use request scope (Request Bean), application scope (Application Bean), or session scope (Session Bean). Many of the projects in this text add properties to these beans. We discuss JSF scoping issues in Chapter 6 (see "Pre-defined Creator Java Objects" on page 226).

Creator displays files in the

The Projects view also lists the resources node, which lives under the Web Pages node. The resources node typically holds file **stylesheet.css** and any image files. Creator uses the libraries listed in the Libraries node to display, build, and deploy your application. These class files (compiled Java classes) are stored in special archive files called JAR (Java Archive) files. You can see the name, as well as the contents (down to the field and method names) of any JAR file by expanding the nodes under Libraries. We show you how to add a JAR file to your project in Chapter 13 (see “Add the asg.jar Jar File” on page 595).

## Files

The Projects window shows you a logical view of your project. Sometimes you may need to access a configuration file that is not included in the Projects view. In such a case, use the Files view, as shown in Figure 2–12.

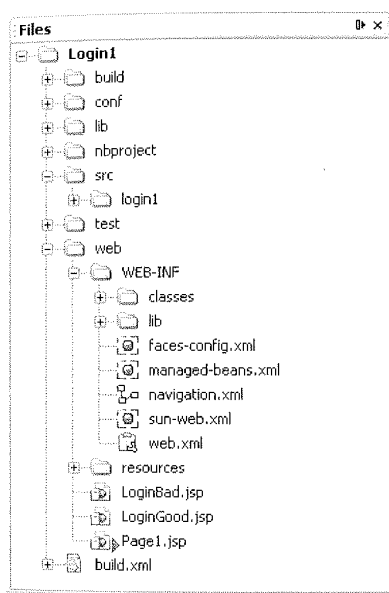


Figure 2–12 Files view for project Login1

The Files view shows all of the files in your project. For example, expand node **web** > **WEB-INF** and double-click file **web.xml**. Creator brings up file **web.xml** in a specialized Creator-configuration XML editor, as shown in Figure 2–13.

a component structure we). To see the files, Creator source editor displays fields, content, and Java file. The pre-configured scopes. (Application this text add r 6 (see “Pre-

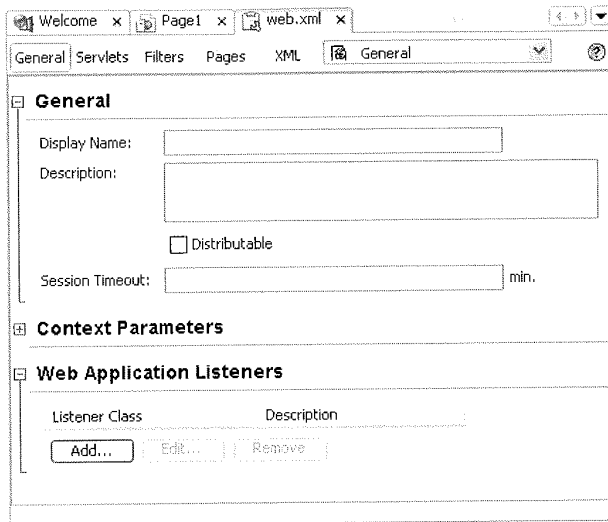


Figure 2-13 Editing file web.xml

File `web.xml` lets you set various project-level configuration parameters, such as Session Timeout, Filters, or special error pages. Close this file by clicking the small x in the `web.xml` file tab.

## JSP Editor

As you drop components on the page and configure them with the Properties window, Creator generates JSP source code for your application. You can view the JSP representation of a page by clicking the JSP button in the editing toolbar, as shown in Figure 2-14.

Normally, you will not need to edit this page directly, but studying it is a good way to understand how Creator UI components work and how to manage their properties. You'll see a close correspondence between the JSP tags and the components' properties as shown in the Properties window. If you do edit the JSP source directly, you can easily return to the design view. Creator always keeps the design view synchronized with the JSP source.

Tags in the JSP source use a JSF Expression Language (EL) to refer to methods and properties in the Java page bean. For example, the login button's action property is set to `#{Page1.login_action}`, which references method `login_action()` in class `Page1.java`.

Creator also generates and maintains code for the "page bean," the Java backing code generated for each page. Let's look at the Java source for `Page1.java` now.

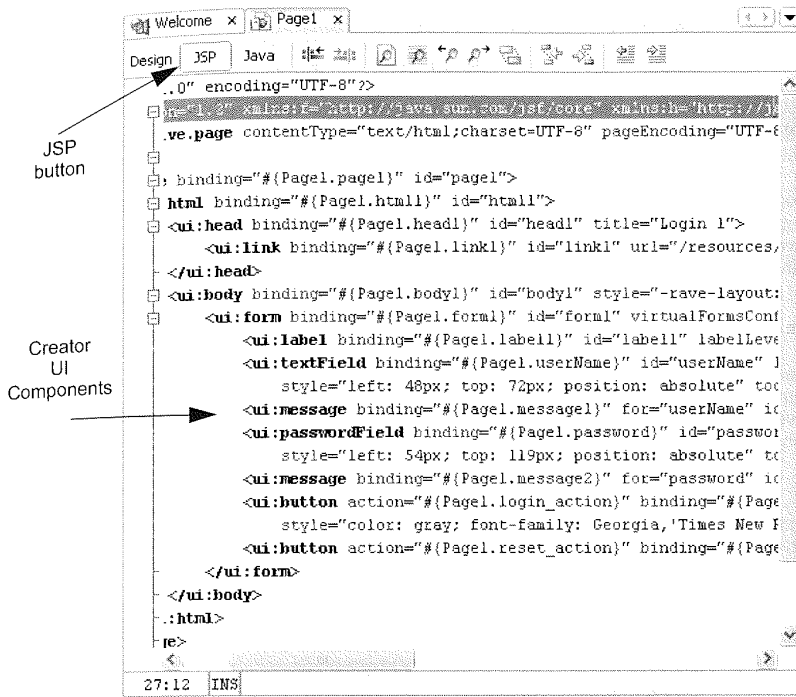


Figure 2-14 Page1.jsp XML Editor

### Java Source Editor

Click the Design button and return to the Page1 design view. As you build your application, not only does Creator generate JSP source that defines and configures your component, but it also maintains the page bean. For example, Creator makes it easy for you to code event handlers (methods that perform customized tasks when the user selects an option from a drop down list or clicks a button). Double-click button Login in the design view. Creator generates a default event handler for this button and puts the cursor at the method in the Java source editor. If this method was previously generated (as it was here), Creator brings up the editor and puts the cursor at the method, as shown in Figure 2-15. Here you see method login\_action() in file Page1.java.

You can always bring up a page's Java code by selecting the Java button in the editing toolbar. This Java file is a bean (conforming to a JavaBeans structure) and its properties consist of the components you place on the page. Each

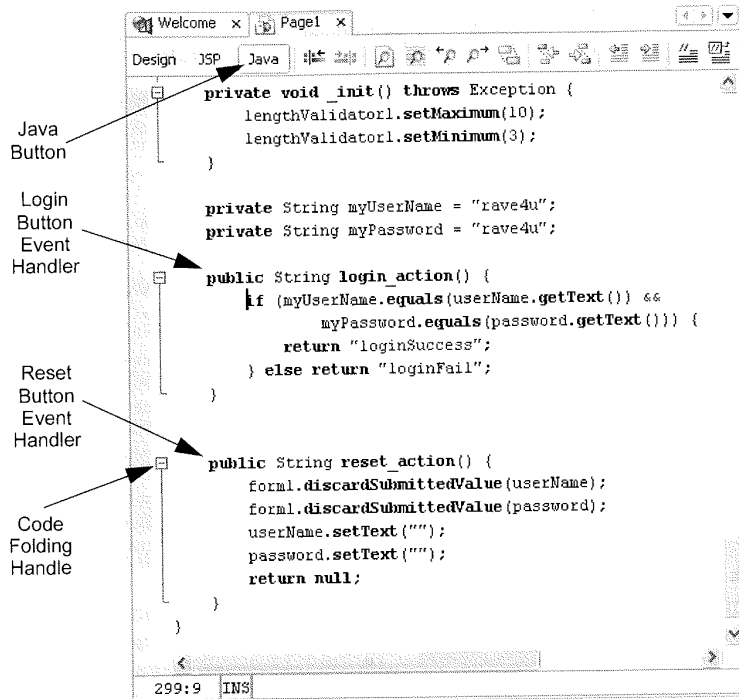


Figure 2-15 Page1.java in Java source editor

component corresponds to a private variable and has a getter and setter. This allows the JSF EL expression to access properties of the page bean.

All of Creator's editors are based on NetBeans. The Editor Module is a full-featured source editor and provides code completion (we show an example shortly), a set of abbreviations, and fast import with **<Alt+Shift+I>**. The editor also has several useful commands: Reformat Code (handy when pasting code from an external source), Fix Imports (adds needed import statements as well as removes unused ones), and Show Javadoc (displays documentation for classes and methods). There are more selections in the context menu (right-click inside the editor to see the menu). Sections of the Creator-generated code are folded by default to help keep the editing pane uncluttered. You can unfold (select '+') or fold (select '-') sections as you work with the source code. You can also view line numbers: place the mouse in the blue left-margin, right-click, and select Show Line Numbers.

To see the set of abbreviations for the Java editor, select Tools > Options from the main menu bar. The Options dialog pops up. Under Options, select Editing

> Editor Settings > Java Editor. On the right side of the display, click the small editing box next to Abbreviations. Creator pops up the window shown in Figure 2-16.

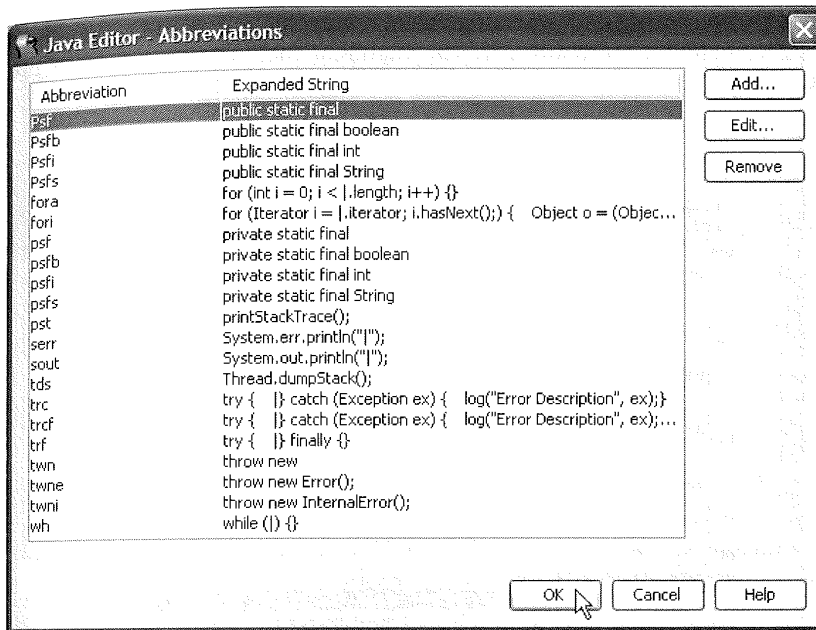


Figure 2-16 Java source editor list of abbreviations

id setter. This

odule is a full-  
an example  
|. The editor  
pasting code  
nents as well  
mentation for  
menu (right-  
nerated code  
ou can unfold  
ode. You can  
n, right-click,

Options from  
select Editing

The window lists the abbreviations in effect for your Java editor. (You can edit, add, or remove any item.) For example, to place a `for` loop in your Java source file, type the sequence `fora` (for array) followed by `<Space>`. The editor generates

```
for (int i = 0; i < .length; i++) {
}
```

and places the cursor in front of `.length` so that you can add an array name. (`.length` refers to the length of the array object. This code snippet lets you easily loop through the elements of the array.)

The Java source editor also helps you with Java syntax and code completion. All Java keywords are bold, and variables and literal Strings have unique colors.

When you add statements to your Java source code, the editor dynamically marks syntax errors (in red, of course). The editor also pops up windows to help with code completion and package location for classes you need to reference (press **<Ctrl+Space>** to activate the code completion window). If available, code completion includes Javadoc help. For example, Figure 2-17 shows the code completion mechanism as you highlight method `equals()` and press **<Ctrl+Space>**.

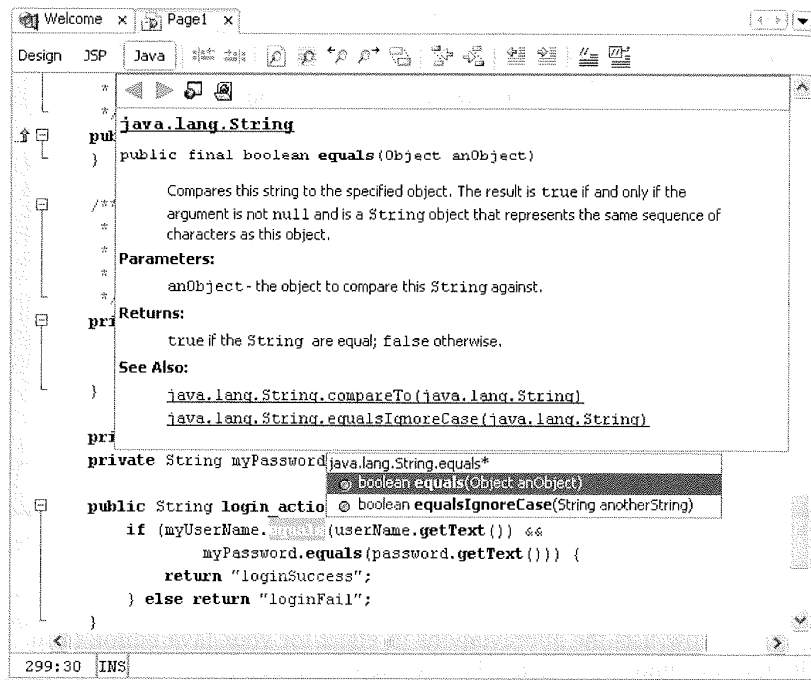


Figure 2-17 Java source editor code completion

When you use the down-arrow to select the second method, `equalsIgnoreCase()`, the help mechanism displays its Javadoc documentation. (To retrieve Javadoc documentation on any class in your source file, select it and press **<Ctrl+Shift+Space>**.) The Java Source editor is discussed in more detail in Chapter 4 (see "Using the Java Source Editor" on page 136).

When the Java source editor is active, Creator also activates the Navigator window, as shown in Figure 2-18. The Navigator window lets you go to a method or field within the Java source editor by clicking its name in the window. In Figure 2-18, the cursor hovers over method `destroy()`, displaying help in a tooltip.



dynamically  
windows to  
eed to refer-  
If available,  
7 shows the  
) and press

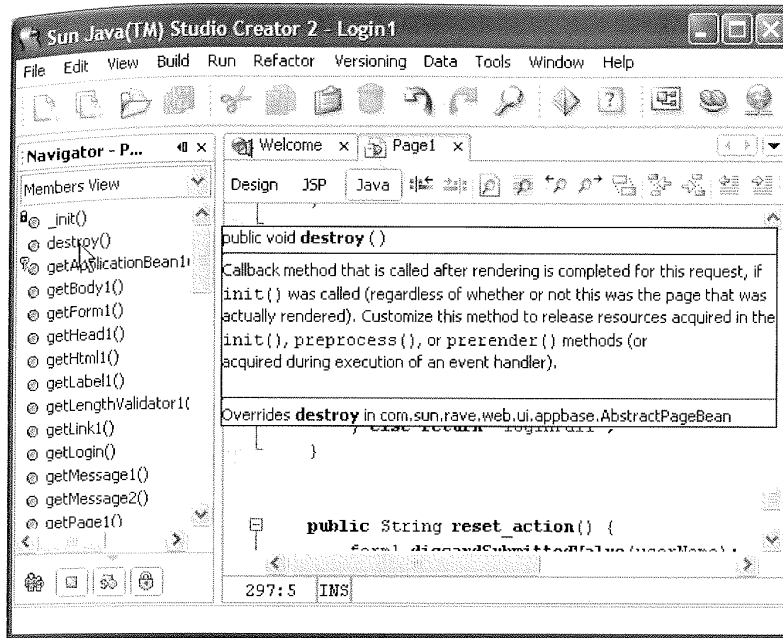


Figure 2-18 Navigator view and help for method `destroy()` displayed

## Code Clips Palette

When the Java Source editor is displayed, Creator replaces the Components palette with the Code Clips palette, as shown in Figure 2-19. Here we show several sections, including the code clips for Application Data. Highlight clip Store Value in Session in this section. If you hold the cursor over the clip name, Creator displays a snippet window. You can drag and drop the clip directly into your Java source file.

To view or edit a clip, select it, right-click, and choose Edit Code Clip. Figure 2-20 shows the Store Value in Session code clip.

The Code Clips palette is divided into categories to show sample code for different programming tasks. For example, if you click Application Data, you'll see a listing of clips that let you access application data from different scopes in your web application.

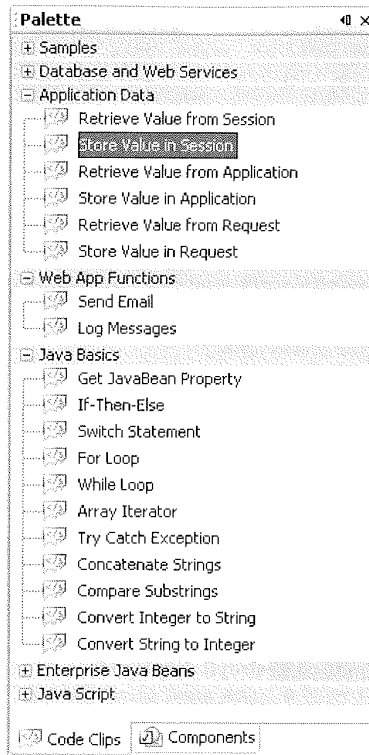


Figure 2-19 Java Clips Palette

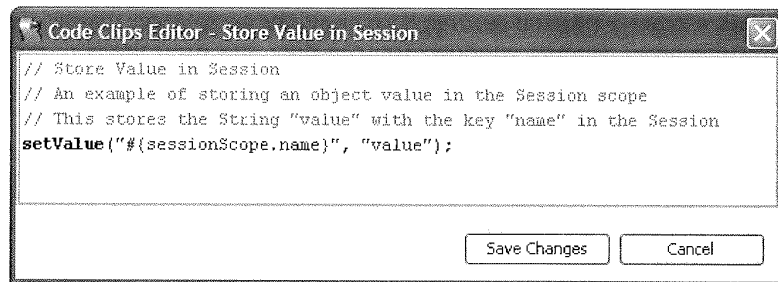


Figure 2-20 Code Clips Editor

## Page Navigation Editor

Return to the Java Source window and examine method `login_action()`. You'll see that `login_action()` returns one of two Strings (either "loginFail" or "loginSuccess") to the action event handler. The action event handler then passes the String on to the navigation handler to determine page flow. Let's look at the Page Navigation editor now.

1. From the top of the Java source window, select the Design button. This returns you to the design canvas for this page.
2. Now right-click in the design canvas and select Page Navigation from the context menu. Creator brings up the Page Navigation editor for project **Login1**, as shown in Figure 2-21.

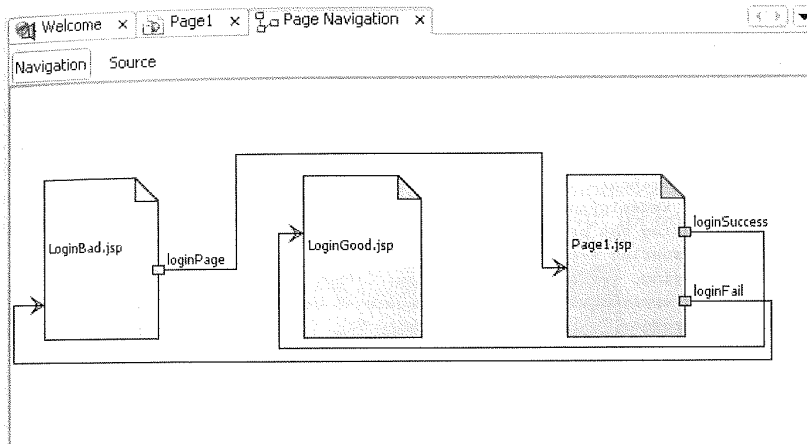


Figure 2-21 Page navigation editor for project Login1

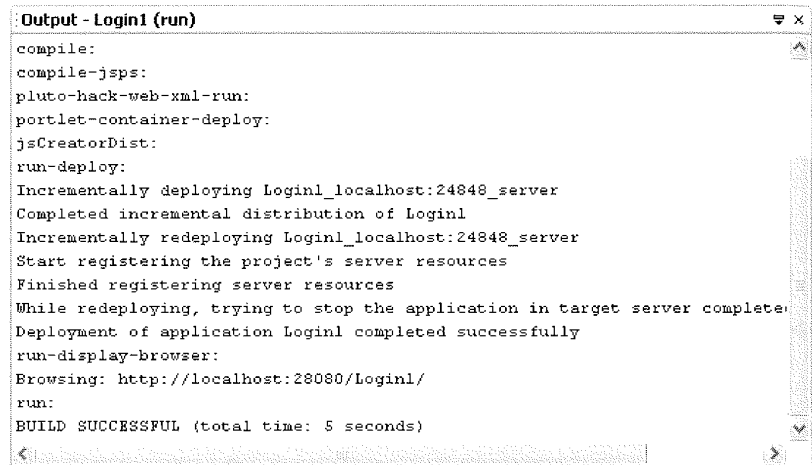
There are three pages in this project. The Page Navigation editor displays each page and indicates page flow logic with labeled arrows. The two labels originating from page **Page1.jsp** correspond to the return Strings in action method `login_action()`.

Chapter 5 shows you how to specify navigation in your applications (see "Page Navigation" on page 188). The Page Navigation editor is also a handy way to bring up any of the project's pages: just double-click inside the page. Once you've visited the Page Navigation editor, Creator displays a file tab called **Page Navigation** so you can easily return to it.

Before we explore our project any further, let's have you deploy and run the application. From the menu bar, select Run > Run Main Project. (Or, click the green Run arrow on the icon toolbar, which also builds and runs your project.)

## Output Window

Figure 2-22 shows the output window after building and deploying project Login1. Creator uses the Ant build tool to control project builds. This Ant build process requires compiling Java source files and assembling resources used by the project into an archive file called a WAR (Web Archive) file. Ant reads its instructions from a Creator-generated XML configuration file, called `build.xml`, in the project's directory structure.



```

Output - Login1 (run)
compile:
compile-jsp:
pluto-hack-web-xml-run:
portlet-container-deploy:
jsCreatorDist:
run-deploy:
Incrementally deploying Login1_localhost:24848_server
Completed incremental distribution of Login1
Incrementally redeploying Login1_localhost:24848_server
Start registering the project's server resources
Finished registering server resources
While redeploying, trying to stop the application in target server complete
Deployment of application Login1 completed successfully
run-display-browser:
Browsing: http://localhost:28080/Login1/
run:
BUILD SUCCESSFUL (total time: 5 seconds)

```

Figure 2-22 Output window after building and deploying project Login1

If problems occur during the build process, Creator displays messages in the Output window. A compilation error with the Java source is the type of error that causes the build to fail. When a build succeeds (the window shows `BUILD SUCCESSFUL`, as you see Figure 2-22), Creator tells the application server to deploy the application. If the application server is not running, Creator starts it for you. If errors occur in this step, messages appear in the Outline window from the application server.

Finally, it's possible that the deployment is successful but a runtime error occurs. In this situation, the system throws an exception and displays a stack trace on the browser's web page. Likely sources for these errors are problems with JSP tags on the JSP page, resources that are not available for the runtime class loader, or objects that have not been properly initialized.

and run the  
(Or, click the  
our project.)

ying project  
his Ant build  
rces used by  
Ant reads its  
file, called



essages in the  
ype of error  
hows BUILD  
n server to  
ator starts it  
ne window

ntime error  
lays a stack  
e problems  
the runtime

When the build/deployment process is complete, Creator brings up your browser with the correct URL. (Here the status window displays "Browsing: http://localhost:28080/Login1/.") To run project **Login1** with the Sun bundled Application Server, Creator generates this web address.

http://localhost:28080/Login1/

You use `localhost` if you're running the application server on your own machine; otherwise, use the Internet address or host name where the server is running. The port number 28080 is unique to Sun's bundled J2EE application server. Other servers will use a different port number here.

Note that the Context Root is `/Login1` for this application. The application server builds a directory structure for each deployed application; the context root is the "base address" for all the resources that your application uses.

Figure 2-23 shows the **Login1** project deployed and running in a browser. The Password field's tooltip is displayed. Both the User Name and Password input fields have asterisks, indicating required input. Type in some values for User Name and Password. If you leave the User Name field empty or type less than 3 characters or more than 10, you'll get a validation error. (The minimum and maximum number of characters only apply if you added a length validator earlier.) The correct User Name and Password is "rave4u" for both fields.

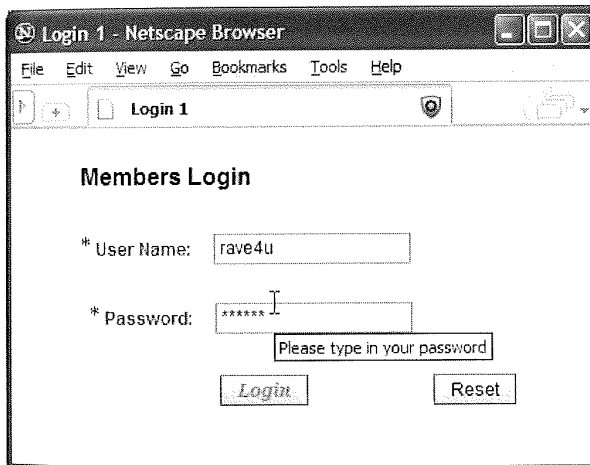


Figure 2-23 Project Login1 running in a browser

If you supply the correct values and click the Login button, the program displays page `LoginGood.jsp`. Incorrect values display `LoginBad.jsp`.

It's time now to explore the Servers window, located in the upper-left portion of your Creator display. Click the tab labeled Servers to see this window.

## Servers

Figure 2-24 shows the Servers window after you've deployed project **Login1**. Various categories of servers are listed here, including Data Sources, Enterprise Java Beans, Web Services, Deployment Server, Remote Deployment Servers, and Bundled Database Server.

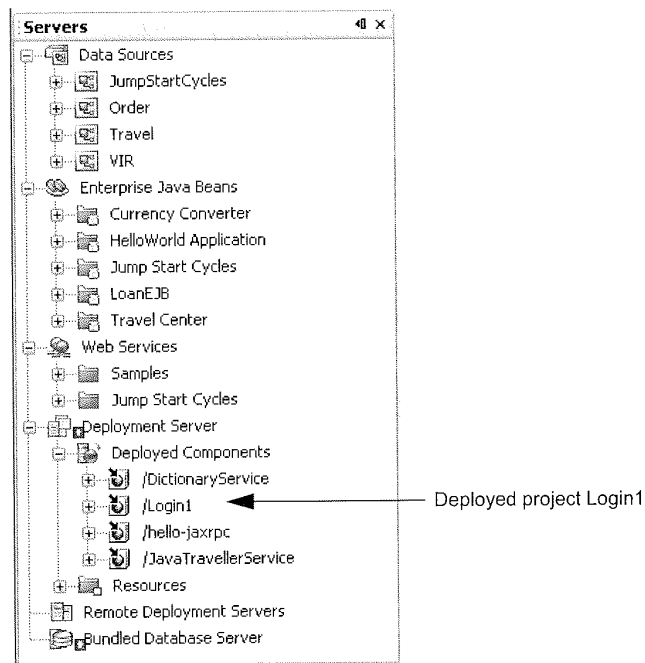


Figure 2-24 Servers window

The Data Sources node is a JDBC database connection. Creator bundles a database server and the Data Sources node connects to the bundled database by default. You can configure a different database. Creator comes configured with several sample databases, which are visible if you expand the Data Sources node.

**Creator Tip**

*The Database Server must be running to inspect the sample database tables. If the Bundled Database Server is not running, right-click node Bundled Database Server and select Start Bundled Database.*



Let's expand the Travel > Tables node and view the database tables. As you select different tables, Creator displays their properties in the Properties window. Expand a table further to examine its database table field names, as shown in Figure 2-25. Here, we expand table PERSON, displaying field names PERSONID, NAME, JOBTITLE, and FREQUENTFLYER.

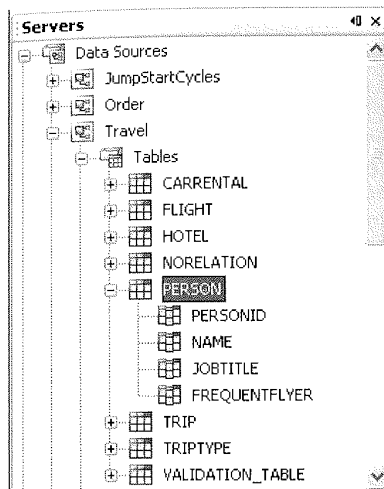


Figure 2-25 Inspecting the Travel Database tables (Person)

When you double-click a table name, Creator displays the data in the editor pane with a default query, as shown in Figure 2-26. You can close the table view by clicking the small x on the Query 1 tab. We discuss creating web applications that access databases in Chapter 9 (see "Accessing Databases" on page 372).

The second resource in the Servers window is the Enterprise JavaBeans node. Creator has a few sample EJBs deployed on the bundled Application Server, which you can access within your projects. Expand node Enterprise JavaBeans > Travel Center > TravelEJB, as shown in Figure 2-27. The TravelEJB provides some of the same data as the Travel database. With Creator, you can bind data to components exactly the same with EJBs as you can with data source tables. We show you how to use EJBs in Chapter 11.

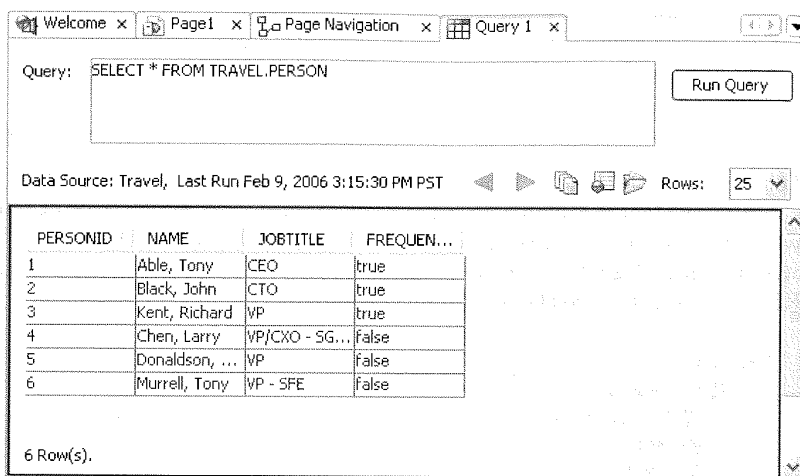


Figure 2-26 Display data from the Person table

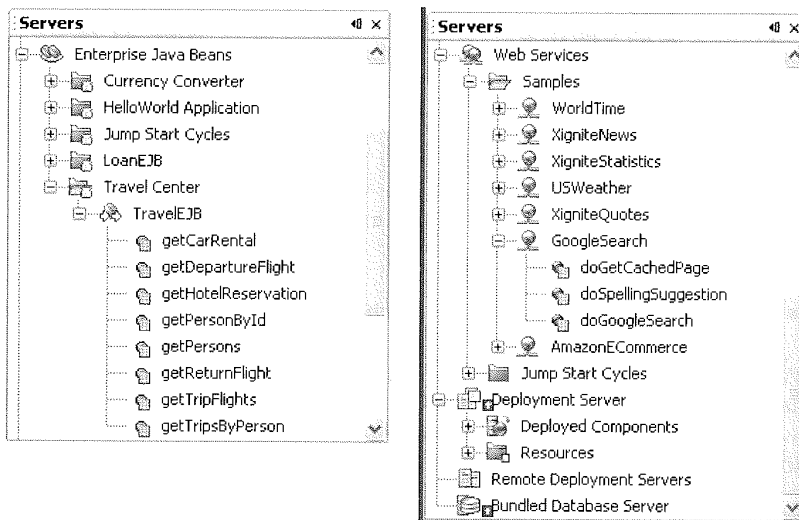


Figure 2-27 EJB and Web Services resources shown in the Servers window

Another server resource is Web Services, which provides access to remote APIs from Creator applications. This requires the cooperation of several Java technologies, which we discussed in Chapter 1. The Creator installation



includes a client to access the Google Web Services. In Chapter 10 we show you how to create an application with the Google web service API. The Google Search web service methods are shown in Figure 2-27.

The bundled Deployment Server allows you to deploy and run Creator applications on your machine. The Deployed Components node shows you the currently deployed components (including the Login1 application you just deployed). From the Deployment Server node, you can start and stop the server, access the Administrative Console, or view the server's log (right-click Deployment Server to view the context menu with these options).

**Creator Tip**

*The application server must be running for access to the administration console. Use user name **admin** and password **adminadmin**.*



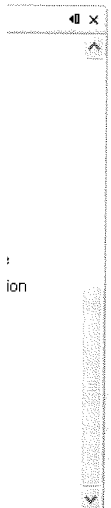
**Debugging Windows**

Creator has a debugger that lets you perform typical debugging tasks, such as setting breakpoints, tracing the call stack, tracking local variables, and setting watches. Use the View > Debugging menu to choose which debugging windows to enable, as shown in Figure 2-28.

- Local Variables Alt+Shift+1
- Watches Alt+Shift+2
- Call Stack Alt+Shift+3
- Classes Alt+Shift+4
- Breakpoints Alt+Shift+5
- Sessions Alt+Shift+6
- Threads Alt+Shift+7
- Sources Alt+Shift+8
- HTTP Monitor Ctrl+Shift+5

Figure 2-28 View > Debugging Menu Choices

To run your application in "debug mode," click on Run > Debug Main Project from the menu bar. The application server has to stop and restart if it's not already in debug mode. In Chapter 14 we walk you through the debugger options, setting breakpoints, stepping through code, and other debugging activities.



s to remote  
everal Java  
installation

## Creator Help System

The Creator Help System is probably the most useful window for readers new to Creator. This help system includes a Dynamic Help display, search capability, contents, and an index. The easiest way to access the help system is to select Help > Dynamic Help from the main menu. The selections displayed are context sensitive.

As an example, in the Page1 design view, select one of the Message components and choose Help > Dynamic Help. Creator displays a help window with topics relating to the message component as shown in Figure 2–29.

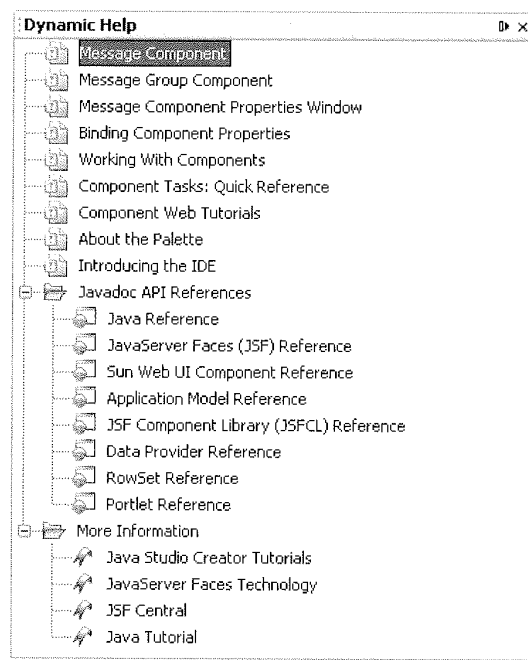


Figure 2–29 Dynamic Help window

When you double-click a selection, Creator displays the help information (see Figure 2–30).

r readers new  
earch capabil-  
em is to select  
ayed are con-

essage compo-  
window with  
9.

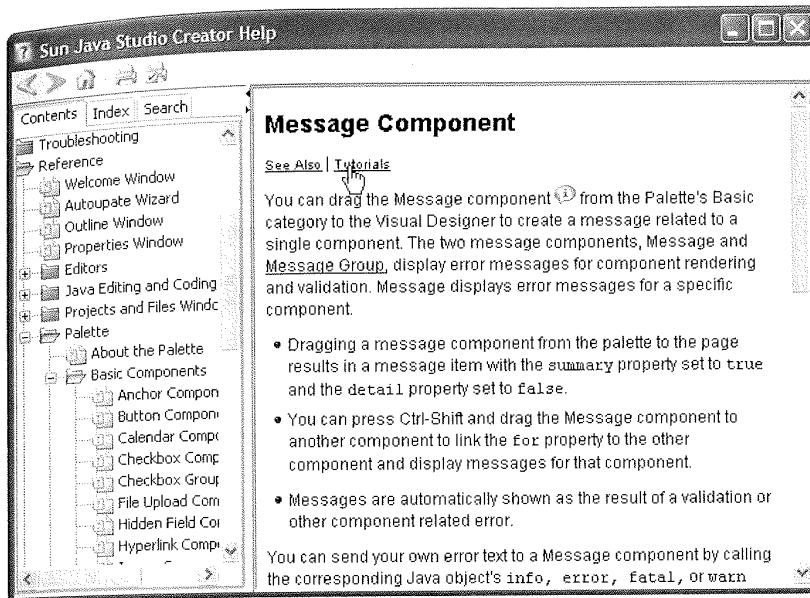


Figure 2-30 Creator Help system

## 2.3 Sample Application

Now that you're comfortable with Creator, let's create a simple web application. Even though this application is simplistic, it shows some of the power in Creator. Figure 2-31 provides a preview of this web application.

### Create a Project

Close project **Login1** if it's open.

1. From the Projects window, right-click the project node **Login1** and select **Close Project** from the context menu.
2. From Creator's Welcome Page, select **Create New Project**. From the New Project dialog, under **Categories** select **Web** and under **Projects** select **JSF Web Application**. Click **Next**.

ormation (see

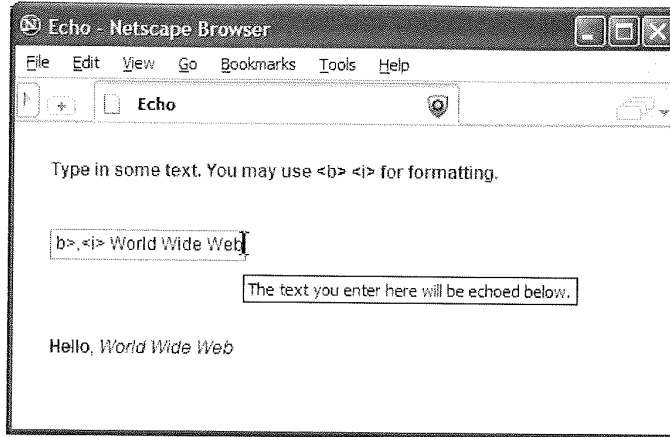


Figure 2-31 Web application Echo running in a browser

3. In the New Web Application dialog, specify **Echo** for Project Name and click Finish.

After creating the project, Creator comes up in the design view of the editor pane. You can now set the title.

4. Select **Title** in the Properties window and type in the text **Echo**. Finish by pressing **<Enter>**.

### **Add Components to the Page**

Creator makes the Components palette visible after you create a project. Using the design editor, you'll add three components to the page: a label, a text field, and a static text component. Figure 2-32 shows the design view with all the components added to the page.

1. From the Basic Components palette, select Label and drag it over to the design canvas. Drop it on the page, near the top on the left side.
2. The label remains selected after you drop it on the page. Supply the text **Type in some text. You may use <b> <i> for formatting.** Finish by pressing **<Enter>**. Creator sets the `text` property to this text (verify this in the Properties view) and displays the text on the page. The default font setting (property `labelLevel`) for a label's text is Medium(2), which can be changed in the Properties window.

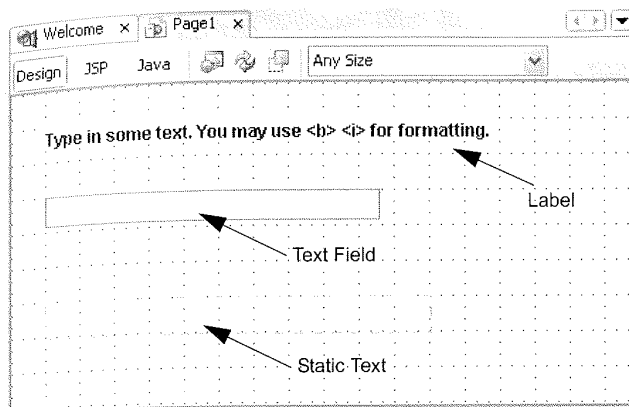


Figure 2-32 Project Echo in the design view

3. From the Basic Components palette, select component Text Field and place it on the design canvas underneath the label you just added.
4. Make sure the text field is selected. In the Properties window under Behavior, change its `toolTip` property to **The text you enter here will be echoed below**. Finish with `<Enter>`. This will appear as a tooltip when the user hovers the mouse over the text field in the browser.
5. From the Basic Components palette, select Static Text component and place it under the text field. Resize it so that it is approximately 11 grids wide, as shown in Figure 2-32.
6. Select the static text component. In the Properties window under Data, *uncheck* property `escape`. This allows HTML formatting tags to pass through unaltered to the browser.

You've finished adding the components. Now you will use property binding to bind the text field component `text` property to the static text component `text` property. Here's how.

1. Select the static text component (`staticText1`), right-click, and choose Property Bindings from the context menu. Creator brings up the Property Bindings dialog as shown in Figure 2-33.
2. Under Select bindable property, choose *text Object*.
3. Under Select binding target, expand **Page1 > page1 > html1 > body1 > form1** by clicking the '+' at each level.
4. Select component `textField1` (the text field component you added). Expand the component by clicking '+' on `textField1` and select *text Object*. (The properties are listed in alphabetical order, so `text` is near the end.)

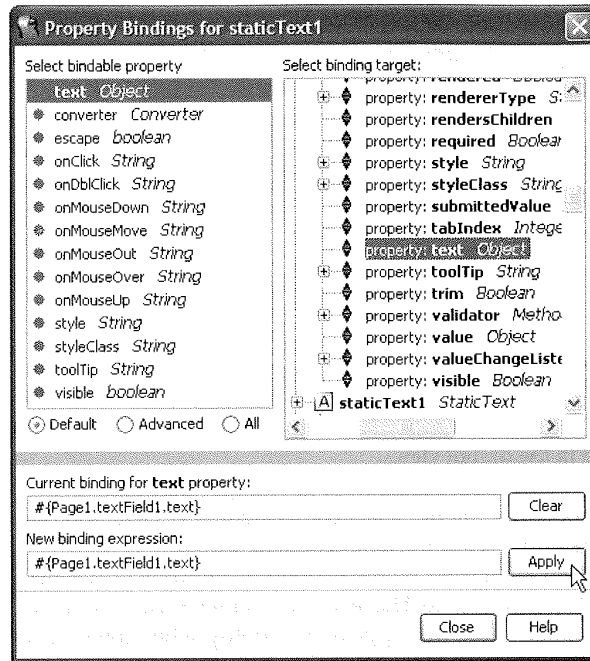


Figure 2–33 Property Bindings dialog

5. Click the Apply button. (If you don't click the Apply button, Creator doesn't set the property binding.) Under Current binding for **text** property, you should see the following JSF EL expression

```
#{Page1.textField1.text}
```

6. Click Close to finish

So, what did all this accomplish? You've just configured *property binding* on the static text component (id `staticText1`). This means JSF gets the `text` property (the text that is displayed on the page) for `staticText1` from the text field's `text` property (component `textField1`). This, in turn, means that whatever you type in for input will be echoed in the static text's display when you press **<Enter>**. Note that Creator and JSF made all this possible without you writing any Java code!

Is a button necessary to submit the page? As it turns out, when you hit **<Enter>** after entering text in the text field, the default action is to submit the

page. This puts the JSF life cycle events in motion and the page is rendered with the new text displayed in the static text component. We discuss the JSF life cycle events in detail in Chapter 6 (see "The Creator-JSF Life Cycle" on page 260). Because you unchecked the `escape` property, any formatting tags are unaltered by the component and passed directly through to the browser.

## Deploy and Run

You've finished creating the application. Now it's time to build, deploy, and run it. From the menu bar, select Run > Run Main Project (or select the Run Main Project green arrow icon on the toolbar). Creator builds the application, deploys it, and brings up a browser with the **Echo** web application running.

Figure 2-31 on page 52 shows what the browser window displays after you type `<b>Hello</b>`, `<i>World Wide Web</i>` inside the text field followed by `<Enter>`. Note that bold tags mark the word Hello and italic marks the phrase World Wide Web. The text field tooltip appears as the user hovers the mouse over the text field component.

This completes our tour of Creator. The next chapter provides a detailed description of the Creator UI components, validators, and converters.

## 2.4 Key Point Summary

- Creator has multiple windows to give you different views of the project that you're working on. The windows can be sized, docked and undocked, or hidden.
- From the main menu, select View and the desired window name to enable viewing.
- Use the Welcome Window to select a Project to open or to create a new project.
- The design canvas allows you to manipulate components on a page and control their size and placement. Grid lines provide an easy way to align components.
- Use the Components palette to drag and drop a component on the design canvas.
- Use the Converters and Validators sections in the palette to select data converters and input validators for your project.
- The Properties window allows you to inspect and edit a component's properties. Each component type displays a different list of properties.
- A component's `text` attribute typically contains text that is rendered on the page (such as labels on buttons, input text fields, and static text fields). Use the `toolTip` property to create a tooltip for the component and the `style`

ator doesn't  
rty, you

y binding on  
e text prop-  
ie text field's  
at whatever  
n you press  
you writing

hen you hit  
) submit the

property to change font characteristics. Property `styleClass` lets you apply previously defined style definitions.

- You can apply Property Binding and “connect” the value of one component to another or to a data object in request, session, or application scope.
- The Outline window shows all of the elements on a page, including nonvisual components such as converters, validators, or EJB or Web Services clients.
- The Projects view gives you a logical view of your project, including Web Pages, Source Packages, Libraries, the pre-configured beans, and Page Navigation.
- The Files view lets you see all the files in your project.
- The JSP Source editor displays a page’s source. Most of the page includes JSP tags for components and their properties. As you make changes to your pages in the design canvas, Creator synchronizes the JSP and Java source.
- Creator’s editors are based on NetBeans and reflect a rich functionality for editing Java source, JSP source, and XML files.
- The Java Source editor displays the Java source for each “page bean,” a JavaBeans component that manipulates each page’s elements. You typically place event handler code or custom initialization code in the Java page bean.
- The Java Source editor includes a code completion mechanism that provides pop-up windows with possible method names (use **<Ctrl+Space>** to invoke) and Javadoc documentation for classes and objects in your program (use **<Ctrl+Shift+Space>** to invoke). The Java editor also includes a dynamic syntax analyzer to warn you about compilation errors before you compile.
- The Code Clips palette provides sample Java code to accomplish common programming tasks. The Clips are organized into categories based on function. You can select a clip and drop the code into your Java source.
- The Page Navigation editor lets you specify page flow. This editor generates a navigation configuration file, **navigation.xml**.
- When you build your project, the Output window provides diagnostic feedback and completion status.
- The Servers window displays Data Sources, Enterprise JavaBeans, Web Services, Deployment Server, and Database Server nodes.
- The Deployment Server node lets you start and stop the application server and undeploy running web applications.
- You can view database table data by expanding the Data Sources node and selecting individual tables. Creator displays the data in the editor pane when you right-click a table name and select View Data.
- The Debugger Window displays several views that are helpful when you are debugging your project. You can set breakpoints and monitor the call stack, local variables, and watches with the debugger.
- The Creator Help system provides a table of contents, index, and search mechanism to help you use Creator effectively. The help system is dynamic



and displays help information based on how you're currently interacting with Creator.

When you apply

component  
scope.  
using  
Web Services

Using Web  
Page

includes JSF  
to your  
a source.  
quality for

bean," a  
usually  
page bean.  
that provides  
to invoke)  
bean (use  
annotation  
compile.  
common  
defined on  
source.  
The generator

nostic

is, Web

on server

node and  
pane

When you are  
call stack,

search  
dynamic

# CREATOR COMPONENTS

## Topics in This Chapter

- JSF Overview
- Component Categories
- Basic Components
- Layout Components
- Composite Components
- Converters and Validators
- Component Library Manager
- Importing a Component Library

# Chapter

# 3

**S**un Java Studio Creator's design palette presents a wide variety of components to choose from. These components include buttons, text fields, checkboxes, listboxes, radio buttons, hyperlinks, images, tables, tree nodes, grid panels, and so on—in short, anything you need to design a web page. You can select a component, drag it to the design canvas, and drop it at the location of your choice. In addition, you can choose validator components to verify user input and converter components for data conversions. Creator maintains a design canvas with your web page layout and generates Java code for you, along with JSP and XML statements to configure and deploy your application.

In this chapter we present a catalog of Creator User Interface (UI) components, validators, and converters. We also provide references to examples in this book where they are used. The examples will help you understand how to use the Creator UI components in your projects.

## 3.1 JSF Overview

The Creator UI components work within a JSF web application environment. With the JSF framework, these components let you handle events, validate and convert user input, define page navigation, and support internationalization. JSF also connects components to server side objects. Let's start with the architecture of JSF to give you the "big picture" of what's going on.

## JSF Architecture

Figure 3-1 shows the architecture used with JSF.

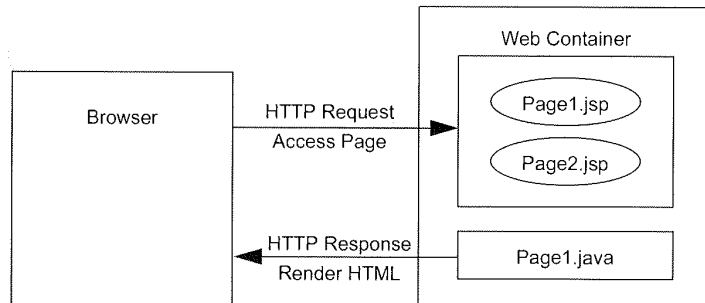


Figure 3-1 JSF architecture

Your browser interacts with the web container through one or more JSP pages (**Page1.jsp** and **Page2.jsp**). These are JSP pages containing JSF tags. The supporting page bean (**Page1.java**) manages the objects referenced by the JSP pages. Note that the JSP pages handle HTTP requests when a page is accessed, whereas the Java files render HTML for the HTTP response.

### The JSP Page

Suppose a web page has a static text component (`staticText1`) that displays "In what year were you born?", a button to click (`button1`), and a text field (`textField1`) for the year (restricted to the range 1900 to 1999). When these components are moved from the palette to the design canvas, Creator generates the component tags in the JSP file. Each Creator UI component also becomes a property in the generated Java page bean. To understand how this all works, let's start with how the static text component is defined in **Page1.jsp**.

```

<ui:staticText id="staticText1"
  binding="#{Page1.staticText1}"
  style="font-size: 18pt; left: 96px; top: 96px;
  position: absolute"
  text="In what year were you born?"/>
  
```

The JSP file is expressed in XML. Creator generates this file for you and keeps it synchronized with your page design. As you modify components with the Properties window, Creator updates the JSP code as well as the Java code as necessary. Creator generates the required tags for your components in the JSP page. You can always access the JSP page by selecting the JSP label in the editing toolbar above the design canvas.

In this example, the static text component displays "In what year were you born" on the page. Its `id` property (unique page identification) is `staticText1` and its `binding` property (the corresponding property in the `Page1` page bean) is also `staticText1`. The `style` property specifies its location on the page with the left and top settings in pixels. This property also makes the text appear in 18-point font size.

### JSF Expression Language (EL)

JSF uses a specialized syntax to access JavaBeans components with its tags. For example, the notation

```
#{Page1.staticText1}
```

references the `staticText1` property in JavaBeans component `Page1`. In the JSP file (**Page1.jsp**), the generated component tags reference properties in the supporting page bean, as follows.

```
binding="#{Page1.staticText1}"
```

Now let's look at the generated tags for a button component in **Page1.jsp**.

```
<ui:button id="button1"
  binding="#{Page1.button1}"
  action="#{Page1.button1_action}"
  style="left: 72px; top: 168px; position: absolute;"
  text="Click for your age"/>
```

Elements `binding` and `action` are UI component tag library properties whose values are set with JSF EL. Element `binding` is the button component's page bean reference, and `action` references a special action event method `button1_action()`, also in `Page1`. Here, method `button1_action()` is called when the users clicks the button controlled by component `button1`.

or more JSP  
JSF tags. The  
d by the JSP  
is accessed,

hat displays  
l a text field  
When these  
erator gener-  
ponent also  
nd how this  
n **Page1.jsp**.

## Converters and Validators

What about the text field component? Recall that this component must read a year (in the range 1900 to 1999) from the user. Here's how the input text field component is configured in `Page1.jsp`.

```
<ui:textfield id="textField1"
  binding="#{Page1.textField1}"
  converter="#{Page1.integerConverter1}"
  style="left: 192px; top: 168px; position: absolute"
  validator="#{Page1.longRangeValidator1.validate}"/>
```

Text field components display and accept text, but `textField1` must work with integer numbers in this example. Consequently, a JSF conversion component (`integerConverter1`) is necessary to convert String input to integer values. Input is restricted to a specific range of numbers (1900 to 1999), so we'll need a JSF validator (`longRangeValidator1`) for the input, too.

```
converter="#{Page1.integerConverter1}"
validator="#{Page1.longRangeValidator1.validate}"
```

In both cases, JSF EL references the components that perform the conversion and validation.

## Event Handling

JSF uses a delegation event model to handle events generated by user actions (clicking a button, changing a selection in a drop down list, pressing `<Enter>` after editing a text field, for example). It's helpful to have an understanding of the pieces that work together to make responding to events a well-behaved system.

The Event Source is a component that is capable of generating an event. Different components generate different event types. Button components and hyperlink components (for example) generate action events. Drop down list components generate value change events.

Event Objects are generated by components (the Event Source). An Event Object is basically a message that is passed from the Event Source to an Event Listener. The Event Object contains information about the Event.

Event Listeners are specialized objects created by JSF that know what to do when an event is generated. Different types of listeners can respond to different types of events. For example, `ActionEventListeners` respond to action events and `ValueChangeListener`s respond to value change events.

Using the "Publish-Subscribe" design pattern, Event Listener Registration keeps track of which objects "care about" an event occurring. Objects that

"care" are those that register themselves through the Event Listener Registration. After registering with the Event Source, Event Listeners are notified when an action occurs. Notification means their special event method is called with the event object as a parameter. Fortunately, Creator generates all the method stubs, event listeners, and event registration for you. Here is an example of the default value change method that JSF calls when a value change event is generated from a drop down list component.

```
public void dropdown1_processValueChange(
    ValueChangeEvent event) {
    // TODO: Replace with your code
    . . .
}
```

Web application developers provide the specialized event-processing code (whatever actions your web application must perform in responding to the value change event).

Action events are common with most applications and Creator generates the action event handlers for you. Action events can be used to write processing code in response to a button click. In addition, action events return String values to a navigation handler, which allow you to invoke a different web page. Here is the default event handler Creator generates for a button (with property id set to button1).

```
public String button1_action() {
    // TODO: Process the button click action.
    // Return value is a navigation
    // case name where null will return to the same page.
    . . .
    return null;
}
```

Note that action events implement navigation by returning String values. A null string means you stay on the same page. A different String ("Button-Click", for example) instructs the navigation handler to go to a different page.

## Java Page Bean

Now let's show you the Java page bean file, **Page1.java**. Creator generates Java code in the Java page bean for the components you select from the design palette. Each component becomes a property of the supporting page bean, and the component instance is bound to that property.

Here's the `Page1.java` file for our simple web application with two text fields and a button. Again, Creator generates this file for you.

```
public class Page1 extends AbstractPageBean {

    private Button button1 = new Button();
    private TextField textField1 = new TextField();
    private TextField textField2 = new TextField();

    private LongRangeValidator longRangeValidator1 =
        new LongRangeValidator();
    private IntegerConverter integerConverter1 =
        new IntegerConverter();

    // getters and setters for components
    . . .

    public Page1() {
    }

    // Creator-generated life cycle code omitted . . .

    public String button1_action() {
        // TODO: Process the button click action.
        // Return value is a navigation
        // case name where null will return to the same page.
        return null;
    }
}
```

Note that `Page1` extends `AbstractPageBean`. The private fields are generated for each UI component you place on the page and the getter and setter methods make them accessible as properties. Here are the getters and setters for the static text component.

```
public TextField getTextField1() {
    return textField1;
}

public void setTextField1(TextField tf) {
    this.textField1 = tf;
}
```

In our example a public `init()` method calls a private `_init()` method in the Java page bean to set the minimum and maximum ranges for the JSF validator component (`longRangeValidator1`). When you set these values for the validator by using Creator's Properties window, Creator generates the state-



ments in the private `_init()` method to configure the validator for you. This is done before the managed components are initialized.

```
private void _init() throws Exception {
    longRangeValidator1.setMaximum(1999L);
    longRangeValidator1.setMinimum(1900L);
}

public void init() {
    super.init();
    try {
        _init();
    }
    catch (Exception e) {
        log("Page1 Initialization Failure", e);
        throw e instanceof FacesException ?
            (FacesException)e : new FacesException(e);
    }
    // Perform application initialization that must complete
    // *after* managed components are initialized
    // TODO - add your own initialization code here
}
```

## 3.2 Components

Creator allows you to select UI components from a design palette for your application. These components are implemented with a JSP custom tag library for rendering components in HTML.

When you select a component and drag it to the design canvas, Creator generates code in the page's JSP source as well as support code in the associated Java page bean. Furthermore, Creator displays each component in the Outline view, including any support components that may not be visible. Once you place a component on the design canvas, you can modify its properties and behavior through the Properties window, through the JSP code, or through modifications to the Java page bean. In general, it's preferable to edit properties of a component with Creator's Properties window. However, writing code to handle action and value change events must be done in the Java page bean file.

### Components Palette

The Components Palette is divided into three groups: Basic, Layout, and Composite. Each component includes the component name and an icon that you

can drag and drop on the design canvas. Figure 3-2 shows you the component groups and all of their components.

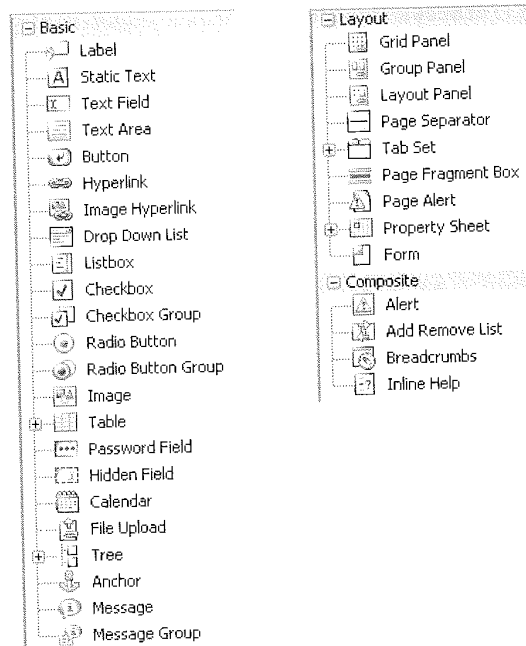


Figure 3-2 Component palette



#### Creator Tip

The Components palette also includes the Standard Components, the JSF Reference Implementation components bundled with the first version of Creator. These are included for backward compatibility with imported Creator1 projects only. For newly created projects, use the components in the Basic, Layout, and Composite sections.

## Component Properties

This chapter presents a catalog of Creator components so that you can easily look up their behavior and use them in your applications. Many components share common properties and code generation features, however. Let's start with the definitions of these properties so that you can see how they're used.

## text

The `text` property stores a component's main textual characteristics. Its meaning depends on the component. For example, `text` stores the text of a button label, the display text of a static text component, or the input for a text field. The `text` property can also store the text for a password field and hidden field components.

The `text` property is a Java Object type. Creator allows you to bind a component's `text` property to a JavaBeans property, a data source, or even a localized message in a properties file.

## label

The `label` property is a text string that provides text labeling for a component on a web page. Examples of components that have label properties are text fields, checkboxes, radio buttons, and drop down lists.

The `label` property is a Java Object type. Creator allows you to bind a component's `label` property to a JavaBeans property, a data source, or a localized message in a properties file.

## toolTip

The `toolTip` property is a text string for a component's tooltip.

## style

The `style` property holds Cascading Style Sheet (CSS) strings for properties such as font family, font size, and position parameters. These determine the type of font used, its point size, and placement on the design canvas. Creator provides a sophisticated CSS style editor that helps you configure a component's `style` property. (For a detailed discussion of the style editor, see "Using the Style Editor" on page 282.)

## styleClass

The `styleClass` property allows you to specify predefined CSS style classes. You can place CSS style class definitions in the default style sheet, `stylesheet.css` (found in the Projects window under Web Pages > resources). We show you examples of property `styleClass` in Chapter 7 (see "Using Property `styleClass`" on page 284).

## id

The `id` property is a page-unique string that identifies a component on the web page. Creator generates the component's `id` for you, but you can use the Properties window to change it.

**Creator Tip**

*We recommend renaming the default id when you have components on the page with event handling methods (action or value change methods). Providing meaningful names for the id property makes Creator generate methods with meaningful names. This makes your Java code easier to read.*

## rendered

The `rendered` boolean property controls whether a component will be rendered during the Render Response Phase of the JSF life cycle.

## visible

The `visible` boolean property controls whether a rendered component will be visible on the page.

## action

The `action` property is important for Action and Link components, such as buttons and hyperlinks. This property references a method in the page bean that returns a String for JSF's navigation handler. Chapter 5 discusses page navigation in detail. The application writer may provide application-specific statements in the action method, process information to determine page flow, or both. To generate an action event handler, double-click the component in the design canvas. Creator brings up the Java source editor and puts the cursor at the first line of the action event handler.

## binding

Creator sets the `binding` property for all components you place on the page. A `binding` property binds the component instance to a property in the page bean. Since Creator maintains this property for you, there is no reason for you to change it. For example, if you add a button component to a web application's initial page (`Page1.jsp`), the default `binding` property for the button component is

```
binding="#{Page1.button1}"
```

This JSF EL expression references the `button1` property of managed bean `Page1` and binds the component instance to the bean property. Now you can write code in the `Page1.java` page bean to access the button component and dynamically control its properties.

## JavaScript

JavaScript allows client side processing activated with mouse events (for example, clicking a component, giving focus to a component, or moving the mouse over a component). The browser executes the JavaScript on the client machine without any server involvement. You can attach a mouse event to a component by specifying a JavaScript element in the component's Properties window for that event. Not all Creator UI components detect the same mouse events.

For example, suppose you want to obtain a confirmation from the user before activating a button's Delete operation. In the design view, select the button. In the Properties view under JavaScript, specify the following JavaScript for property `onClick`.

```
return confirm('Are You Sure You Want To Delete?');
```

When the user clicks the button, a confirmation window appears, as shown in Figure 3-3. If the user selects OK, the button's action event handler method is invoked. Otherwise, the button click is ignored.

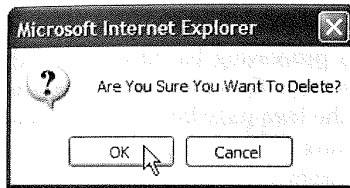


Figure 3-3 JavaScript confirmation dialog defined for property `onClick`

## Input Components

Components that collect input (text field, password field, text area, drop down list, listbox, for example) share common properties to control and validate input. Let's look at some of these properties now.

### validator

The `validator` property references a method that performs validation on its value. JSF provides three standard validators: a length validator for strings, a long range validator for integral types, and a double range validator for floating types. You can also write your own custom validation method. See "Add a Validation Method" on page 617 (Chapter 13).

### converter

The `converter` property references a converter component that builds the correct type of object. Once the conversion has taken place, you can retrieve the object by casting it to the desired type.

### maxlength

The `maxlength` property limits input to a specified number of characters. (This is *not* the same as length validation.) Setting `maxlength` causes a component to stop accepting input after the user has typed in the maximum characters allowed. No error messages are produced.

### required

The boolean `required` property specifies whether or not input is necessary for the component. If the user leaves an input component's field empty and `required` is set, an error message is produced during the validation phase.

### valueChange

A value change event occurs when an input component's selection changes or its text changes. If you want to perform processing based on input change, double-click the component in the design view. Creator generates a `processValueChange()` event method for you in the Java page bean. You can add your own processing code to this method.

### Auto-Submit on Change

The Auto-Submit on Change feature submits the page for processing when an input component generates a value change event. To enable Auto-Submit on Change, select the component in the design canvas, right-click, and choose Auto-Submit on Change. This sets the `onChange` property to the following JavaScript element, shown here for a Text Area component.

```
common_timeoutSubmitForm(this.form, 'textArea1');
```

The input component has `id` property `textArea1`. When the input value of the text area changes, the page is submitted, allowing immediate processing (instead of waiting for a button click or hyperlink selection).

### Virtual Forms

Virtual Forms allow the application developer to build web pages that provide more than one function (or use case). For example, a single web page might

allow a user to either login or create a new username. The login use case requires a username and password before clicking the "Login" button. The create new username use case requires additional fields (perhaps a new password that must be entered twice, as well as a username). By grouping input components into separate virtual forms, you avoid interference when a validator requires input for a component that is not needed to fulfill another use case. Here are several examples of virtual forms use shown in the text.

### *Book Examples*

- "Configure Virtual Forms" on page 216 (Chapter 5). Uses virtual forms to allow a Reset button to clear input fields.
- "Virtual Forms" on page 417 and "Configure Virtual Forms" on page 420 (Chapter 9). Uses virtual forms to provide add, update, and cancel use cases on the same page.

## **Data-Aware Components**

Creator offers a selection of data-aware components that can bind a data provider to a database table, a web services method, an EJB method, or a JavaBeans component. The table component is particularly suited for displaying data, but you can also choose from among the drop down list, checkbox, listbox, or radio button components.

Creator automatically supplies a converter for non-String data fields when you bind to a data provider with known data types. If there are any conversion errors, you will only see error messages if you have placed message components on the page.

### **Creator Tip**

---

*We recommend placing a message group component on the page when you're using the data-aware components (see "Message Group" on page 90).*

---



## **Data Providers**

A data provider is an abstraction for a data source. Creator has data providers for database tables, web services, EJBs, maps, arrays, and lists. Data providers are useful because they offer a common interface for accessing different sources of data.

When you drop a database table on a data-aware component, Creator configures the appropriate data provider for you. Similarly, if you drop a web services method or EJB method on a component, Creator configures a data provider. You can also explicitly select data providers for arbitrary objects, such as arrays or lists. Chapter 8 introduces data providers and Chapter 9 uses

data providers with database accesses. Chapter 10 shows you data providers with web services and Chapter 11 shows you data providers with EJB methods.

### 3.3 Basic Components

The following catalog of basic components describes each component and gives you common usage scenarios. To show you how a basic component can be useful in a Creator project, we also point you to relevant examples in other chapters of this book. The basic components are listed alphabetically for easy lookup.

#### Anchor

The anchor component helps position link targets within a page. Anchor components are non-visual and often used with hyperlinks to scroll pages. By default, an anchor is rendered in HTML as `<a name=targetname></a>`.

Figure 3-4 shows an anchor component dropped on the design canvas.



Figure 3-4 Anchor component

Suppose, for example, you place a hyperlink at the bottom of a page and drop an anchor component called `anchorTop` at the top of the page. To jump to the top of the page, set the `url` property of the hyperlink to the following.

```
/faces/Page1.jsp#anchorTop
```

It's also possible to link to anchor components in other pages.

#### *Book Examples*

- “Add Components to the Page” on page 298 (Chapter 7). Uses anchor components with hyperlinks to control page scrolling.

#### Button

The button component is an example of a “command component.” Buttons perform an action when they are activated (clicked). This can happen during server-side processing (a method that processes an action event) or with a navigational action that determines page flow. The button component is one of the



ata providers  
EJB methods.

most-often-used components in web design. By default, buttons are rendered as HTML `<input type=button>` tags.

Figure 3-5 shows a button component and tooltip in a web page with a browser. The message shown was set in the button's `toolTip` property.

component and  
component can  
aples in other  
cally for easy

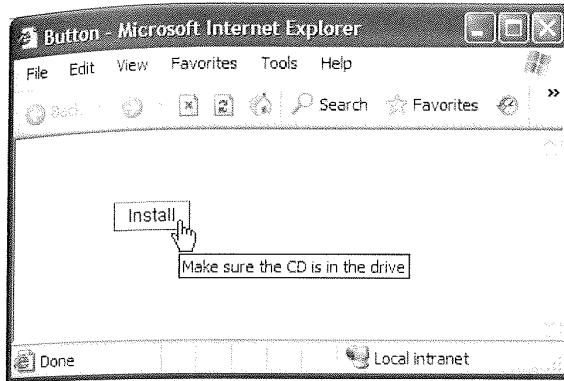


Figure 3-5 Button component

Anchor com-  
oll pages. By  
<</a>.  
i canvas.

Buttons can be used for “simple” or dynamic navigation between web pages. With simple navigation, a button's `action` method returns a `String` that matches a case label in the navigation rules generated for the application. We show you how to create this type of navigation in “Add Page Navigation” on page 195. Dynamic navigation is useful when you need to figure out the next page based on some sort of processing. In this case, the `action` method returns a `String` based on the processing. See “Create New Web Pages” on page 212 for an example of dynamic navigation.

of a page and  
ge. To jump to  
llowing.

In Creator, you connect the a button click (an action event) to an event processing method by double-clicking the button component in the Creator design canvas. This brings up the matching `button_action()` method (where `button` is the component's `id` property) in the Java page bean so that you can add your processing code.

The button's `text` property is its label. You can bind this value to a property or to a value in a properties file.

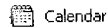
anchor

#### Book Examples

- “Add Button Components” on page 193 (Chapter 5). Uses a button to initiate navigation.
- “Place Button, Label and Static Text Components” on page 257 (Chapter 6). Uses a button to submit a page for processing.

ent.” Buttons  
appen during  
or with a nav-  
t is one of the

- “Add a Button Component” on page 452 (Chapter 10). Uses a button to invoke an action event method.
- “Modify the Components for Localized Text” on page 599 (Chapter 13). Configures a button for internationalization.



## Calendar

The calendar component lets users enter dates on a page, either by typing in a specific date or by selecting a date from a pop-up calendar. Figure 3-6 is an example of a calendar component in the visual editor with a Enter Date label.

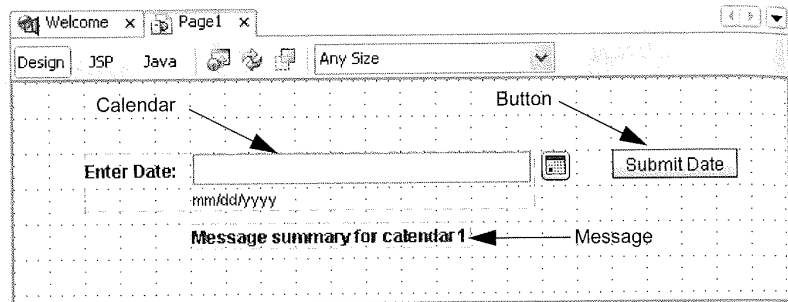


Figure 3-6 Calendar component

When you click the calendar icon, a pop-up window lets you select a date from a drop down list of months and years. Otherwise, just type a specific date into the component’s text field.

The calendar component automatically validates the input date. You control the range by setting properties `minDate` and `maxDate`. The minimum date defaults to the current date and the maximum defaults to today’s date four years from now. Most applications will need to customize these values. Here is an example that sets the minimum date to January 1, 1975 and the maximum to December 31, 2020. You typically add this initialization code to the page bean’s `init()` method.

```
// Set minimum date to January 1, 1975
// Method getTime() returns java.util.Date
calendar1.setMinDate(
    new GregorianCalendar(1975, 0, 1).getTime());
// Set maximum date to December 31, 2020
calendar1.setMaxDate(
    new GregorianCalendar(2020, 11, 31).getTime());
```

**Creator Tip**

*You cannot supply String values for these properties in the Properties window, but you can provide property binding expressions (to objects of type Date) or you can set the minimum and maximum values as shown in the page bean. Use a message group or message component to display validation error messages on the page.*



The date format defaults to the default format for the locale, but you can use the `dateFormatPattern` property to select different date format patterns.

Property `selectDate` holds the user-supplied date, which you can bind to an object or a data provider. You can also right-click the calendar component and select **Edit Event Handler** from the pop-up menu. The `validate` option lets you insert Java code that validates user input, and the `processValueChange` option lets you insert Java code that executes when a component's value has changed.

**Book Examples**

- “Configure the Calendar Component” on page 355 (Chapter 8). Uses a calendar component and shows you how to configure its settings.

**Checkbox**
 Checkbox

The checkbox component uses a boolean on/off setting as a choice for a user. Checkboxes are often used as standalone components (that is, not part of a checkbox group) on a page. Figure 3-7 shows part of a page with a checkbox component. Here we set the `label` property to the text string shown and the `selected` property to `true`, which displays a check mark.

 Check here for email confirmation

Figure 3-7 Checkbox component

There are two important properties with checkboxes. The `selected` property indicates whether or not a checkbox is selected and checked on the page. The `selectedValue` property allows you to store and retrieve an arbitrary data value associated with the checkbox. A check box is considered to be selected when the value of the `selected` property is equal to the value of the `selectedValue` property. You can bind the `selected` property of a checkbox to an object, such as a JavaBeans property or a data source object.

Use method `isChecked()` to determine if the component is selected.

*Book Examples*

- “Configure Checkbox Components” on page 430 (Chapter 9). Uses checkboxes in columns with table components.
- “Specify Property Bindings” on page 582 (Chapter 12). Uses standalone checkbox components and binds them to SessionBean properties.



Checkbox Group

**Checkbox Group**

The checkbox group component groups a set of checkboxes. You can specify their items with a dialog accessed from the Properties window or dynamically fill them from a database or JavaBeans component. When you use a checkbox group, users may select any number of checkbox options (including none unless the `required` property is checked). Checkboxes are rendered as an HTML `<table>` element with rows and columns. Figure 3–8 shows a checkbox group component when you drop it on the design canvas.

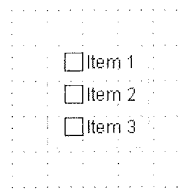


Figure 3–8 Checkbox group component

A checkbox group component is appropriate when you want to give the user a list of choices with the phrasing “please check all that apply.”<sup>1</sup> The selection items may be hardcoded with the Properties window or generated dynamically at run time. Creator automatically supplies a converter for non-String data fields when you fill the list from a database source. The checkbox group component also accepts data binding. The `selected` property of the checkbox group returns an array of `Objects` consisting of the checked selections.

Adding a checkbox group component to your web page creates three elements: the checkbox group component, an embedded selection list, and a “default items” list used for initializing the selection choices. To specify the choices, select `checkboxGroup1DefaultOptions` in the Outline view. In the Properties window, click the editing box opposite property `options`. Creator pops-up a dialog so that you can add, edit, or remove items. (This is the same

1. If you’d like to limit the choice to only one from a list, use the radio button group component (see “Radio Button Group” on page 93).

dialog used to specify Display/Value pairs for the Listbox component. See Figure 3-20 on page 87.)

### Example

Figure 3-9 shows a checkbox group component on a web page. The default layout for a checkbox group is a single vertical column, so we set the `columns` property to the number of checkboxes (4) to get a horizontal layout. Note that a user may select more than one choice (including none or all).

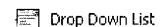


Figure 3-9 Checkbox group component

Here is the Java code to display choices selected from the checkbox group (whose `id` property is `checkboxGroup1`) in a static text component. Note that we assign the selected values to a `sides` `String` array. Casting is necessary since the `getSelected()` method returns an `Object` array. The `for` loop concatenates selected values with space delimiters. This code is placed in the button event handler method.

```
public String button1_action() {
    String choices = "";
    String[] sides = (String[])checkboxGroup1.getSelected();
    for (int i = 0; i < sides.length; i++) {
        choices = choices + " " + sides[i];
    }
    staticText1.setValue(choices);
    return null;
}
```

## Drop Down List



The drop down list component is an extremely versatile component, rendered as an HTML `<select>` element (a drop down list). A drop down list allows a user to select one item from a set of items, as shown in Figure 3-10.

The selection items may be hardcoded with a dialog accessed from the Properties window or generated dynamically at run time. Creator automatically supplies a converter for non-String data fields when you fill the list from a



Figure 3-10 Checkbox group component

database source. This component also accepts data binding. The selected property determines the value of the currently selected item.

When a drop down list component is used with a data provider that wraps a database data source, you typically bind the database table's primary key field with the Value field. You select an appropriate field from the data table for the dropdown component's Display field. Figure 3-11 shows the Bind to Data dialog that lets you configure the drop down list and the data provider. Here the Value field is the RECORDINGID field (the primary key) and the Display field is RECORDINGTITLE. Thus, the dropdown component's `getSelected()` method will return the primary key. This is useful for setting SQL query parameters from a drop down list component's selection value (see Chapter 9, "Connect Dropdown List to Query" on page 408).

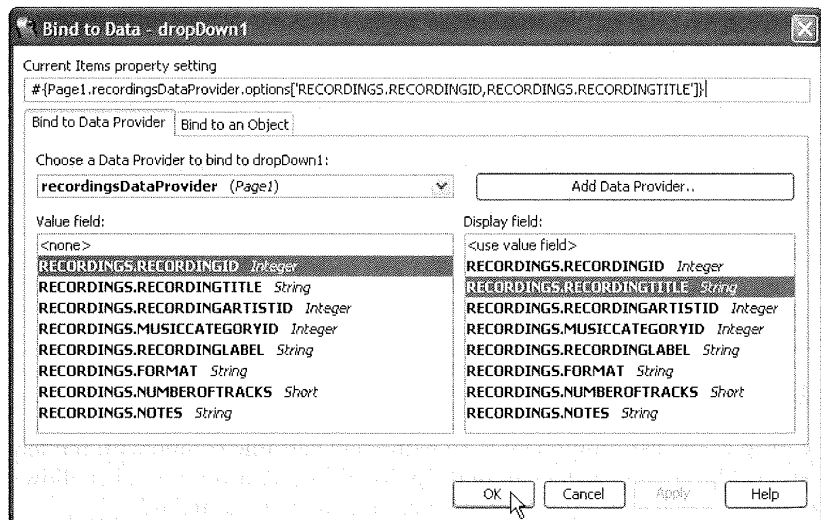


Figure 3-11 Bind to Data dialog with a drop down list

The nonvisual component `dropDown1DefaultOptions` supplies text for the selections. To input text items, select the `dropDown1DefaultOptions` element in

the Outline view and click the options property in the Properties window. A dialog pops up that lets you type in Display/Value pairs for each selection item (see Figure 3-20 on page 87). To supply options for a drop down list using data stored in application scope (for example, see "Add ApplicationBean1 Data" on page 577).

When the user selects a different item from a drop down list component, the system generates a value change event. To submit the page for immediate processing on a value change event, set the Auto-Submit on Change feature. To provide event handling code for a value change event, double-click the drop down list component on the design canvas. Creator generates a default `processValueChange()` method and brings up the Java source editor for you.

### Book Examples

- "Add a Drop Down List" on page 202 (Chapter 5). Uses a drop down list with navigation.
- "Add a Data Source" on page 401 (Chapter 9). Fills the selection list from a database data source.
- "Add ApplicationBean1 Data" on page 577 (Chapter 12). Shows how to instantiate data in an `Options[]` array appropriate for a selection component, such as drop down list or listbox.
- "Specify Property Bindings" on page 582 (Chapter 12). Uses a drop down list to bind a SessionBean property.
- "Add Components to Page1" on page 607 (Chapter 13). Uses a drop down list to specify locale.

### File Upload



The file upload component lets users locate a file on their system and upload it to a server. You can upload text files, images, and other types of data files (.zip or .jar files, even executables). This component is similar to an HTML `<input type="file">` element. Figure 3-12 shows a file upload component on the design canvas.

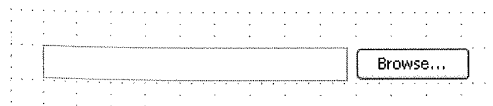
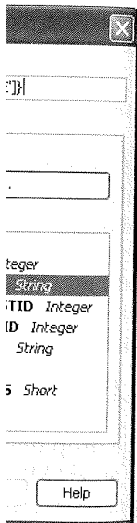


Figure 3-12 File upload component

For security reasons, file upload components are not supported in portlet projects. You can upload files up to one megabyte in size by default. To upload



larger files, modify the `maxsize` parameter for the `UploadFilter` entry in the web application's `web.xml` file.

The read-only `uploadedFile` property provides a `UploadedFile` interface with methods that let you read the file and write it to disk. There are also methods to access the file's name, size in bytes, and type (`text/plain` or `image/jpeg`).



#### Creator Tip

*Be careful with file names that have spaces, they are not supported. It is also not possible to nest a file upload component within a tab set component.*

#### Example

Figure 3-13 shows you a web page with a file upload component. Note that the file upload component has a built-in Browse button to let users locate a file on their system. When the "Get File Now" button is clicked, the file contents are written to the server and displayed in the scrollable text area on the page. A message group component displays status.

Here is the Java code for the button handler that uploads the file, displays the file in the text area, and writes the data to the server.

```
public String filer_action() {
    // read file from client
    UploadedFile uploadedFile =
        (UploadedFile)fileUpload1.getUploadedFile();
    String uploadedFileName = uploadedFile.getOriginalName();
    String FileName = uploadedFileName.substring
        (uploadedFileName.lastIndexOf(File.separatorChar) + 1);
    info("Uploaded file from " + uploadedFileName +
        ", size is " + uploadedFile.getSize() + " bytes");

    // write data to text area component
    textArea1.setText(uploadedFile.getAsString());

    // save file contents to server on C:
    try {
        File file = new File("C:" + File.separatorChar + "Saved" +
            File.separatorChar + FileName);
        info("Saved file to " + file);
        uploadedFile.write(file);
    } catch (Exception ex) {
        error("Cannot upload file: " + FileName);
    }
    return null;
}
```



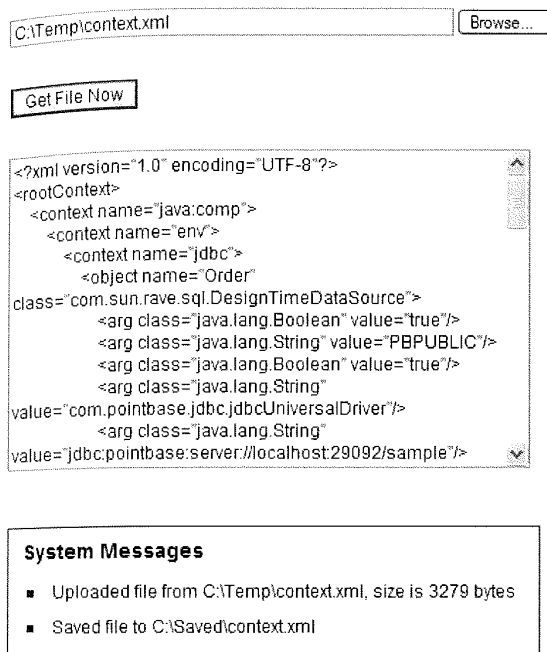


Figure 3-13 File upload component

## Hidden Field

Hidden Field

The hidden field component is a non-visual form field that is not displayed on the design canvas or in a browser window. The hidden field component is generated as an HTML `<input type="hidden">` element. Hidden field components do not appear in the design view, but you can access their properties from the Outline window.

Web developers typically use hidden field components to store data used by Javascript on the page. Hidden field components are also handy for storing page data, as an alternative to saving and restoring in session scope. The `text` property of a hidden field component holds the data that is sent to the server.

Note that anyone can examine an HTML document's source to locate a "hidden" field. Hidden fields, like password fields, are extended from the same

component classes as text field and therefore have the same configurable properties.

### Hyperlink **Hyperlink**

Hyperlink components are action components that provide navigation to other pages as well as to a location on a page (with the anchor component). A hyperlink component is also useful when a page's URL information is data driven and no processing is necessary (you set property `url`). Figure 3-14 shows a hyperlink for a home page in a browser window.



Figure 3-14 Hyperlink component

To add event handling code, select the hyperlink component in the design canvas and double-click. This brings up the `hyperlink1_action()` method (where `hyperlink1` is the component's `id` property) in the Java source editor.



#### Creator Tip

*If you want the hyperlink to show an image rather than text, use the image hyperlink component (see "Image Hyperlink" on page 84).*

#### Book Examples

- "Add Components to Page LoginBad" on page 213 (Chapter 5). Uses a hyperlink component with navigation.
- "Using Hyperlink with a Nested Static Text" on page 458 (Chapter 10). Uses a hyperlink and a nested static text component (for formatting) to provide a link to URLs returned from a Google web services search.

- “Modify the Components for Localized Text” on page 599 (Chapter 13). Configures a hyperlink component for localization.



## Image

Image components display graphics from a file or a URL. The image component is rendered as an HTML `<img>` element. Figure 3–15 shows an image component after dropping it on the design canvas.

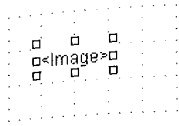


Figure 3–15 Image component

Once you place an image component on the design canvas, there are several ways to set its image. The image may be a file (JPEG, GIF, PNG), a URL web location, or a built-in theme. To set the image, right-click on the image component and choose Set Image. The Image Customizer dialog appears with radio buttons Choose File, Enter URL, and Set Theme Icon. Figure 3–16 shows the Image Customizer dialog with the Set Theme Icon selected and `ALARM_CRITICAL_MEDIUM` set for the image.

Selecting Choose File in this dialog lets you navigate to an image file in your file system and copy it to the **resources** node in your project. When you choose this option, the image component’s `url` property is set to `resources/image_filename`, where `image_filename` is the image file.

### Creator Tip

You can also add an image by dragging its file node from the file explorer dialog to your page.



The dialog also lets you enter a URL to a web location for the file. As before, the `url` property of the image component will be set to the URL you enter. Alternatively, you can select the `url` property in the Properties window and select the Use Binding radio button. This allows you to bind the property to a data object.

### Book Examples

- “Add the Google Logo” on page 451 (Chapter 10). Puts an image on the page.

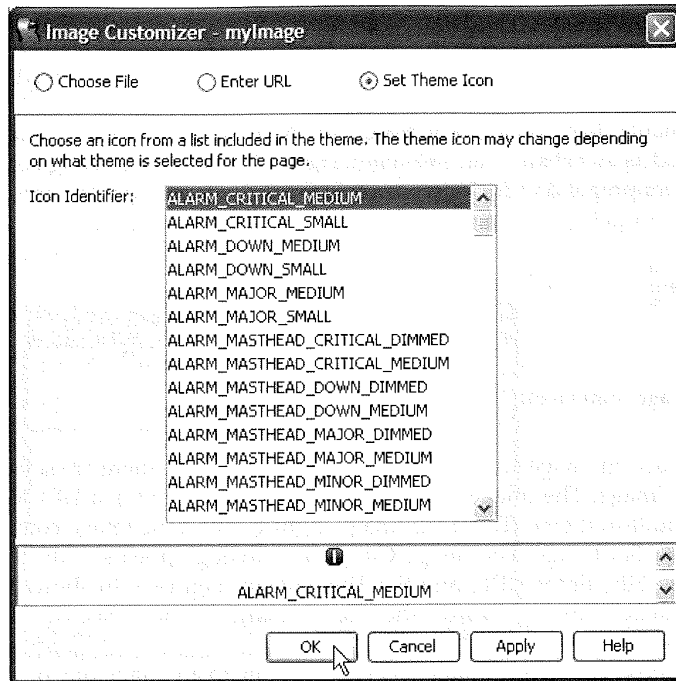


Figure 3-16 Image customizer dialog

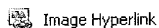


Image Hyperlink

## Image Hyperlink

The image hyperlink component is similar to a hyperlink, except that it supports images in addition to text. When you right-click an image hyperlink component, the Image Customizer dialog lets you set the image to a file (JPEG, GIF, PNG), a URL web location, or a built-in theme (see Figure 3-16).

The `imageURL` property specifies an image file or a URL on the web. The `icon` property holds the theme. Figure 3-17 shows an image hyperlink in a browser window.

As with hyperlinks, you can specify an action event handler. To add event handling code, select the image hyperlink component in the design canvas and double-click. Creator generates a default action event method and brings up the page bean in the Java source editor. The cursor is set to the `imageHyperlink1_action()` method (where `imageHyperlink1` is the component's `id` property).

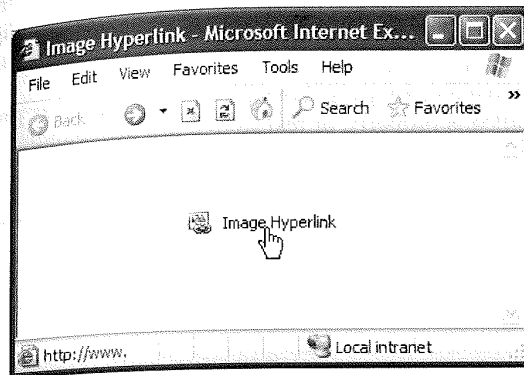


Figure 3-17 Image hyperlink component

### Book Examples

- “Banner Page Fragment” on page 309 (Chapter 7). Uses an image hyperlink with a page fragment.
- “Add an Image Hyperlink Component” on page 481 (Chapter 10). Uses image hyperlinks to page through Google search results.
- “Add Components to the Page” on page 562 (Chapter 12). Uses image hyperlinks to page through Google search results.

### Label



The label component is typically used to associate text with input components, such as text fields and checkboxes. Labels are rendered as HTML `<label>` elements when they are associated with components and as `<span>` elements when they are not. Figure 3-18 shows a label on the design canvas.

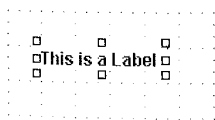


Figure 3-18 Label component

A label's `for` property associates the label with another component. When you bind the `for` property of a label to a text field, for instance, the label com-

ponent displays an asterisk if the text field's `required` property is set to `true`. Furthermore, if invalid input is supplied to the server, the page will highlight the label component's text in red. These behaviors make labels highly useful in pages where input components are heavily used.

Label components also have data binding. You can bind a label's `text` property to a data source, a JavaBeans property, or text from a resource bundle.



#### Creator Tip

*Input components have a dedicated `label` property that you can alternatively use for label text, but these labels cannot be easily resized or aligned. Instead, use the label component for more flexibility with component placement and style control.*

#### Book Examples

- “Add a Label Component” on page 207 (Chapter 5). Uses a label to place a heading on a page.
- “Modify the Components for Localized Text” on page 599 (Chapter 13). Sets the label's text from a localized `.properties` file.
- “Add Components for Input” on page 620 (Chapter 13). Sets the label's `for` property to a text field component and sets the label's `text` property from a localized `.properties` file.



Listbox

## Listbox

The listbox component allows users to select items from a list of items. The selection items may be hardcoded with a dialog accessed from the Properties window or generated dynamically at run time. Figure 3–19 shows a listbox component on the design canvas after configuring the options list.

Creator automatically supplies a converter for non-String data fields when you fill the list from a data provider source. The `multiple` property determines whether the user may select one item or multiple items in the listbox. (With `multiple` set, press `<Ctrl+Click>` to select more than one item.) The `rows` property controls the number of items to display.

When a listbox component's data provider wraps a database data source, you typically bind the database table's primary key field with the `Value` field. You select an appropriate field from the data table for the listbox component's `Display` field. Figure 3–11 on page 78 shows the (equivalent) `Bind to Data` dialog for a drop down list that lets you configure the data provider.

The nonvisual component `listbox1DefaultOptions` supplies text for the selections. To specify selection items, select the `listbox1DefaultOptions` element in the Outline view and click the `options` property in the Properties win-



Figure 3-19 Listbox component

down. A dialog pops up that lets you type in Display/Value pairs for each selection item, as shown in Figure 3-20. Text in the Display column appears on the page. The selected property (or method `getSelected()`) returns the corresponding text of the selected item from the Value column.

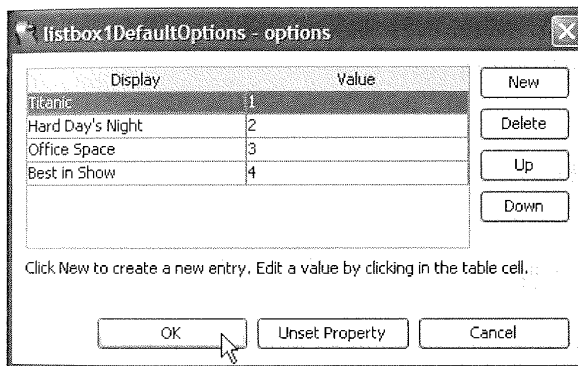


Figure 3-20 Dialog to select text items for listbox component

When the user selects a different item from a listbox component, the system generates a value change event. To submit the page for immediate processing on a value change event, set the Auto-Submit on Change feature. To provide event handling code for a value change event, double-click the listbox component on the design canvas. Creator generates a default `processValueChange()` method and brings up the Java source editor for you.

*Book Examples*

- “Add a Listbox Component” on page 400 (Chapter 9). Fills the selection list from a database data source. Uses listbox for a master-detail database read.
- “Add Components to the Page” on page 516 (Chapter 11). Fills the selection list from an EJB data source.
- “Add ApplicationBean1 Data” on page 577 (Chapter 12). Shows how to instantiate data in an `Options[]` array appropriate for a selection component, such as drop down list or listbox.



Message

**Message**

The message component displays error messages generated by other components. Typically, these messages are data conversion or input validation errors. When the validator or converter detects errors, it sends a message to the JSF context on behalf of the component. Message components can retrieve and display these messages. Message components are particularly useful on a web page that contains multiple input components. When you associate a unique message component with each input component, validation or conversion error messages clearly indicate the source of the input error. By default, a message component has its `ShowSummary` property set to `true` and its `ShowDetail` property set to `false`. Figure 3–21 shows a message component initially dropped on the design canvas.

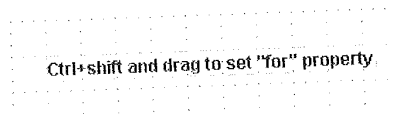


Figure 3–21 Message component

To associate a message component to an input component, select the message component, press **<Ctrl+Shift>**, and drag the arrow generated by Creator to the target component. This sets the `for` property to the `id` property of the input component. Figure 3–22 is an example page with a submit button and two message components associated with input text fields. Note that the names of the text fields (`textfield1` and `textfield2`) appear in the message text, indicating that the `for` property has been set for each message component. You can use the message component’s `style` property to format the appearance of your error messages.

You can also send your own message to a message component with the `info()`, `error()`, `fatal()`, or `warn()` methods. These methods are all rendered using distinct styles. You must include the message component’s target compo-



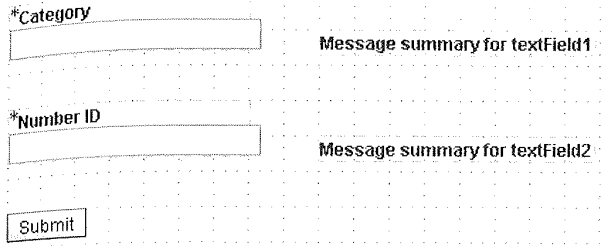


Figure 3-22 Message components with `for` property set

component `id` with the call, however. The following code shows the approach. Here, inside the listbox's value change event handler, we send a warning that the listbox component's value has changed. Note that component `id` `listbox1` is the first parameter for method `warn()`.

```
public void listbox1_processValueChange(
    ValueChangeEvent event) {
    warn(listbox1, "Value changed!");
}
```


#### Creator Tip

*The message component displays a single message only. If you need to display multiple messages, or you don't want to specify a particular component, use component `Message Group` instead.*



#### Book Examples

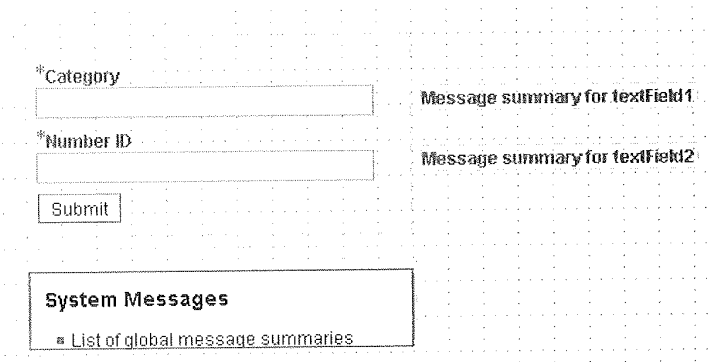
- "Use Validators and Converters" on page 252 (Chapter 6). Uses a message component to report data conversion errors for a text field component.
- "Add a Message Component" on page 474 (Chapter 10). Uses a message component to report validation errors for a text field component.
- "Add Components to the Page" on page 562 (Chapter 12). Uses a message component to report validation errors.
- "Add Components for Input" on page 620 (Chapter 13). Uses a message component to report validation errors from a custom validator method.

 Message Group

## Message Group

Message group components display run-time errors for page-level messages originating from multiple components or for system (global) messages. With message group components, you can limit the message group to display global errors only (that is, exclude component errors), or display errors for all components on the page, including errors with the page itself.

Figure 3-23 is a page with a submit button, input text fields with associated message components, and a message group component to report global errors. The message group component's `showGlobalOnly` property is set to true.



The figure shows a web page layout on a grid background. At the top left, there are two text input fields. The first is labeled '\*Category' and has a message component 'Message summary for textField1' associated with it. The second is labeled '\*Number ID' and has a message component 'Message summary for textField2' associated with it. Below these fields is a 'Submit' button. At the bottom of the page, there is a box titled 'System Messages' containing a bullet point: '▪ List of global message summaries'.

Figure 3-23 Message group components

Set property `showGlobalOnly` when one or more message components appear on the page with a message group component. This prevents a component's validation or conversion error message from appearing twice.

You can also use the message group `style` property to format the appearance of your error messages in the System Messages box.



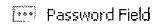
### Creator Tip

*It's a good idea to routinely place a message group component on your pages, especially when accessing a database, web service, or EJB. Recall that JSF writes `FacesException` messages to the JSF context. You will only see these messages if a message group or message component is on the page.*

Book Examples

- "Add a Message Group Component" on page 386 (Chapter 9). Uses a message group component to report database access errors.
- "Message and Message Group Components" on page 474 (Chapter 10). Uses a message group component to report system (global) errors.
- "Add Components to the Page" on page 502 (Chapter 11). Uses a message group component to show all error messages when there is only one input component.
- "Add the Google Web Service Client" on page 566 (Chapter 12). Uses a message group component to report system (global) errors.

Password Field



The password field component allows users to input a single line of text. Echoed text is replaced by a single character, such as a black dot or an asterisk. Password field components are useful for handling sensitive data input, like passwords and PIN numbers. The password field component is rendered as an HTML `<input type=password>` element. When a password field is rendered, its previous value is always cleared. Figure 3-24 shows a password field dropped on the design canvas.

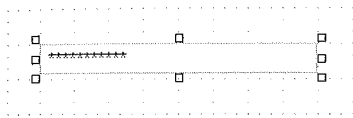


Figure 3-24 Password field component

In all other respects, a password field component behaves just like a text field. The input string is stored in the component's `password` property. When you change the text, a value change event is generated. The password field's `getPassword()` method reads the text and `setPassword()` sets it. You can also bind the `password` property to an object or data provider.

Password field components can have validators. Length validators, required validators, and range validators are all possible to check input text. Note that value change events occur only if no validation errors are detected.

When a password field component generates a value change event, the JSF implementation invokes the value change event handler for that component. You can use the password field's `label` property to set the label text on a page. It's a good idea have message components associated with password fields to report validation or conversion errors. To create a tooltip, set the password field's `toolTip` property.

*Book Examples*

- “Create the Form’s Input Components” on page 208 (Chapter 5). Uses a password field to gather input for a password field.
- “Bind Input Components” on page 237 (Chapter 6). Shows property binding with the password field component.
- “Modify the Components for Localized Text” on page 599 (Chapter 13). Shows how to localize an application that contains a password field (the password component’s `toolTip` is bound to the properties file).



Radio Button

**Radio Button**

The radio button component uses a boolean on/off setting as a choice for a user. Radio buttons can appear as standalone components on a page (not part of a radio button group). Figure 3–25 shows part of a page with two radio button components. Here we set the `label` property to the text strings shown. You can treat two or more radio buttons as a group by setting each radio button’s `name` property to the same value. When radio buttons are part of the same group, only one radio button can be selected (set to true).



*Figure 3–25* Radio button component

There are two important properties with radio buttons. The `selected` property indicates that a radio button is selected and clicked on the page. The `selectedValue` property allows you to pass data values for the radio button. It’s also possible to bind the `selected` property of a radio button to an object, such as a JavaBeans or a data source. A radio button is considered to be selected when the value of the `selected` property is equal to the value of the `selectedValue` property.

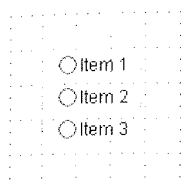
Use the `isChecked()` method to determine if the component is selected. To use radio buttons in table component columns, set the `name` property to the same value to group all the radio buttons in the column.

**Creator Tip**

*Radio buttons by themselves (not in a group) are used sparingly in web pages because users cannot deselect a single radio button once it is selected. If you want users to select and deselect their choices, use checkboxes (see "Checkbox" on page 75).*

**Radio Button Group**

The radio button group component lets you group radio buttons on a page. When a user selects a choice from a radio button group, each choice deselects the previous one. This means only one button within the group is "on" at a time.<sup>2</sup> Radio button groups are rendered as an HTML `<table>` element with rows and columns. Figure 3-26 shows a radio button group component after you drop it on to the design canvas.



**Figure 3-26** Radio button group component

The selection items can be hardcoded with the Properties window or dynamically generated at run time. Creator automatically supplies a converter for non-String data fields when you fill the list from a database source. This component also accepts data binding. The `selected` property of the radio button group returns the selected item.

To specify the choices, select `radioButtonGroup1DefaultItems` in the Outline view. In the Properties window, click the editing box opposite property options. Creator pops up a dialog so that you can add, edit, or remove items. (This is the same dialog box used to specify Display/Value fields for the Listbox component. See Figure 3-20 on page 87.)

**Example**

Figure 3-27 shows a radio button group component on a web page. The default layout for a radio button group is a single vertical column, so we set the `col-`

2. If you need to select more than one item at a time, use the checkbox group component (see "Checkbox Group" on page 76.)

## Chapter 3 Creator Components

umns property to the number of radio buttons (3) to get a horizontal layout. Note that a user may select only one choice.

Age Bracket  21-34  35-49  over 50

Your age is 35-49

Figure 3-27 Radio button group component

Here is the code in the button's event handler that displays the user's selection in a static text component. The code that accesses the radio button group component is bold.

```
public String button1_action() {
    String choice = (String)ageBracket.getSelected();
    staticText1.setText("Your age is " + choice);
    return null;
}
```

 Static Text

### Static Text

Of all the components, static text is probably used the most often in web pages. A static text component lets you display any kind of textual information, like instructions, titles, and headings. Static text components typically display String data, but you can bind them to objects, JavaBeans properties, and data providers. Figure 3-28 shows a static text component after dropping it on the design canvas.

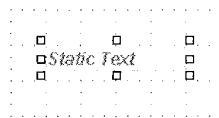


Figure 3-28 Static text component

With data converters and formatters, static text components can display almost any type of data. An embedded static text component is the default for a table component. Static text components may be embedded in hyperlink components to allow embedded HTML in the hyperlink's text display.

The text property of a static text component stores the text that is displayed. From a user's point of view, static text components are read-only. The `setText()` method sets its text and `getText()` reads it. You can resize static

text components on the design page, but Creator expands them if you leave them unsized.

Static text components are rendered as plain text, which may include HTML formatting tags. This means you can build entire HTML pages by concatenating a string of HTML tags with text and assigning it to the component's `text` property.

#### Creator Tip

*To enable correct rendering of HTML tags, make sure you set the `escape` property to `false` in the Properties window. Also, avoid using static text as labels for other components. Use either a separate label component or the label property of the component.*



#### Book Examples

- “Place Button, Label and Static Text Components” on page 257 (Chapter 6). Binds an output component to a JavaBeans property and applies a number converter.
- “Add Components to the Page” on page 298 (Chapter 7). Uses an embedded static text component in a grid panel. Builds the static text component's `text` by concatenating HTML tags and unchecking the `escape` property.
- “Using Hyperlink with a Nested Static Text” on page 458 (Chapter 10). Uses an embedded static text component in a hyperlink to store HTML text.
- “Configure the Table” on page 477 (Chapter 10). Uses an embedded static text component to improve HTML formatting.
- “Add Static Text Component to the Page” on page 586 (Chapter 12). Uses a static text component for formatting text using HTML tags in a portlet.

### Table



Table

The table component is a composite component with rows and columns. Tables typically have nested columns, which in turn contain other display components (such as static text components, buttons, or text fields). Table components render as HTML `<table>` elements.

A table node in the design palette has nested column and row group components (see Figure 3-29). Dragging these components to a table in the design canvas adds columns and row groups to the table. Figure 3-29 also shows a table component initially dropped on the design canvas with the default of five rows by three columns.

Figure 3-30 shows the Outline view for the default table component. Note that each column in a table has a static text field to display data, but you can replace it with other components (checkboxes, hyperlinks, for instance). Every

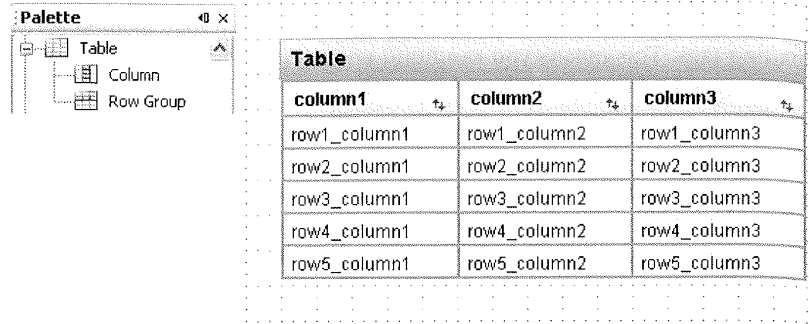


Figure 3-29 Table component

table component also has a default data provider (defaultTableDataProvider, a non-visual component).

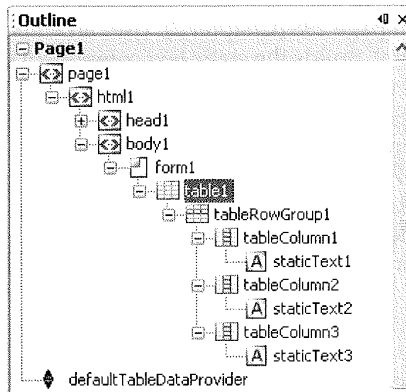


Figure 3-30 Table component Outline view



**Creator Tip**

Creator provides an enhanced selection mechanism for composite components, such as the table component. In the design canvas, click on any row in a table column and look in the Outline view and Properties window to see which component is selected. Now click again and you'll see the outer nested component's properties. Successive clicks let you cycle through each nesting level of a component.



column3
1_column3
2_column3
3_column3
4_column3
5_column3

:TableDataPro-

Table components are typically filled dynamically with data from a data provider attached to a data source (database table), web services method, EJB method, or JavaBeans property. When you fill a table component with data from a data provider, Creator lets you control the layout, including the columns to display, the number of headers and footers, and the component you use in each column. You can also apply data converters to any field (column).

When you bind a data provider to a table component in the design canvas, Creator automatically fills the table with the data and generates the needed columns. Creator also generates headers from the field names and applies the necessary converters.

**Example**

Figure 3-31 shows Creator's design canvas with a table component bound to the TRACKS table from the Music Database in Chapter 9. This table has three columns (with headings from the database metadata). Creator shows the column's data type as "123" for numeric data and "abc" for String data. For data that is not text, Creator applies a converter for you.<sup>3</sup> In this table, an embedded static text component is used for the display.

TRACKNUMBER	TRACKTITLE	TRACKLENGTH
123	abc	abc
123	abc	abc
123	abc	abc

Figure 3-31 Binding table component with an external database table

- For example, if a primary key field is integer data, Creator applies an Integer converter to the component. Creator performs this action for all the data-aware components.

Creator automatically sizes the columns and dynamically generates the correct number of rows. When you bind a table component to a database table, Creator generates a default query for you. You can modify this query by selecting the associated rowset from the nonvisual display. We show you how to work with database queries in Chapter 9 (see “Modify the SQL Query” on page 405.)

Figure 3-32 shows Creator’s dialog for manipulating a table component’s layout. Select the table component in the Design or Outline view, right-click, and choose Table Layout. Here, the dialog shows the columns from the TRACKS database table. You can choose which columns to display, the header and footer text, and the underlying component that holds the data.

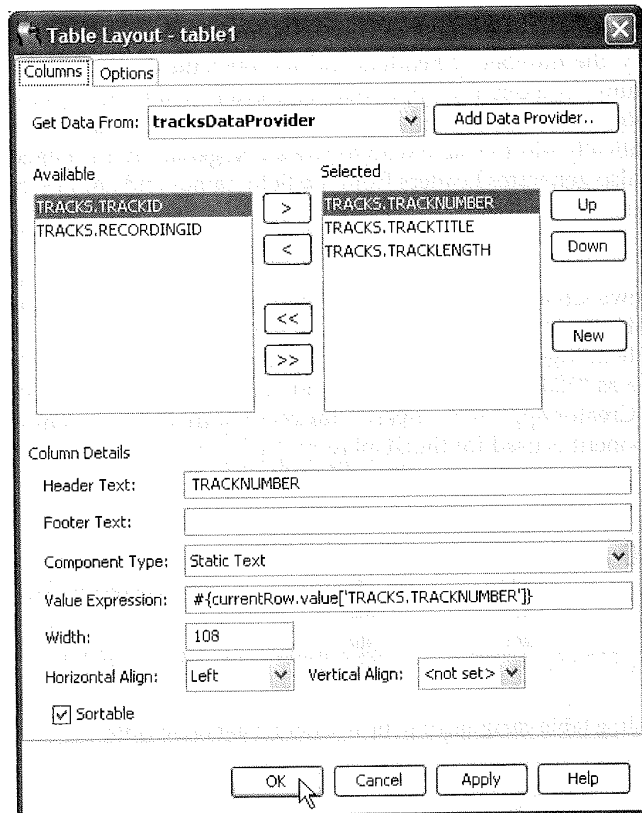


Figure 3-32 Table Layout dialog: specifying columns

generates the correct  
a database table,  
s query by select-  
how you how to  
SQL Query" on

able component's  
view, right-click,  
columns from the  
isplay, the header  
data.

Figure 3-33 shows Creator's dialog for specifying options for the table. Here you can set the table's title, description (summary), footer, and a message to display if the table is submitted without data. The checkboxes let you enable various options for the table component, including buttons to select all rows, clear sorting, open or close the table's sort panel, and enable pagination and a page size number. After making your selections, click Apply, then OK.

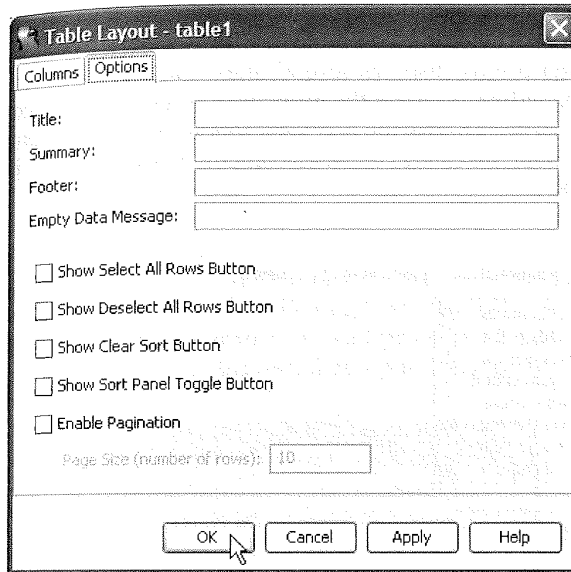


Figure 3-33 Table Layout dialog: specifying options

### Book Examples

- "Configure the Table" on page 360 (Chapter 8). Uses a table component with an object list data provider. Enables table pagination.
- "Configure Table Component" on page 397 (Chapter 9). Uses a table component with an SQL query parameter.
- "Add a Table Component" on page 402 (Chapter 9). Builds a master-detail relationship using data binding with a table component.
- "Modify the Table Layout" on page 412 (Chapter 9). Uses a table component with text field components for updating data.
- "Add Components" on page 429 (Chapter 9). Uses a table component with checkboxes.

- “Add a Table Component” on page 477 (Chapter 10). Uses a table component with an object array data provider and web services.
- “Configure Table Component” on page 556 (Chapter 12). Uses a table component with portlets and database access.



Text Area

## Text Area

Text area components gather textual information for multiple lines. This component is similar to a text field, but you build it with rows and columns (see Figure 3–34). Its standard look displays several lines, and a vertical scrollbar appears if the number of lines exceeds the number of rows. Text area components let you specify their size, provide text for a tooltip, and bind their `text` property to objects or data providers. Text area components are rendered as an HTML `<textarea>` element.

Please provide any additional information in the box below:

When you select components, Creator generates code in the page's JSP source as well as support code in the associated Java page bean. Furthermore, it places the component in the Outline Window where you can edit its properties and change

Figure 3–34 Text area component on a web page

Text area components are common with web applications that solicit free-form text. Examples are composing letters, listing comments, sending email, posting to guest books, filing bug reports, or reviewing products.

The `getText()` method retrieves the text and `setText()` sets it. The text is sent to the server when the page is submitted. Like the listbox component, text areas work with value change events and the `processValueChange()` event handler. To configure this method, double-click the text area component in the design view. Creator generates the event handler method in the Java page bean for you.

### Example

You can bind the `text` property to a session bean property to automatically save submitted text in session scope. For example, here is the generated JSP

es a table  
services.  
. Uses a table

code for a text area component that binds its `text` property to session bean property `userInfo` (shown in bold).

```
<ui:textArea binding="#{Page1.textArea1}" id="textArea1"
  style="height: 120px; left: 48px; top: 72px;
  position: absolute"
  text="#{SessionBean1.userInfo

```

## Text Field

 Text Field

ple lines. This com  
s and columns (see  
a vertical scrollbar  
. Text area compo  
nd bind their text  
are rendered as an

The text field component enables users to input a single line of text. The input string is stored in the component's `text` property, and a value change event is generated when you change the text. The component's `getText()` method reads the text and `setText()` sets it. The text is sent to the server when the page is submitted. Text field components are rendered as an HTML `<input type="text">` element.

Figure 3-35 shows a text field in a browser window that prompts for a person's first name. The text field's label and `toolTip` properties are set to the strings shown. The boolean `required` property makes a red asterisk appear with the label and alerts the user that input is mandatory.

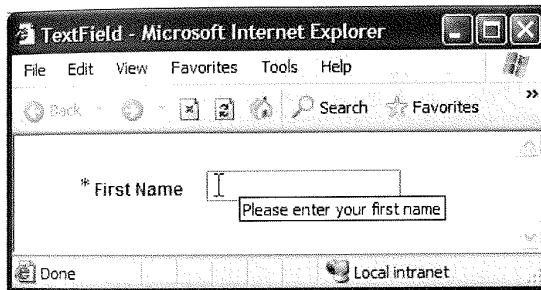


Figure 3-35 Text field component on a web page

that solicit free-  
s, sending email,  
cts.

ets it. The text is  
component, text  
Change () event  
component in the  
e Java page bean

With text fields, you can attach a length validator, a required validator, or range validators (with converted numerical values). Value change events occur only if no validation errors are detected. When a text field component generates a value change event, the JSF implementation invokes the value change event handler for that component. Message components are handy for reporting validation or conversion errors with text fields. (See "Message" on page 88.)

You can also attach a data converter to a text field. To do this, select the converter you want from the `converter` property in the Properties window under

o automatically  
e generated JSF

Data. When you apply a data converter, the type of the `text` property changes from `String` (the default) to the converted type. If you don't want all input components on the page to be validated, use virtual forms (see "Configure Virtual Forms" on page 216).

Text fields may be embedded in table components and you can bind them to data or other objects. Figure 3-36 shows the Property Bindings dialog box for binding a text field component to a JavaBeans property. Here, we bind text field `username` with the `username` property in the JavaBeans component `loginBean`.

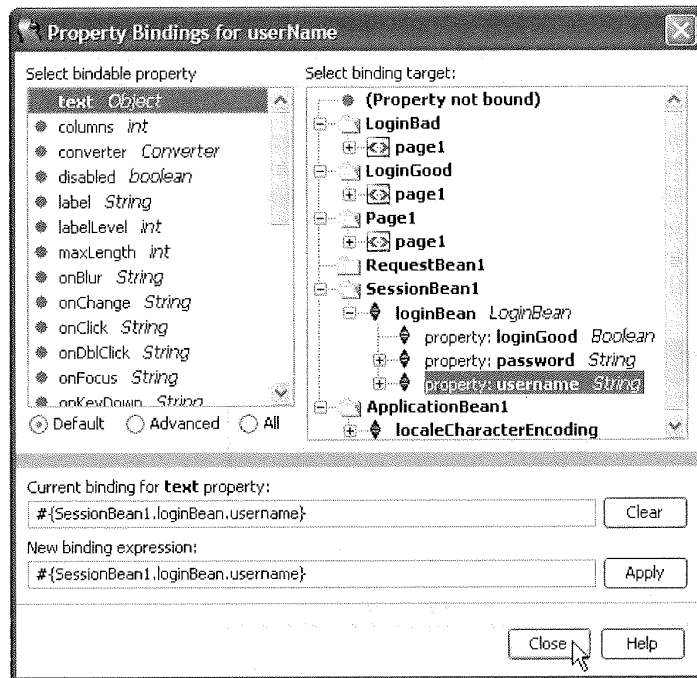


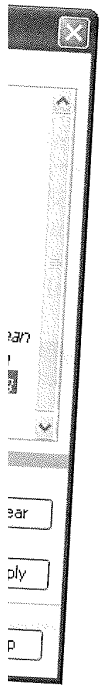
Figure 3-36 Property Bindings dialog with text field component

#### Book Examples

- "Configure Virtual Forms" on page 216 (Chapter 5). Excludes validation of text field components with virtual forms.
- "Bind Input Components" on page 237 (Chapter 6). Shows binding properties with a text field.

xt property changes  
want all input com  
"Configure Virtual

ou can bind them to  
lings dialog box for  
Here, we bind text  
s component login-



- "Create the Form's Input Components" on page 250 (Chapter 6). Shows text fields with converters and validators.
- "Add Components to the Page" on page 298 (Chapter 7). Uses a text field in a nested grid panel.
- "Modify the Table Layout" on page 412 (Chapter 9). Uses text fields with a table component.
- "Add Components" on page 417 (Chapter 9). Uses text fields to gather input for database row insert operations. Uses virtual forms.
- "Add a Text Field Component" on page 451 (Chapter 10). Shows validators.
- "Add Components for Input" on page 620 (Chapter 13). Uses text fields with a custom validator method.

## Tree



The tree component lets you render data in an expandable list with a hierarchical tree structure. In web applications, trees are useful for navigating through nested data, like file systems and categories. A tree component contains tree nodes, which act like hyperlinks. In the design palette, a nested tree node component appears when you expand a tree node, as shown in Figure 3-37.

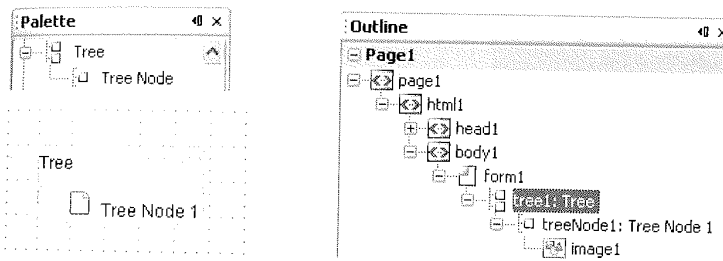


Figure 3-37 Tree component

Figure 3-37 also shows you the design canvas for a tree component and its Outline view. Note that a tree node has an embedded image component. Once you expand a tree component and drop it on the design page, you can drop tree node components to build nested structures.

Initially, when you drop a tree component on a page, the root node is labeled Tree and the subnode is labeled Tree Node 1. The `text` property lets you set the strings to be rendered for these nodes, and the `toolTip` property gives users more information about the node.

validation of  
inding

**Creator Tip**

When you drop tree node components on tree components, pay attention to what Creator outlines in blue. If the entire tree component is blue, the tree node will render as a sibling of the tree component. Otherwise, the tree node will render as a nested node underneath the node outlined in blue.

There are several important properties with tree components. The `url` property lets you navigate to another page or display data like a PDF or JPEG file. Binding the `action` property to an action event handler makes the tree node automatically submit the current page. The `clientSide` boolean property controls whether a request to the server is made each time a user expands or collapses a node.

**Example**

Figure 3–38 shows a tree component called Download Site with tree nodes Home Page and Music. Underneath the Music node are the nested tree nodes Jazz, Rock, and Country.



Figure 3–38 Example tree component

Note that the image for a tree node is a page icon if it is not nested. Otherwise, a folder icon appears with an arrow if the node has children (nested nodes). On a web page, users may expand or collapse the folder to see the nested nodes by clicking the arrow icon.

Suppose a web application displays PDF files for the Jazz, Rock, and Country music categories. When you select a tree node on the design page and click the `url` property customizer box, a dialog appears to set the property. Clicking the Add File button lets you browse for the location of the PDF file you want to display. Figure 3–39 shows this dialog for the Jazz tree node.



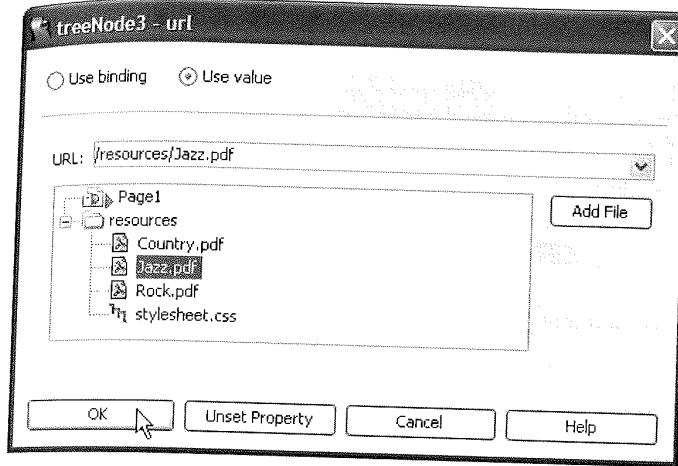


Figure 3-39 Url dialog for tree node component

#### Creator Tip

*At the time of this writing, tree node selection events do not work in portlet projects.*



## 3.4 Layout Components

The following catalog of layout components describes each component and gives you common usage scenarios. To show you how layout components can be useful in a Creator project, we also point you to relevant examples in other chapters of this book. The layout components are listed alphabetically for easy lookup.

### Form



The IDE makes sure that every new page that you create already has one form component. If you want to add more forms, drag the form component from the design palette and drop it on the page. This is usually not necessary in most applications, but you may want to manage certain components in their own forms. If you add a new form component to a page, it appears in the Outline

view along with `form1`, the default form (see Figure 3-40). New form components render as selected boxes in the design canvas.

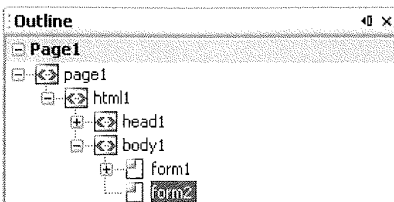


Figure 3-40 Form component Outline view



#### Creator Tip

*If you need nested forms, use a virtual form (see “Virtual Forms” on page 70). Although you can always delete form components that you add to a page, it is not possible to delete the default form component, since every page must have one.*



Grid Panel

## Grid Panel

A grid panel component is a general-purpose container that groups other components and controls their layout. When you drop a grid panel on the design canvas, you can place other components inside of the grid panel. Creator fills the grid panel with your components in a grid (rows and columns) layout. The components appear on the grid panel in the order that you drop them. (You can rearrange components later by “re-dropping” them on the grid panel.) Grid panels render as HTML `<table>` elements.

By default, grid panels have one column but you can modify the `columns` property to add more columns. The grid panel displays its components left to right to fit the number of columns you specify. It also resizes the number of rows based on how many components you have in the grid panel. Figure 3-41 shows a grid panel (box of dashed lines) in the design canvas containing a button, checkbox, and radio button. The grid panel on the left is a vertical layout (one column, the default). Next to it is a grid panel with `columns` set to 3.

The grid panel is particularly useful when you don’t know how much space a component will take up on the page. For example, if a static text component is built dynamically and you want to place another component after it on the



Figure 3-41 Grid panel components

page, you can nest both components in a grid panel. The layout mechanism adjusts the relative position of each component appropriately.

The grid panel component has other properties that control its appearance. These properties include `bgcolor` for background color, `cellspacing` and `cellpadding` for cell width spacing, and `border` for the width of the grid panel's border lines.

#### Creator Tip

Use the Outline view rather than the design canvas to work with nested components. It's much easier to place components on top of a desired target with the Outline view. Rendering in the design view often obscures the specific target component that you're trying to drop onto.



#### Book Examples

- "Add a Grid Panel Component" on page 193 (Chapter 5). Uses a grid panel to hold button components.
- "Add Components to the Page" on page 298 (Chapter 7). Uses nested grid panels to help with component layout.
- "Layout and Grouping with Grid Panel" on page 457 (Chapter 10). Uses a grid panel to group different components and control their rendering.
- "Add Components to the Page" on page 502 (Chapter 11). Uses a grid panel and nested grid panel to help with layout. Uses static text components as placeholders in grid panels.
- "Add Components to the Page" on page 521 (Chapter 11). Uses a grid panel with table components to help with layout.
- "Using Grid Panel to Improve Page Layout" on page 597 (Chapter 13). Uses a grid panel to handle layout for components rendered with text read from properties files.
- "Adding Components to the Page" on page 619 (Chapter 13). Uses a grid panel to help with layout.



## Group Panel

A group panel is a general-purpose container that groups components and controls their layout. Whereas grid panels place components in a grid configuration (you specify the number of columns), a group panel component uses a flow layout. Depending on the width of the panel, group panels arrange components one after the other in a flow. When there's not enough room in the first row, Creator continues with placement in a second row. Like grid panels, the order in which you drop components on a group panel is the same order that they appear on the page.

A group panel component renders as an HTML `<span>` element and the page bean implements a group panel as a `PanelGroup` object. (If you set property `block` to true, a group panel renders as an HTML `<div>` element.) Figure 3-42 shows a group panel (box of dashed lines) in the design canvas containing a button, checkbox, and radio button.

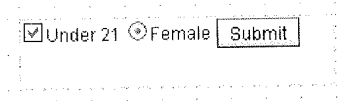
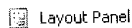


Figure 3-42 Group panel component



### Creator Tip

Group panels are handy for grouping nested components. It's possible, for example, to place a group panel inside cells of a grid panel. This technique lets you create interesting web pages by placing groups of components in each cell of a grid panel. From the grid panel's perspective, these nested components are treated as a single cell.



## Layout Panel

The layout panel component is a container that groups components and lets you choose a layout mode. When you drag a layout panel component from the component palette and drop it on the design canvas, the IDE gives you a Flow Layout by default. As you drop components in the layout panel, the IDE aligns them from left to right on the top line, moving them to the next line if there is not enough room. This makes layout panels behave like group panels as you add components.

If, on the other hand, you'd like to use the design canvas to position components at arbitrary (absolute) places in the panel, change the `panelLayout` prop-

erty to Grid Layout (use the drop down list in the Properties window). Now each component will be positioned relative to the nearest grid lines. This makes layout panels behave like Creator's design canvas (the default grid layout).

Figure 3-43 shows a layout panel (box of dashed lines) in flow layout mode containing a drop down list, a listbox, and a button.

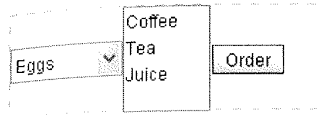


Figure 3-43 Layout panel component

Layout panel components are also the default for tab set components (see "Tab Set" on page 114).

#### Book Examples

- "Add Components to the Page" on page 289 (Chapter 7). Uses a layout panel in Grid Layout mode to position components.
- See TabSet3 project in the Creator download file (**FieldGuide2/Examples/WebPageDesign/Projects/TabSet3**). Uses a tab set component with embedded layout panels for each tab selection.

### Page Alert



The page alert component displays messages on a separate page. If you don't want to use a separate page, use an alert component from the Composite palette (see "Alert" on page 118). Page alerts are useful because they have recognizable icons and configurable messages. Figure 3-44 shows the page alert component after you drop it on the design canvas.



Figure 3-44 Page alert component

The `type` property in the Properties view contains a drop down list of icons and alert types. There are four types of alerts: error, warning, information, or question. The `summary` property displays a brief text message for the alert, and the `detail` property lets you display a longer, more detailed message. You can also right-click a page alert component and bind its properties to a Javabeans property or another object.

Figure 3-45 shows two page alerts in a browser window. The left alert displays a brief information message. The right alert shows an error alert with a brief reason for the alert followed by a detailed suggestion.



Figure 3-45 Page alert components

 Page Fragment Box

## Page Fragment Box

Page fragments are separate, reusable components that you include in multiple web pages. The Page Fragment Box component generates a JSP directive that includes a JSP file fragment in your page. Page fragments let you build web pages that have consistent form. You may, for instance, use page fragments to include the same graphic header in all pages of an application.

When you select a page fragment component and drop it on the design canvas, Creator pops up a dialog that lets you create a new page fragment or select an existing one. Figure 3-46 shows the Select Page Fragment dialog.

Page fragment files show up in the Projects view Web Pages > resources node as a `.jspxf` file. In the Outline view, a page fragment box appears as a node underneath an HTML `<div>` element. The name of this node has the format `directive.include:fragment_filejspxf`, where `fragment_file` is the name of your page fragment file. Figure 3-47 shows the Outline view for the `Fragment1jspxf` page fragment.

Once you have a page fragment box, you can add visual elements to it as needed. A typical example is a page fragment consisting of a banner with a company's logo (an image component). As you create pages in your web application, drag a page fragment box component to the page, position it, and specify the page fragment name.

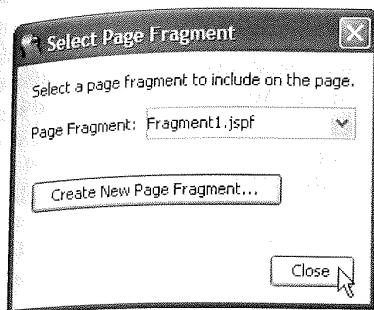


Figure 3-46 Select Page Fragment dialog

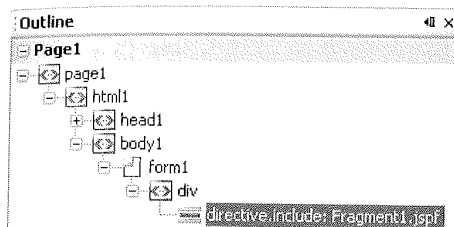


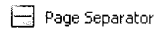
Figure 3-47 Page fragment Outline view

**Creator Tip**

When adding components to page fragments, make sure the id of any new component does not conflict with any component id names on the including page. Also, virtual forms are not allowed within page fragments.

**Book Examples**

- “Banner Page Fragment” on page 309 (Chapter 7). Uses page fragment box components to create uniform looking pages.
- “Using Tab Sets and Page Fragments” on page 327 (Chapter 7). Uses a page fragment box with tab set components.



Page Separator

## Page Separator

The page separator component creates a horizontal line on your page. This lets you separate other components for a better visual layout. Page separator components are rendered as HTML `<hr>` elements. You can change a page separator's width and appearance in the Properties view. In the page bean, a page separator component is a `PageSeparator` object.

Figure 3-48 shows a page separator with a drop-down list, text field, and submit button.

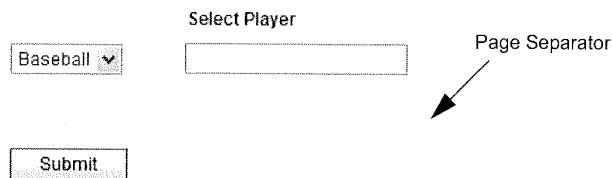


Figure 3-48 Page separator component



Property Sheet

## Property Sheet

The property sheet component is a layout composite component. In the design palette, property sheets contain nodes for nested property sheet section components and property components (see Figure 3-49).

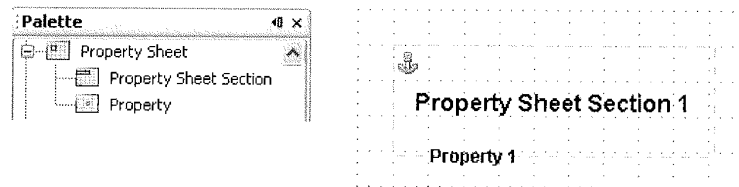


Figure 3-49 Property sheet component

When you drag a property sheet component to the design canvas, the initial layout is one sheet section containing one property, as shown in Figure 3-49. It's possible to have multiple sheet sections on a page with header strings ini-



tialized with the sheet section's label property. You can also have multiple property components within each sheet section.

Property components are containers with labels, optional help text, and default formatting. By default, the property component displays read-only data, but you can attach input components, such as calendars, drop-down lists, or text fields. To add new properties, drop a property component on a property sheet section, or right-click the property sheet section component and select Add Property. After creating property components, the Outline view is helpful for dropping input components on a selected property. Figure 3-50 shows a layout with one sheet section (section1) and four properties. Each property has its own input component.

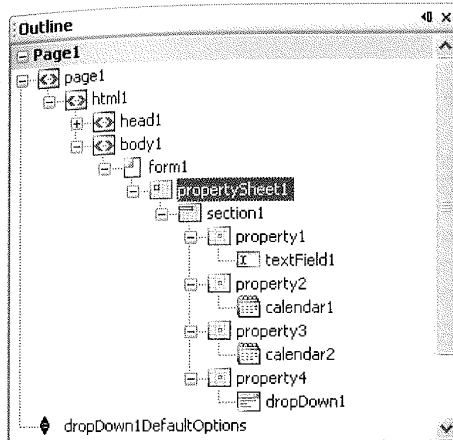


Figure 3-50 Property sheet Outline view


There are several important properties for property sheet components. The `requiredFields` property displays a required fields message (and red asterisk) when set to `true`. A property sheet component also contains an anchor component by default (see "Anchor" on page 72). If you set the `jumpLinks` property, the property sheet displays links to its sections at the top of the property sheet. If a property has an input component, you can set the optional `required` property for that component to force data entry on the page.

#### Example


Property sheets are handy for creating data entry forms. Figure 3-51 shows an entry form for a rental car reservation in a browser window.


Here, we have one property sheet set up with properties and input components. Note that this property sheet has required fields for most of the input


\* Indicates required field

 **Rental Car Reservation**

\* Pickup City

\* Pickup Date    
mm/dd/yyyy

\* Return Date    
mm/dd/yyyy

Car Type  

*Figure 3-51 Property sheet example*

components. The drop-down selection is not required for input and defaults to the initial string `Sub Compact`. The `requiredFields` property is set here to display the required fields message.

### Tab Set **Tab Set**

The tab set is a layout composite component. In the design palette, tab sets contain a nested tab component node (see Figure 3-52). Tab set components let you click tabs to view alternate sets of components or navigate to different pages. Each tab in a tab set is a tab component with configurable properties. To add a new tab, right-click the tab set component and choose `Add Tab` or drop a new tab component on a tab set (or another tab component for nested tabs).

In the Outline view, tab components provide a default layout panel (see Figure 3-52) to hold components that become visible when a user clicks a tab. Each layout panel's `panelLayout` property is set to `Grid Layout` by default, but you can change it to `Flow Layout` in the Properties view. If you use tab sets to navigate between pages, be sure to delete each tab component's layout panel. The `selected` property of a tab set component determines which tab is initially selected. Tab selections also change color when you select them.

You can create an event handler by double clicking any tab component of a tab set in the design view. It's also possible to bind a tab component's text property to an object or data provider.

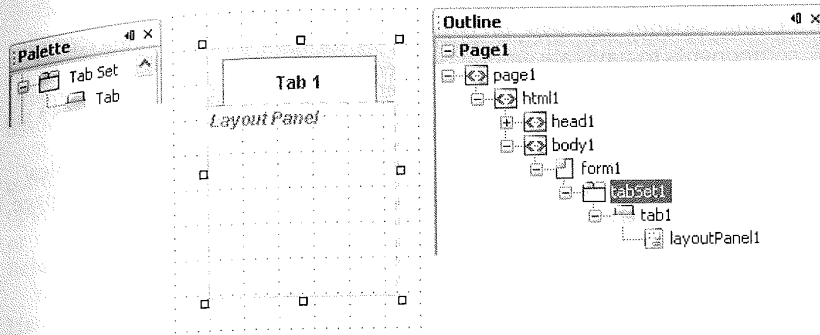


Figure 3-52 Tab set component

**Creator Tip**

When you drop a tab component to the left or right of an existing tab in a tab set, the tab component appears in the same row of tabs. Otherwise, the tab component will be a child of the tab component that you drop it on. You can have at most three levels of tabs in any tab set.



Figure 3-53 shows the design canvas for a tab set with three tab components. In the Outline view, each tab component has its own layout panel.

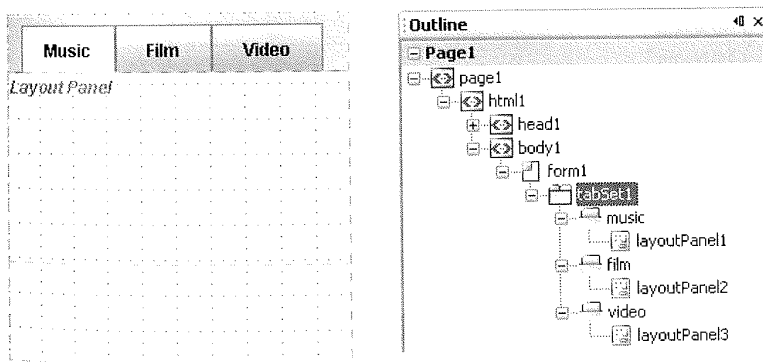


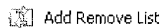
Figure 3-53 Design canvas and Outline view of tab set

*Book Examples*

- “Using Separate Tab Sets” on page 324 (Chapter 7). Uses separate tab set components to navigate among pages.
- “Add Tab Set and Tabs to CactusBanner” on page 329 (Chapter 7). Uses a tab set component with a page fragment for navigation.
- See TabSet3 project in the Creator download file (**FieldGuide2/Examples/WebPageDesign/Projects/TabSet3**). Uses a tab set component on one page to display different sets of components.

### 3.5 Composite Components

The following catalog of composite components describes each component and gives you common usage scenarios. The composite components are listed alphabetically for easy lookup.



Add Remove List

#### **Add Remove List**

The add remove list component lets users select items from one list and add or remove them from another list. The component displays two listboxes and two buttons. One listbox displays available options and the other displays selected options. The buttons let you add or remove options from the two listboxes.

Figure 3–54 shows the layout after you drop the add remove list component on the design canvas and fill in the selection items in the available listbox.

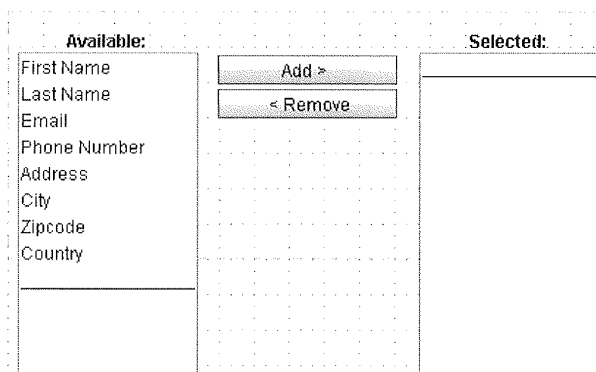


Figure 3–54 Add remove list component

The selection items can be set through the Properties window or dynamically generated at run time. Creator automatically supplies a converter for non-String data fields when you fill the list from a data provider. The `selected` property of the add remove list returns the selected items. The `items` property associates the component with a data provider.

To specify the choices, select the non-visual `addRemoveList1DefaultItems` component in the Outline view. In the Properties window, click the editing box opposite property `options`. Creator pops up a dialog so that you can add, edit, or remove items. (This is the same dialog box used to specify Display/Value fields with the Listbox component. See Figure 3-20 on page 87.)

You can also right-click the add remove list component and select Edit Event Handler from the pop-up menu. The `validate` option lets you insert Java code that validates user input, and the `processValueChange` option lets you insert Java code that executes when a component's value has changed.

#### Example

Let's add a submit button and a static text field to the sample page shown in Figure 3-54. In a browser window, the submit button determines which selections were made and the static text field displays them. Figure 3-55 shows the results of clicking the Submit button after adding an email, city, and country to the selected listbox. The static text field displays the selected string items.

<p>Available:</p> <div style="border: 1px solid black; padding: 5px;"> <p>First Name Last Name Phone Number Address Zipcode</p> <hr style="width: 100%;"/> </div>	<div style="border: 1px solid black; padding: 2px; width: 100px; margin: 5px auto;">Add &gt;</div> <div style="border: 1px solid black; padding: 2px; width: 100px; margin: 5px auto;">&lt; Remove</div>	<p>Selected:</p> <div style="border: 1px solid black; padding: 5px;"> <p>Email City Country</p> <hr style="width: 100%;"/> </div>
<div style="border: 1px solid black; padding: 2px; width: 80px; margin: 0 auto;">Submit</div>	<p> dilbert@yahoo.com Anywhere USA </p>	

Figure 3-55 Add remove list component with selections

Here is the Java code for the Submit button event handler method that reads the selections from the add remove list component and displays the strings in the static text component.

```
public String submit_action() {
    String selections = addRemoveList1.getSelectedValues();
    staticText1.setText(selections);
    return null;
}
```



Alert

## Alert

The alert component lets you display messages on a page. Alerts have recognizable icons and configurable messages. When you drag an alert component from the palette and drop it on to the design canvas, the default is an error alert, as shown in Figure 3-56.



Figure 3-56 Alert component

The `type` property in the Property view contains a drop down list of icons and alert types. There are four types of alerts: success, error, warning, and information. The `summary` property displays a brief text message for the alert, and the `detail` property lets you display a longer, more detailed message. If the `summary` property is empty, the component won't display on the page. You can also right-click a page alert component and bind its properties to a Javabeans property or another object.

Component alert includes an embedded hyperlink component, which you access by setting property `linkText`. To specify an action event handler for the hyperlink component, right-click the alert component and select Edit action Event handler.

### Example 1

Figure 3-57 shows a page with a success alert directing users to a second page. To show the check mark icon, we set the alert component's `type` property to success. The `summary` and `detail` properties are set to "Item in Stock" and "To Process Your Order", respectively. Users are directed to another page via the `linkText` property, set to the string "Click Here".

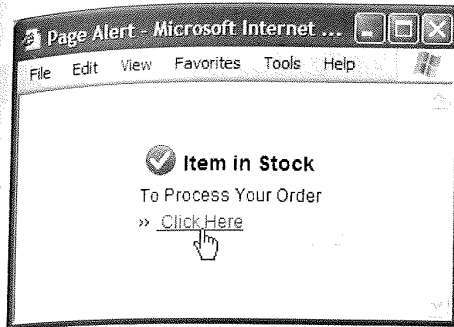


Figure 3-57 Success alert with page navigation

To implement page navigation, we use Creator's page navigation editor to specify navigation. Figure 3-58 shows the page navigation editor connecting the two JSP pages with string "alertOutcome".

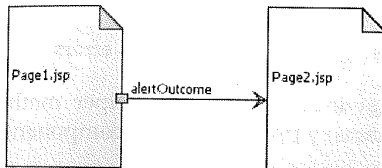


Figure 3-58 Page navigation editor

The IDE generates an event handler when you double-click the alert component. Here is the Java code for the alert event handler method, which simply returns the same string used by the page navigator.

```
public String alert1_action() {
    return "alertOutcome";
}
```

#### Example 2

You can use alert components in place of message or message group components. Figure 3-59 shows project Color1 (see "Custom Validation Method" on page 611) running in a browser. We replaced the three message components

with alert components (with property id of redAlert, greenAlert, and blueAlert).

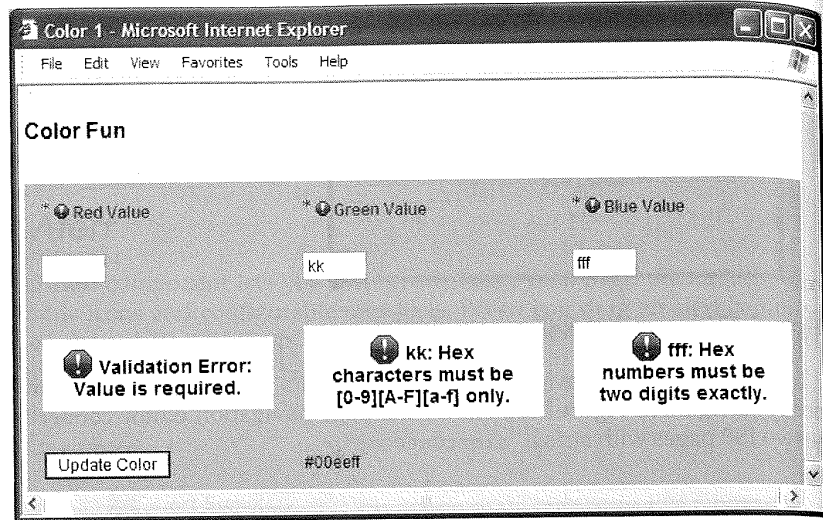


Figure 3-59 Using alert instead of message to display validation errors

In the page bean method `prerender()`, we invoke helper method `setAlertMessage()` to set the alert's summary property with a component-specific message from the `FacesContext`.

```
public void prerender() {
    setAlertMessage(alertRed, redInput);
    setAlertMessage(alertGreen, greenInput);
    setAlertMessage(alertBlue, blueInput);
}
```

Method `setAlertMessage()` obtains the `FacesContext` and any messages associated with `inComp`, its argument's component. `FacesContext` method `getMessages()` returns an `Iterator` of `FacesMessages` for the component's `clientId` passed as an argument. Conveniently, the alert component does not display if its summary property is empty.

```
private void setAlertMessage(Alert ac, UIComponent inComp) {
    FacesContext context = FacesContext.getCurrentInstance();
    Iterator mi = context.getMessages(
        inComp.getClientId(context));
}
```




```

String newMessage = "";
while (mi.hasNext()) {
    newMessage += ((FacesMessage)mi.next()).getSummary()+" ";
}
ac.setSummary(newMessage);
}

```

## Breadcrumbs

 Breadcrumbs

The breadcrumbs component is a default layout for hyperlinks. The name comes from an old hiker's trick where you drop breadcrumbs on a trail to find your way back and not get lost. With applications having many different web pages, breadcrumbs typically show a user's location by displaying the path through the page hierarchy to the current page.

When you drag a breadcrumbs component from the palette and drop it on the design canvas, the IDE includes a nested hyperlink component for every page in the application. Figure 3-60 shows the Design view and Outline view for a breadcrumbs component with two pages.

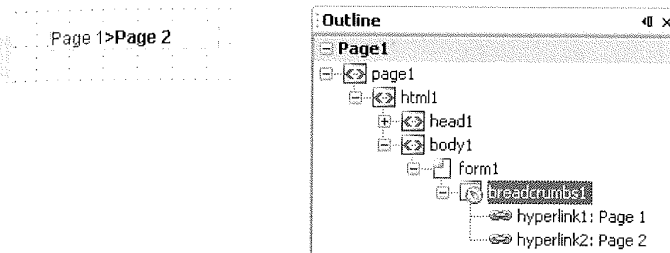
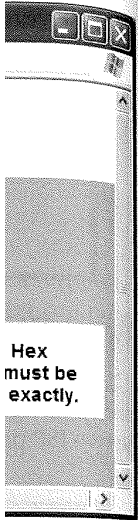


Figure 3-60 Breadcrumbs component

In the Design view, the breadcrumbs component separates hyperlinks by right angle brackets (>). By default, the initial component has a single hyperlink that points to the current page.

The `url` and `action` properties of each hyperlink are set the same way for breadcrumb components (see "Hyperlink" on page 82). You populate a list of hyperlinks by setting the `pages` property of a breadcrumb component to point to any array or list of `HyperLink` objects. You can also bind the `pages` property to a JavaBeans component or data provider.

art, and blue-



rs

method set-  
component-specific

any messages  
method get-  
component's  
component does not

```

nComp) {
    ance();
}

```

**Creator Tip**

*With portlets, the IDE does not provide a default hyperlink for breadcrumb components. You must add the hyperlinks yourself.*

**Example**

Suppose an application has separate web pages to help users edit, compile, debug, and test a program. On the test page, it may be important to refer to the previous pages for information that relate to testing. Figure 3-61 shows a breadcrumbs component in a browser with links to the previous pages visited by the user.

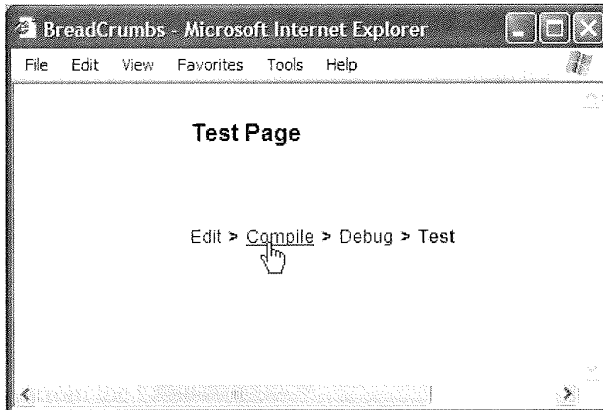


Figure 3-61 Breadcrumbs Example

Property `url` of each hyperlink in the breadcrumbs component is set to the page that was previously visited.



Inline Help

**Inline Help**

The inline help component is similar to a label. However, inline help components are restricted to displaying short help information for users on web pages. Once you drop an inline help component on the design canvas, you can type text directly in the component box. You may resize the box and the text wraps automatically. Figure 3-62 shows an inline help component on the design canvas.

This is Inline Help text

Figure 3-62 Inline help component

The inline help component has a `type` property which may be set to `page` (the default) or `field` in the Properties view. Page view is a larger font that applies to a page, whereas field view is a smaller font to help describe individual components. You can set the `style` property of an inline component using the Style Editor and the `styleClass` property using the styleClass Property editor. The `text` property can also be bound to an object or data provider.

#### Example

Figure 3-63 shows a page in a web browser with an inline help component at the top with `type` property set to `page`. Below the Confirm Selection button, a second inline help component appears with its `type` property set to `field`.

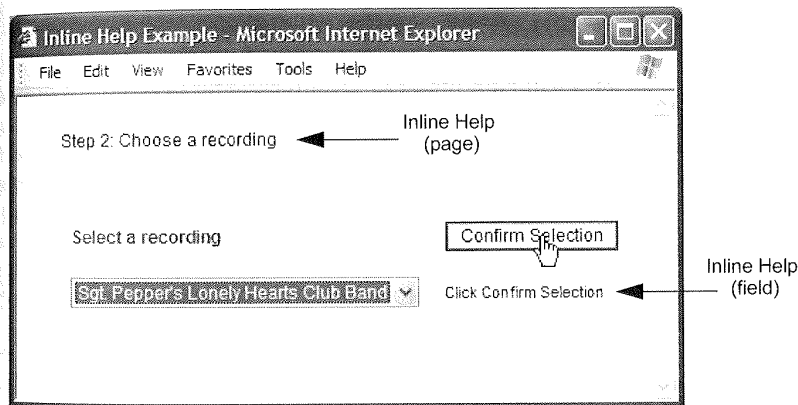


Figure 3-63 Inline help example

## 3.6 Validators

Creator provides a set of standard objects that validate user input gathered through UI components. The JSF architecture builds validation into the page request life cycle process, making validation an easy task for the developer to specify. Figure 3-64 shows the available validators in the Creator Components palette.

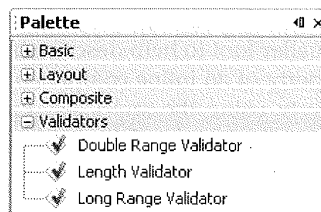


Figure 3-64 Validators

### Validation Model

In Creator, you attach a validator object to a component by selecting a validator from the design palette and dropping it onto the component in the design canvas. You can also select a validator from the drop down list opposite property validator in an input component's Properties view. Validators have properties that you can manipulate to specify range limits (for example).

The JSF life cycle (see Figure 6-16 on page 262) includes a Process Validations phase. For components that have registered validators, JSF will validate the component's data. When validation errors occur, the affected component is marked "invalid" and an error message is sent to the JSF context.

Validation errors affect the life cycle process. Validation errors cause the page to proceed directly to the Render Response phase, skipping the Update Model Values phase and Invoke Application phase. This means events such as button clicks are not processed. When a page has multiple components with registered validators, all input is validated. This is helpful to the user since feedback (error messages) for the entire page can be displayed. Table 3.1 describes the validators in more detail.

There are three standard validators: a Double Range Validator for floating types, a Length Validator for strings, and a Long Range Validator for integral values. You can also write your own custom validation method. Note that each standard validator has properties for minimum and maximum values. The Length validator works with String data, and the Double Range and Long

Table 3.1 JSF Validators

<i>Name</i>	<i>Description</i>	<i>Example</i>
Double Range Validator	Specify minimum and maximum values.	"Use Validators and Converters" on page 252 (Chapter 6). Uses a Double Range Validator with a text field to check the range of a double.
Length Validator	Specify minimum and maximum values. Does not detect empty input fields (you use required property of component).	"Add a Validator" on page 472 (Chapter 10). Uses a Length Validator with a text field.
Long Range Validator	Specify minimum and maximum values.	"Place Interest Rate and Term Components" on page 254 (Chapter 6). Uses a Long Range Validator with a text field to check the range of an Integer value.
Custom Validate Method	<code>validateHexString()</code> method checks for a 2-digit hex string	"Add a Validation Method" on page 617 (Chapter 13). Shows how to implement your own validation method.

Range validators are typically used with converters to convert a component's data to the correct type.

Table 3.1 also points you to examples in the book that show you how to use the validators. This includes an example of a custom Validate Method called `validateHexString()` that checks for a 2-digit hexadecimal string in a web application.

## 3.7 Converters

JSF strives to separate presentation data (the data that users read and possibly modify) from internal data or model data. To accomplish this, you should use JavaBeans components, EJB components, JDBC cached rowsets, and other application-specific structures to represent model data and behaviors. JSF also makes sure that any data conversions between the two views are consistent and well-defined.

Figure 3-65 shows the available converters in the Creator design palette.

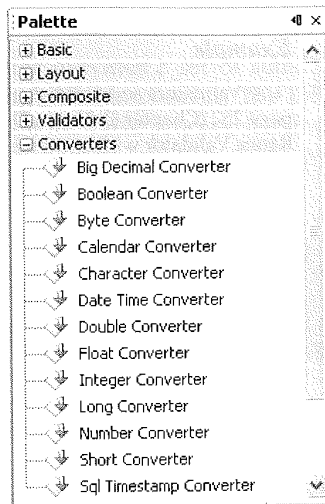


Figure 3-65 Converters

## Conversion Model

A UI component (components for input such as text fields or components for output such as labels and static text components) can take a data converter to convert its data to a specific type. Typically (but not always), the component may be bound to a JavaBeans property of that type. For example, in project **Payment1** (see "LoanBean" on page 242), we bind a text field component (`loanAmount`) to the `amount` property of `LoanBean`. Property `amount` is a `Double`, so we apply a `Double` converter to the text field component.

Like the validation process, JSF sets aside specific times to perform conversions. For an input component, conversion applies to the submitted input before validation. When errors occur, the affected component is marked "invalid" and conversion error messages are sent to the JSF context. JSF proceeds to the Render Response phase in this case.

Creator applies converters automatically to data-aware components when the source data type is not a `String`. Table 3.2 describes the converters available on Creator's palette.

Most of the converters are straightforward and provide a conversion that's obvious from their name. Note that all converters use wrapper classes (sub-classed from `Object`) instead of the primitive types. This allows the text property (type `Object`) to accept all of these types.

Table 3.2 JSF Converters

<i>Name</i>	<i>Description/Example</i>
Big Decimal Converter	Converts between String and <code>java.math.BigDecimal</code> .
Boolean Converter	Converts between String and Boolean.
Byte Converter	Converts between String and Byte.
Calendar Converter	Converts between String and <code>java.util.Calendar</code> .
Character Converter	Converts between String and Character.
Date Time Converter	"Configure the Table" on page 360 (Chapter 8). Converts between String and <code>java.util.Date</code> .
Double Converter	"Use Validators and Converters" on page 252 (Chapter 6). Shows a double converter with an interest rate value and a loan amount value.
Float Converter	Converts between String and Float.
Integer Converter	"Place Interest Rate and Term Components" on page 254 (Chapter 6). Shows an integer converter with a loan term value.
Long Converter	Converts between String and Long.
Number Converter	"Place Button, Label and Static Text Components" on page 257 (Chapter 6). Shows a number converter with a currency value.
Short Converter	Converts between String and Short.
Sql Timestamp Converter	Converts between String and <code>java.sql.Timestamp</code> .

The Date time converter, Number converter, and Sql Timestamp converter require a bit more explanation, however, so let's do that now.

## Date Time Converter

The Date Time Converter converts a component's data to a `java.util.Date`. When you apply a Date Time Converter to a text field, the textual input is converted. The field on the page is updated with a standard format during the Render Response phase. You can always configure a Date Time Converter's format if you need to. If you don't specify a locale, the Date Time Converter uses the default locale (see "A Word About Locales" on page 593).

The Date Time Converter uses the format rules and patterns of the `DateFormat` class. See the tutorial at <http://java.sun.com/docs/books/tutorial/i18n/format/dateFormat.html> for more information on formatting; see also the Javadoc for the `DateFormat` class at <http://java.sun.com/j2se/1.4.2/docs/api/java/text/DateFormat.html>.

The Date Time Converter uses a default pattern if you don't configure it differently. The data are assumed to be a date (as opposed to time) using the pat-

tern MMM d, yyyy. Although full names for the month are accepted, the Date Time Converter shortens it to three letters and rejects numerical values. On input, you must supply a comma. See “Configure the Table” on page 360 for an example of applying the Date Time Converter to a table column.

Of course, your choices for other formats are more flexible. Table 3.3 shows the results of applying the Date Time Converter to a specific date (April 5, 1985) in String format. The table shows several formatting patterns and the effect of setting the `dateStyle` property to `medium`, `long`, and `full`.

**Table 3.3** Date Time Converter

<i>Property/Pattern</i>	<i>Result</i>
<code>medium</code>	Apr 5, 1985
<code>long</code>	April 5, 1985
<code>full</code>	Friday April 5, 1985
<code>MM-dd-yy</code>	04-05-85
<code>EEE, MMM d, "yy</code>	Fri, Apr 5, '85

## Number Converter

A Number Converter lets you manipulate numerical data using either a pattern, or specifying minimum and maximum digits and fraction digits. Since numbers are sensitive to language and locale, a Number Converter can use locale.

The Number Converter uses a pattern with separate properties for manipulating a format (such as currency symbol, integer digits, fraction digits, and locale). We use a Number Converter to convert a double to a dollar (String) value here (see “Place Button, Label and Static Text Components” on page 257). Also see Figure 11-8 on page 505, which shows the Number Format dialog. We use it in Figure 11-8 to convert a `BigDecimal` value to String for output, using pattern “USD #,###.00” for a currency amount in U.S. dollars.

## Sql Timestamp Converter

The Sql timestamp converter converts data between String values and `java.sql.Timestamp` data types. It is also useful for binding a component to a database column of type `TIMESTAMP`. You can use to convert input data to type `TIMESTAMP` or display `TIMESTAMP` values on the web page.

## 3.8

Asynch  
technic  
allow a  
and wil  
library  
compo  
and ad  
Compo

Figure

In t  
to add  
AJAX  
AJAX  
nents

C

S:  
m  
L

In

The fi  
target



## 3.8 AJAX Components

Asynchronous JavaScript Technology and XML (AJAX) is a web development technique for building interactive web applications. Its main purpose is to allow asynchronous updates on a web page without refreshing the whole page and without performing a submit and postback. Creator provides a component library that includes experimental-technology AJAX components. To use these components, install the Update Center's most recent AJAX component library and add it to the Components palette. Figure 3-66 shows the BluePrints AJAX Components and Support Beans installed in the Components palette.

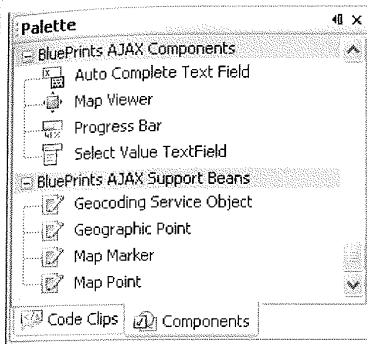


Figure 3-66 BluePrints AJAX Components and Support Beans

In this section, we'll show you how to use the Component Library Manager to add the AJAX component library to the palette. We show you how to use the AJAX-enabled Auto Complete Text Field component in Chapter 13 (see "Using AJAX-Enabled Components" on page 628 and "Using AJAX-Enabled Components with Web Services" on page 638).

### Creator Tip

*Since the AJAX-enabled components are under development, you should make sure you have installed the most recent component library from the Update Center.*



### Importing a Component Library

The first step in using one of the AJAX-enabled components is to import the target component library into Creator.

1. From the Creator main menu, select **Tools > Component Library Manager**. Creator brings up the Component Library Manager dialog, as shown in Figure 3-67. (Alternatively, right-click on one of the Component sections in the Components palette and select **Manage Component Libraries**.)

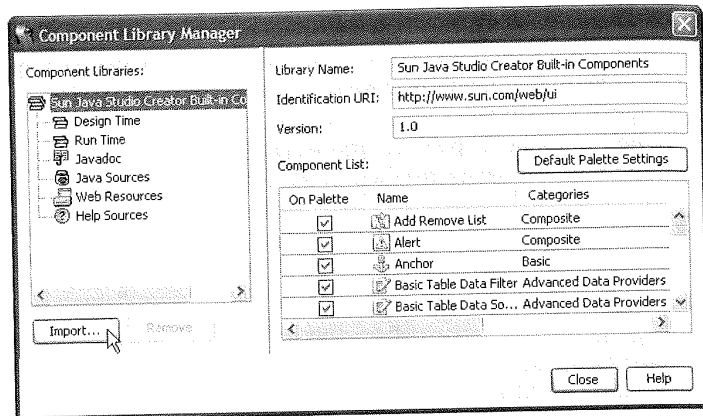


Figure 3-67 Component Library Manager Dialog

2. Click **Import**. Creator brings up the **Import Component Library** dialog.
3. Click **Browse** and navigate to the directory **samples/complib**.
4. Select **ui.complib** and click **Open**.
5. You'll see **BluePrints AJAX Components** and **Support Beans** in the text field under radio button **Import into Palette Categories** defined by Library, as shown in Figure 3-68. Click **OK**.
6. The **Component Library Manager** dialog now shows the **BluePrints AJAX Components** listed under the **Component Libraries**. The **Component List** includes the **Auto Complete Text Field**, **Map Viewer**, **Progress Bar**, and **Select Value TextField** (and support beans) as shown in Figure 3-69. Click **Close** to close the **Component Library Manager** dialog.
7. From the **Components palette**, open the **BluePrints AJAX Components** section to see these components added to the palette (as shown in Figure 3-66).



#### Creator Tip

*More components will be included in the BluePrints AJAX Components Library as they are developed.*

Library Manager dialog, as shown in component sections in libraries.)

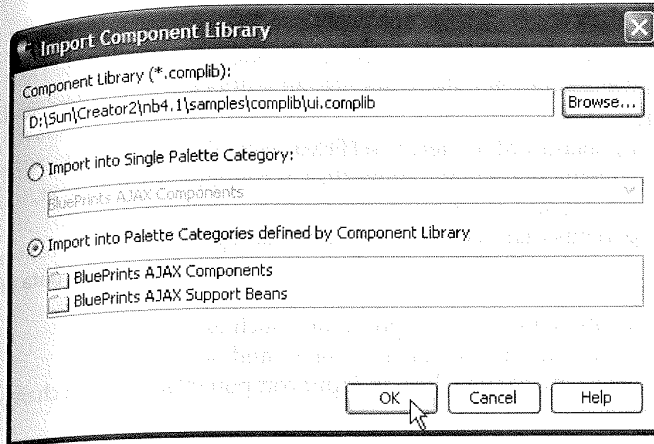


Figure 3-68 Import Component Library Dialog

Library dialog.

as in the text field by Library, as

BluePrints AJAX Component List, and Figure 3-69. Click

Component sections in Figure 3-66).

Components

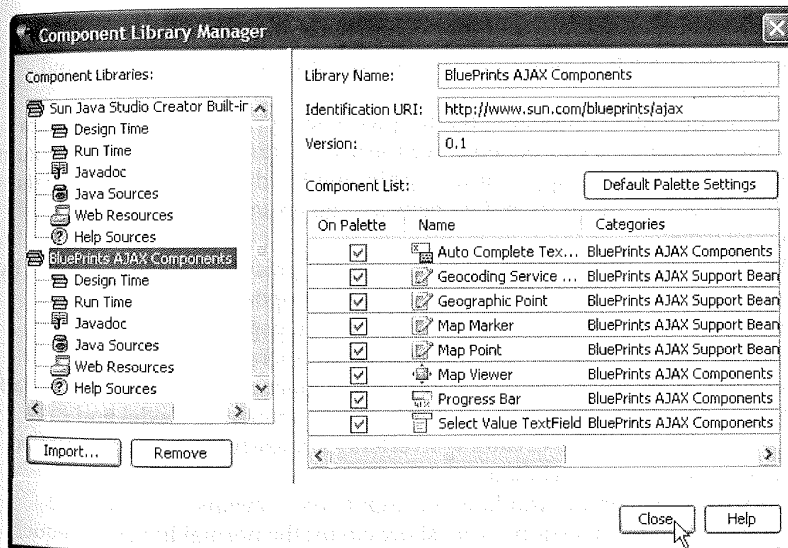


Figure 3-69 After importing the BluePrints AJAX component library

### 3.9 Key Point Summary

- Creator's design palette contains components, validators, converters, and data providers.
- Creator's components are rendered in HTML.
- Creator components share many properties in common, such as `text`, `toolTip`, `style`, `id`, and `binding`.
- Creator components that manipulate data can accept converters to convert data to an from String form. You can also use converters to format data on output.
- Input components share common properties, such as `validator`, `maxLength`, `required`, `valueChangeListener`, and `onChange`.
- A value change event occurs when an input component's selection changes or its text changes.
- Creator generates a `processValueChange()` method when you double-click an input component.
- The Auto-Submit on Change feature submits a page when an input component fires a value change event. Creator generates a JavaScript element and configures the component's `onChange` property to implement this feature.
- Table components (table and grid panel) have properties to control appearance, such as `bgcolor`, `border`, `cellspacing`, `cellpadding`, and `columns`.
- Creator provides component binding with data providers that wrap data sources, JavaBeans components, web services return objects, and EJB return objects. You can also apply property binding to arbitrary application data. This simplifies transferring data between the presentation view and the model view.
- A table component is data aware and offers sophisticated layout choices. By specifying headers, footers, and embedded component types for its columns, the page designer can build a custom page for displaying data.
- You can enable paging controls with table components. This is useful for database queries or other data that produce more than a single page of data.
- JSF has data converters that encourage the separation of model and presentation data. The Creator converters seamlessly convert presentation data to and from model data.
- The Creator validators validate user input before events are processed. Validation and conversion errors short-circuit the normal life cycle request mechanism and re-render the page with error messages.
- Use a message or message group component to display validation or conversion errors on a web page.
- Use message group components to display system or global errors.

- You can write your own custom validation method and hook it into the JSF validation cycle.
- Use the Component Library Manager to import component libraries to Creator's Components palette, as well as to configure the palette.
- Creator includes a bundled BluePrints AJAX Components library, an experimental technology set of components that use AJAX.

verters, and

as text,

ers to convert  
rmat data on

tor,

ction changes

u double-click

input  
aScript  
o implement

ntrol  
ling, and

wrap data  
nd EJB return  
cation data.  
v and the

it choices. By  
or its  
ying data.  
useful for  
page of data.  
and  
resentation

ccessed.  
r/cle request

on or

ors.

# SOFTWARE DEVELOPMENT

## Topics in This Chapter

- Editing Java Code
- Refactoring
- Source Code Control with CVS
- Creating Non-Web Projects

# Chapter

# 4

**S**un Java Studio Creator has an integrated development environment (IDE) that greatly simplifies the “edit-compile-deploy” cycle of complex web applications. Based on NetBeans, the IDE has code generation and navigation features that make it easy and pleasurable to edit and compile programs. In addition to keyboard shortcuts and code completion, the IDE also provides code refactoring and CVS source code control. All of these features make up a development environment that helps you create, manage, and maintain your web applications.

This chapter shows you how to use the Source Editor to write Java code effectively. You will learn how to customize the IDE to your tastes and create a comfortable environment to develop applications. We’ll also show you how to refactor your code when it becomes necessary to make large changes, like changing the name of a method or a heavily-used class. Because source code maintainability is so vital today with complex web projects, we’ll show you how to put your code under CVS source code control. Along the way, there will be plenty of examples to help you understand how to use these features<sup>1</sup> in Creator projects.

- 
1. This chapter focuses only on editing, refactoring, and versioning in the IDE. To learn more about Creator’s software development features beyond these topics, consult the NetBeans documentation.

## 4.1 Using the Java Source Editor

The Java source editor is where you'll spend a lot of your time in Creator. This section shows you useful features that make it easier to develop your applications.

### Finding What You Need

Creator allows you to customize the Java editor to suit your individual tastes. If you select Tools > Options from the Creator toolbar and select Java Editor under the Editing > Editor Settings node, you will see General and Expert settings for the editor, as shown in Figure 4-1.

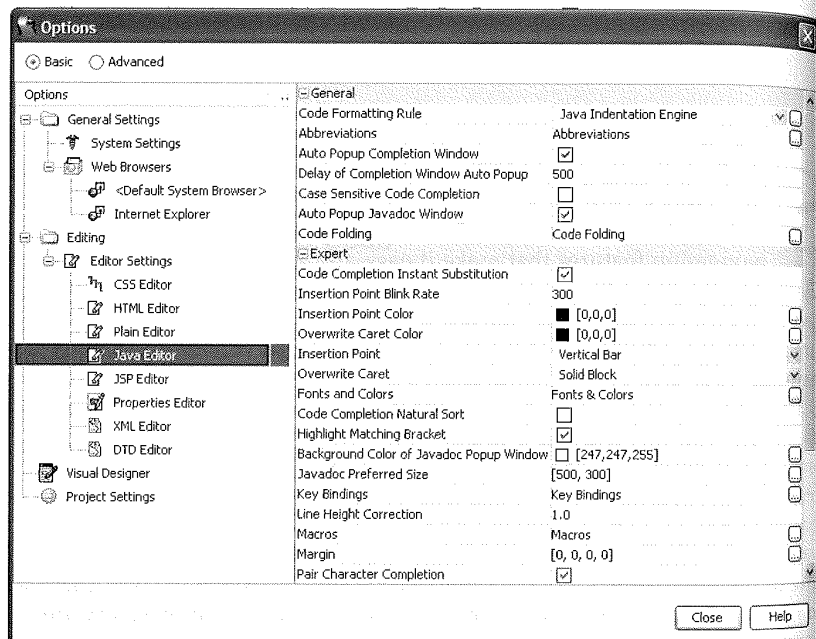


Figure 4-1 Java Editor Basic Options

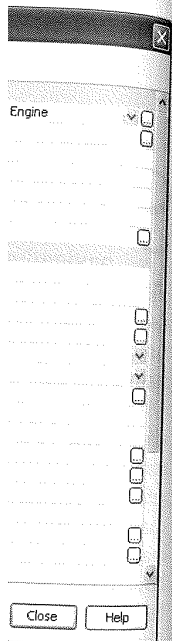
Note that you can do basic things like change code formatting rules, add new editor abbreviations, or modify code folding for imports and methods. If you are feeling like an expert, you can change fonts and colors, key bindings, and even the insertion blink rate (to save on your eyes). Take a few moments to click on the customizer boxes with several of the features here, and you will learn a lot about what you can do to customize the editor.



or

me in Creator. This  
relop your applica-

ndividual tastes. If  
select Java Editor  
ral and Expert set-



If you click the Advanced radio button in the Options dialog, you will see the Java Code Formatting Rule settings under the Editing > Code Formatting Rules node. Figure 4-2 shows the list of configurable rules that affect code formatting.

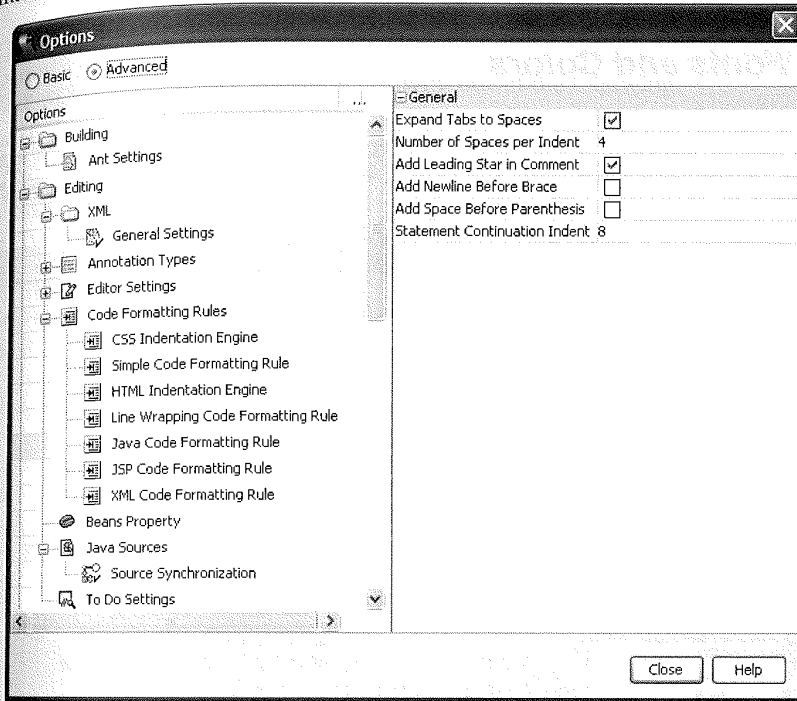


Figure 4-2 Java Editor Advanced Options

## Formatting Code

Java code is automatically formatted in Creator according to default rules. Members of classes, for example, are indented four spaces, continued statements are indented eight spaces, and any tabs that you enter are converted to spaces. No spaces are placed before an opening parenthesis, and an opening curly brace is put on the same line as a class or method declaration.

To reformat all the code in any file in Creator, type **<Ctrl+Shift+F>**. (Alternatively, right-click inside the editor and select Reformat Code from the context menu.) This is very handy right after you paste a code fragment from another file into your source code. To indent blocks of code manually, type **<Tab>** or **<Ctrl+T>**. Typing **<Shift+Tab>** or **<Ctrl+D>** reverses indents.

ting rules, add  
and methods. If  
s, key bindings,  
ew moments to  
e, and you will

You can change any of Creator's default settings for code formatting by accessing the Java Indentation Engine. This can be done directly with the Java Code Formatting Rule in the Advanced Options dialog (see Figure 4-2), or by clicking on the customizer box for Code Formatting Rule in the Basic Options dialog (see Figure 4-1).

## Fonts and Colors

The Basic Options dialog (Figure 4-1) lets you configure font size and style, as well as the foreground and background colors of the editor. Under Expert, click the customizer box for Fonts and Colors. Figure 4-3 shows the settings for Java Method calls.

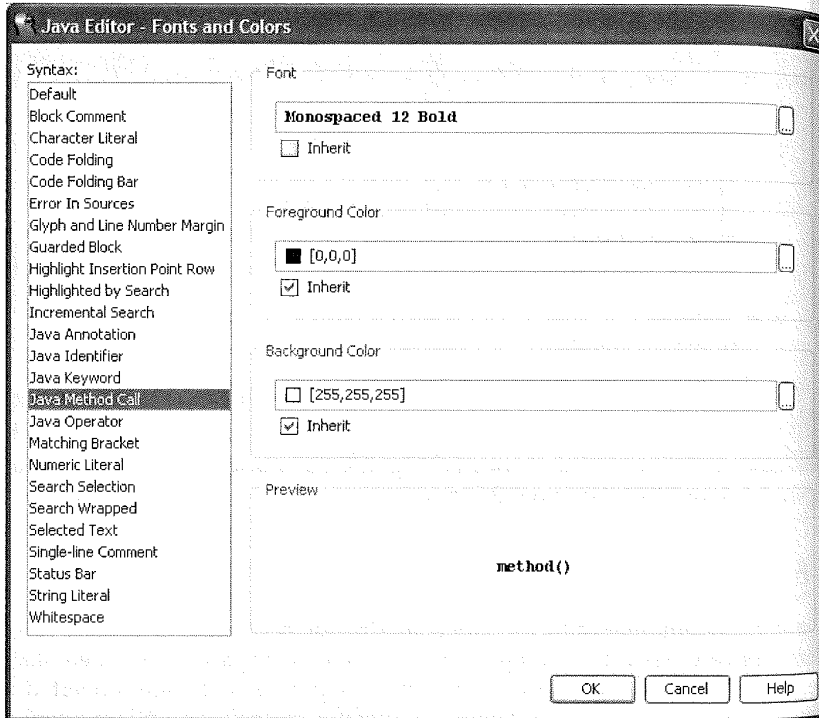


Figure 4-3 Fonts and Colors Dialog

Clicking the customizer box for a Foreground or Background Color brings up a color palette to choose a different RGB value. Likewise, clicking the customizer box for a Font allows you to change its style and point size. The Inherit

code formatting by directly with the Java (see Figure 4-2), or by in the Basic Options

checkbox indicates whether or not a font or color should be inherited from the Default syntax category.

## Code Completion

One of the handier features of the IDE is code completion, which lets you type part of a Java identifier and let the IDE finish the expression for you. To use this feature, activate a code completion box with one of the following:

- Type a few characters in an expression, then press **<Ctrl+Space>** or **<Ctrl+>**.
- Pause after you type a period (.) in an expression (this gives you a choice of method names).
- Type the import keyword followed by a space.

A code completion box contains a list of choices to select from. After you choose what you want, just press **<Enter>** to finish. To close the code completion box without choosing anything, press the **<Esc>** key.

Figure 4-4 shows a code completion box and associated Javadoc popup window. Here we typed new List followed by **<Ctrl+>** and selected ListDataProvider in the code completion box. Note that the Javadoc window provides the documentation for this class. After you press **<Enter>**, the IDE completes the class name and adds the import statement for the class to your code.

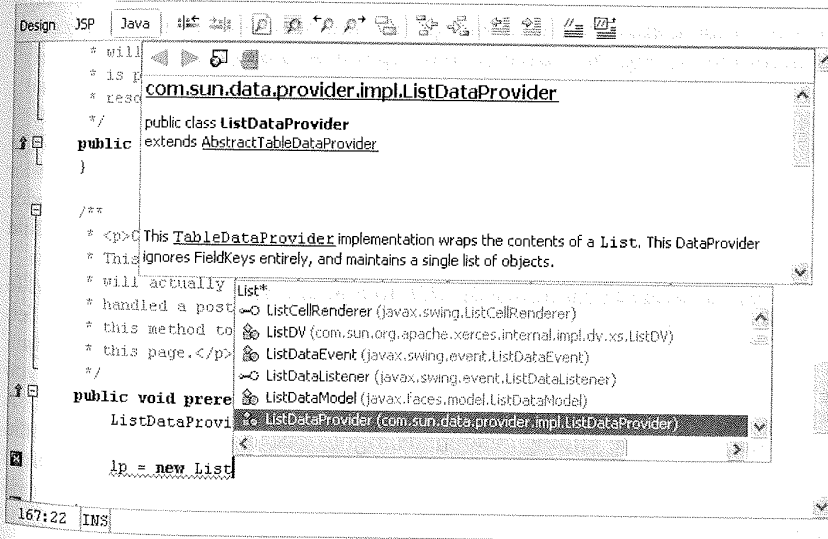


Figure 4-4 Code Completion and Javadoc popup

## Disabling Code Completion

You can disable the code completion box and Javadoc popups if you don't want them active. To do this, choose Tools > Options from the Creator menu, expand the Editing > Editor Settings node, and select Java Editor. Under General, select from any of the following.

- To disable the code completion box - uncheck the checkbox for Auto Popup Completion Window.
- To disable Javadoc popups - uncheck the checkbox for Auto Popup Javadoc Window.
- To change the code completion box display delay time - modify the default of 500 milliseconds for the Delay of Completion Window Auto Popup.

Note that changing these options disables only the automatic appearance of what the IDE does. Once disabled, you can still manually activate code completion with <Ctrl+Space> (or <Ctrl+>). Likewise, typing <Ctrl+Shift+Space> manually activates Javadoc popups.

## Show Line Numbers

You can toggle whether or not the Java source editor shows line numbers. Place the cursor in the blue left-margin area and right-click. Select Show Line Numbers from the context menu to enable line numbers. This is especially convenient for identifying the source of an exception or debugging activities.

## Code Folding

The Editor lets you collapse (or fold) certain sections of code to make room for other lines. You may fold methods, inner classes, import blocks, and Javadoc comments. Clicking a box icon in the left margin allows you to fold/unfold code that is bracketed by a vertical line extending down from the icon.

It's also possible to configure the IDE to fold code for you automatically. To do this, click Tools > Options from the toolbar and select Editing > Editor Settings > Java Editor. In the property window for Code Folding, click the customizer box (see Figure 4-1) and select the checkbox for any code element that you would like folded by default.

To access the code folding commands, right-click in the editor window and select Code Folds from the context menu. Or, select Window > Code Folds from the toolbar. Figure 4-5 shows the context menu for Code Folding and its shortcuts.



Figure 4

## Ha

There a choices

- Fast for th Impe only state
- Fix I for t in th
- Cod com box

## Us

A Java <Ctrl+S brows Show brows

Go To			
Select in			
Find Usages	Alt+F7		
Show Javadoc	Alt+F1		
Refactor			
Reformat Code	Ctrl+Shift+F		
Fix Imports	Alt+Shift+F		
Run File	Shift+F6		
New Watch...	Ctrl+Shift+W		
Toggle Breakpoint	F9		
Cut	Ctrl+X		
Copy	Ctrl+C		
Paste	Ctrl+V		
Code Folds			
		Collapse Fold	Ctrl+NumPad -
		Expand Fold	Ctrl+NumPad +
		Collapse All	Ctrl+Shift+NumPad -
		Expand All	Ctrl+Shift+NumPad +
		Collapse All Javadoc	
		Expand All Javadoc	
		Collapse All Java Code	
		Expand All Java Code	

Figure 4-5 Code Folds

## Handling Imports

There are several ways to manage import statements in Creator. Here are the choices:

- Fast Import (<Alt+Shift+I>) - lets you add an import statement to your code for the currently selected identifier. Figure 4-6, for example, shows an Import Class dialog for the selected identifier, `ListDataProvider`. Although only one import shows up here, this technique lets you choose the import statement you want from a list in the dialog.
- Fix Imports (<Alt+Shift+F>) - lets you insert any missing import statements for the entire file. Figure 4-7 shows the context menu when you right-click in the editor window and move the cursor to the Fix Imports selection.
- Code Completion - you can also generate import statements with code completion. Just type part of the class name with an open code completion box and an import statement will be added to your code automatically.

## Using Javadoc

A Javadoc popup window appears for any selected class when you type <Ctrl+Shift+Space>. Press <Esc> to remove it. Additionally, you may open a web browser for a selected class within the IDE. Just right-click the class and choose Show Javadoc (or <Alt+F1>) from the context menu. You may close the internal browser window by clicking the x in the Javadoc tab.

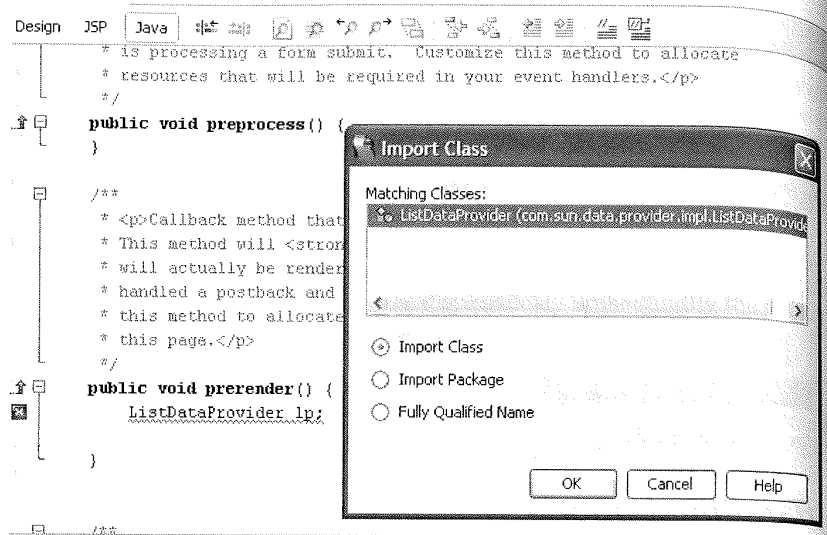


Figure 4-6 Fast Import using <Alt+Shift+I>

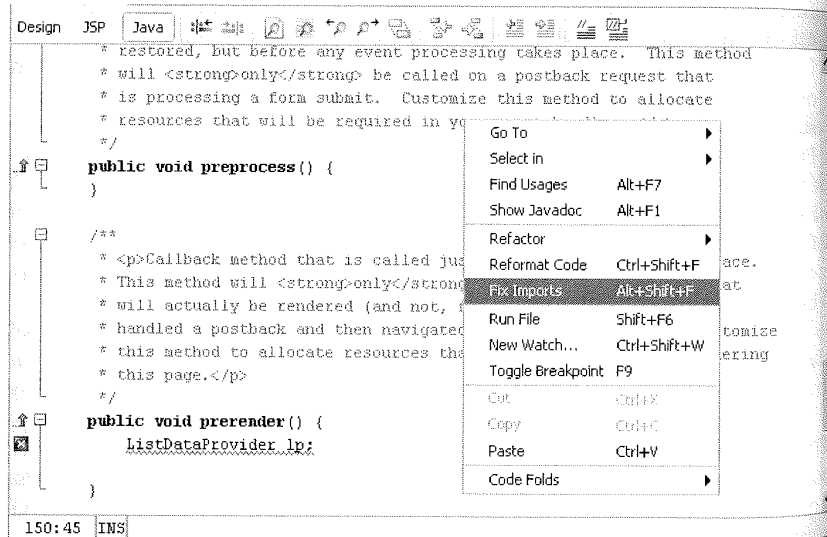


Figure 4-7 Fix Imports using <Alt+Shift+F>

## Abbreviations

The editor has an internal list of abbreviations that generate commonly used keywords, identifiers, and code idioms. Just click on the Abbreviations customizer box in the Basic Options dialog under General (see Figure 4-1). Figure 4-8 shows the default list of abbreviations. Type the abbreviation, press the `<Space>`, and the editor fills in the expanded keywords or expressions for you. If an abbreviation is the same as the text you want to type, press `<Shift+Space>` to keep it from expanding.

The `fora` and `fori` abbreviations are very handy for generating `for` loops and the `trc`, `trcf`, and `trf` abbreviations for `try/catch/finally` can save a lot of typing time. Note that the Abbreviations dialog allows you to edit or remove an abbreviation or add your own.

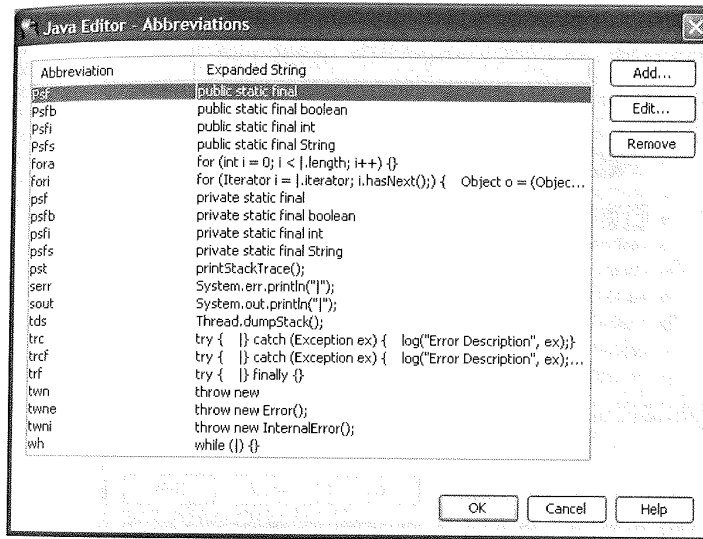


Figure 4-8 Abbreviations Dialog

## Generating Methods

The IDE helps you generate code when extending a class or implementing an interface. Let's show you how to do this now.

## Overriding Methods

When you extend a class, overriding multiple methods and getting everything right can be a tedious process. Creator's IDE has an Override and Implement

Methods dialog to help you generate the code from a list of allowable methods for an extended class. Here are the steps.

1. Define your class and type extends *ClassName*.
2. Select Tools > Override Methods from the toolbar (or press <Ctrl+I>).
3. Select the method(s) you want the IDE to override for your extended class.

Figure 4–9 shows the Override and Implements Methods dialog for a class extended from `ListProvider`. Here we override the `appendRow()` and `canInsertRow()` methods. The Generate Super Calls checkbox makes the IDE include calls to the super implementation of the method. Uncheck this box if you don't want this behavior.

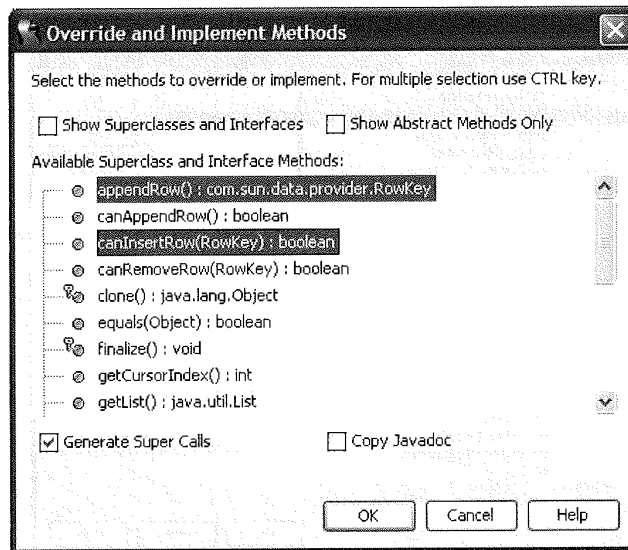


Figure 4–9 Override and Implement Methods Dialog

## Implementing Interfaces

When you create a class that implements an interface, there can be many methods to implement. The IDE's Synchronize feature helps you generate the necessary methods. Here are the steps.

1. Define your class and type implements *InterfaceName*.
2. Select Tools > Synchronize from the toolbar.
3. Select the method(s) you want the IDE to implement.



You can also have the IDE automatically prompt you to generate methods when you create a class that implements an interface. Here are the steps.

1. Select Tools > Options from the toolbar and click the Advanced radio button.
2. Expand the Editing > Java Sources node and select Source Synchronization. Under General in the properties window, select the Synchronization Enabled checkbox.

Figure 4-10 shows a Confirm Changes dialog when creating a class that implements the `DataProvider` interface.

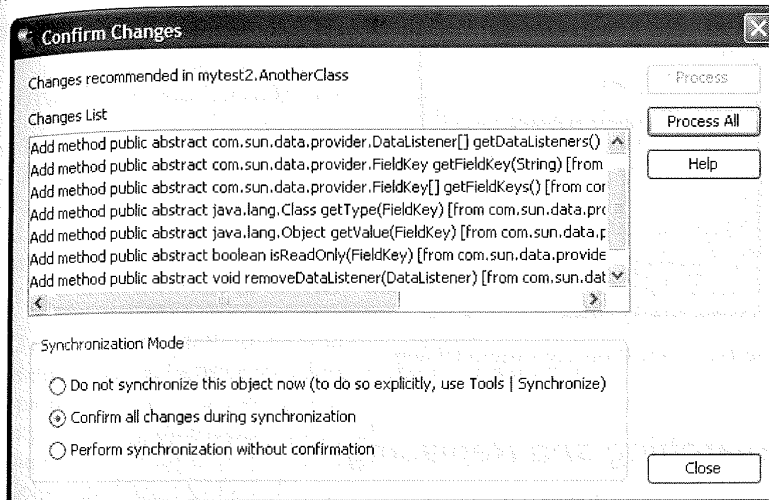


Figure 4-10 Confirm Changes Dialog for Implementing Interfaces

## Generating Properties

With the IDE, it's easy to generate properties that conform to the JavaBeans component model. Here are the steps.

1. In the Projects window, expand your project node.
2. Right-click on a bean pattern node (Session Bean, Application Bean, etc.)
3. Choose Add > Property.
4. In the New Property Pattern dialog, type in the name of your property and select its type (String is the default). Under Mode, select Read/Write (default), Read Only, or Write Only.
5. Choose the options you want for code generation of the property.
6. When you click OK, the IDE will generate a field for the property and the getter and setter methods for the field.

Figure 4-11 shows a New Property Pattern dialog for the status property.

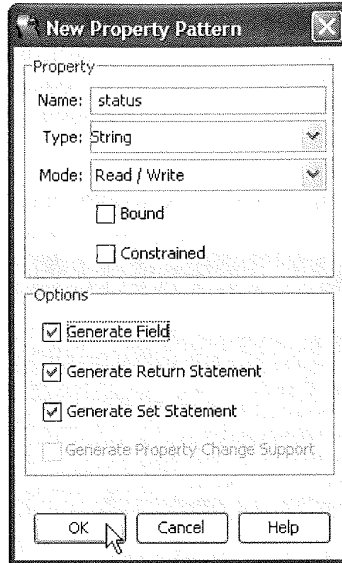


Figure 4-11 New Property Pattern Dialog

## Searching and Replacing

The IDE has several find commands that help you search and replace in your code. These commands work with the current open file or with other project files. Let's show you how to use these different find commands.

### Find Command

Selecting Edit > Find from the toolbar (or typing **<Ctrl+F>**) lets you find specific character combinations in your current open file. You can match case, look for whole words, search backwards, and use regular expressions in your search. After you close a Find dialog, you can move to the next occurrence with **<F3>** or move to the previous occurrence with **<Shift+F3>**.

To search and replace, select Edit > Replace from the toolbar (or type **<Ctrl+H>**) and fill in the fields for Find What and Replace With. Figure 4-12 shows a Find command that searches for `isRowAvailable` in the current open file.

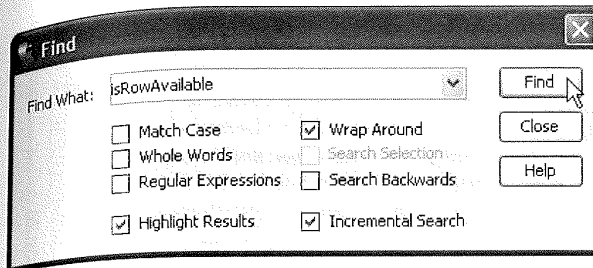


Figure 4-12 Find command

## Find Usages Command

The Find Usages command displays lines in your project according to what you specify. Just select **Edit > Find Usages** from the toolbar (or type **<Alt+F7>**). You may also bring up this command by right-clicking on a class, method, or field name and selecting **Find Usages** from the context menu. The Find Usages command is case-insensitive and doesn't match parts of words, but you can have it look for a variety of different things, such as:

- Class, interface, method, or field declarations
- Method declarations or variables of classes and interfaces
- Specific occurrences, like new instances, imports, extending classes, implementing interfaces, casts, and throwing exceptions
- Methods or fields of a specific type
- Getters and setters of a field
- Method invocation
- Overriding methods
- Comments that refer to an identifier

The results of the Find Usages command appear in a separate Usages window at the bottom of your screen (like the Output window). Figure 4-13 shows an example of a Usages window. Here we show the results of a Find Usages command for the variable `returnValue`. Double-clicking on any of the `returnValue` occurrences takes you to the spot in the file where it's used.

## Find in Projects Command

The Find in Projects command lets you search project files for characters in a file, filename characters, file type, file modification dates, and version control status. Just select **Edit > Find in Projects** (or type **<Ctrl+Shift+F>**) from the toolbar, or right-click a folder in the Files window and select **Find** from the context menu. Figure 4-14 shows the Find in Projects dialog for the text `SessionBean`.

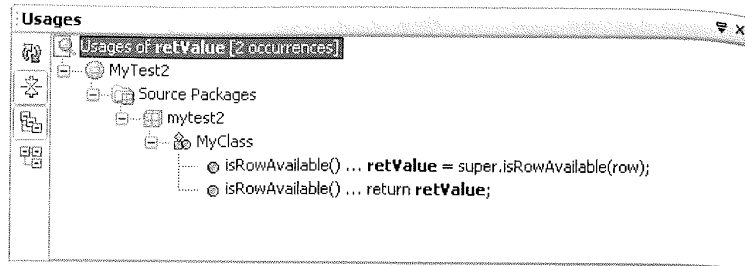


Figure 4-13 Find Usages command

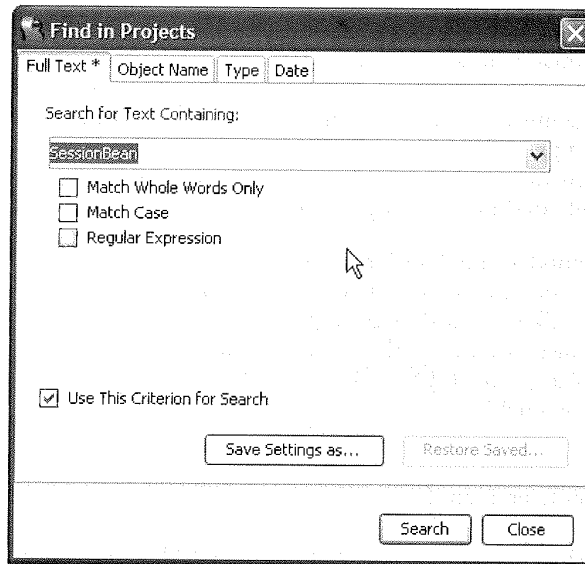


Figure 4-14 Find in Projects command

The results of a Find in Projects command appear in a Search Results window at the bottom of your screen. With full-text searches, you can expand nodes to see which files contain your patterns. Double-clicking on any of the occurrences takes you to the spot in the file where it is used. Figure 4-15 shows the Search Results window for the text `SessionBean`.

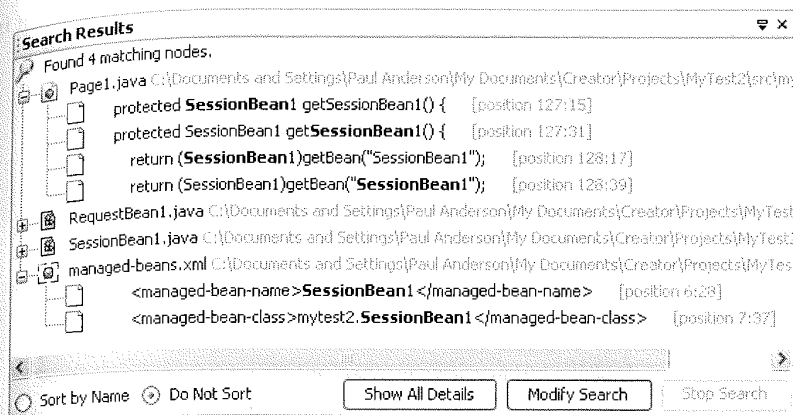


Figure 4-15 Search Results window

## Navigating Files

When you right-click in the editor window and select Go To, a context menu lists ways to navigate from your current file to other places. You may navigate to the super implementation of a class, a specific line number, a declaration, or to the source code for a class, method, or field. Figure 4-16 shows the context menu for Go To and its shortcuts.

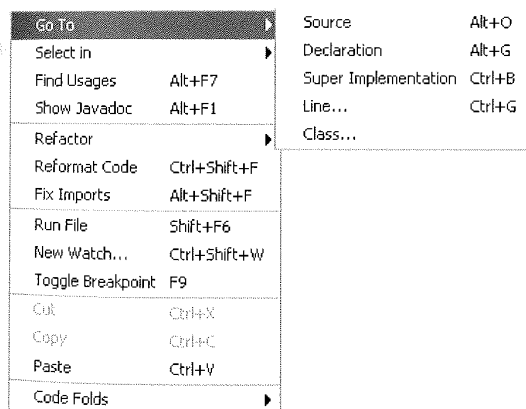


Figure 4-16 Navigation with Go To

If you right-click in the editor window and choose Select in, you can navigate to other project files or to other files in the same package. These options

results win-  
an expand  
any of the  
4-15 shows

are also available from the toolbar by clicking Window > Select Document in. Figure 4-17 shows the context menu for Select In and its shortcuts.

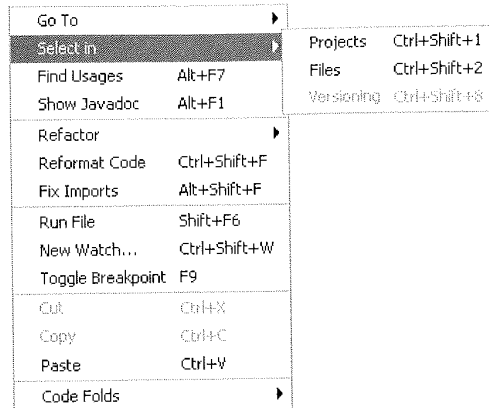


Figure 4-17 Navigation with Select in

## Task Lists

During a hectic software development project, who wants to write notes on post-its to remind themselves to do something important? To help with this, the IDE supports *task lists*. Task lists provide a way to document and clean up any loose ends in your code.

Task lists manage special “tag” words that you mark in your code. These tag words typically appear in comments, such as

```
// TODO: add your event handler here..
// PENDING: Gail will write this code
```

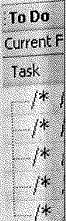
To see what tag words are available for your task list, click Tools > Options from the toolbar, click the Advanced radio button, and select the Editing > To Do Settings node. Click the Task tags customizer box to see the list of tags and their priorities. Figure 4-18 shows the Task Tags dialog.

Note that it’s possible to change or delete the default list of tags, and you can add your own tag to the task list. You can also change a tag’s priority. The available priorities are High, Medium-High, Medium, Medium-Low, and Low. By default, all tags have Medium priority, except the <<<<<< tag, which has High priority.

To view the task list, select View > To Do (or type <Ctrl+6>) from the toolbar. This brings up the To Do window, which appears at the bottom of your screen (like the Output window). Inside the To Do window, you can view tasks for the

Figure 4

current  
window an  
ority. In  
source  
ple of a



Figure

The  
you to  
left of  
To cre  
Name  
or all  
the cc  
Fig  
make  
PENE

Select Document in  
shortcuts.

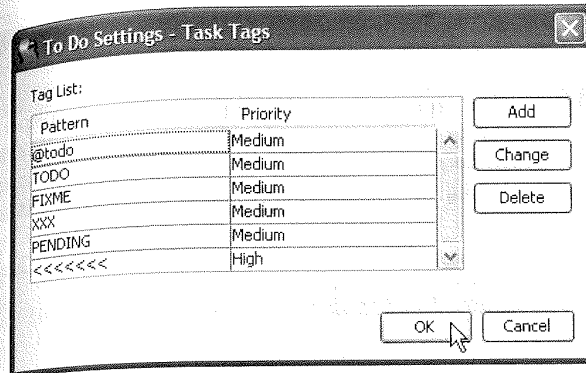


Figure 4-18 Task Tags dialog

current file, all open files, or for a specific folder. If you right-click in the window and select List Options, you can sort the task list by task, location, or priority. If you double-click any tag line in the task list, the editor highlights the source code line in the file where the tag appears. Figure 4-19 shows an example of a task list in the To Do window for opened files.

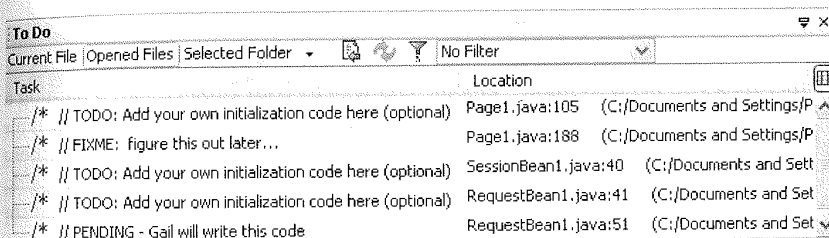


Figure 4-19 To Do window

The To Do window also supports a handy feature called *filters*, which allow you to limit what you see in this window. If you click on the filter icon (to the left of the combo box on the toolbar), the IDE brings up an Edit Filters dialog. To create a new filter, click on the New button and type a filter name in the Name field. Below this, you may specify more than one criteria and match any or all of the criteria. When you're done, your newly defined filter will appear in the combo box on the toolbar of the To Do window.

Figure 4-20 shows you how to create a filter for the PENDING tag. This makes the To Do window display only the PENDING tag lines when you select PENDING in the Combo box.

to write notes on  
To help with this,  
nent and clean up

ir code. These tag

: Tools > Options  
the Editing > To  
e list of tags and

gs, and you can  
riority. The avail-  
w, and Low. By  
which has High

rom the toolbar.  
1 of your screen  
ew tasks for the

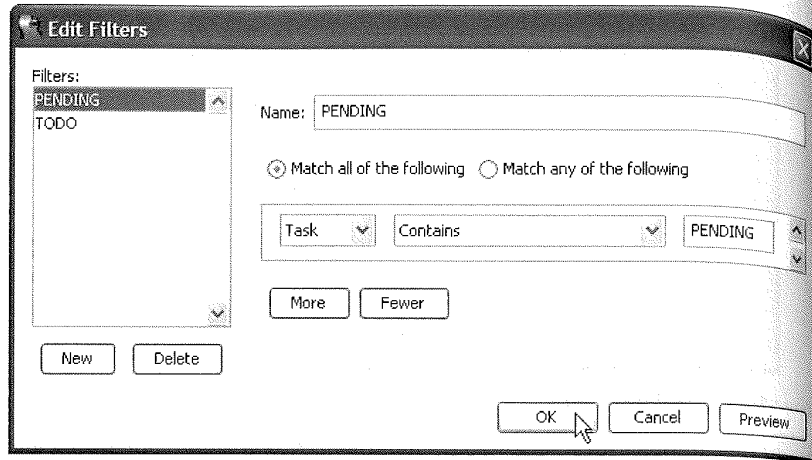


Figure 4-20 Task Edit Filters dialog

## 4.2 Refactoring

Sometimes you need to make “global” changes to your project, like renaming a heavily-used class, field, or method. You might also have to add a new parameter to a method or move a class to a different package. You could do these things manually, but it would be tedious, error-prone, and well, a lot of work. A better approach is to have the IDE help you. Making these kinds of modifications is called *refactoring*. In this section we’ll show you how to use the refactoring features of the IDE. This knowledge can save you a lot of time, especially in large projects with many files.

### ***What is Refactoring?***

Refactoring is transforming and restructuring source code so that the refactored code behaves the same as the original source. In an object-oriented development environment like Java, refactoring must apply to classes, fields, and methods. Some examples of refactoring are relatively simple, like renaming a class, field, or a method. Other types of refactoring are more complicated, like changing the signature of a method or moving a class to a different package.

Here are several reasons why you would refactor your source code.

- You want to add a new feature to your code.



- You need to remove unnecessary repetitions.
- You want to reduce complexity for better understanding.
- You want to make your code more maintainable for others.

Let's explore how Creator's IDE helps you refactor. We'll show you how to use the refactoring features in the IDE and explain how to use them with existing projects. As you will see, the IDE not only lets you preview the changes before you make them, but the IDE also gives you a chance to undo your refactoring changes if you make a mistake.

Here are the refactoring features in the IDE.

- **Find Usages** - determine where classes, fields, and methods are used in your source code.
- **Renaming** - change the name of a class, field, or method. Automatically updates all the references to these elements in your source code.
- **Encapsulating Fields** - generates getter and setter methods for fields. Optionally updates all references to a field using the getters and setters.
- **Change Method Signatures** - add parameters to methods and change the method's visibility.
- **Move Classes** - move a class to another package or inside another class. Automatically updates your source code to reference the class from its new location.

## Refactoring Window

All the refactoring commands make use of a refactoring window, which appears at the bottom of your screen in the IDE (in the same place as the output window). This window is created when you execute a refactoring command. The window provides a preview of files and class elements that are affected by each refactoring command.

Here's what you can do in the Refactoring window.

- Allow or disallow a refactoring change.
- Open the file in the editor for the line(s) to be refactored.
- Refresh the refactoring preview.
- Exit without making any changes.
- Apply the refactoring changes.

We'll show you how to use the refactoring window in the forthcoming examples.

## Payment Project

Let's begin with an existing Creator project called **Payment1**, a monthly payment calculator. (Project **Payment1** is in the download for this book under **FieldGuide2/Examples/JavaBeans/Projects**.) Open Project **Payment1** and deploy it by selecting Run Main Project on the Creator toolbar (or click the Run icon). When the web page comes up in your browser, you will see the payment for a default loan amount, interest rate, and loan term. Try different values for each parameter and click the Calculate button to see the recalculated loan payment.

We'll actually have you build this project from scratch in a later chapter, but let's use it now to demonstrate refactoring.

## Copy Project

This step is optional. If you don't want to copy the project, simply skip this section and make modifications to the **Payment1** project.

1. Bring up project **Payment1** in Creator, if it's not already opened.
2. From the Projects window, right-click node **Payment1** and select Save Project As. Provide the new name **PaymentRF**.
3. Close project **Payment1**. Right-click **PaymentRF** and select Set Main Project. You'll make changes to the **PaymentRF** project.
4. Expand **PaymentRF > Web Pages** and open **Page1.jsp** in the design view.
5. Click anywhere in the background of the design canvas of the **PaymentRF** project. In the Properties window, change the page's **Title** property to **Payment Calculator - Refactor**.

## Find Usages

When it's time to refactor, the first thing you'll want to do is issue a Find Usages command. You already know how to determine where classes, fields, and methods are used in your source code (see "Find Usages Command" on page 147), but let's look at this technique again as it relates to refactoring.

The Payment Calculator project uses a **LoanBean** class to calculate payments from the input parameters supplied on the web page. Let's use the Find Usages command to determine where this **LoanBean** class is used in our source code.

1. Bring up the Projects view, if it is not already opened.
2. Expand the Source Packages > **asg.bean\_examples** node.
3. Right-click **LoanBean.java** and select Find Usages in the context menu.
4. In the Find Usages dialog, click the Search in Comments checkbox, then click the Next button.

- The IDE displays a Usages window. Click the Show Physical View icon in the bottom left margin of the window. The Usages window shows you all the occurrences of the `LoanBean` class in the project (see Figure 4–21).
- Double-click any of the `LoanBean` references in the window. This takes you to the appropriate line in `LoanBean.java` or `SessionBean1.java` where these statements appear.

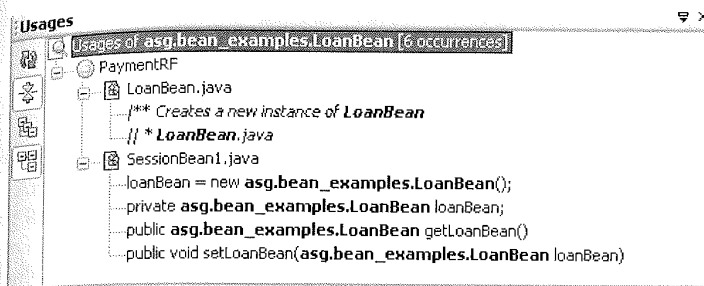


Figure 4–21 Find Usages for `LoanBean`

## Renaming Classes

Let's change the name of the `LoanBean` class in this project. The Find Usages command shows the `LoanBean` object is instantiated in `SessionBean1.java`. It also lists the other places in this file where the `LoanBean` class is referenced.

Here are the refactoring steps to change the name of the `LoanBean` class and all its references in the project.

- Double-click `LoanBean.java` in the Projects view. This brings up this file in the editor window.
- Find the `LoanBean` class declaration in `LoanBean.java` and right-click the `LoanBean` name. Select Refactor > Rename from the context menu.
- In the Rename dialog, type `MyLoanBean` in the New Name field and click the Apply Rename on Comments checkbox.
- Make sure the Preview All Changes checkbox is checked in the Rename dialog.
- Click the Next button.
- The IDE displays a Refactoring window. Click the Show Physical View icon in the bottom left margin of the window. Figure 4–22 shows seven occurrences (including comments) of the `LoanBean` class in the project. The Refactoring window lists all the occurrences of `LoanBean` in two files, `LoanBean.java` and `SessionBean1.java`. The checkboxes next to each refactoring line let you allow (checked) or disallow (unchecked) the refactoring. There are buttons in the left margin of the window to refresh the refactoring data, col-

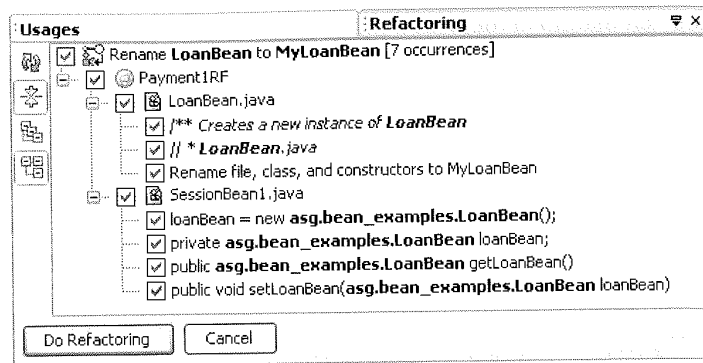


Figure 4-22 Refactoring window

collapse the nodes in the tree, and show the logical and physical views of the refactoring lines.

Let's finish the class name refactoring now.

1. Leave all the checkboxes checked in the Refactoring window.
2. Click the Do Refactoring button.
3. Click the x in the top right corner of the Usages window to remove this window from the display.
4. Verify that the file name **MyLoanBean.java** now appears in the Projects view.
5. Right-click the MyLoanBean class name in **MyLoanBean.java** and select Find Usages in the context menu. In the Find Usages dialog, make sure the Search in Comments checkbox is still checked.
6. Click the Next button. In the Usages window, click the Show Physical View icon in the bottom left margin of the window. You should see the newly applied changes from the refactoring, including MyLoanBean as the new class name (Figure 4-23).
7. Right-click the PaymentIRF project in the Projects window and select Clean and Build Project (this may take a few moments). Verify that your project compiles without errors.

## Undo and Redo

Everything should have worked fine here, but let's show you the Undo and Redo commands anyway. Select Refactor > Undo [Rename] from the Creator toolbar (you can also right-click in the editor and select this from the context menu). You'll see all your refactoring changes restored back to their original values. This can be very valuable when you realize that a refactoring did not do exactly what you wanted.

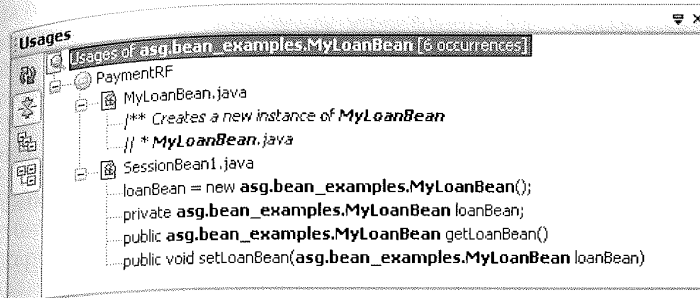


Figure 4-23 Find Usages for MyLoanBean

After an undo command, it's possible to redo refactoring changes by selecting Refactor > Redo [Rename] from the toolbar or from the context menu. Let's leave our changes undone for now and show you another way to refactor classes.

## Refactoring for Renamed Files

Refactoring is also done when you rename class files in the Projects or Files window. This brings up the Refactor Code for Renamed File(s) dialog. Let's show you how to rename your LoanBean class with this technique.

1. Switch from the Projects view to the Files view.
2. Under PaymentRF, open the src > asg > bean\_examples node.
3. Right-click **LoanBean.java** and select Rename.
4. In the Rename dialog, type **MyLoanBean** in the New Name field. Click OK.
5. The Refactor Code for Renamed File(s) dialog appears (see Figure 4-24). Click Next.
6. The IDE displays a Refactoring window. Click the Show Physical View icon in the bottom left margin of the window. The refactoring changes should be the same as what you did before (see Figure 4-22 on page 156).
7. Click Do Refactoring.
8. Verify that the file name **MyLoanBean.java** now appears in the Files view.
9. Right-click the PaymentRF project in the Projects window and select Clean and Build Project (this may take a few moments). Verify that your project compiles without errors.

### Creator Tip

*Make sure you use the IDE to rename class files. If you rename your files with Windows explorer or other file system utilities, Creator won't be able to track your changes.*



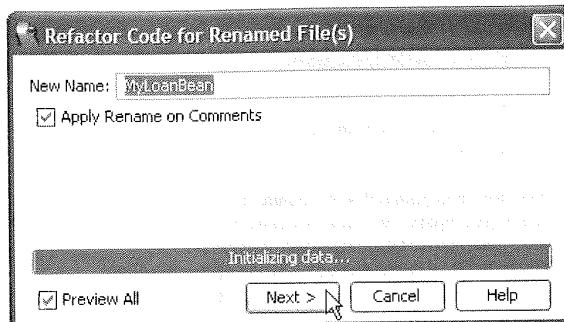


Figure 4-24 Refactor code for renamed file

## Renaming Fields and Methods

Renaming a class field or method is done the same way as renaming a class. Here are the steps.

1. Find the field or method you want to rename in the editor. In the Projects view, expand the source file nodes until you find the field or method you want.
2. Right-click the field or method and select Refactor > Rename from the context menu.
3. In the Rename dialog, type the New Name for the field or method.
4. Click the Next button.
5. In the Refactoring window, review the lines of code that will be refactored. Clear any checkboxes for code that you do not want changed.
6. Click the Do Refactoring button to make the changes.
7. Right-click on your project name in the Projects view and select Clean and Build Project. Verify that your project compiles without errors.

### Creator Tip



*Be careful with refactoring fields and methods of JavaBeans components. With JavaBeans, the setters and getters use naming conventions which could be disrupted by an improper refactoring. Refactoring JavaBeans could also adversely affect bindings and other assumptions made by the IDE.*

## Encapsulating Fields

Refactoring lets you encapsulate fields, which insures that class fields can only be accessed by getter and setter methods. This type of encapsulation enforces

data hiding and improves maintainability. Typically, a class field's visibility is restricted to private, whereas the getter and setters for the field are marked as public. Other visibility choices are possible (protected with inheritance access, for instance).

The IDE supports the following refactoring features for Encapsulating Fields.

- Generate getter and setter methods for fields.
- Modify the visibility modifier for the fields and the getters and setters.
- Replace references to field names with calls to the getters or setters.

Let's modify our Payment Calculator project and show you how to encapsulate a field and generate setters and getters for the field. Here are the steps.

1. Open **MyLoanBean.java** in the editor if it is not already open.
2. Add the following field declaration to your code, right below the MyLoanBean constructor.

```
private String version = "Version 1.0";
```

3. Select the `version` field. Choose Refactor > Encapsulate Fields from the toolbar (or right-click the `version` field and select this option from the context menu).
4. In the Encapsulate Fields dialog, make sure the `version` field's checkbox is checked. Select `protected` in the combo boxes for both the Fields' Visibility and the Accessors' Visibility (see Figure 4-25). Click Next.
5. The IDE displays a Refactoring window. Click the Show Physical View icon in the bottom left margin of the window. The Refactoring window shows the changes that will be made to encapsulate the `version` field (Figure 4-26).

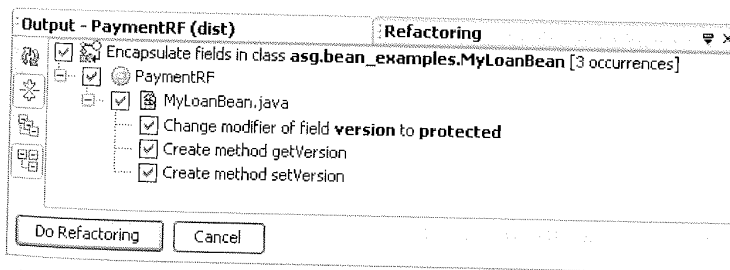


Figure 4-26 Refactoring window for Encapsulate Fields

6. Click Do Refactoring. Verify that the code for setter method `setVersion()` and getter method `getVersion()` now appear in **MyLoanBean.java** with protected visibility. The `version` field should be protected as well.

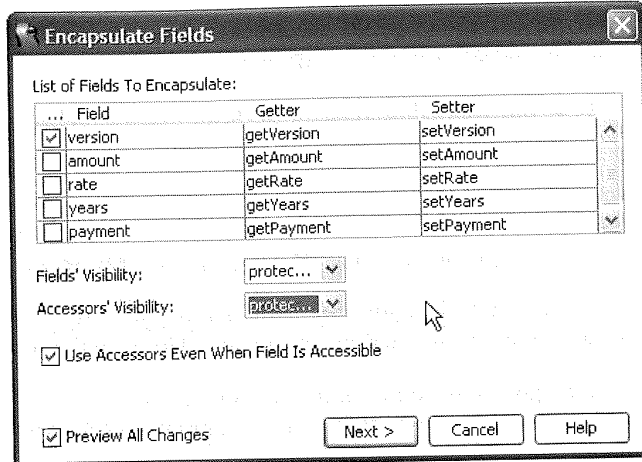


Figure 4-25 Encapsulate Fields dialog

7. Right-click your project name in the Projects view and select Clean and Build Project (this may take a few moments). Verify that your project compiles without errors.

## Changing Method Signatures

The design of class methods is crucial to the behaviors of object-oriented designs and reusable classes. During the early stages of development, it's easy to develop methods with several parameters and change them when you need to. But near the end of a large development cycle, changing the signature of a heavily used class method can be a time-sink, because the change often propagates to a large number of invocations in source code. Refactoring can be a big help here, because the IDE can update all the method calls for you.

The IDE supports the following refactoring features for Changing Method Signatures.

- Add parameters to a method's signature.
- Reorder the parameters in a method's signature.
- Change the visibility for a method.

### Creator Tip



*Refactoring does not allow you to remove a parameter from a method's signature. You can't refactor a method's return type, either. If you need to do these things in your project, you'll have to do it manually.*



## Generate New Method

Before we show you how to refactor a method's signature, let's add a new method to the `MyLoanBean` class and call it from the `Payment Calculator`. Here are the steps.

1. Open `MyLoanBean.java` in the editor if it is not already open.
2. Add the following method to your code, right below the `setVersion()` and `getVersion()` methods.

```
public String getInfo() { return version; }
```

3. Return to `Page1.java` in the editor and click the `Design` button to bring up the design view.
4. Double-click the `Calculate` button. This generates a `calculate_action()` method in `Page1.java`. Add the following code before the return statement (new code is in **bold**).

```
public String calculate_action() {
    // TODO: Process the button click action...
    log(getSessionBean1().getLoanBean().getInfo());
    return null;
}
```

5. Click the `Run` icon on the toolbar to deploy the `Payment Calculator` application. Type input values on the page and click the `Calculate` button.
6. In the `Servers` window, right-click `Deployment Server` and select `View Server Log`.
7. In the `Output` window, you should see the string "Version 1.0" appear in the server log.

## Add Method Parameter

Now let's add a new parameter to our `getInfo()` method and refactor it in our project. Here are the steps.

1. Select the `getInfo()` method in the editor. Select `Refactor > Change Method Parameters` from the toolbar (or right-click the `getInfo()` method and select this option from the context menu).
2. In the `Change Method Parameters` dialog, type **who** for Name, **String** for Type, and "paul" for Default Value. To edit these, you'll need to double-click each cell.
3. Leave the Access Modifier `public` and make sure the `Preview All Changes` checkbox is checked (see Figure 4-27). Click `Next`.

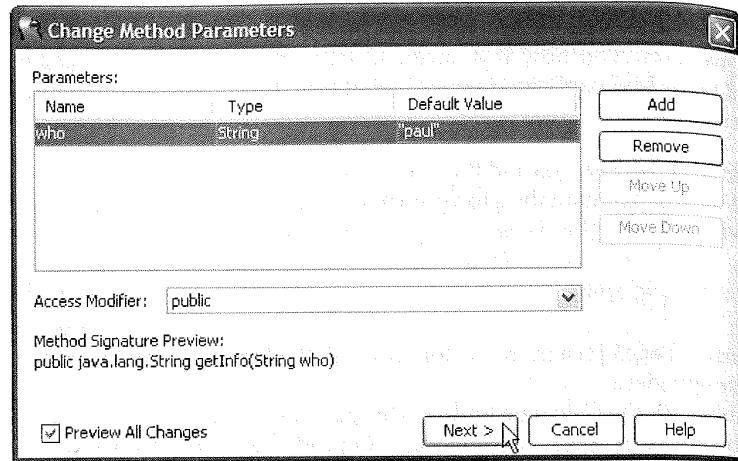


Figure 4-27 Change Method Parameters dialog

4. The IDE displays a Refactoring window. Click the Show Physical View icon in the bottom left margin of the window. The Refactoring window shows the changes that will be made to refactor the `getInfo()` method (see Figure 4-28).

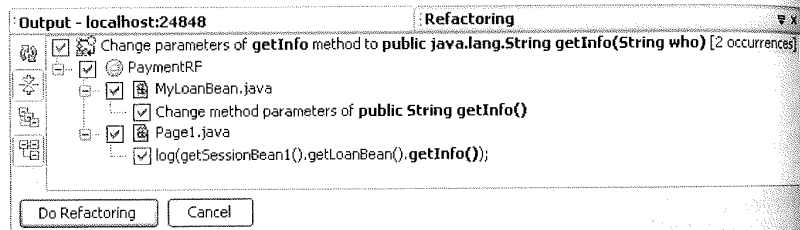


Figure 4-28 Refactoring window for Changing a Method's signature

5. Click Do Refactoring. Verify that a new parameter was added to the `getInfo()` method in `MyLoanBean.java`. Modify this code as follows (new code is **bold**).

```
public String getInfo(String who) {
    return version + "-" + who;
}
```

6. Verify that the call to `getInfo()` in `Page1.java` was modified, too. Here's what it should look like (new code is **bold**).

```
public String calculate_action() {
    // TODO: Process the button click action...
    log(getSessionBean().getLoanBean().getInfo("paul"));
    return null;
}
```

Now deploy, run, and test the project.

1. Right-click your project name in the Projects view and select Clean and Build Project (this may take a few moments). Verify that your project compiles without errors.
2. Click the Run icon on the toolbar to deploy the Payment Calculator application. Type input values on the page and click the Calculate button.
3. In the Servers window, right-click Deployment Server and select View Server Log.
4. In the Output window, you should see the string "Version 1.0-paul" appear in the server log.

## Reordering Parameters

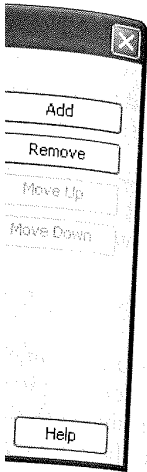
Reordering parameters in a method's signature is done the same way as adding a parameter. Here are the steps.

1. Select the method you want in the editor. Select Refactor > Change Method Parameters from the toolbar (or right-click the method and select this option from the context menu).
2. Select the parameter you want to move and click Move Up or Move Down. This changes its position in the list. Click Next.
3. In the Refactoring window, review the lines of code that will be refactored. Clear any checkboxes for code that you do not want changed.
4. Click Do Refactoring to make the changes.
5. Right-click your project name in the Projects view and select Clean and Build Project. Verify that your project compiles without errors.

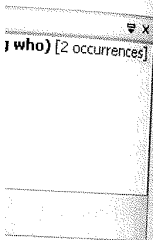
## Changing a Method's Visibility

The refactoring commands for a method's visibility are very similar to the others. Here are the steps.

1. Select the method you want in the editor. Select Refactor > Change Method Parameters from the toolbar (or right-click the method and select this option from the context menu).



ysical View icon  
indow shows the  
d (see Figure 4-



ed to the get-  
llows (new

2. Select an Access Modifier for the method's visibility from the combo box (options are public, protected, private, or default). Click Next.
3. In the Refactoring window, review the lines of code that will be refactored. Clear any checkboxes for code that you do not want changed.
4. Click the Do Refactoring button to make the changes.
5. Right-click your project name in the Projects view and select Clean and Build Project. Verify that your project compiles without errors.

#### Creator Tip



Note that the code for the `getInfo()` method refers to the `version` field directly. By refactoring, you can make this method use the getter method for the field instead. To do this, select the `version` field and choose Refactor > Encapsulate Fields. Make sure the checkbox is checked for Use Accessors Even When Field is Accessible. Click Do Refactoring. You will see code in `getInfo()` that now calls `getVersion()` to get the `version` field's value.

## Moving Classes to Different Packages

Another important refactoring feature is moving a class from one package to another. This kind of code change can certainly be a hassle to do manually, so refactoring is a big help here.

There are two approaches for moving a class between packages, so let's show you how to do both. Here are the steps for the first approach.

### Moving Classes

1. Bring up the Projects view, if it is not already opened.
2. Under `PaymentRF`, expand the `Source Packages > asg.bean_examples` node. Note that `MyLoanBean.java` is contained in this package (Figure 4-29).
3. Right-click `MyLoanBean.java` and select Refactor > Move Class from the context menu.
4. In the Move Class dialog, select `payment1` from the combo box for To Package (see Figure 4-30).
5. Make sure the Preview All Changes checkbox is checked. Click Next.
6. The IDE displays a Refactoring window. Click the Show Physical View icon in the bottom left margin of the window. The Refactoring window shows the refactoring statements that move the `MyLoanBean` class to the `payment1` package (Figure 4-31).
7. Click Do Refactoring to make the changes. Verify that `MyLoanBean.java` has been moved to the `payment1` package in the Projects View (see Figure 4-32).

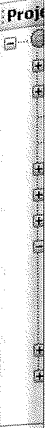


Figure 4-29



Figure 4-30

8. Right-click Build Project.
9. Click the verification button.

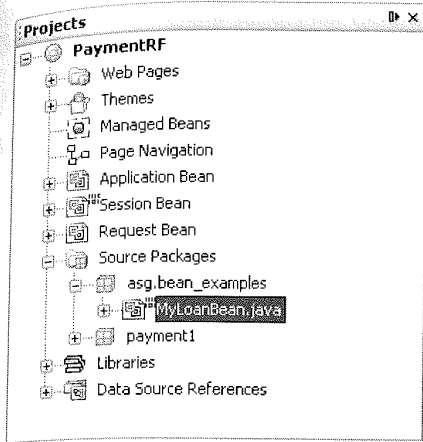


Figure 4-29 Projects window before refactoring

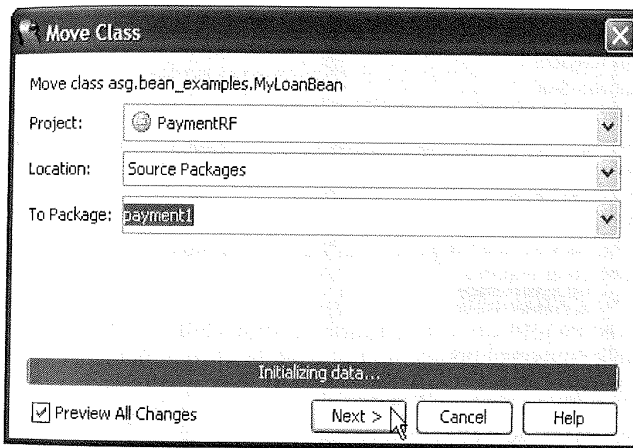


Figure 4-30 Move Class dialog

8. Right-click your project name in the Projects view and select Clean and Build Project (this may take a few moments). Verify that your project compiles without errors.
9. Click the Run icon on the toolbar to deploy the Payment Calculator application. Verify that everything works.

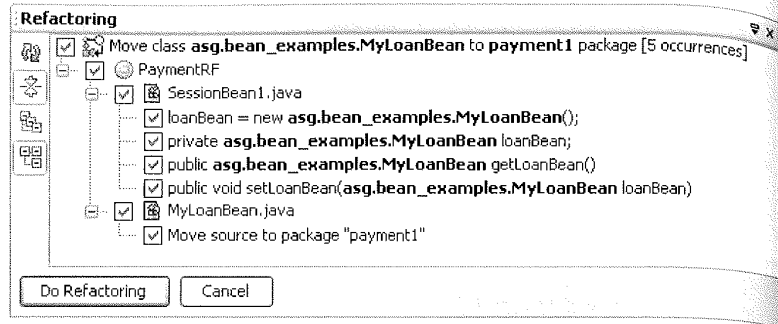


Figure 4-31 Refactoring window for Move Class

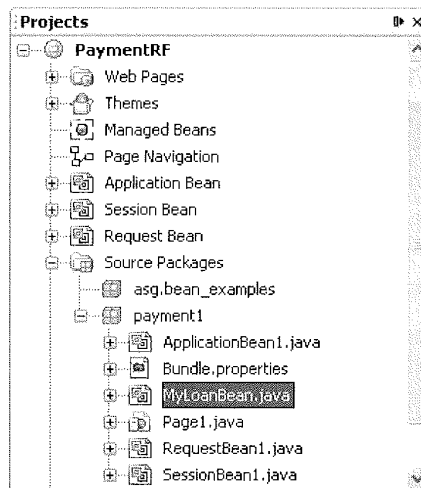


Figure 4-32 Projects window after refactoring

## Refactoring for Moved Files

The IDE has other ways to move class files between packages. The Refactor Code for Moved Class dialog opens whenever you perform the following actions:

- Cut and paste files in the Projects or Files window.
- Drag and drop files in the Projects or Files window.

- Type a new package name in the Projects window for a package node.
- Type a new folder name in the Files window for a folder node.

Let's show you how to use this technique. Here are the steps.

1. Select Refactor > Undo [Move class] to put the MyLoanBean class back in the original package.
2. In the Projects view under PaymentRF, open the Source Packages > asg.bean\_examples node.
3. Right-click **LoanBean.java** and select Cut.
4. Right-click package **payment1** and select Paste.
5. The Refactor Code for Moved Class dialog appears (Figure 4-33). Click

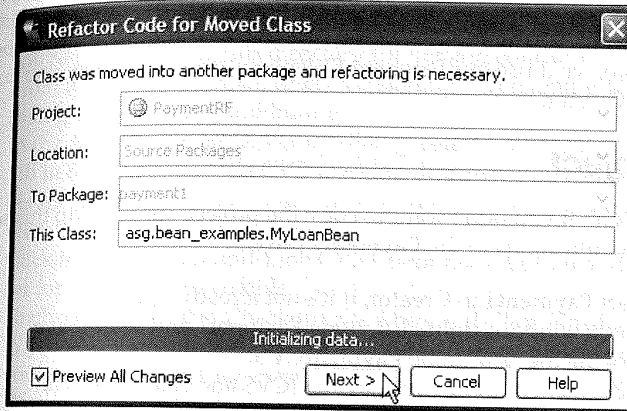


Figure 4-33 Refactor Code for Moved Class dialog

Next.

6. The IDE displays a Refactoring window. Click the Show Physical View icon in the bottom left margin of the window. The refactoring changes should be the same as what you did before (see Figure 4-31 on page 166).
7. Click Do Refactoring.
8. Verify that **MyLoanBean.java** has been moved to the **payment1** package in the Projects View (see Figure 4-32 on page 166).
9. Right-click the PaymentRF project in the Projects window and select Clean and Build Project (this may take a few moments). Verify that your project compiles without errors.

. The Refactor  
the following

## 4.3 Source Code Control with CVS

Version control allows developers to track the changes they make to their source code. With version control, you can determine when a change was made and by whom. You can also use version control to track bugs and generate specific builds that might customize certain parts of your system. All this works for a single developer working on a project as well as a group working on the same project code.

Creator supports several Version Control Systems (VCS), but we'll show you CVS (Concurrent Versioning System) which is very popular with developers. You'll learn how to create CVS working directories and repositories, import source code into CVS, and check out modules. You'll also see how to commit editing changes to CVS, compare revisions, and examine log histories of code changes. As before, we'll use our Payment Calculator project to show you how to use CVS with Creator.

### Copy Project

This step is optional. If you don't want to copy the project, simply skip this section and make modifications to the **Payment1** project.

1. Bring up project **Payment1** in Creator, if it's not already opened.
2. From the Projects window, right-click node **Payment1** and select **Save Project As**. Provide the new name **PaymentCVS**.
3. Close project **Payment1**. Right-click **PaymentCVS** and select **Set Main Project**. You'll make changes to the **PaymentCVS** project.
4. Expand **PaymentCVS > Web Pages** and open **Page1.jsp** in the design view.
5. Click anywhere in the background of the design canvas of the **PaymentCVS** project. In the Properties window, change the page's **Title** property to **Payment Calculator - CVS**.

### Setting up CVS

The IDE provides two ways to work with CVS in Creator. You can use the built-in CVS client in Creator (written in Java) that helps you connect to CVS repositories on remote machines. Or, you can install CVS locally and have the IDE work with CVS directly on your machine. This is the approach we'll show you here.

### Installing CVS

If you don't have CVS installed on your system already, it's fairly easy to find an open source version on the web for CVS. Download the appropriate version



for your machine and install it. Make sure you can run the `cv`s command from a command prompt window.

## Create CVS Profile

In CVS, you will need to setup two directories: a *repository* directory, which stores a project's full revision history, and a *working* directory to store the files in your project. Here are the steps.

1. Outside the IDE, create a directory or folder for your CVS repository. Make sure this is a safe place where accidental deletions are unlikely (a `temp` directory, for instance, would be a poor choice).
2. Outside the IDE, create a directory or folder for your CVS working directory. (You can skip this step if you already have a directory with source files and you're willing to use this directory for version control.)
3. From the Creator toolbar, select Versioning > Versioning Manager. In the dialog box, click the Add button.
4. In the Add Versioned Directory dialog, select CVS for the System Profile.
5. Fill in the location of the CVS working directory you created in Step 2.
6. Set the Repository Path to the name of the CVS repository you created in Step 1.
7. Click the Use Command-Line CVS Client radio button and make sure `cv`s is set for your CVS executable.
8. Uncheck the Perform Checkout checkbox and Click Finish (see Figure 4-34).
9. When you return to the Add Versioned Directory dialog, you should see your working directory appear. Click Close to exit this dialog.

## Initialize CVS Repository

Now that the CVS repository directory has been created, you need to initialize it. Here are the steps.

1. From the Creator toolbar, select Versioning > CVS > Init Local Repository. This brings up the CVS Init dialog.
2. Select your Repository Path in this dialog, click the Set As Default button to remember these values, then click OK (see Figure 4-35).

## Importing Files

Now that you've setup and initialized your CVS directories, the next step is to import source files into the CVS repository. This is very straightforward, the only thing you have to watch are your own binary file types, like images and jar files.

**Add Versioned Directory**

**Steps**

1. Choose Template
2. Profile

**Add Profile**

Version Control System Profile: CVS

Working Directory: C:\Documents and Settings\CVS

CVS Server Type: local

CVS Server Name:

Port: 2401

User Name:

Repository Path: C:\Documents and Settings\CVSRepository

Use Built-In CVS Client

Use Command-Line CVS Client

CVS Executable: cvs

Remote Shell:

Log In or Set Offline Mode

Login to Server at

Password:

You are not logged in.

Set Offline Mode

Perform Checkout

To get additional profiles, visit: <http://vcsgeneric.netbeans.org/profiles/index.html>

< Back   Next >   Finish   Cancel

Figure 4-34 CVS Profile Dialog

**Creator Tip**

Version control works by storing your changes as textual diff statements. You don't want to import your own binary files (images, jar files, etc.) into the repository, because textual diffs won't work. The easiest thing to do is to remove binary files from your project directory before you import. After you checkout the files into your working directory, you can use the `CVS > Add` command to put the binary files back in your project. (See "Creator Tip" on page 180.)

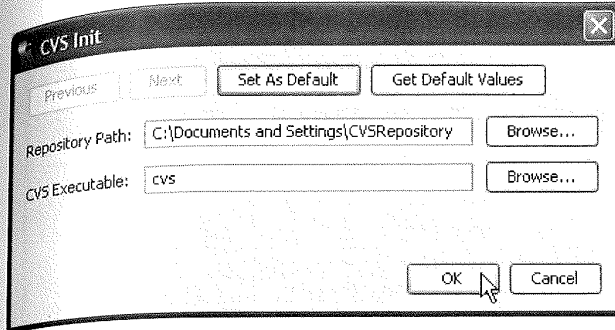


Figure 4-35 Initialize CVS Repository

Here are the steps to import the Payment source files into the CVS repository.

1. From the Creator toolbar, select Versioning > CVS > Import. This brings up the CVS Import dialog.
2. In this dialog, set Directory to Import to the location of your source files to import.
3. Choose **local** for the CVS Server Type.
4. Set the Repository Path to your CVS repository directory.
5. Type **PaymentCVS** for your Repository directory.
6. Click the Use Command-Line CVS Client radio button and make sure **cvs** is set for your CVS executable.
7. Fill in Logging Message, Vendor Tag, and Release Tag as shown, and uncheck the Perform Checkout After Import checkbox.
8. Click the Set As Default button to store these values.
9. Click OK (see Figure 4-36). In the VCS Output window under Standard Output, you should see a list of imported files in your repository.

## Checking Out Files

Now that the files are residing in the CVS repository, you must check them out into your CVS working directory. This directory is where you will make your changes under version control. Here are the steps.

1. From the Creator toolbar, select Versioning > CVS > Check Out. This brings up the CVS Checkout dialog.
2. In this dialog, set the Working Directory to the name of your CVS working directory.
3. Choose **local** for the CVS Server Type.
4. Set the Repository Path to your CVS repository directory.

ements. You  
) into the  
do is to  
. After you  
VS > Add  
or Tip" on

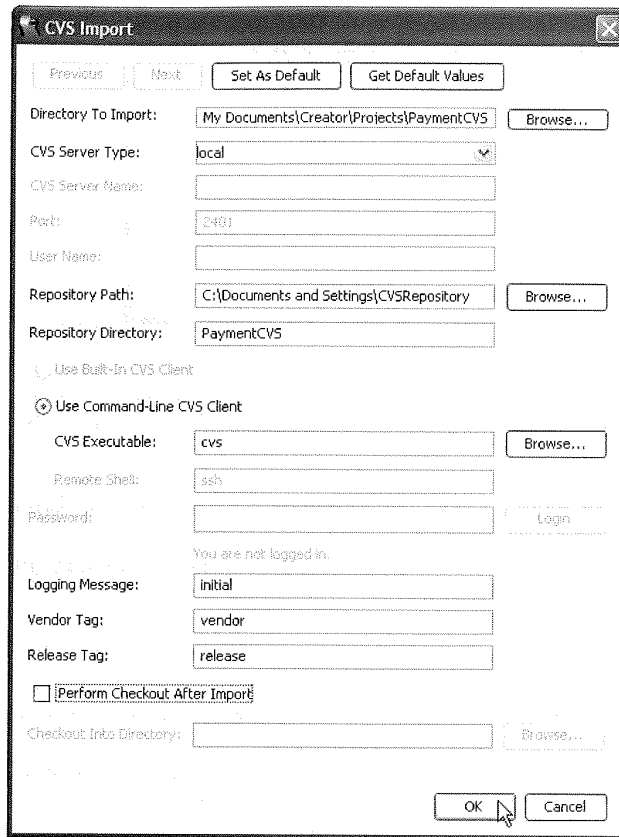


Figure 4-36 CVS Import Dialog

5. Click the Use Command-Line CVS Client radio button and make sure `cvs` is set for your CVS executable.
6. Click the Module(s) radio button and type in **PaymentCVS**.
7. Click OK (see Figure 4-37). In the VCS Output window under Standard Output, you should see a list of checked-out files in your working directory.

### Updating Source Files

In this section we'll show you how to access files in your working directory, edit their contents, then update them under CVS version control.

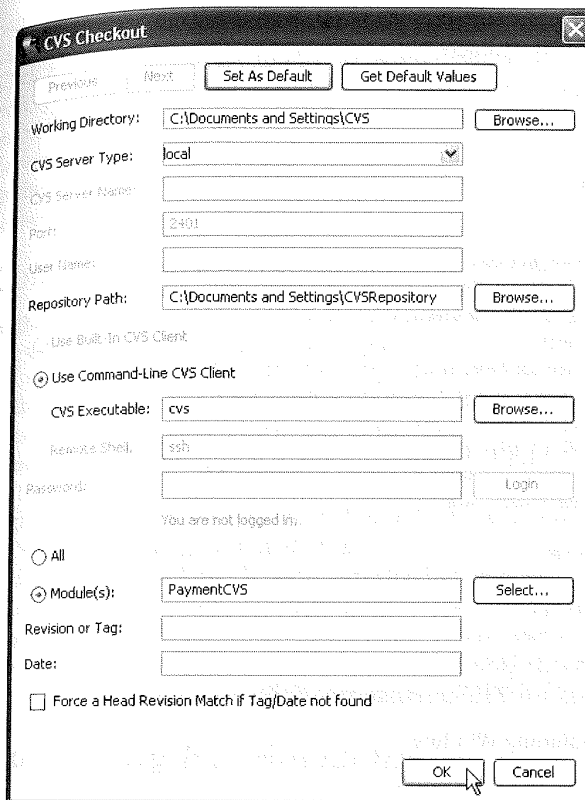


Figure 4-37 CVS Checkout Dialog

## Versioning Window

Before you make a change to your code, let's open the Versioning window and see what the PaymentCVS project looks like. Here are the steps.

1. From the Creator toolbar, Select View > Versioning > Versioning (or type <Ctrl+S>).
2. The Versioning window appears in the top left portion of the screen. Expand the source nodes for src > asg > bean\_examples > LoanBean.java, src > payment1 > Page1.java, and web > Page1.jsp (see Figure 4-38).

Note that all **.java** and **.jsp** files are listed as Up-to-date with 1.1.1.1 being the Initial revision.

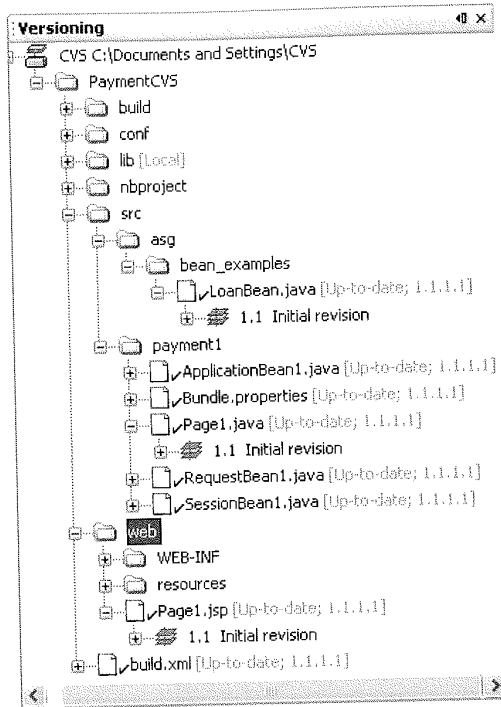


Figure 4-38 CVS Versioning Window

## Editing Source Files

Let's modify our project code and test it. Here are the steps.

1. Bring up **Page1.jsp** in the Design view. On the page for the Payment Calculator, double click the Calculate button. This generates a `calculate_action()` button handler code in **Page1.java**.
2. Let's write to the log file for a button push. Add the following code before the return statement (new code is in **bold**).

```
public String calculate_action() {
    // TODO: Process the button click action...
    log("Version 1.1");
    return null;
}
```

3. Save your changes. Note that **Page1.java** and **Page1.jsp** are marked as Locally Modified in the Versioning window.
4. Click the Run icon on the toolbar to deploy the Payment Calculator application. Type input values on the page and click the Calculate button.
5. In the Servers window, right-click Deployment Server and select View Server Log.
6. In the Output window, you should see the string "Version 1.1" appear in the server log.

## Committing Source Files

Now that you've tested the code to make sure it works, let's store your changes under version control. Here are the steps.

1. In the Versioning window, right-click **Page1.java** and select CVS > Update. This ensures that your local copies of the files are up-to-date. You should see a successful update appear in the VCS Output window under the Standard Output tab.
2. Right-click **Page1.java** again and choose CVS > Commit. This brings up a CVS Commit dialog for **Page1.java**.
3. In the window for Enter Reason, type **log button push**.
4. Follow these same steps for **Page1.jsp**. Provide the same input **log button push** for Enter Reason when you commit.
5. The Versioning window should now mark the **Page1.java** and **Page1.jsp** files Up-to-date as revision 1.2 (see Figure 4-39).

## Comparing File Revisions

During a hectic software development cycle, it's often necessary to compare file revisions and track down what happens as code evolves. This helps everyone understand what changes were made, by whom, and when.

At this point, you have revision 1.1 (initial) and revision 1.2 (log button push) for both the **Page1.java** and **Page1.jsp** files. Let's look at **Page1.java** and show you how to see the changes you just made to the project for that file. Here are the steps.

1. In the Versioning window, right-click on the 1.1 Initial revision under **Page1.java** and select Diff Graphical.
2. A **Page1.java [VCS Diff]** graphical visualizer window appears, showing you the changes between revision 1.1 on the left and your working file (revision 1.2) on the right (see Figure 4-40).

The Graphical Visualizer gives you a side-by-side view of the changes in the main window. You'll see changed lines highlighted in blue, lines added since an earlier revision highlighted in green, and lines removed highlighted in red.



Figure 4-39 Commit CVS files

Try the Graphical Visualizer with `Page1.jsp` to see the differences between its two revisions. You can also **<Shift+Click>** revision nodes in the Versioning window and right-click either of the nodes to select Diff Graphical.

#### Creator Tip



*The Visualizer also lets you see the differences between revisions as text differences. To see your changes in this format, click the Visualizer combo box and select Textual Diff Viewer.*

### Viewing History

During software development, it's important to track the changes that have been made to a module and by whom. This section shows you how to use the CVS History command to get this information.



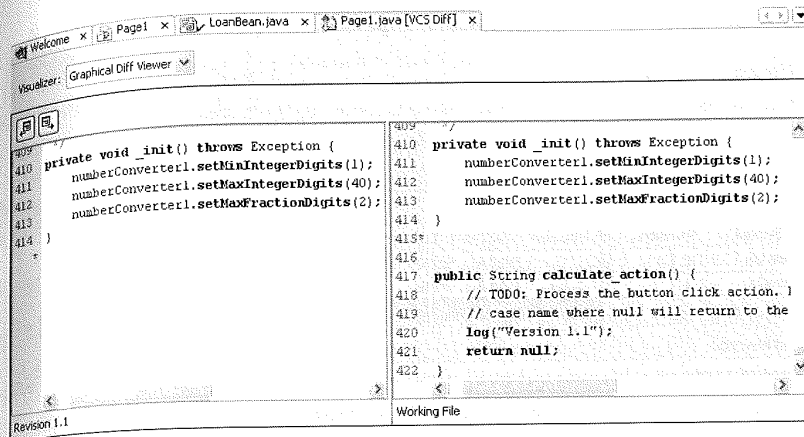


Figure 4-40 CVS Graphical Visualizer

## History Log

The CVS History Log command gives you a full list of a file's revisions, tags, and commit history. Let's show you what this looks like for the changes we made to **Page1.java**. Here are the steps.

1. Right-click **Page1.java** in the Versioning window.
2. Select CVS > History > Log.
3. A history log should appear in the IDE main window displaying the complete revision history of **Page1.java** (see Figure 4-41).

## History Annotation

Another useful CVS history command is annotation. The CVS History Annotation command displays information about each line in source file, including when a line was changed and by whom. Let's try this out with our **Page1.java** file. Here are the steps.

1. Right-click **Page1.java** in the Versioning window.
2. Select CVS > History > Annotate.
3. A history annotation should appear in the IDE main window displaying the revision history of **Page1.java** line-by-line. In Figure 4-42, we show you which lines in the file were introduced for revision 1.2 and by whom.

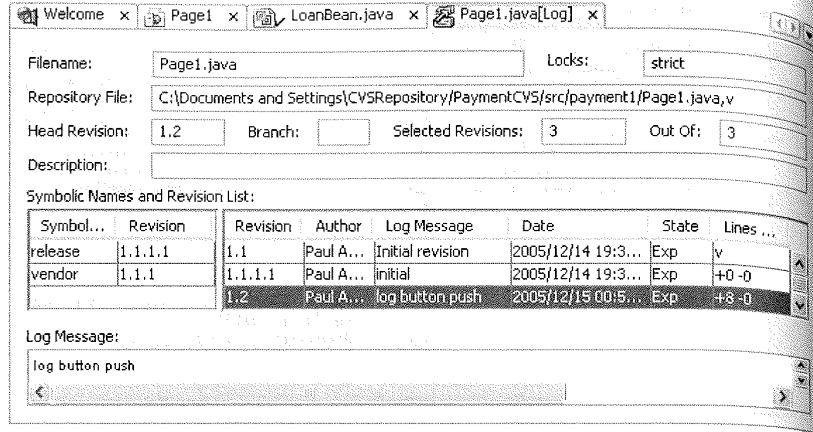


Figure 4-41 CVS History Log

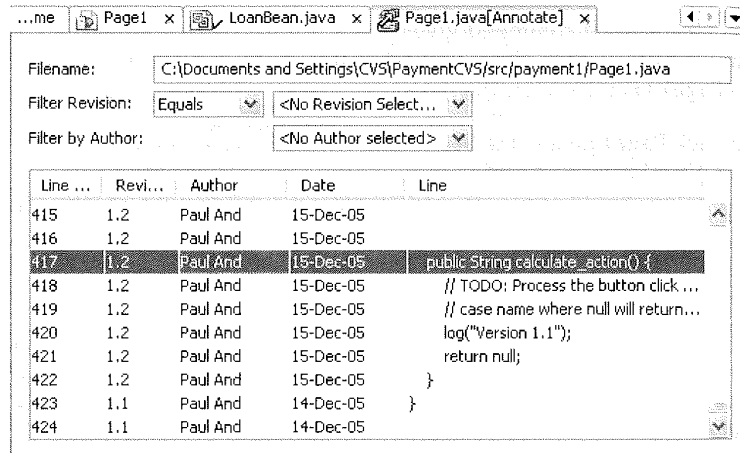


Figure 4-42 CVS History Annotation

## Adding and Removing Files

After you import your source files and check them out, CVS allows you to add and remove source files from your CVS working directory. This section shows you how to use these commands.

## Add Command

The CVS Add command lets you schedule a new file to be added to your working directory. The CVS status of a file must be set to Local, or the CVS Add command is not available for that file.

Let's create a new java file in our working directory and add it to the repository. We'll also commit this file and put it under version control. Here are the steps.

1. In the Files window, right-click `src > asg > bean_examples` and select `New > Java Class`.
2. Type **DemoBean** for the New Class Name in the New Java Class dialog. Click the Finish button.
3. In the Versioning window, the **DemoBean.java** file should appear under the `bean_examples` node, marked as Local.
4. Right-click **DemoBean.java** and select `CVS > Add`. This brings up the CVS Add dialog.
5. Type **Demo Bean** for the File Description and click the Textual radio button.
6. Check the Proceed with Commit If Add Succeeds checkbox. Click OK (see Figure 4-43).

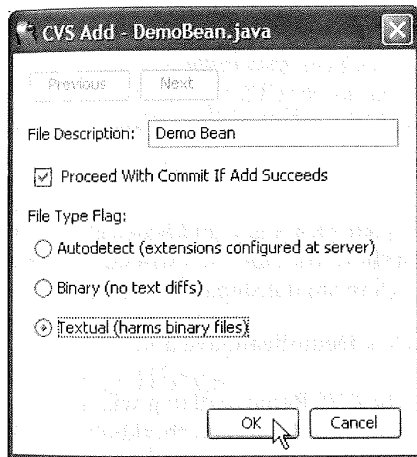


Figure 4-43 CVS Add Dialog

7. In the VCS Output window, an Update tab will open and display status. Since you clicked the commit radio button in the CVS Add dialog, CVS will commit the file after the Add succeeds. In the CVS Commit dialog, type **Initial revision** in the window for Enter Reason.

8. The Versioning window should now reflect the CVS status change of **DemoBean.java** to Up-to-date with revision 1.1 (see Figure 4-44).

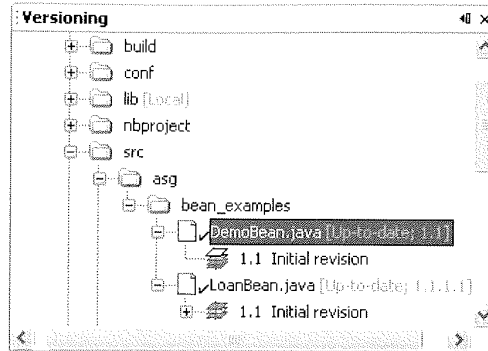


Figure 4-44 CVS Add New File



#### Creator Tip

Check the **Textual** radio button for text files and the **Binary** radio button for binary files. Use the **CVS > Add** command to restore binary files (images and jar files) that you removed during a CVS import. (See “Creator Tip” on page 170.) If you do not check the commit checkbox in the Add dialog, your file will not be added to CVS until you run the **CVS > Commit** command.

## Remove Command

The CVS Remove command deletes your local copy and schedules the file for removal from the CVS repository. To show you how this works, let’s delete the **DemoBean.java** file you just added. Here are the steps.

1. In the Versioning window, right-click **DemoBean.java** and select **CVS > Remove**.
2. Click **Yes** in the Question dialog. The CVS Remove dialog will appear.
3. Check the **Proceed with Commit If Remove Succeeds** checkbox. Click **OK** (see Figure 4-45).
4. In the VCS Output window, an **Update** tab will open and display status. Since you clicked the commit radio button in the CVS Remove dialog, CVS will commit the file after the Remove succeeds. In the CVS Commit dialog, type **Not necessary** in the window for Enter Reason.
5. In the Versioning and Projects windows, the **DemoBean.java** will not appear.

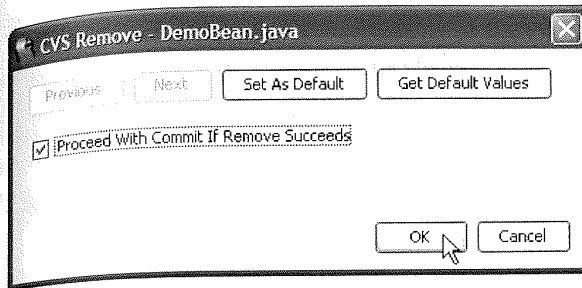


Figure 4-45 CVS Remove File

## Configuring CVS Settings

The IDE lets you configure CVS with settings that apply to a single local working directory or globally for all projects under version control. Let's show you how to access these settings for CVS management.

### Local Settings

The IDE lets you view or change settings for your working directory. Here are the steps for the PaymentCVS project.

1. From the Creator toolbar, select Versioning > Versioning Manager. In the Versioning Manager dialog, select the working directory and click Edit. (Or, right-click the working directory in the Versioning window and select Customize.)
2. The Customizer dialog has four tabs: Profile, Advanced, Environment, and Properties. Figure 4-46 shows the settings under the Properties tab. Note that changes made in the Customizer apply only to the working directory you select.

### Global Settings

It's also possible to view or modify global settings that apply to all CVS working directories and repositories. Here are the steps.

1. From the Creator toolbar, select Tools > Options.
2. Click the Advanced radio button.
3. Expand the Source Creation and Management node and Version Control Settings node.
4. Click CVS. Figure 4-47 shows the General Properties window. Clicking any customizer box here grants you access to a wide variety of different configuration parameters that you can view or modify.

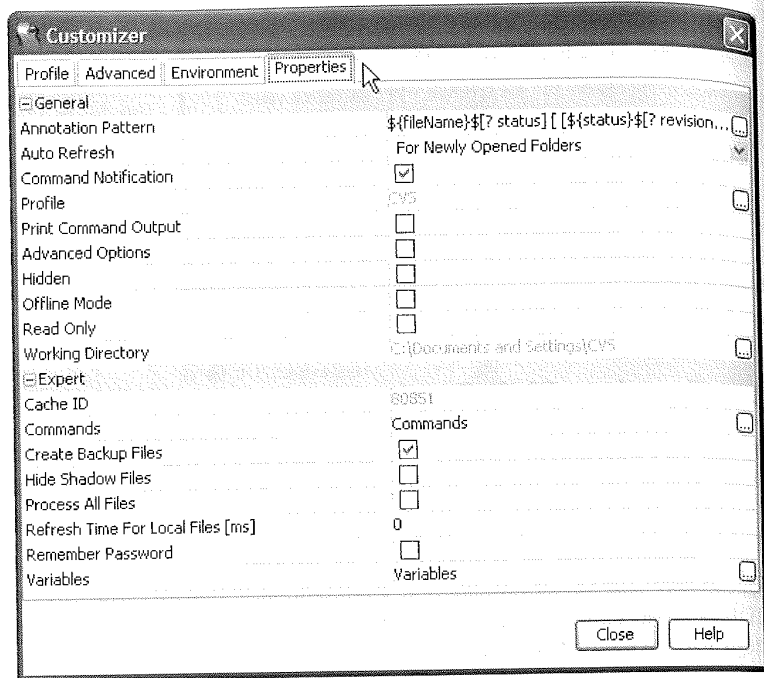


Figure 4-46 CVS Customizer for Working Directory

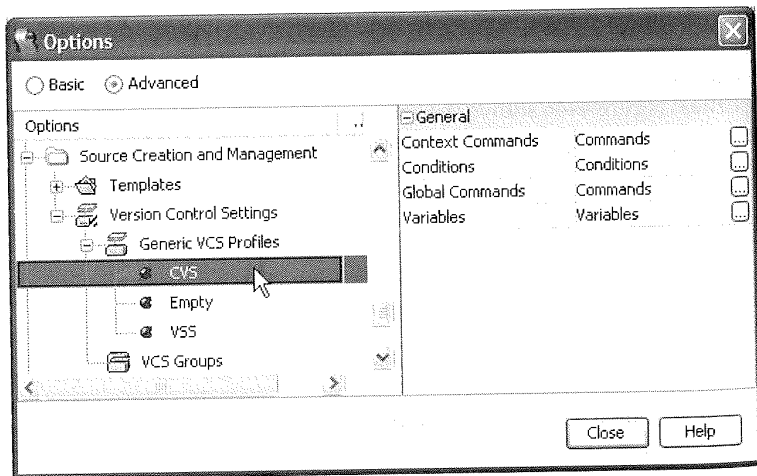


Figure 4-47 CVS Global Options

## Advanced CVS Features

The IDE also implements more advanced features of CVS, such as branches and merging. A *branch* allows you to maintain different versions of a code base. This can be handy when a customer requires a different version of your code or you need to build a demo program. After creating a branch, any committed changes that you make apply only to that branch.

CVS also supports code *merging*. This can be useful when it's time to incorporate branch code back into a code "trunk" or to merge file revisions. Merging in CVS can be a bit tricky because *merge conflicts* are possible. This can happen when more than one developer changes the same line of source code. The IDE helps you graphically resolve merge conflicts before committing the code to version control.

Branches and merging are beyond the scope of this book, but you should know enough about CVS now to apply these advanced features if you need them.

## 4.4 Creating Non-Web Projects

While Creator is a great IDE for creating and managing web applications, you might also want to create non-web projects, such as stand-alone Java programs. With general projects, the IDE generates an Ant script to build, run, and debug your project. You can also add testing. In this section, you'll step through building a general project with a very useful goal: the project creates a sample database that you'll use later on in this book to explore Creator's database access facilities.

The MusicBuild project consists of a single Java class, a library that you'll add through the Libraries node, and the default JDK that comes installed with Creator. When you run the project, it generates sample database tables, table constraints, and records for a Music Library.

This project is included in the book's download. If you don't want to step through the building process, you can bring up the project in the IDE. The project is located at `FieldGuide2/Examples/Projects/MusicBuild`.

### Create a General Project

Here are the steps to create the MusicBuild project.

1. Close any projects that are open. From the Creator Welcome page, click the Create New Project button.
2. In the New Project dialog, select **General** under Categories and **Java Class Library** under Projects. Click Next.

3. In the New Java Class Library dialog, specify project name as **MusicBuild**.
4. Click Finish.

After creating the project, Creator builds the structure for your project which you can inspect through the Projects window.

### **Add a Java Package**

Here are the steps to add a Java package under the Source Packages node.

1. In the Projects window, expand the Source Packages node. You'll see that Creator generates a default package node for you.
2. Right-click the Source Packages node and select **New > Java Package**. Creator displays the New Java Package dialog.
3. For Package Name, specify **asg.databuild**. Click Finish. Creator replaces the default package node with package **asg.databuild**.

### **Add a Java Class File**

Here are the steps to add the main Java file to this project.

1. In the Projects window, select package **asg.databuild**. Right-click and select **New > Java Class**.
2. In the New Java Class dialog, specify class name **DerbyMusicDB**. Creator generates class file **DerbyMusicDB.java**.
3. Copy and paste the contents of **DerbyMusicDB.java** found in your Creator book download at **FieldGuide2/Examples/Database/utills**.
4. Go ahead and build the project (don't run it yet, though). From the main menu, select **Build > Build Main Project**. There should be no build errors in the Output window.

### **Add a JAR/Folder**

The **DerbyMusicDB** class uses Java's JDBC package to connect to the bundled database. In order for this program to work, you must make the database driver class available at runtime (found in jar file **derbyclient.jar**). You must also make sure that the bundled database server is running (before running the project).

Here are the steps to add the **derbyclient** JAR file to the project.

1. From the Projects window, select Libraries, right-click, and select **Add JAR/Folder** from the context menu. Creator displays the Add JAR/Folder dialog.



2. Browse to the Creator installation directory and locate the derbyclient JAR file, `<Creator-Installation-Directory>/rave2.0/core/derbyclient.jar`. Click Open, as shown in Figure 4-48.

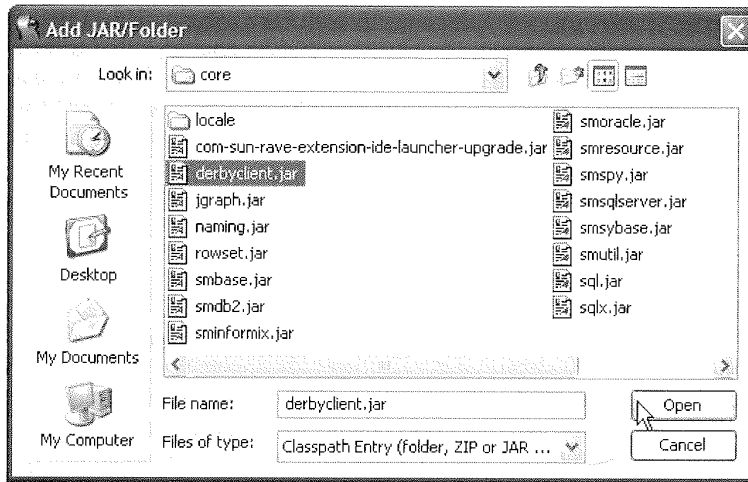


Figure 4-48 Add JAR/Folder dialog

After adding the `derbyclient` JAR file to your project, the Projects window should display its name under Libraries, as shown in Figure 4-49.

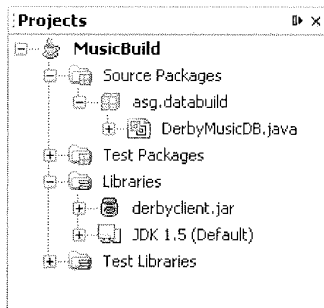


Figure 4-49 Projects view for project MusicBuild

## ***Build and Run Project***

Now you are ready to build and run project **MusicBuild**. Click the green Run Main Project icon from the icon toolbar or select Run > Run Main Project from the main menu.

After running the program, the Output window should tell you the Music database was created. Once you add the Music schema as a data source to Creator's IDE, you can build web applications with design-time support for data-aware components. We show you how to do this in Chapter 9. See "Configuring the Bundled Database" on page 376.

## **4.5 Key Point Summary**

- The IDE greatly simplifies the "edit-compile-deploy" cycle of complex web applications.
- Keyboard shortcuts and code completion help make coding easier.
- The IDE lets you format your code, change fonts and colors, collapse (fold) sections of code, generate import statements, and use abbreviations for heavily used Java keywords and expressions.
- Javadoc popup windows make it easy to locate documentation for Java classes.
- The IDE helps you generate code when extending a Java class or implementing a Java interface.
- The IDE generates properties that conform to the JavaBeans component model.
- Task lists provide a way to document and clean up loose ends in your code.
- Refactoring is transforming and restructuring source code so that the refactored code behaves the same as the original source.
- With refactoring, you may rename a class, field, or method, generate getter and setter methods for fields, change method signatures, and move classes to another package.
- The IDE supports Undo/Redo for refactoring commands.
- Creator supports CVS (Concurrent Version System), one of several Version Control Systems (VCS).
- With CVS, you may place source code under version control, generate revisions, compare revisions, and examine log histories of code changes.
- A CVS repository is a directory that stores a project's full revision history.
- A CVS working directory stores the source code of your project.
- The IDE lets you setup CVS profiles and configure the CVS environment when you work with project code under version control.
- Importing files in CVS is placing project source code under version control.
- Committing source files is storing your edited changes in CVS.

- The IDE has a Graphical Visualizer to help you compare different revisions in CVS.
- History logs in CVS help you document what source code lines were changed in each revision and by whom.
- After importing source code files into the repository and checking them out to your working directory, you may add new files to your project or remove them.
- The IDE also lets you create non-web projects, such as stand-alone Java programs.

n Run  
t from

Music  
o Cre-  
: data-  
ifigur-

: web

fold)  
r

a

nt

code.

getter  
asses

rsion

e  
jes.  
ory.

ent

ontrol.

# WEB PAGE DESIGN

## Topics in This Chapter

- Component Style
- Themes
- CSS Style Editor
- Page Layout
- Page Fragments
- Project Templates
- Navigation with Page Fragments
- Tab Sets

Facebook's Exhibit No. 1003  
Page 00220

# Chapter 7

**S**un Java Studio Creator provides layout components, visual design editors, and style sheet editors to help you design visually pleasing pages for a coherent, unified-looking web application. In this chapter we explore some of the Creator tools and components available to you for page design.

Once you've designed your pages, you'd like to reuse the components, style settings, and logos and images on subsequent pages or in other projects. Cascading style sheets, page fragments, and project templates help designers build artifacts that can be reused. Although you will see some event handling code, page initialization code, and property patterns in the upcoming examples, this chapter concentrates on Creator's visual page design.

## 7.1 Using the Visual Design Editor

Creator's visual design editor runs in the main editor pane by default when you create a new project. Its main purpose is to help you select components and position them on the page using a page grid. You can also turn off the grid, temporarily disable it, or change the grid size.

## Create a Project

Let's explore the main features of the design editor. To do this, you'll build a simple project with three static text components. Figure 7-1 shows the project in the design view with these components.

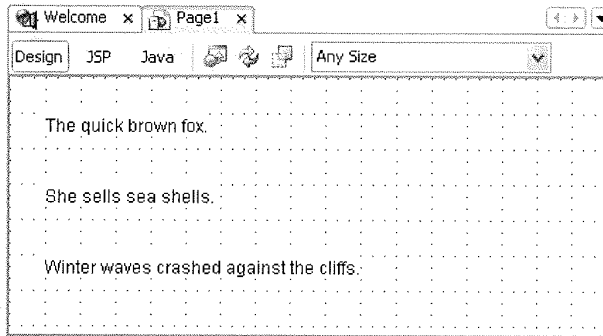


Figure 7-1 Visual design editor in the editor pane

1. From Creator's Welcome Page, select button Create New Project. Creator displays the New Project dialog. Under Categories, select Web. Under Projects, select JSF Web Application. Click Next.
2. In the New Web Application dialog under Project Name, specify **Design1**. Click Finish.

After initializing the project, Creator comes up in the design view of the editor pane. You can now set the title.

3. In the Properties window, select property Title and specify title **Design1**. Finish by pressing <Enter>.

## Add Components to the Page

Now let's add the static text components and specify the text for each one.

1. From the Basic Component palette, select component Static Text and drop it on the page. Don't worry about positioning yet.
2. The component is selected so that you can type in some text. Type in the text **The quick brown fox** followed by <Enter>.
3. Select component Static Text again and add it to the page under the first component.
4. Specify its text **She sells sea shells**.

5. Add a third static text component and specify its text **Winter waves crashed against the cliffs.**

## ***Working with Components on the Page***

By default, a project's layout is in grid mode. This allows you to position components on the page at an absolute location. The grid helps you with component alignment.

1. Select the first static text component. Creator marks the component selected and displays component resizing handles for you. With the component selected, you can move it to a different location. You'll note that it automatically snaps to the grid.
2. Select the same component a second time. The text area is selected (Creator displays a blue background) and the component is enabled for text editing. In this mode you can now issue typical editing short cuts, such as **<Ctrl+C>** for copy or **<Ctrl+V>** for paste. You can also select a word, use the left and right arrow keys, or type replacement text.
3. Sometimes you'll want to move the component so that it doesn't snap to the grid lines. To temporarily turn off grid alignment, select the component, hold the Shift key, and use the mouse to adjust the component on the page.

You can configure Creator to change the grid size or disable it using the Tools menu.

1. In the main menu, select **Tools > Options**. In the Options dialog, select **Visual Designer**. Creator displays the options for the Visual Design Editor, as shown in Figure 7-2.
2. After making changes, click Close.

The Show Grid option controls whether or not the grid is visible in the editor. This is true by default. When the Snap to Grid option is set to true, the components align with the grid lines in the designer. The Grid Height and Grid Width values control the grid size and the Target Browser Window controls the size of the application's window in the browser.

## ***Component Alignment***

Each component has a context-sensitive menu that becomes visible when you select a component and right-click the mouse. The Align menu option provides component alignment criteria. While frequently you can align components by simply using the default behavior of snapping to the grid lines, occasionally you'll want to align components using other criteria. For example, here's how to center the three static text components.

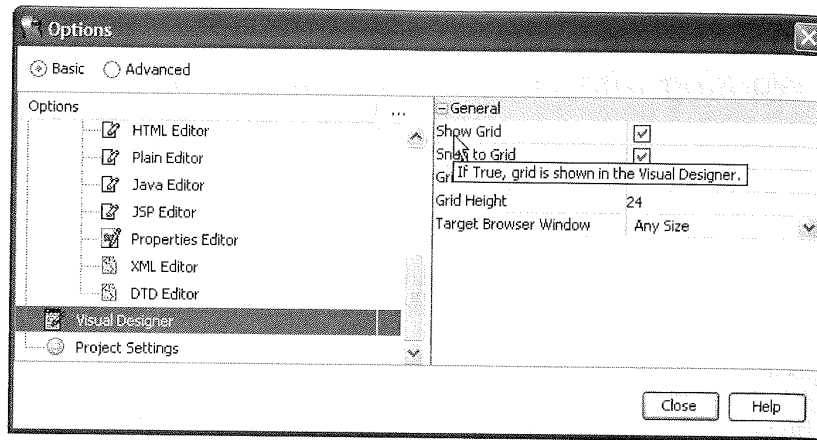


Figure 7-2 Visual Designer Options dialog

1. Choose one static component and position it on the grid at the desired location. Use the Shift key if you'd like to disable the snap to grid lines.
2. You can select multiple components by selecting one, then selecting others while holding the Shift key. Alternatively, you can draw a box around the components you'd like to select, as shown in Figure 7-3. Click the mouse at a spot above and to the left of all the components. Drag the mouse towards the lower-right until all the components are enclosed in the selection box. When you release the mouse, all three components are selected.

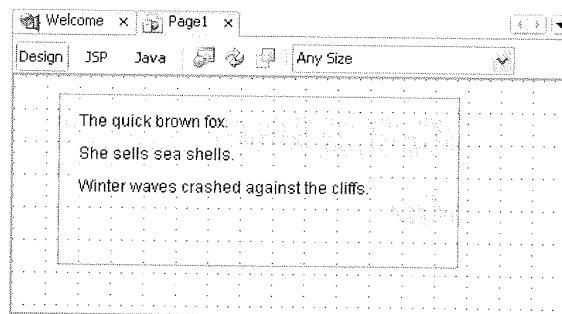


Figure 7-3 Selecting multiple components



- Place the mouse over the component that you want to use as the alignment reference, right-click, and select **Align > Center**. The three components will be horizontally centered using the selected component as the alignment reference.

For horizontal alignment options, select Left, Center, or Right. For vertical alignment options, select Top, Middle, or Bottom.

## Deploy and Run

After aligning the components, deploy and run the application. Figure 7-4 shows project Design1 running in the browser. The components were centered horizontally using the third component as the reference for the alignment.

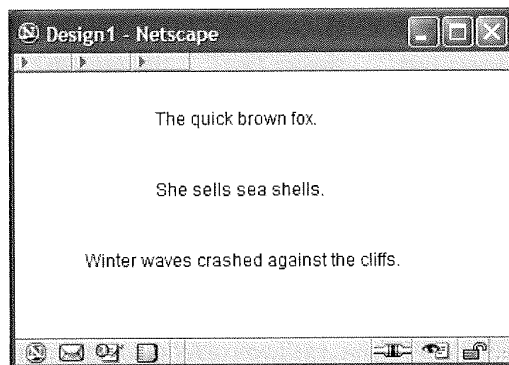


Figure 7-4 Project Design1 running in a browser

## 7.2 Themes

Creator gives web page designers a number of choices for specifying the look of a web page. The components from the Basic, Layout, and Composite sections of the palette are rendered using *themes*. A theme is a bundled set of cascading style sheets, JavaScript files, and images that apply to the components and the web page. Creator currently ships with four configured themes. The available themes for a project are listed in the Projects window under node Themes, as shown in Figure 7-5. The currently selected theme is marked with a triangle badge. To change the current theme, right-click a new theme selection and select **Set As Current Theme**.

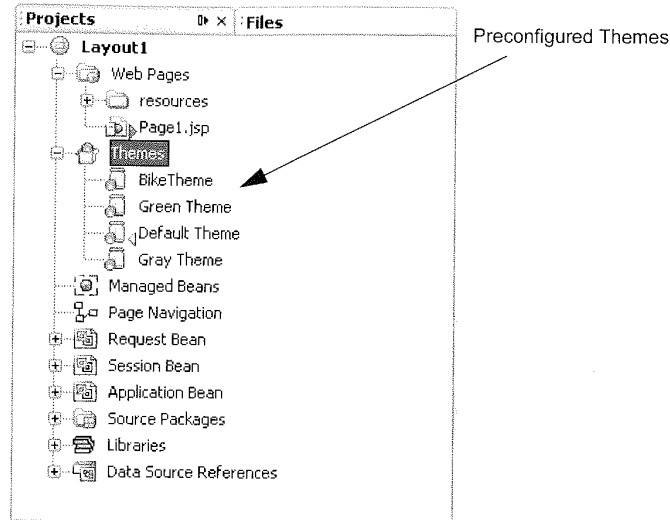


Figure 7-5 Creator Themes available for projects



#### Creator Tip

*To make a new theme take effect for deployment, first stop the application server, then clean and rebuild the project.*

The Default Theme provides a gradient blue color, giving the components a unified look. The Gray and Green Themes provide color variations with the same gradient appearance. The Gray Theme is useful when you want to give the components a neutral look (for example, if your color scheme does not mesh well with either blue or green). The Bike Theme is used with **Jump Start Cycles**, one of Sun's sample applications. Access sample applications at the following url:

<http://developers.sun.com/prodtech/javatools/jscreator/reference/code/sampleapps/index.html>

### Changing the Look with Themes

Let's create a simple application, deploy it, and change its current theme. This application won't do much, but you'll see how several components are affected by theme selection. Figure 7-6 shows the project running in a browser. The

page contains a hyperlink component, text field, label, static text component, and table. The application is built with the Default (blue) Theme.

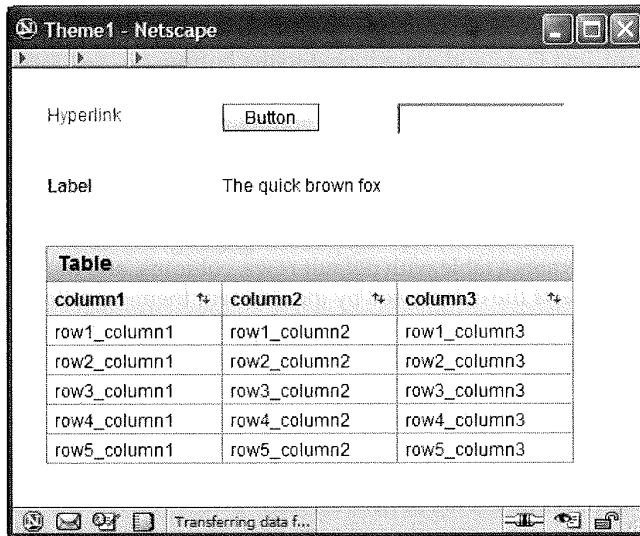


Figure 7-6 Project Theme1 running in a browser

1. From Creator's Welcome Page, select button Create New Project. Creator displays the New Project dialog. Under Categories, select Web. Under Projects, select JSF Web Application. Click Next.
2. In the New Web Application dialog under Project Name, specify **Theme1**. Click Finish.

After initializing the project, Creator comes up in the design view of the editor pane. You can now set the title.

3. In the Properties window, select property `Title` and specify title **Theme1**. Finish by pressing `<Enter>`.
4. In the Projects window, expand the Themes node and make sure that the Default Theme is selected as the current theme (it should display a triangle badge).

### Add Components to the Page

Using Figure 7-6 as a guide, add components to the page. Except for the static text component, all will retain their default settings.

1. From the Basic Component palette, select component Hyperlink and drag it to the top-left portion of the page.
2. Select component Button and place it to the right of the hyperlink component.
3. Select component Text Field and place it next to the button.
4. Select component Label and place it under the hyperlink component.
5. Select component Static Text and place it to the right of the label component. When you drop it on the design canvas, it remains selected and you can begin typing. Type in some text (**The quick brown fox**) and finish with **<Enter>**.
6. Finally, select component Table and place it on the page below the label component. Creator generates a table with default rows, columns, and data. The table component reflects the colors used by the different themes particularly well.
7. Deploy and run the application by clicking the Run arrow on the toolbar.

### **Change the Current Theme**

Now let's change the current theme for this project and redeploy the application.

1. In the Projects window, expand the Themes node. Right-click Green Theme and select **Set As Current Theme**. Creator reminds you that you must stop the application server, then clean and rebuild the project before redeploying.
2. In the Projects window, expand the Libraries node and scroll down until you find the JAR file associated with the Green Theme library.
3. Expand the Green Theme library. You'll see the **defaulttheme-green.css** file as well as the images, properties, and JavaScript files associated with this theme. (You'll look at a cascading style sheet (.css) file shortly.)
4. In the Servers window, right-click Deployment Server and select **Start/Stop Server**. Click the Stop Server button.
5. Return to the Projects window, right-click the Theme1 project name, and select **Clean and Build Project**. (This step is necessary to make the application server use the correct JAR file for the selected theme.)
6. Deploy and run the application by clicking the Run arrow on the toolbar. The application should now display green-colored components.
7. Repeat Steps 1-6 above to change the current theme to the Bike Theme.



#### **Creator Tip**

*Instead of deploying the application each time, you can right-click inside the visual design editor and select **Preview in Browser** for a quick look at a newly selected theme.*

## Modifying the Default Theme

The Default, Gray, and Green Themes are variations of the same theme. Is it possible to modify themes for different colors or use a different theme altogether? The style sheets and images that apply to components can theoretically be modified. The theme JAR files are installed in the Creator2 directory (currently at `rave2.0/modules/ext`). You can unpack the JAR file, edit the CSS file and images, and repackage them. Until Creator includes a theme-based editor, however, this remains a non-trivial task. Still, there's much you can do to control the appearance of your application. Let's continue to explore web page design options beginning with style.

## 7.3 About Style

You've probably noticed by now that each component has a `style` property that allows you to change the appearance of various features such as font (size, color, family, style), position, height, width, etc. Some components also contain "pass-through" HTML attributes, such as border and cellpadding that apply to table-type components. You specify style attributes by modifying the component's property sheet directly. This gives you control over the look of a specific component. It is also a handy way to experiment with different styles until you decide on an overall style for your application.

Property `style` accepts style declarations in the form

```
property1: value1; property2: value2; . . . propertyN: valueN
```

Let's look at an example.

### Copy the Project

You'll make a copy of project Theme1 (call it Theme2) for this section. This step is optional. If you don't want to copy the project, simply skip this section and continue making modifications to the Theme1 project.

1. Bring up project Theme1 in Creator, if it's not already opened.
2. From the Projects window, right-click node Theme1 and select **Save Project As**. Provide the new name **Theme2**.
3. Close project Theme1. Right-click Theme2 and select **Set Main Project**. You'll make changes to the Theme2 project.
4. Expand **Theme2 > Web Pages** and open **Page1.jsp** in the design view.

5. Click anywhere in the background of the design canvas of the Theme2 project. In the Properties window, change the pages's Title property to **Theme2**.
6. Select the table component, right-click, and select **Delete** from the context menu to remove the component from the page (to simplify the page).

### Using the Style Editor

Let's use the style editor to manipulate a component's look.

1. For project Theme2, restore the current theme to the Default Theme (blue-toned). Remember to stop the application server and clean and rebuild the project before proceeding.
2. In the Design View, select component `staticText1`.
3. In the Properties view, click the small editing box opposite property `style`. Creator pops up the Style Editor, as shown in Figure 7-7.

Creator provides a sophisticated Style Editor that lets you specify a component's style attributes. The Property Selection window lets you choose a property to edit. Depending on this selection, the editor displays windows that let you select values from a list or specify a custom value. When you change an attribute, the Results Display Window applies the style to the component.

Use the Style Editor to modify the static text component's `style` property.

1. Select **Font** in the Property Selection window.
2. In the Font-Family selection window, select the list of font-family values beginning with **Verdana**.
3. In the Size window, select font size **18**.
4. In the Style selection window, choose **italic** from the drop down menu. Note that the text in the Results Display windows reflects your selections.
5. Now select **Background** in the Property Selection window. The editor displays a different set of selection windows.
6. Select color **yellow** in the Background Color drop down menu.
7. Choose **Border** in the Property Selection window.
8. In the All selection window for Style select **solid**, for Width select **1px**, and for Color select **gray**.

Here is the new CSS Style setting for this component.

```
border: 1px solid gray; background-color: yellow;
font-family: Verdana, Arial, Helvetica, sans-serif;
font-size: 18px; font-style: italic; left: 120px; top: 72px;
position: absolute
```

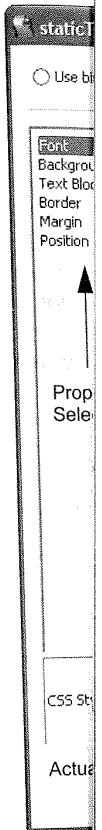


Figure 7-7

9. Click

**De**

Right-  
depl

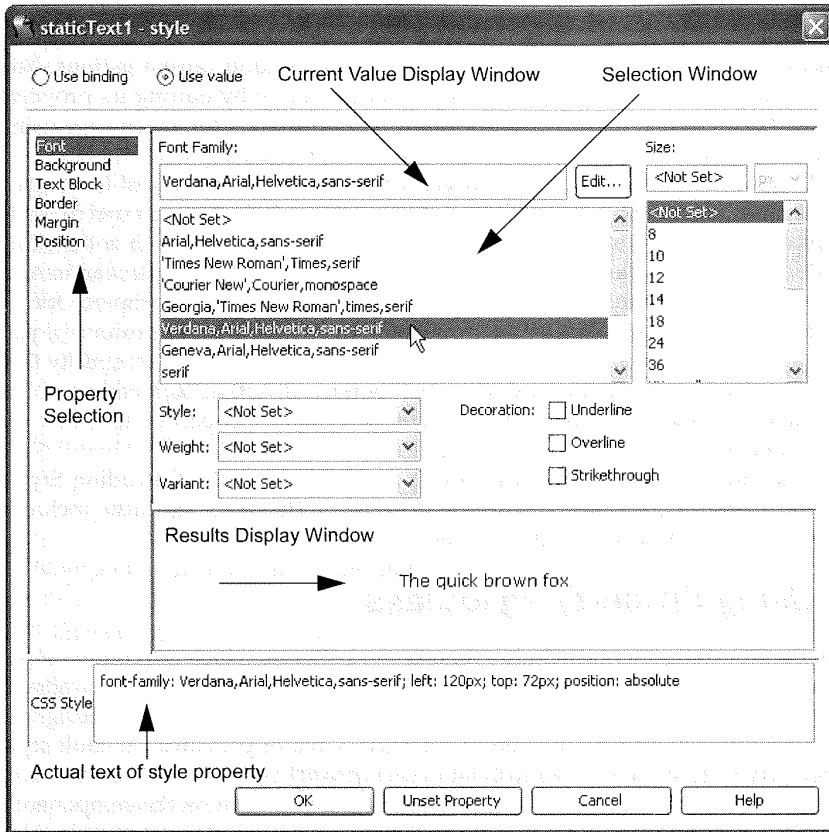


Figure 7-7 Using the Style Editor

9. Click OK. The page in the design view reflects the new style characteristics.

### Deploy and Run

Right-click in the visual design editor and choose Preview in Browser, or deploy and run the application to check its appearance.

## 7.4 Cascading Style Sheets

Using a component's `style` property to control its look can be tedious. You must specify attributes for each component manually by editing its property sheet. Furthermore, it's difficult to employ a uniform look for a web page using only `style` property settings.

Creator uses Cascading Style Sheets (CSS) to control the look of its components and pages. CSS is a standard that allows a web designer to specify style characteristics. The style characteristics apply to a document in a *cascading* fashion: that is, a style applies to a given level and subsequent styles can in turn apply on top of these "inherited" styles. If you don't specify a property for an element, it generally inherits the property from its "parent." For example, you can specify that all text in a document is (color) navy. You can then specify that text in a footer is a smaller text size. The footer text will be *both* navy and the smaller size since the footer-specific style inherits all properties specified for the global style.

You can read more about Cascading Style Sheets at the Cascading Styles Home Page: <http://www.w3.org/Style/CSS/>. The web site also includes tutorials about how to use style sheets.

### Using Property `styleClass`

All Creator components include property `styleClass`, which is a comma separated list of style classes. You define and store a style class in a text file called a *style sheet*. As stated earlier, using property `styleClass` helps web designers create a uniform look to all pages in a project. Creator provides a default style sheet, `stylesheet.css`, that is included in each project you create. When you add style classes to the style sheet, you can then reference them in the component's `styleClass` attribute. (Note that the bundled themes include a set of style rules that also apply to the components.)

A style sheet is a collection of style *rules*. Each rule consists of a *selector* and a *declarator*. The selector identifies an HTML element(s) or style class name(s) to which the rule applies. The curly braces encompass the declarator, which is the semi-colon separated list of property-value pairs. While the component's `style` attribute lists the property-value pairs for a given component, a rule is a collection of property-value pairs that is named.

Let's examine the default style sheet, `stylesheet.css`, in the Style Sheet Editor. In the Projects window for project Theme2, select **Web Pages > resources** and double-click file `stylesheet.css`. Creator brings up the style sheet in the Style Sheet Editor and highlights the first rule, `.list-header`.

```
.list-header {
    background-color: #eee;
    font-size: 1.2em;
    font-weight: bold;
}
```

There are many ways to define a style class. You can use a class name in a table header, a global setting, a color, and so on.

You can use a class name in a table header, a global setting, a color, and so on.

1. Scroll to the bottom of the page.

```
/* Custom style class */
```

2. Now add a new style class.

```
/* Custom style class */
```

```
body {
}
```

3. Put the cursor at the end of the line.

4. Create a new style class.

5. Create a new style class.

230, 230

6. Continue to add style classes.

```
body {
    background-color: #eee;
    font-size: 1.2em;
}
```



```
.list-header {  
  background-color: #eeeeee;  
  font-size: larger;  
  font-weight: bold;  
}
```

There are three property-value pairs here: property `background-color` has value `#eeeeee`, property `font-size` has value `larger`, and property `font-weight` has value `bold`. Rules that contain an initial dot are *style classes*. Once you define them in the style sheet for your project, you can specify the class names in a component's `styleClass` attribute.

You can also define rules that apply to HTML elements such as `<body>`, `<th>` (table heading), `<td>` (table data). The `body` style rule is a good place to list global settings for your web application, such as basic font characteristics, text color, and background color. Let's do this now.

1. Scroll to the top and add the following comment.

```
/* Custom style rules */
```

2. Now add a rule for `body` followed by opening and closing braces.

```
/* Custom style rules */
```

```
body {  
}
```

3. Put the cursor after the open brace and hit **<Enter>**. You see that Creator pops up a property selection dialog. Select **background-color** followed by **<Enter>**.
4. Creator now pops up a value selection dialog, as shown in Figure 7-8. Scroll down to the bottom and choose **more ...** and hit **<Enter>**.
5. Creator pops up a Choose Color dialog. Select tab RGB and specify values **230, 230, 200**, as shown in Figure 7-9. Click OK. Creator fills in value `#e6e6c8` for property `background-color`.
6. Continue editing style class `body`. You can also use the style selection windows below the editing pane. Here is the style to use for `body`.

```
body {  
  background-color: #e6e6c8;  
  font-family: Verdana,Arial,Helvetica,sans-serif  
}
```

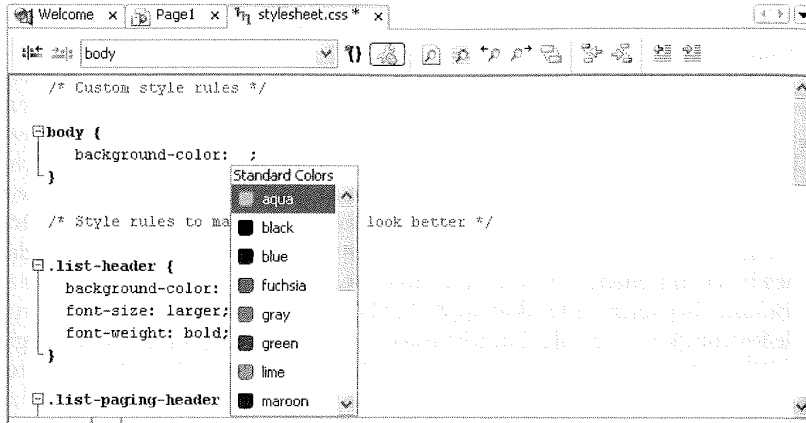


Figure 7-8 Using the Style Sheet (CSS) Editor

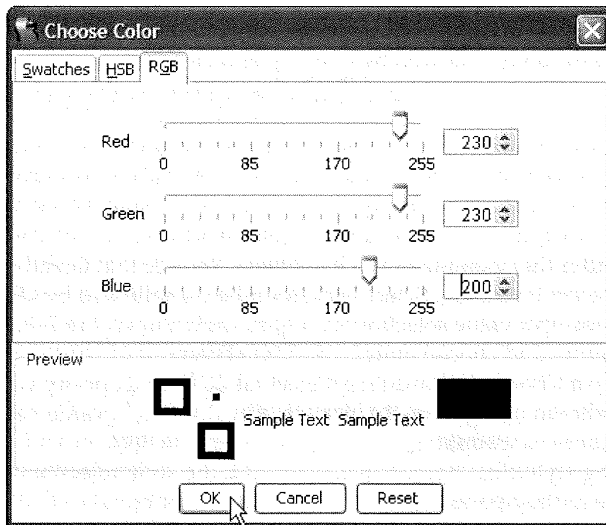


Figure 7-9 Choose Color dialog

7. Add  
wind

.head  
for  
for  
}

8. Selec  
Sheet

The b  
declared  
erty-val  
Return  
the font  
apply th

1. In the  
brow
2. Select  
pops  
positi
3. Select
4. Make  
for pr  
follow  
size ar

### Dep

Deploy a  
browser.

## 7.5

Creator p  
your web  
panel, an  
order to c

7. Add a style class called `.headingStyle`, as follows. Use the style selection windows to specify `font-size` and `font-weight` (or just type them in).

```
.headingStyle {  
    font-size: XX-large;  
    font-weight: bold  
}
```

8. Select the Save All icon on the toolbar to save the changes and close the Style Sheet Editor (click the small x in the `stylesheet.css` tab).

The `body` rule applies to all HTML `<body>` elements, as well as any elements declared inside of `<body>`. Thus, nested (“children”) elements inherit the `property-value` settings from their enclosing (“parent”) elements.

Return to the Page1 design view. You’ll see that the background color and the font-family setting reflect the `body` style rule you defined. Now you’ll apply the `.headingStyle` style class to the static text component.

1. In the design view, select the static text component (its text is “The quick brown fox”).
2. Select the editing box opposite the `style` property. When the style editor pops up, select **Unset Property**. Creator clears its `style` setting (including its position value). The component is now in the upper-left corner.
3. Select it and move to its previous location (restoring its position values).
4. Make sure the component is still selected and type in the text **headingStyle** for property `styleClass` (do not use the initial dot from the style sheet file) followed by **<Enter>**. The static text component now has the extra large font size and is bold.

## ***Deploy and Run***

Deploy and run the project. Figure 7-10 shows the application running in a browser.

## **7.5 Page Layout**

Creator provides several components that help you manipulate the layout of your web page. In this section, we’ll examine components layout panel, grid panel, and anchor (paired with hyperlink). We’ll keep the projects simple in order to concentrate on page layout issues.

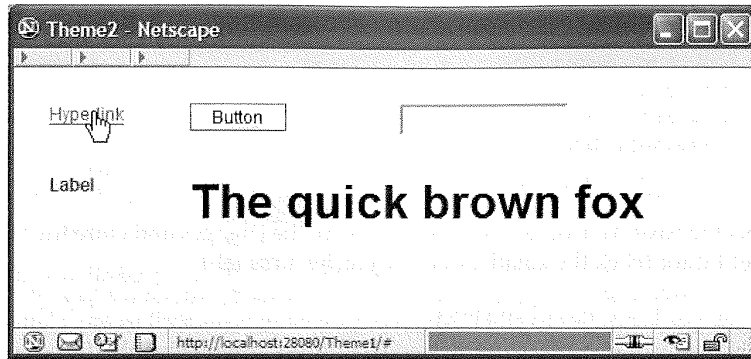


Figure 7-10 Project Theme2 running in a browser

## Layout Panel

Creator's palette includes several component containers that group or nest embedded components. With containers, you can uniformly control the "children" components' appearance, including position, style, and rendering. The layout panel component positions its components with either a flow layout, placing each component directly after the previous one, or a grid layout, letting you position components using the design editor. For this project, we'll also show you how to inspect the CSS style rules and HTML rendering that Creator generates for you. Finally, we'll show you how to center components on the page, even when the user resizes the browser window.

## Create a Project

In the following project, you'll control components by grouping them together, allowing the components to share common style, position, and rendering attributes. You'll see how the layout panel lets you position components using the standard grid.

1. From Creator's Welcome Page, select button Create New Project. Creator displays the New Project dialog. Under Categories, select Web. Under Projects, select JSF Web Application. Click Next.
2. In the New Web Application dialog under Project Name, specify **Layout1**. Click Finish.

After initializing the project, Creator comes up in the design view of the editor pane. You can now set the title.

3. In the Properties window, select property `Title` and specify title **Layout and Style**. Finish by pressing **<Enter>**.

## Add Components to the Page

Figure 7-11 shows the design view of project `Layout1`. Use this as a guide as you add components to the page.

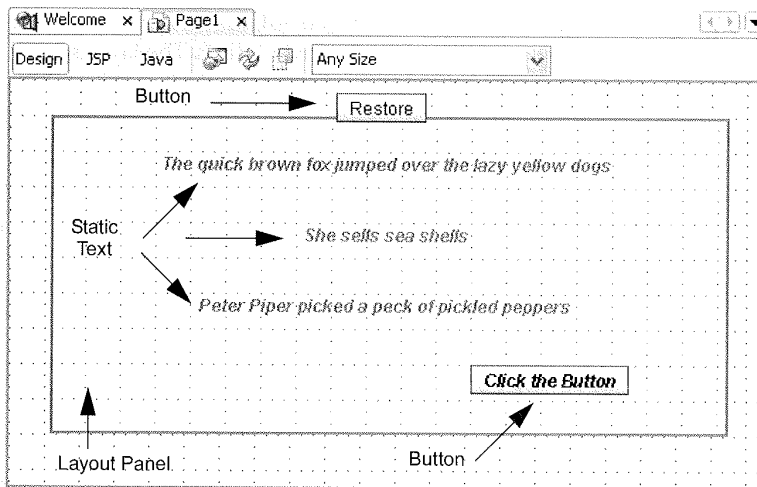


Figure 7-11 Design view for project `Layout1`

1. From the Layout section of the Component palette, select `Layout Panel` and drop it on the page. Enlarge it so that it is approximately 20 grids wide and 10 grids high.
2. Make sure the layout panel component is selected. In the Properties window opposite property `panelLayout`, select **Grid Layout** from the drop down list. This lets you use the design view's grid to position components that you'll add to the panel.
3. Still in the Properties window for the layout panel, click the small editing box opposite property `style`. Creator pops up the Style Editor.

As you've seen, there are several ways to modify a components' style property. We'll step you through using the editor, but you can also type in the style attributes manually. Refer to Figure 7-7 on page 283 for the window labels used here.

4. In the Property Selection window, select **Font**. In the center Font Family Selection Window, choose **Georgia, Times New Roman, times, serif**. In the Size Selection Window, choose **12**. In the Style window, select **italic** from the drop down list. In the Weight window, select **bold** from the drop down list. In the Color window, select **gray** from the drop down list.
5. Now select **Background** in the Property Selection window. In the Background Color window, type the value **rgb(255,255,204)** followed by **<Enter>**. The small color indicator on the right will change to a muted yellow.
6. Select property **Border**. In the top row labeled All, for Style select **solid** and for Width select **2px**. Select OK to close the Style Editor. The layout panel now has a border and a new background color.

By changing the layout panel's `style` settings, you'll see how the children components are affected by the layout panel's `style`. Some style attributes are inherited (such as font characteristics); others are not (such as border). And some settings are overridden by more specific settings. We'll examine this in more detail after you add components to the layout panel.

1. From the Basic Components palette, select Static Text and drop it on the layout panel component. The static text component appears in the Page1 Outline view nested under the layout panel.
2. Type in the text **The quick brown fox jumped over the lazy yellow dogs** followed by **<Enter>**.
3. Change the static text `id` property to `line1`.
4. Add two more static text components with text **She sells sea shells** and **Peter Piper picked a peck of pickled peppers**. Change the `id` properties to `line2` and `line3`. All three static text components will be nested under the layout panel in the Page1 Outline view, as shown in Figure 7-12. (The screen shot also shows two button components, which you'll add later.)

Let's position the three static text components so that they're centered on the layout panel.

1. First, use the grid lines to evenly space the static text components vertically. Leave some room at the bottom of the panel for a button component.
2. Select the first static text component. Use **<Shift+Click>** to select all three static text components, as well as the layout panel.
3. Position the cursor inside the layout panel (anywhere in the background) and right-click. Select **Align > Center** from the context menu. Creator centers all three text components horizontally, using the layout panel as the reference component. The components should now be centered, as shown in Figure 7-11 on page 289.

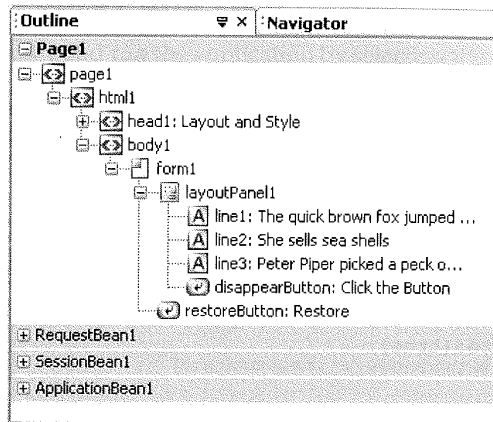


Figure 7-12 Page1 Outline view for project Layout1

#### Creator Tip

*Note that the components are positioned relative to the layout panel component. If you re-position the layout panel on the page, the nested components retain their relative position inside the panel.*



Now let's add two button components: one inside the layout panel and one outside.

1. From the Basic Component palette, select component Button and place it inside the layout panel. Position it under the three text components off-center to the right.
2. Change its text label to **Click the Button** (note that the button's font inherits the font style from the panel).
3. Change the button's `id` property to `disappearButton`.
4. From the Basic Component palette, select component Button again and place it on the page, centered above the layout panel (outside of the panel). The button component is not nested inside the panel component in the Page1 Outline view. Its font characteristics are therefore independent of the layout panel's settings.
5. Change its text label to **Restore**.
6. Change the button's `id` property `restoreButton`.

The event handling code for the buttons will make the layout panel disappear from the page (`disappearButton`) and then will restore it on the page (`restoreButton`).

1. In the design view, double-click the first button (`disappearButton`). Creator generates a default action event handler and brings up the Java source editor so that you can add event handling code.
2. Add the following code to the `disappearButton_action()` event handler (the added code is shown in bold).

```
public String disappearButton_action() {  
    layoutPanel1.setRendered(false);  
    restoreButton.setRendered(true);  
    return null;  
}
```

The event handler sets the `rendered` property of the layout panel to false, causing it (and all of its nested components) to disappear from the page. It makes the Restore button appear on the page.

1. Click label Design in the editing pane to return to the design view.
2. Double-click the Restore button, which brings up the action event handler in the Java source editor.
3. Add the following code to the `restoreButton_action()` event handler. The added code is bold. When the user clicks the button, the layout panel and all of its nested components will be rendered on the page. At the same time, the Restore button will disappear.

```
public String restoreButton_action() {  
    layoutPanel1.setRendered(true);  
    restoreButton.setRendered(false);  
    return null;  
}
```

4. Click label Design in the editing pane to return to the design view.
5. Select the Restore button. In the Properties view under Advanced (scroll down to see it), *uncheck* property `rendered`. The button disappears from the design view.



#### Creator Tip

*Even though the component no longer appears on the design view, you can still select it in the Page1 Outline view. If you want to adjust it visually, turn the rendered property back on, make adjustments, and then turn it off again.*



## Deploy and Run

Deploy and run the application by selecting the green arrow in the icon toolbar. Figure 7-13 shows project Layout1 running in a browser. When you click the inside button, the layout panel disappears and the Restore button is rendered. Clicking the Restore button makes the layout panel reappear.

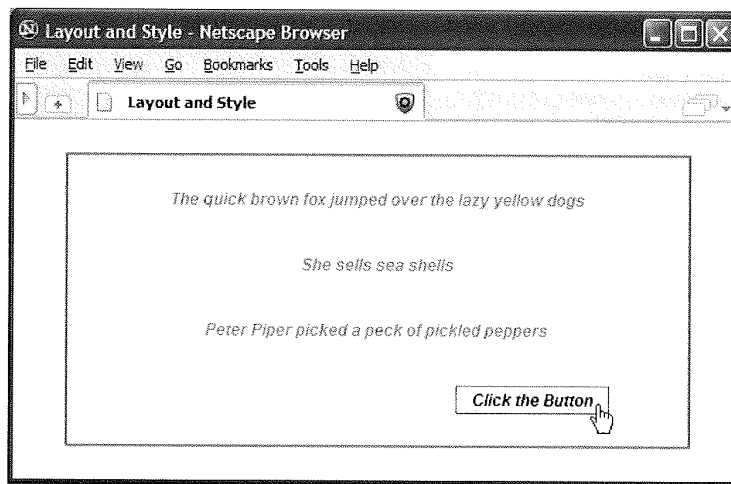


Figure 7-13 Project Layout1 running in a browser

## More CSS Style Issues

Although this project is very simple, there are subtle style issues illustrated here. The style that a component finally acquires is an amalgamation from various sources, some of which may not be obvious. For example, the Creator components acquire a basic style from the pre-configured Theme style sheets. (This is why the button component has a gradient blue background image.) When you nest components inside container components, the nested ones can inherit styles from the enclosing component. (Thus, the button and the text components have a bold, italic font.) To help you figure out where style definitions originate, Creator has a hidden Document Object Model (DOM) inspector (*hidden* because it is not a formal part of the product). Let's examine several components in this project with the DOM inspector.

You access the DOM inspector by selecting a component with **<Ctrl+Alt+Click>**. Creator displays a Layout Inspector window that contains a tree of the page's components. HTML components are shown in angle `<>` brackets and the Creator component `id` appears (if there is one). The Properties

window displays property values that can help you with various style attributes.

For example, select the third static component and type **<Ctrl+Alt+Click>**. Depending on where you actually place the cursor, a word is highlighted in red on the design view. Figure 7-14 shows the Layout Inspector (on the left) and the corresponding Properties window (on the right).

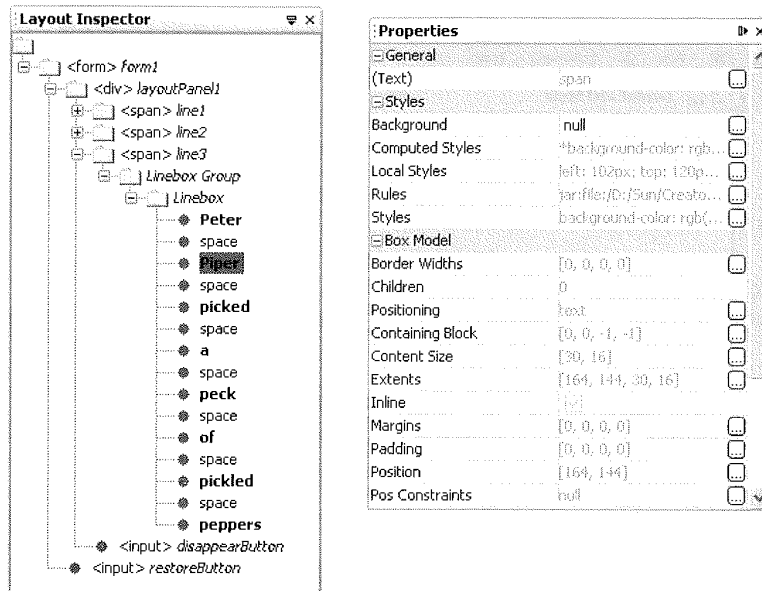


Figure 7-14 Creator DOM Inspector

Let's say you want to determine exactly where the text is set to italic. In the Properties window under Styles, there are several helpful windows. Computed Styles tells you where a style setting originates. Local Styles are the style rules set for this element (values not inherited), Rules are the CSS rules that apply to this element, and Styles is the grand total of all the styles that apply to this element.

In the Properties window, click the small box opposite property **Local Styles**. In the property customizer, you'll see that this element contains only positioning styles. Click Close. Now select property **Rules**. These are the style rules that apply to this element (found in file `css_master.css`). Click Close. Now select property **Computed Styles**. Creator pops up the customizer shown in Figure 7-15. Scroll up until you find the property setting for `font-style` (shown highlighted in the figure). You see that it's set to italic and it references Line 12 in `Page1.jsp`. Click Close and select JSP in the editing pane to open `Page1` in the

yle  
ck>  
red  
and

JSP editor. Line 12 contains the component definition for `<ui:panelLayout>`, the layout panel that contains the text components. Thus, the text component inherited its `font-style` property value from the layout panel.

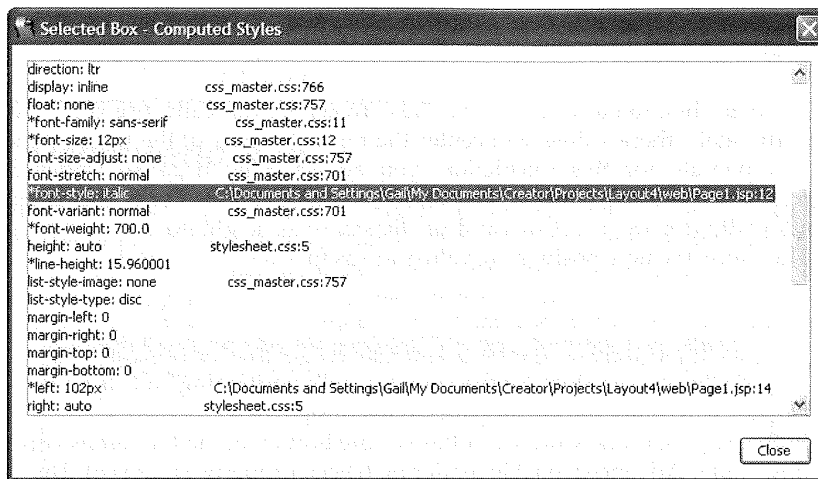


Figure 7-15 Computed Styles property

Finally, return to the design view (click Design in the editing toolbar) and re-select the text component with `<Shift+Alt+Click>`. Now select **Styles** (under Styles) in the Properties window. You'll see all of the styles that apply to the static text component.

## Centering Components on a Page

the  
m-  
yle  
hat  
to

Let's continue our exploration of manipulating style attributes to center the components in your browser window. You can apply centering horizontally, vertically, or both. To center a component horizontally, you must know the *width* of the target component. Likewise, to center a component vertically, you must know its *height*. The convenience of using containers is that you can center the container on the page, and then all of its children components retain their relative positions in the centered container.

les.  
on-  
hat  
lect  
: 7-  
gh-  
! in  
the

We're going to center the components on the page both horizontally and vertically. You center the layout panel and separately center the Restore button (you might want to enable rendering for the Restore button until you're done modifying its style).

1. From the Page1 design view, select the layout panel. In the Properties window, select property `style` and bring up the style editor.

- At the bottom on the window, you'll see the style settings for the component. Note the property settings for `width` and `height`. It will be something similar to the following (depending on how you sized the layout panel).

```
height: 212px; width: 460px;
```

To center horizontally, use `left: 50%`. To center vertically, use `top: 50%`. Unfortunately, these values will center the top-left corner of the layout panel. To compensate for this calculation, you adjust using negative values for `margin-left` and `margin-top`. The value should be *half the size* of the component's width (for `margin-left`) and *half the size* of its height (for `margin-top`).

Therefore, the new positioning values are as follows.

```
margin-left: -230px; margin-top: -106px; left: 50%; top: 50%;
```

- Provide the above values for the layout panel's positioning and click OK. Creator will center the layout panel in the design view.
- In the Page1 Outline view select the Restore button. In the Properties window under Advanced, enable rendering (check property `rendered`). The button will appear on the design canvas.
- In the Properties window, select property `style` and bring up the style editor.



#### Creator Tip

*Note that there are no set values for a button's height and width, because the component automatically sizes itself according to the text label. To find out its approximate size, you can resize it slightly and Creator will then make its size static. Use these values for the centering calculations and then return the component to automatic sizing by removing the static values for width and height.*

- The position values for the Restore button are as follows. Provide the values for the button in the Style editor and click OK.

```
margin-left: -33px; margin-top: -12px; left: 50%; top: 50%;
```

- In the Restore button's Properties window under Advanced, *uncheck* property `rendered`.

**Creator Tip**

*Note that if you adjust the position of the layout component or the Restore button in the design view, Creator replaces the percentage values you supplied for left and top with absolute position values. You'll have to re-edit the style property and supply the percentage settings.*

**Deploy and Run**

Deploy and run project Layout1. Resize the browser window and check that the layout panel component remains centered. After you click the button, the Restore button appears. It should also be centered on the page. Figure 7-16 shows the component centered in the browser window.

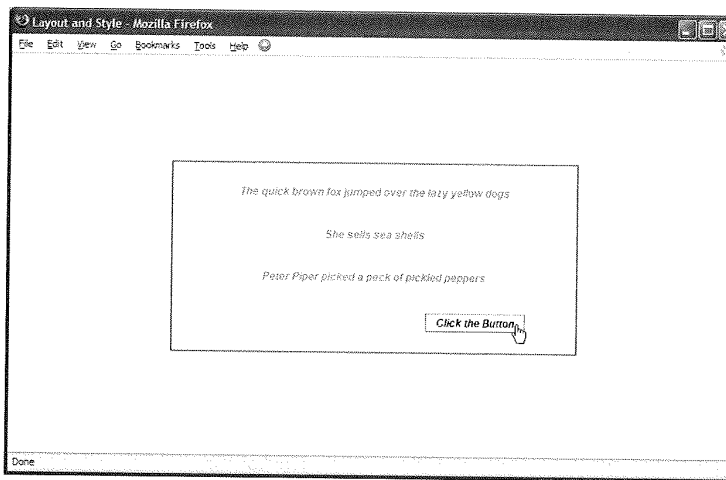


Figure 7-16 Project Layout1 with centered components

**Grid Panel**

Creator provides another container component called a grid panel. The grid panel (as its name implies) provides a grid layout, whereby you specify the number of columns if you require more than the default of one. Creator places each component in the grid, positioning the component in the next available cell. With a single column, a component goes into a cell in the next row.

The grid panel gives the page designer additional options for controlling the page layout. For example, you can nest grid panels to achieve some advanced layout designs. In this section, we'll use the grid panel to control the page lay-

out. We'll show you how to control component placement when you want to position a component after a variable-sized component (such as a table that contains an indeterminate number of rows).

## Create a Project

In the following project, you'll control page layout by nesting components inside grid panels. Project LayoutMadness displays a table of numbers and their squares. The user specifies how many numbers should be displayed. To keep everything simple, the event handling code will generate HTML tags on the fly to build the table. This is a handy technique when you want to generate your own HTML tags.

1. From Creator's Welcome Page, select button Create New Project. Creator displays the New Project dialog. Under Categories, select Web. Under Projects, select JSF Web Application. Click Next.
2. In the New Web Application dialog under Project Name, specify **LayoutMadness**. Click Finish.

After initializing the project, Creator comes up in the design view of the editor pane. You can now set the title.

3. In the Properties window, select property `Title` and specify title **LayoutMadness**. Finish by pressing `<Enter>`.

## Add Components to the Page

You'll add a grid panel to help with page layout. Inside, you'll add a nested grid panel that will hold a text field and button to gather and process the input. A static text component will display (using generated HTML elements) the table of squares and two hypertext/anchor component pairs will help with page scrolling. Figure 7-17 shows the design view.



### Creator Tip

*Note that we assigned contrasting background colors to the grid panels. This is helpful when you want to see how the grid panel is rendered and how it lays out its nested components. When you're satisfied with the layout, you can restore the grid panels' default background colors.*

Figure 7-18 shows the Page1 Outline view for this project. You might want to consult it from time to time as you add the components to make sure that the nesting levels for the components are correct.

Figure 7

Figure 7

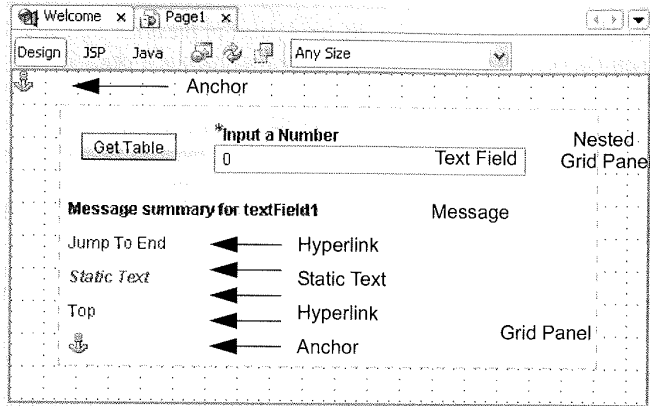


Figure 7-17 Design view for project LayoutMadness

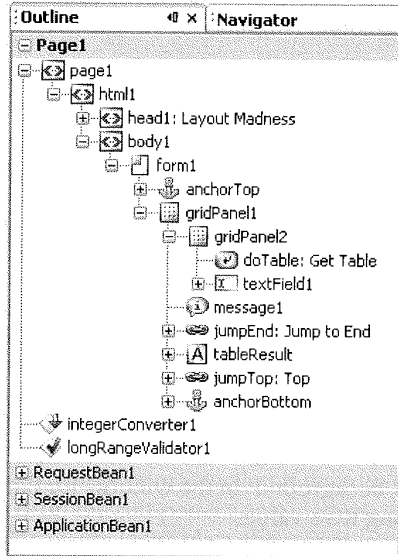


Figure 7-18 Page1 Outline view for project LayoutMadness



### Creator Tip

*Creator lists the components nested in the grid panel in the order that you place them on the page. If you need to rearrange the order, you can select a component, drag it up to its parent container, and re-drop it. This moves the component to the end of the list for that container.*

1. From the Basic Components palette, select Anchor and place it on the page in the upper-left corner.
2. In the Properties window, change its id to **anchorTop**.
3. From the Layout Components palette, select Grid Panel and place it on the page.
4. In the Properties window for the grid panel, select property `style` and bring up the Style Editor.
5. In the Style Editor, select property Background. In the window for Background Color, provide the following RGB values.

```
rgb(255, 255, 215)
```

6. Select property Text Block. For Horizontal Alignment, select **center** from the drop down list.
7. Select property Position. Under Size, set Width to **400px**. Click OK to close the Style Editor. The grid panel will have a muted yellow background in the design view.
8. In the Properties window, set property `cellpadding` to **3** and property `cellspacing` to **3**. This will create space around the nested components.

Now you'll add a second grid panel and nest it inside the first one.

1. In the Layout Components palette, select a Grid Panel component and drop it on top of the previous grid panel.
2. In its Properties window, set property `columns` to **2**, property `cellpadding` to **6**, and property `cellspacing` to **2**.
3. Click the editing box opposite property `style` and bring up the Style Editor for the nested grid panel.
4. In the Style Editor, select property Background. In the window for Background Color, provide the following RGB values.

```
rgb(232, 245, 202)
```

5. Select property Text Block. For Horizontal Alignment, select **center** from the drop down list. Click OK to close the Style Editor.



You'll place a button and a text field component inside the nested grid panel (component `gridPanel2`).

1. From the Basic Components palette, select Button and drop it on the nested grid panel. Make sure that the smaller, light-green panel is outlined in blue before you release the mouse.
2. The button's label is selected. Change its label to **Get Table**.
3. In the Properties window, change its `id` property to `doTable`.
4. From the Basic Components palette, select Text Field and drop it on top of the nested grid panel. Again, make sure that the panel is outlined in blue before you release the cursor.
5. In the Properties window for the text field, *check* property *required*.
6. In the Properties window for the text field, set property `label` to **Input a Number**. Because the field is required, Creator prepends an asterisk to the label.

The button and text fields components should be nested inside the second grid panel. Since the nested grid panel has two columns, these components are rendered side-by-side (each in its own cell in the same row). Let's configure the text field component now: you'll add an integer converter and a long range validator.

1. From the Converters Components palette, select Integer Converter and drop it on top of the text field component. The `converter` property for the text field is now set to `integerConvert1`.
2. From the Validators Components palette, select Long Range Validator and drop it on top of the text field component.
3. In the Page1 Outline view, select `longRangeValidator1`. In its Properties window, set `maximum` to **200** and `minimum` to **1**.

Now you'll add the rest of the components to the outer grid panel (component `gridPanel1`).

1. Since you've attached a validator and converter to the text field, you'll need a message component to display error messages. From the Basic Components palette, select Message and drop it on top of the outer grid panel. (Check component selection if you use the design view. Alternatively, you can drop the component on `gridPanel1` in the Page1 Outline view.)
2. In the design view, place the cursor inside the message component. Type **<Ctrl+Shift>**, hold, and left-click the mouse, releasing the cursor when it's over the text field component. The message component now displays "Message summary for `textField1`" on the design view.
3. From the Basic Components palette, select Hyperlink and drop the component on the outer grid panel.

4. Its text is selected. Change its `text` property to **Jump to End** followed by `<Enter>`. Change its `id` property to **jumpEnd**. You'll set its `url` property later.
5. From the Basic Components palette, select Static Text and drop it on the outer grid panel. In the Properties window, change its `id` property to **tableResult**. Under Data, *uncheck* property `escape`. This allows HTML tags to be interpreted.
6. From the Basic Components palette, select a second Hyperlink component. Drop it on the outer grid panel.
7. Change its text to **Top** followed by `<Enter>`. Change its `id` property to **jumpTop**.
8. From the Basic Components palette, select an Anchor component and drop it on the outer grid panel. Change its `id` property to **anchorBottom**.

The Page1 Outline view should now match the one shown in Figure 7-18 on page 299. Let's configure the two hyperlinks and connect them to the anchor components.

1. In the design view, select hyperlink component `jumpEnd`. In the Properties window, click the editing box opposite property `url`. Creator pops up the `url` property customizer, as shown in Figure 7-19.

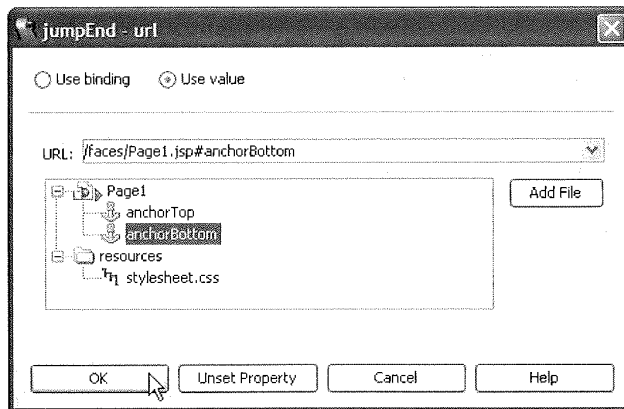


Figure 7-19 Customizer for property `url`

2. Select **anchorBottom** and click OK. This sets property `url` to  
`/faces/Page1.jsp#anchorBottom`
3. Repeat steps 1 and 2 for the `jumpTop` hyperlink component and set its `url` property to **anchorTop**.

Let's finally add the event handling code for the button component.

1. In the design view, double-click button Get Table. Creator generates the default event handler and brings up `Page1.java` in the Java source editor.
2. Supply the following code. Copy and paste from the Creator download file `FieldGuide2/Examples/WebPageDesign/snippets/layout_doTable_action.txt`. The added code is bold.

```
public String doTable_action() {
    String str = "<table border=\"2\" +
        " cellpadding=\"2\" width=\"400px\">";
    int nrows = ((Integer)textField1.getValue()).intValue();

    str = str + "<tr><th>Number</th><th>Square</th></tr>";
    for (int i = 1; i < nrows+1; i++) {
        str = str + "<tr><td>" + (i) +
            "</td><td>" + (i*i) + "</td></tr>";
    }

    str = str + "</table><p>";
    if (nrows > 0)
        tableResult.setValue(str);
    else tableResult.setValue(null);
    return null;
}
```

Method `doTable_action()` reads the value from the text field component (`textField1`) and uses it to compute a table of squares for that many numbers. The method generates the HTML code to dynamically build the table.

There's a few layout and design decisions we made that affect this project.

- First, we used grid panel to hold the components because we can't tell ahead of time how much space the static text (that holds the table of squares) will consume. By using a grid panel, Creator places all the components after each other in the next cell. If you tried to use absolute positioning you would not be able for format cleanly any component that came after the static text component.
- Second, we used anchor components since there is a possibility that the table won't fit on the page. This way, the user can easily jump to the end to view the bottom of the table. For the same reason, we added an anchor component so that the user can jump back to the top of the page.
- We configured the nested grid panel to have two columns, which holds both the button and the text field component in a single row. Then, the message component (which can display rather long messages) is in the outer grid panel in its own row.

- You can optionally center the outer grid panel using the component centering technique presented in the previous section. However, because the height of the grid panel is unknown, you cannot center it vertically. To center it horizontally, supply the following style positioning values for `gridPanel1`.

```
width: 400px; left: 50%; margin-left: -200px
```

## Deploy and Run

Deploy and run project `LayoutMadness`. Figure 7–20 shows the project running in a browser (centered). The cursor is about to click the hyperlink component to jump to the end of the page.

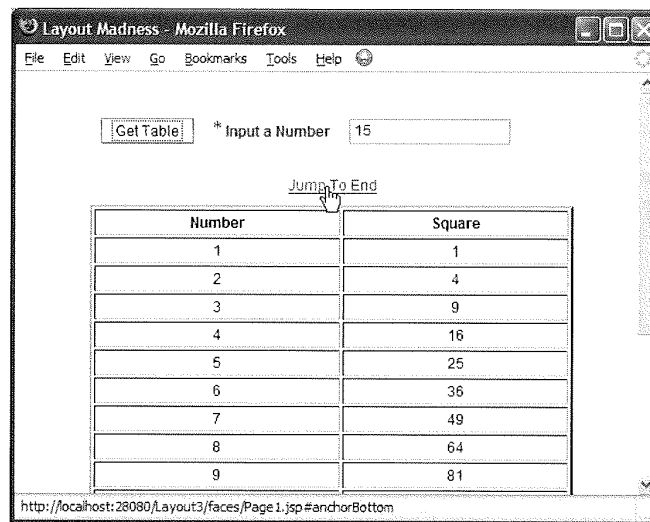


Figure 7–20 Project `LayoutMadness` running in a browser

## 7.6 Page Fragments

Page fragment components can be valuable to web page designers because they define building blocks for web pages. You can place components inside page fragments and then use the fragments within subsequent pages. Typically, page fragments hold parts of a web page that are standardized for a uni-

form look, such as images used as page headers, standard menus or navigation links, or even footers that contain copyright notices.

A page fragment is a helpful mechanism for reuse, but it does have some caveats. For one, page fragments are inserted inline into their containing document on the server. This means that a page fragment can only contain elements that are valid at the point of inclusion. As you work through the example in this section, note that page fragments are embedded in a `<div>` element (generated by Creator) and do not contain elements such as `<head>` or `<body>`, which already exist in the containing page.

To use page fragments in Creator, you first create a page fragment and then place it on the page. As an example, let's build a project for the hypothetical company called Cactus Consortium. This project has three pages: a Home (login) page, a Courses page, and a Books page. Figure 7-21 shows the layout of the Home page. The header is a page fragment that contains an image hyperlink component, the footer is a page fragment that contains a static text component, and the left menu is a page fragment that consists of a grid panel component holding navigation links (hyperlink components).

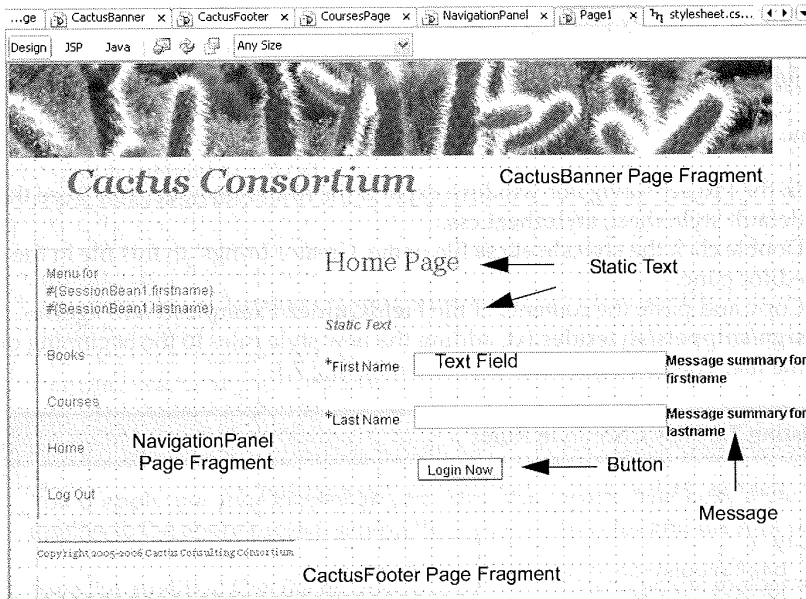


Figure 7-21 Page layout using page fragments

This web application requires users to login with their first and last names. The names are stored in session scope and adorn the navigation menu on the left. The user must login before navigating to subsequent pages.

The Books and Courses pages are prototype pages containing titles (and the page fragments for the uniform look).

### **Create a New Project**

1. From Creator's Welcome Page, select button Create New Project. Creator displays the New Project dialog. Under Categories, select Web. Under Projects, select JSF Web Application. Click Next.
2. In the New Web Application dialog under Project Name, specify **Cactus1**. Click Finish.

After initializing the project, Creator comes up in the design view of the editor pane. You can now set the title.

3. In the Properties window, select property **Title** and specify title **Cactus Consulting**. Finish by pressing **<Enter>**.

### **Modify Default Style Sheet**

Now you'll add some style rules to the default style sheet.

1. In the Project Navigator window, expand the resources node. You'll see the default style sheet, **stylesheet.css**.
2. Double click the **stylesheet.css** file name. Creator brings up this file in the editor pane.
3. Copy and paste the contents of file **FieldGuide2/Examples/WebPageDesign/snippets/stylerrules.txt**, adding the new style rules to the beginning of the file. The new style rules are shown Listing 7.1.

---

#### **Listing 7.1** New CSS Style Rules

---

```
/* Custom Style Rules for Cactus Consortium */

body {
    background-color: rgb(230,230,200);
    color: olive;
}
```

---

**Listing 7.1** New CSS Style Rules (*continued*)

---

```
.headerStyle {
    font-family: Georgia,'Times New Roman',times,serif;
    font-size: 200%
}

.footerStyle {
    font-family: Georgia,'Times New Roman',times,serif;
    border-top-color: olive;
    border-top-style: solid;
    border-top-width: 1px;
    font-size: 75%
}

.bannerStyle {
    font-family: Georgia,'Times New Roman',times,serif;
    font-size: 24pt;
    font-style: italic;
    font-weight: bold
}

td, th {
    padding-left: .5em;
    padding-top: 1em;
    padding-bottom: 1em;
    padding-right: 1em;
}

.tableStyle {
    font-family: Georgia,'Times New Roman',times,serif;
    width: 200px;
    border-left-color: olive;
    border-left-style: solid;
    border-left-width: 1px;
}
```

---

You'll apply the style classes as you build the project. The `body` style rule applies to the entire project, setting the background color and the font color.

4. Save the modified CSS file by selecting the Save All icon on the toolbar and close file `stylesheet.css` (click the small x on the `stylesheet.css` tab). Note that Page1 now has a new background color.

## Use the Gray Theme

The image and background colors for this application look better with the more neutral gray theme components.

1. In the Projects view, expand the Themes node, right-click Gray Theme and select **Set as Current Theme**.
2. After you restart the application server, Clean and Build the project to have the new theme take effect.

## Add SessionBean1 Properties

This application displays the user's first and last names in different places (both in the navigation panel and the home page display). It also keeps track of whether the user is logged in or not. Since the application must save these values for each user session, you store all three variables in session scope. Program data scope is covered in detail in Chapter 6 (see "Scope of Web Applications" on page 224). Here's how to add these three properties (first name, last name, and login status) to SessionBean1, which puts the data in session scope.

1. In the Projects window, right-click Session Bean and select **Add > Property**. Creator pops up the New Property Pattern dialog.
2. For Name specify **firstname**, for Type specify **String**, and for Mode, specify the default **Read/Write**, as shown in Figure 7-22.
3. Click OK.
4. Repeat steps 1 through 3 and add property `lastname` to SessionBean1.
5. Add property `loggedIn` to SessionBean1. For this property, specify Name **loggedIn**, Type is **Boolean** (with an uppercase 'B'), and Mode is **Read/Write**. Click OK.

When you add properties to SessionBean1, Creator generates code for the data field, as well as the getters and setters you need to access the data. Now you'll supply initialization code that you'll add to the `SessionBean1()` constructor.

1. In the Projects window, double-click Session Bean to bring **SessionBean1.java** up in the Java source editor.
2. Find the `SessionBean1()` constructor and add the following code after the comment, as shown. Copy and paste from the Creator download file



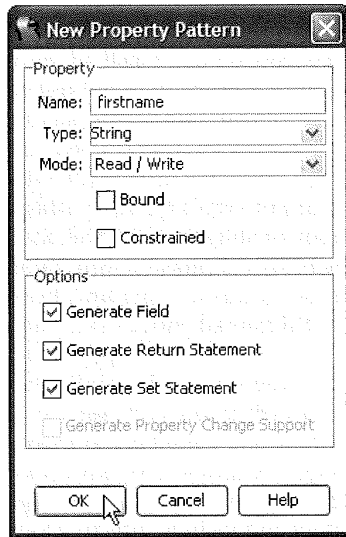


Figure 7-22 New Property Pattern dialog

FieldGuide2/Examples/WebPageDesign/snippets/cactus1\_session\_init.txt.  
The added code is bold.

```
public SessionBean1() {
    // Creator-managed Component Initialization (folded)
    // TODO: Add your own initialization code here (optional)
    firstname = "";
    lastname = "";
    loggedIn = new Boolean(false);
}
```

## Banner Page Fragment

Figure 7-21 on page 305 shows the general layout of the home page, **Page1.jsp**. The first step is to create the banner page fragment, **CactusBanner.jspf**, placed across the top portion of the page.

1. Bring up Page1 in the design editor.
2. From the Layout Components palette, select Page Fragment Box and drag it over to the top-left corner of the page. Creator pops up the Select Page Fragment dialog.
3. Click Create New Page Fragment. Creator displays the Create Page Fragment dialog.

- Specify Name **CactusBanner** and click OK. When you return to the Select Page Fragment dialog, **CactusBanner.jspf** appears in the selection window, as shown in Figure 7–23.

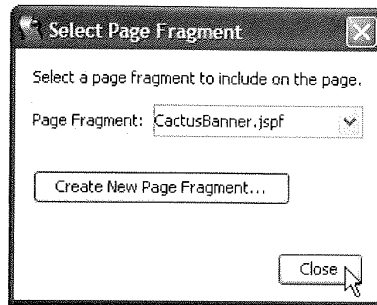


Figure 7–23 Select Page Fragment dialog

- Click Close. Creator displays the (empty) CactusBanner page fragment in the design view and adds a `<div>` component and include directive to the Page1 Outline view.

Now you'll add components to the CactusBanner page fragment, as shown in Figure 7–24.

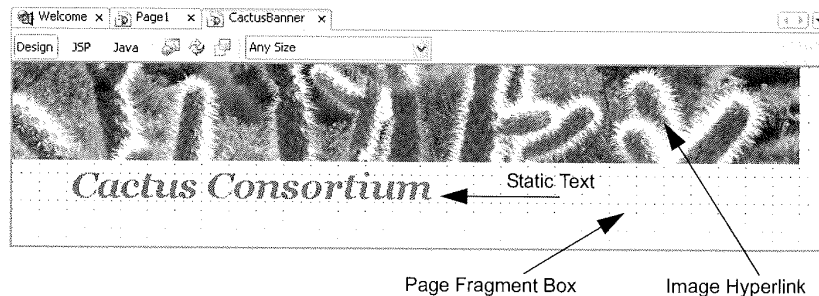


Figure 7–24 CactusBanner Page Fragment

- In the Page1 design view, double-click the CactusBanner page fragment to bring up the design view for editing the page fragment.
- Creator sets a page fragment's default size to 400px (width) by 200px (height). The white area in the design view indicates the page fragment's boundary.
- In the Properties view, change the Height to **130px** and the Width to **700px**.

Place an image hyperlink on the page fragment.

1. From the Basic Components palette, select Image Hyperlink and place it in the top-left of the design view. Its top and left position parameters should both be zero. (Hold the mouse cursor over the `style` property in the Properties view to check its value.)
2. In the Properties window, click the editing box opposite property `imageUrl`. Creator pops up a custom property editor.
3. Click Add File, navigate to your Creator download directory, and select file **FieldGuide2/Examples/WebPageDesign/images/cactus\_banner.JPG**.
4. Select Add File. Creator copies the file to your project's resources directory. Make sure **cactus\_banner.JPG** is selected and click OK. The image appears in the design view.
5. In the Properties window, click the editing box opposite property `text`. Click **Unset Property** in the property editor dialog. This removes the image hyperlink's default text from the design view.
6. In the Properties window under Behavior, set the `toolTip` to **Return to Home Page**. (When the user clicks on the image, you'll navigate back to the home page.)

Place a static text component on the page fragment.

1. From the Basic Components palette, select Static Text and place it inside the page fragment under the image on the design canvas.
2. It will be selected. Type in the text **Cactus Consortium** followed by `<Enter>`.
3. In the Properties window opposite property `styleClass`, specify **banner-Style**. (This is one of the styles rules you added earlier to the project's style sheet.) The text now appears in a larger italic font and its color is olive.
4. Select the Save All icon on the toolbar to save these changes to your project.

### ***Navigation Page Fragment***

This project uses a grid panel to hold hyperlink components for navigation. You'll put this in a separate page fragment, **NavigationPanel.jspf**.

1. Return to the Page1 design view by selecting the Page1 tab above the editing pane.
2. From the Layout Components palette, select Page Fragment Box and drag it over to the left side of the page under the CactusBanner page fragment. Creator pops up the Select Page Fragment dialog.
3. Click Create New Page Fragment. Creator displays the Create Page Fragment dialog.

4. Specify Name **NavigationPanel** and click OK. When you return to the Select Page Fragment dialog, **NavigationPanel.jspf** appears in the selection window.
5. Click Close. Creator displays the (empty) NavigationPanel page fragment in the design view.

Now you'll add components to the NavigationPanel page fragment, as shown in Figure 7–25.

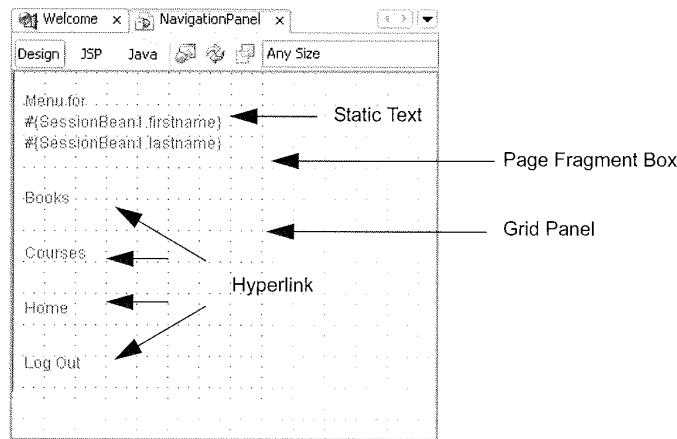


Figure 7–25 NavigationPanel Page Fragment

1. In the Page1 design view, double-click the NavigationPanel page fragment to bring it up in the design view.
2. In the Properties view, change the Height to **250px** and the Width to **200px**.

Place a grid panel to hold the navigation links.

1. From the Layout Components palette, select Grid Panel and place it on the page fragment in the top-left corner.
2. In the Properties view for property `styleClass`, specify **tableStyle**.

Add components to the grid panel.

1. From the Basic Components palette, select Static Text and drop it on the grid panel component. Make sure that the grid panel component is outlined in blue before you release the mouse.
2. The static text component is selected. Specify **Menu for #{SessionBean1.firstname} #{SessionBean1.lastname}** for the component's text property. This con-

catenates Session Bean property `firstname` and `lastname` with some text for the menu's heading.

As you add components to the grid panel, you'll see the effects of the style class you applied to the grid panel. For example, Creator wraps the text onto multiple lines instead of stretching the grid panel component because its width is fixed at 200 pixels. Also, the grid panel's cells have a generous margin because of the style rules applied to HTML elements `<th>` and `<td>` (Creator's grid panel is rendered with an HTML `<table>` element). Finally, the grid panel has a solid, 1px olive border on its left margin, which lengthens as you add components.

1. In the Properties window for the static text component, change the `id` property to `leftHeader`.
2. From the Basic Component palette, select Hyperlink and drop it on the grid panel component. (Again, make sure the grid panel is outlined in blue.)
3. Specify `Books` for its `text` property.
4. In the Properties window, change its `id` property to `booksPage`.
5. Repeat steps 4 through 6 to add a hyperlink component with `text Courses` and `id coursesPage`.
6. Repeat steps 4 through 6 to add a hyperlink component with `text Home` and `id homePage`.
7. Finally, add a hyperlink component with `text Log Out` and `id logout`.
8. Select the Save All icon on the toolbar to save these changes.

You'll specify the navigation for this project after you've added the Books and Courses pages.

### ***CactusFooter Page Fragment***

Each page in this project has a footer with a copyright designation. This information goes in its own page fragment, `CactusFooter.jspf`.

1. Return to the Page1 design view.
2. From the Layout Components palette, select Page Fragment Box and drag it over to the bottom-left of the design view under the NavigationPanel page fragment. Creator pops up the Select Page Fragment dialog.
3. Click Create New Page Fragment. Creator displays the Create Page Fragment dialog.
4. Specify Name `CactusFooter` and click OK. When you return to the Select Page Fragment dialog, `CactusFooter.jspf` appears in the selection window.
5. Click Close. Creator displays the (empty) `CactusFooter` page fragment in the design view.

Now you'll configure the CactusFooter page fragment, as shown in Figure 7-26.

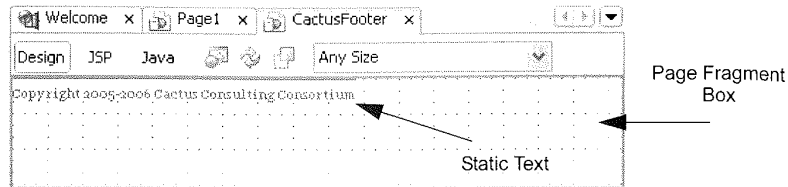


Figure 7-26 CactusFooter Page Fragment

1. In the Page1 design view, double-click the CactusFooter page fragment to bring it up in the design view.
2. In the Properties view, change the Height to **50px** and leave the Width at the default 400px.
3. From the Basic Components palette, select Static Text and drop it on the page fragment. Place it in the top-left corner.
4. The static text component is selected. Specify **Copyright 2005-2006 Cactus Consulting Consortium** for the component's text property.
5. In the Properties window, specify **footerStyle** for property `styleClass`. You'll see the font size shrink and a top border appear above the text.
6. Select the Save All icon on the toolbar to save the changes to your project and select the Page1 tab to return to the Page1 design view.

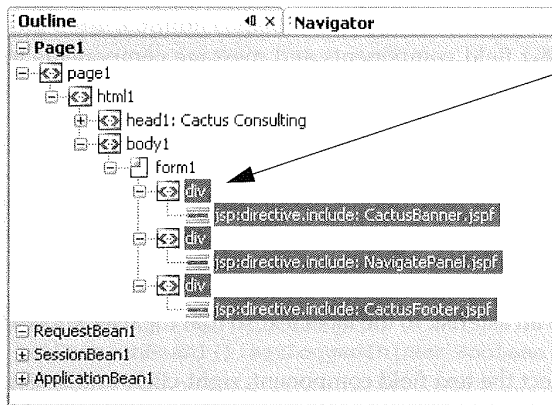
## Add Pages

You've finished creating the page fragments. Now you'll create two more pages and add the page fragments to these pages as well.

1. Close the page fragments to keep the editor pane uncluttered. For each page fragment, click the small 'x' on the tab.
2. In the Projects window, right-click the Web Pages node and select **New > Page**. Creator displays the New Page dialog.
3. For File Name specify **BooksPage** and click Finish. Creator creates the new page and brings it up in the design view. Note that it has the new default background color you configured for this project.
4. Click anywhere in the background of the design view. In the Properties window for property `Title`, specify **Cactus Consulting - Books**.
5. Repeat Steps 1 through 3 and add another page with file name **CoursesPage** and page property `Title` **Cactus Consulting - Courses**.
6. Select the Save All icon on the toolbar to save the changes to your project and select the Page1 tab to return to the Page1 design view.

You have several ways to add the page fragments to CoursesPage and BooksPage. The brute-force approach is to simply add the three page fragments one at a time, positioning each one on the page at the same location. An easier approach is to copy and paste the three page fragments as a group onto the new pages. This second approach is more efficient and the one we'll use now.

1. Bring up Page1 in the design view. In the Page1 Outline view, use **<Shift+Click>** to select all three `div` elements and their nested page fragments, as shown in Figure 7-27. All three page fragments will also be selected in the design view.



Use **<Shift+Click>** to select all three `div` elements

Figure 7-27 Selecting all three `div` elements and the nested page fragments

2. From the main menu, select **Edit > Copy** to copy the page fragments.
3. Now select the BooksPage tab on top of the editing pane to bring up BooksPage in the design view. In the BooksPage Outline view, expand nodes `page1 > html1 > body1` and select node `form1`. Right-click and select **Paste** from the context menu. Creator copies all three page fragments to the BooksPage, placing them in the equivalent positions on the page.
4. Select the CoursesPage tab and repeat the **Paste** operation on the `form1` component in the CoursesPage Outline view.

## Page-Specific Content for Page1

You've created three pages that all share the same content. Now you'll add the page-specific components to each page. Let's start with Page1.

1. Select the Page1 tab above the editing panel to bring it up in the design view. Figure 7-21 on page 305 shows the design view with the page fragment

Facebook's Exhibit No. 1003

Page 00263

boxes and the page-specific components. Note that the page has a heading text component, a second static text component, text field components to provide login information, message components, and a button.

2. From the Basic Components palette, select Static Text and drop it on the page to the right of the navigation panel.
3. Specify the text **Home Page**.
4. In the Properties window, set the `id` property to **pageHeader**.
5. In the Properties window, set the `styleClass` property to **headerStyle**. The font-size and font-family now reflect the `headerStyle` style rule.
6. From the Basic Components palette, select Static Text and drop it on the page under static text component `pageHeader`.
7. In the Properties window, set its `id` property to **instructText**.

You'll now add two text field components and message components to go with them.

1. From the Basic Components palette, select Text Field and drop it on the page under the static text components you added.
2. In the Properties window, set the component's properties as follows. Set property `id` to **firstname**, property `label` to **First Name**, property `labelLevel` to **Weak (3)**, and property `required` to **true** (it should be checked). When you set the `required` property to true and provide label text, Creator prepends an asterisk to the label text so that the user knows the field is required.
3. In the design view, select the text field component, right-click, and select Property Bindings. Creator pops up the Property Bindings dialog. Under Select bindable property, click **text Object**. Under Select binding target, expand `SessionBean1` and select **firstname**. Click Apply and Close. This binds the text field to the `SessionBean1.firstname` property, `#{SessionBean1.firstname}`.
4. From the Basic Components palette, select Text Field and drop it on the page under the text field component you just added.
5. In the Properties window, set the component's properties as follows. Set property `id` to **lastname**, property `label` to **Last Name**, property `labelLevel` to **Weak (3)**, and property `required` to **true** (it should be checked).
6. In the design view, select the text field component, right-click, and select Property Bindings. Creator pops up the Property Bindings dialog. Under Select bindable property, click **text Object**. Under Select binding target, expand `SessionBean1` and select **lastname**. Click Apply and Close. This binds the text field to the `SessionBean1.lastname` property, `#{SessionBean1.lastname}`.

You need a message component to display error messages if the user does not provide input for the text components.



1. From the Basic Components palette, select Message and place it on the page to the right of the `firstname` text field component.
2. Press and hold **<Ctrl+Shift>** and left-click the mouse inside the message component. Drag the mouse and release it over the text field component. This sets the message component's `for` property to `firstname`, the `id` of the text field component. This means that the message component will display messages from the Faces context that are designated for the text field component. The message component's display text now reads "Message summary for `firstname`."
3. Repeat Steps 1 and 2 and set the `for` property to the `lastname` text field component.

You'll use a button component to submit the login information.

1. From the Basic Components palette, select Button and drop it on the page below the text field components.
2. Change the button's `text` property to **Login Now**.
3. Change the button's `id` property to **login**.
4. In the design view, double-click the Login Now button. Creator generates a default action handler and brings up **Page1.java** in the Java source editor.
5. Add the following event handler code (add the code in bold).

```
public String login_action() {  
    getSessionBean1().setLoggedIn(new Boolean(true));  
    return null;  
}
```

This appears to be a rather terse event handler. The code sets the session bean property `loggedIn` to true. No special code is needed to check whether or not the user provided login information (validation does that for you) or specifically set the `firstname` and `lastname` session bean properties (the text field components property bindings do that for you). The only task left, then, is to set the `loggedIn` property.

When the page is rendered, it should display the logged in values stored in properties `firstname` and `lastname`. And, if the user has not logged in, it should display an instruction line requesting the user to log in. You'll put this logic in the predefined `prerender()` method.

1. **Page1.java** should still be active in the Java source editor. Locate method `prerender()`.

2. Add the following code. Copy and paste from **FieldGuide2/Examples/WebPageDesign/snippets/cactus1\_prerender.txt**. The added code is bold.

```
public void prerender() {
    // see if user is logged in
    if (getSessionBean1().getLoggedIn().booleanValue()) {
        instructText.setValue(
            "Welcome, " + getSessionBean1().getFirstname() + " "
            + getSessionBean1().getLastname());
    } else
        instructText.setValue(
            "Please login using the form below.");
}
```

### ***Page-Specific Content***

Now let's add the header text for the BooksPage and CoursesPage.

1. Select the BooksPage tab from the top of the editor pane to bring up BooksPage in the design view.
2. From the Basic Components palette, select Static Text and place it on the page at the same location as the Home Page static text component on Page1.
3. Set its text property to **Books Page**, its id property to **pageHeader**, and its styleClass property to **headerStyle**.
4. Repeat Steps 1 through 3 to add a static text component to CoursesPage. Use the text **Courses Page**, id property **pageHeader**, and styleClass **headerStyle**.
5. Make sure that the pageHeader static text component is in the same location for all three pages. You can check visually or hold the cursor over the style property in the Properties window for the static text component and verify that the position attributes for all three components are the same.

### ***Page Fragments and Navigation***

You'll now provide the navigation rules for this application. The page fragment, **NavigationPanel.jspf**, contains the hyperlink components for navigation. Since this page fragment is on *each page*, you must specify navigation rules for each page. You can certainly do this in the Navigation Editor. You'll need six cases: two navigation arrows originate from each page to specify the other two pages. However, just a slight increase in the number of pages results in a messy graph using the Page Navigation visual editor. Therefore, you're going to cheat! Basically, you want three navigation cases (both the hyperlink compo-

nents for Home and Log Out should navigate to Page1; the image hyperlink component also navigates to Page1), as follows.

1. Navigation label **Books** specifies page **BooksPage.jsp**.
2. Navigation label **Courses** specifies page **CoursesPage.jsp**.
3. Navigation label **Home** specifies page **Page1.jsp**.

As it turns out, JSF provides a sophisticated navigation handler that allows *wildcard* expressions. You'll define some basic rules and then modify the navigation configuration in the source editor to provide the wildcard expression.

1. Bring up the Page Navigation. Right-click anywhere in the background of any of the pages in the design editor and select **Page Navigation** from the context menu. You'll see the three pages in the Page Navigation editor.
2. Select **Page1.jsp** and when it enlarges, select the `booksPage` hyperlink and drag a navigation arrow to **BooksPage.jsp**.
3. Creator displays a navigation arrow. Change the case label to **Books**.
4. Select **Page1.jsp** and draw a navigation arrow from the `coursesPage` hyperlink to **CoursesPage.jsp**.
5. Change the case label to **Courses**.
6. Now select **CoursesPage.jsp** and when it enlarges, select the image hyperlink component and drag a navigation arrow to **Page1.jsp**. Change the case label to **Home**.
7. Starting with **CoursesPages.jsp** again, repeat this two more times, selecting the hyperlink components `logout` and `homePage`. For both of these navigation cases, change the case label to **Home**.

Using the navigation editor, you've configured all of the components so that their `action` property contains the correct navigation case label. Now you just have to generalize the cases so that the navigation handler goes to the correct page from any starting page. You have one rule and three cases.

1. Click the Source button in the navigation editor's toolbar to bring up **navigation.xml** in the source editor.
2. Modify the configuration file so that you have only one navigation rule with three navigation cases. Change the `<from-view-id>` element to `/*` (which matches *any* page).

3. Here is the modified file. (You can copy and paste from file **FieldGuide2/Examples/WebPageDesign/snippets/cactus1\_navigation.txt** or provide the modifications by hand.)

```
<faces-config>
  <navigation-rule>
    <from-view-id>/*</from-view-id>
    <navigation-case>
      <from-outcome>Books</from-outcome>
      <to-view-id>/BooksPage.jsp</to-view-id>
    </navigation-case>

    <navigation-case>
      <from-outcome>Courses</from-outcome>
      <to-view-id>/CoursesPage.jsp</to-view-id>
    </navigation-case>

    <navigation-case>
      <from-outcome>Home</from-outcome>
      <to-view-id>/Page1.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>
</faces-config>
```

4. Click the Save All icon on the toolbar and click the Navigation button to return to the Navigation editor. Figure 7-28 shows the Navigation view after you modify the **navigation.xml** source file.

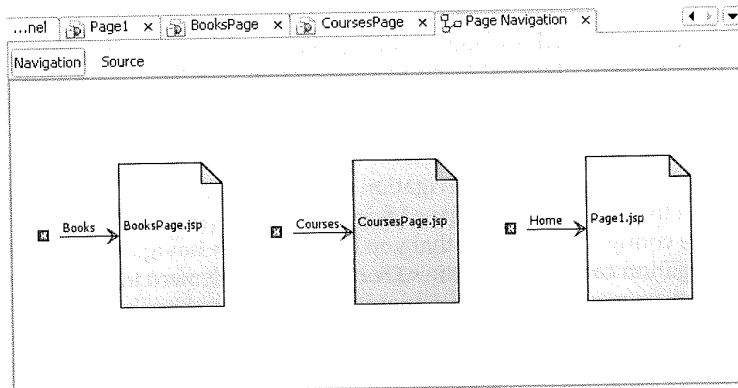


Figure 7-28 Navigation editor with source code wildcard expressions

## Logout Event Handler

The last task is to add the logout event handler code, which will reset the session bean properties. The hyperlink `logout` is in the `NavigationPanel` page fragment.

1. Bring up `NavigationPanel.jspf` page fragment in the design editor.
2. Double-click hyperlink component `logout`. Creator generates the hyperlink's event handler, `logout_action()`. Note that Creator transforms the `action` property "Home" to the correct return String value in the event handler.
3. The `logout_action()` method will reset the values for the three session bean properties. Copy and paste file `FieldGuide2/Examples/WebPageDesign/snippets/logout_action.txt`. The added code is bold.

```
public String logout_action() {  
    getSessionBean1().setFirstname("");  
    getSessionBean1().setLastname("");  
    getSessionBean1().setLoggedIn(new Boolean(false));  
    return "Home";  
}
```

## Deploy and Run

Deploy and run project `Cactus1`. Figure 7-29 shows project `Cactus1` running in the browser displaying `CoursesPage`. The image hyperlink's tooltip is visible.

## Reuse with Project Templates

Once you have the look and feel of your application defined, Creator lets you save the project as a template. Project templates promote reuse and uniformity within an organization. When you create a new project, you can select a project template as a starting point. Let's create a template from project `Cactus1`.

1. In the `Projects` window, right-click the project node `Cactus1` and select **Save Project As**. Creator pops up the `Save Project As` dialog.
2. For `Project Name` specify **CactusTemplate**. Click the `Add Project to Template List` checkbox, as shown in Figure 7-30. Click `OK`.

Now when you create a project, you can select `My Templates` and view a list of saved project templates for your new project. This is the approach we'll take with project `Cactus2` (see "Using Tab Sets and Page Fragments" on page 327.)

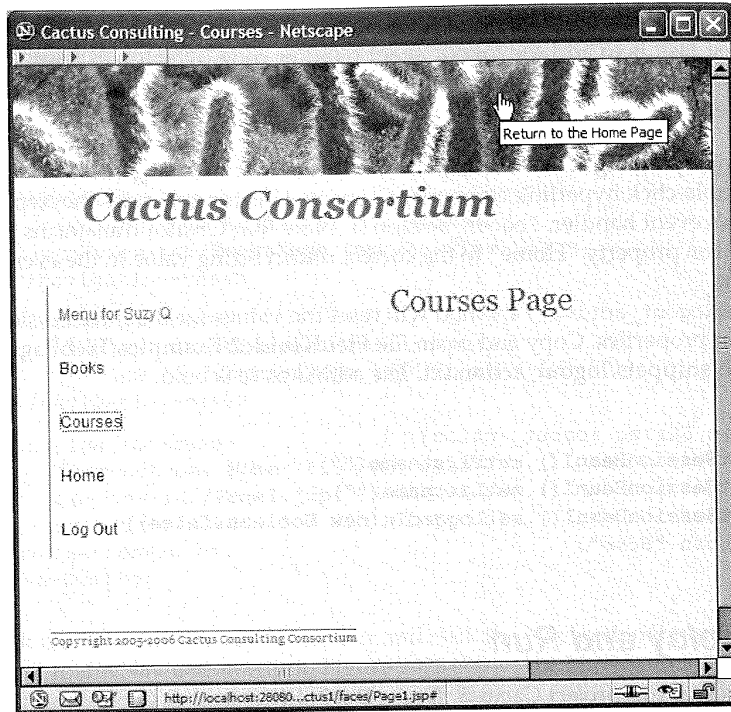


Figure 7-29 Project Cactus1 running in a browser

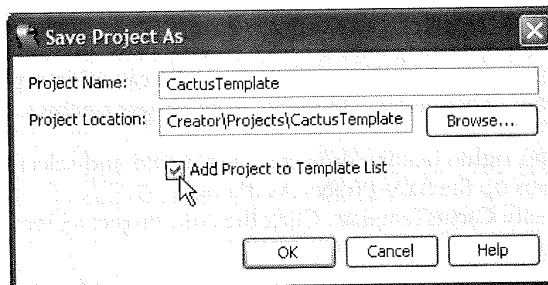


Figure 7-30 Adding a project to the Template List

## 7.7 Introducing Tab Sets

As we continue exploring Creator's components and configuration options for web applications, let's show you another useful layout component, the tab set. Many applications use tab set components for navigation and complex page management. With tab sets, you can display only those components that are relevant to the task at hand.

The tab set is a composite component. You place a tab set component on a page and then add tab components to the tab set. You have more than one way to manage an application with tab sets, however.

The simplest way places a separate tab set component on each page. If your application consists of three pages, for example, each page will hold its own distinct tab set and each tab set holds three tab components. Furthermore, each tab set has one tab that is always selected. The other two non-selected tabs act as hyperlink components and provide navigation to the other two pages where a different tab is always selected. This method is straightforward because you never have to manage the state of the tab sets. You place the components on each page with the visual design editor. Furthermore, you don't have to nest the page's components under the tab set.

A second approach is to put the tab set and tabs in a page fragment and include the page fragment on each page. Again, you place the components for each page directly on the page (you don't nest them under the tab set component). Because there is only one tab set component, you must maintain the state of its selected tab, however. Like the first approach, the tabs are used to navigate to the other pages. The advantage of using a page fragment for the tab set is that any customizing for the tab set must only be specified once.

A third approach uses a single page: the tab set and its tabs all go on one page. In this approach, you use the nested layout panel to hold the components corresponding to each tab. When the user selects a tab, the tab set renders only those components under the selected tab. Furthermore, the tab set automatically marks the selected tab, so you don't need to do anything to maintain its state. The disadvantage of this approach is that it is more difficult to design the page and the page is more complex.

We'll take you through building a project using the first approach. Then, you'll implement a version of the Cactus project (Cactus2), incorporating both a tab set and a navigation panel containing hyperlink components to perform navigation. The Cactus2 project uses the page fragment method for the tab set component. We illustrate the third approach in project TabSet3, which is included in the Creator2 download directory under **FieldGuide2/Examples/WebPageDesign/Projects/TabSet3**.

## Using Separate Tab Sets

The first example uses a separate tab set on each page. For this project, you'll create three pages. Each page contains a tab set component with three tabs. The tabs enable users to navigate to the other pages.

### Create a New Project

1. From Creator's Welcome Page, select button Create New Project. Creator displays the New Project dialog. Under Categories, select Web. Under Projects, select JSF Web Application. Click Next.
2. In the New Web Application dialog under Project Name, specify **TabSet1**. Click Finish.

After initializing the project, Creator comes up in the design view of the editor pane. You can now set the title.

3. In the Properties window, select property **Title** and specify title **TabSet1 - Page1**. Finish by pressing **<Enter>**.

### Add Components to Page1

Figure 7-31 shows the design view of Page1.

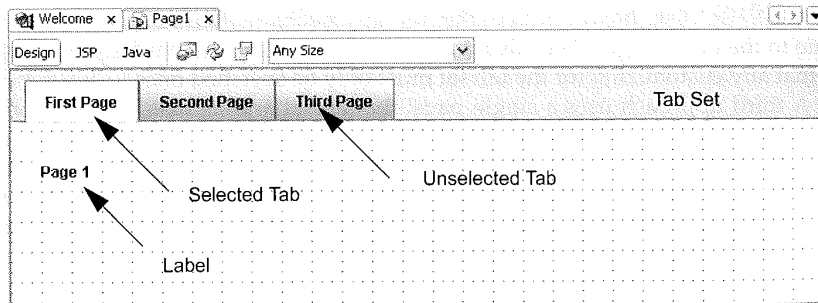


Figure 7-31 Page1 design view for project TabSet1

1. From the Layout Components palette, select Tab Set and drop it on the page. Position it in the top-left corner.
2. When you add the tab set component, Creator preconfigures a single tab nested in the tab set. Inside the tab component, Creator also preconfigures a layout panel. The tab should be selected. Change its text property to **First Page**, followed by **<Enter>**.



3. In the Page1 Outline view, select the nested layout panel, `layoutPanel1`. Right-click and select Delete from the context menu. Since you're placing a separate tab set on each page, you don't need the layout panel to hold the page's components.
4. In the Page1 Outline view, now select the tab set component. In the Properties window for the tab set component, add the property-value `;width: 100%` to the `style` property that is already configured. The background behind the tabs will expand.
5. From the Layout Components palette, expand node Tab Set and select the nested component Tab. Drop it on top of the tab set component that's on the page. Creator configures this as component `tab2`. In the Page1 Outline view, make sure that component `tab2` is at the same nesting level as `tab1`.
6. Component `tab2` should be selected. Change its `text` property to **Second Page**.
7. In the Page1 Outline view, delete the nested layout panel component (repeat the step you followed to delete the nested layout panel in the first tab).
8. From the Layout Components palette, select another Tab component and drop it on top of the tab set component. This is component `tab3`. Change its `text` property to **Third Page** and delete its nested layout panel component.
9. In the design view, click on component `tab1` (First Page) to make it selected.

#### Creator Tip

*As you configure the tab set on each page, you specify the selected tab for that page. Each page has a different tab selected. Figure 7-31 shows `tab1` selected (the Page1 configuration).*



Now you'll add a label component to the page to hold a page header.

1. From the Basic Components palette, select component Label and drop it on the page. (Drop it on the *page*, not on the tab set component.) In the Page1 Outline view, the label component should be nested under `form1`, at the same level as component `tabSet1`.
2. Set its `text` property to **Page 1**.

You'll now create a second and third page and copy and paste these components to the new pages.

1. In the Projects view, right-click the Web Pages node and select **New > Page**. Creator pops up the New Page dialog.
2. Specify File Name **Page2** and click Finish. Creator brings up Page2 in the design view.
3. Set the `Title` property to **TabSet1 - Page 2**.

- Repeat steps 1 through 3 to add a third page with file name **Page3** and Title **TabSet1 - Page 3**.

Since each page holds the same components (the tab set, three tabs, and a label) it's easier to copy and paste these components from Page1 and reconfigure them as needed for Page2 and Page3.

- Select the Page1 tab above the editor pane to bring up Page1 in the design view.
- In the Page1 Outline view, select component `tabSet1` and `label1` (use **<Shift+Click>** for multiple selection). The nested tabs will also be selected.
- From the main menu bar, select **Edit > Copy**.
- Now select the Page2 tab above the editor pane. In the Page2 Outline view, expand `page1 > html1 > body1` and select component `form1`.
- Right-click on `form1` and select **Paste** from the context menu. This copies the Page1 components to Page2, maintaining the same position and other style attributes.
- Repeat Steps 4 and 5 with Page3, pasting the components on this page.

Now you'll configure the tab set and label components for Page2 and Page3.

- Select the Page2 tab above the editor pane to bring up Page2 in the design view.
- In the design view, click on component `tab2` (Second Page) to make it selected. The tab will turn white and the other two tabs will be blue.
- Select the label component and change its `text` property to **Page 2**.
- Repeat Steps 1 through 3 for Page3. Make `tab3` (Third Page) selected and change the label's `text` property to **Page 3**.

### Configure Navigation

The tab components behave like hyperlink components because you can define action event handlers or action labels for navigation. You'll define six navigation cases: each page will have two cases that originate from a tab component and terminate in one of the other two pages.

- In the design view, right-click anywhere in the background and select Page Navigation from the context menu. Creator brings up the Navigation Editor.
- You see the three pages that you defined for this project. Select **Page1.jsp**. When it enlarges, you'll see the three tab components.
- Select `tab2`, click, and drag an arrow to **Page2.jsp**. Change the case label to **second**.
- Select **Page1.jsp** again, click `tab3`, and drag an arrow to **Page3.jsp**. Change the case label to **third**.

5. Now repeat this sequence for **Page2.jsp**. Draw an arrow from component **tab1** to **Page1.jsp**. Use case label **first**. Draw a second arrow from component **tab3** to **Page3.jsp**. Use case label **third**.
6. Finally, configure the navigation cases for **Page3.jsp**. Draw an arrow from component **tab1** to **Page1.jsp**. Use case label **first**. Draw the second arrow from component **tab2** to **Page2.jsp**. Use case label **second**. Figure 7-32 shows the Navigation Editor with all of the cases configured and **Page2.jsp** enlarged.

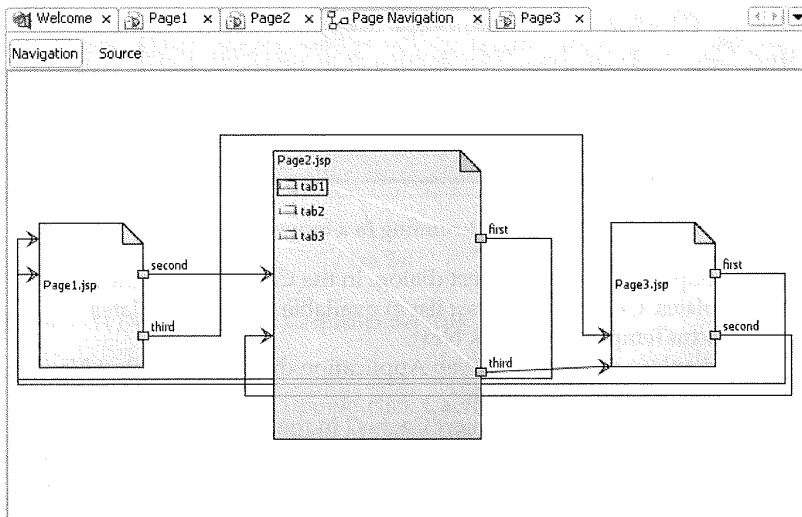


Figure 7-32 Navigation Editor for project TabSet1

## Deploy and Run

Deploy and run project TabSet1. Figure 7-33 shows Page2 in the browser and the cursor is over the third tab.

## Using Tab Sets and Page Fragments

You'll now add a tab set to the CactusBanner page fragment from project Cactus1. Instead of copying project Cactus1, you'll create a new project and use the project template you saved earlier.

1. From the Welcome page, click **Create New Project**.

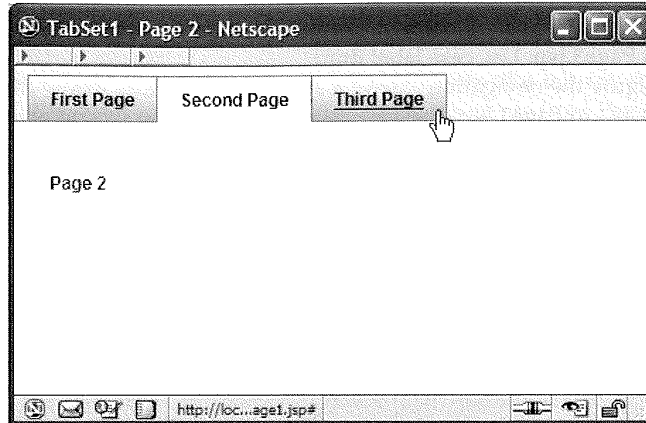


Figure 7-33 Page2 of project TabSet1 running in a browser

2. Creator pops up the New Project dialog. In the Categories window, select **My Templates**. Creator displays a list of available project templates.
3. Select **CactusTemplate** and click Next.
4. Creator displays the New JSF Web Application dialog. For Project Name specify **Cactus2** and click Finish.
5. Creator brings up Page1 of project Cactus2 in the design view. Click anywhere in the background of the design canvas of the Cactus2 project. In the Properties window, change the page's **Title** property to **Cactus Consulting 2**.

Let's add another style rule to **stylesheet.css** to make the tab set blend with the pages better.

1. In the Projects window, expand the **Web Pages > resources** node and double-click **stylesheet.css** to bring it up in the Style Editor.
2. Add the following style rule for TabGrp.

```
.TabGrp {
    background-color: rgb(230,230,200);
}
```



#### Creator Tip

*The standard Themes uses style rule TabGrp to apply styles to tab sets. By customizing this rule, you can change the background color of the page behind the tabs. The default color is a gradient gray for the Gray Theme.*

## Add Tab Set and Tabs to CactusBanner

First, let's add components to the CactusBanner page fragment. Figure 7-34 shows the design view for the CactusBanner page fragment with the tab set component added.

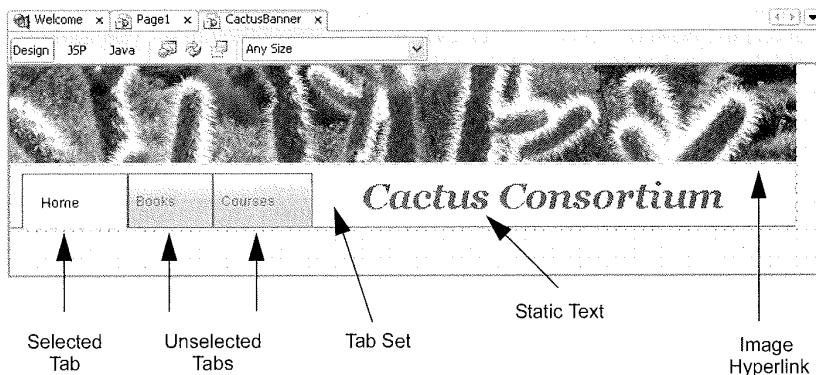


Figure 7-34 Design view for CactusBanner.jspf

1. In the Projects window, double-click **CactusBanner.jspf** to bring up the CactusBanner page fragment in the design view.
2. Click in the blue area to select the page fragment.
3. In the Properties view, change the page fragment's height to **150px**.
4. Select the Cactus Consortium static text component and move it over to the right to make room for the tab set component.
5. From the Layout Components palette, select Tab Set and drop it on the page fragment below the image hyperlink component.
6. The first tab is selected. Change its text to **Home**.
7. In the Outline view, select the nested layout panel, `layoutPanel1`, right-click and select **Delete** from the context menu.
8. In the Outline view, select the nested tab component, `tab1`. In the Properties window for the tab, change its `id` property to **homeTab**.

### Creator Tip

The selected property of the tab set component takes the `id` (passed as a String) of the tab that's selected. Renaming the component's `id` with meaningful names makes your code more readable.



9. Now select component `tabSet1` in the Outline view. In the Properties window for the tab set, check the property `mini`. This changes the appearance of the tab set and makes the tabs smaller.

Add two more tabs to the tab set.

1. From the Layout Components palette, expand the Tab Set component and select the nested Tab component. Drop it on top of the tab set you added. (Make sure that the new tab is at the same nesting level as component `homeTab`.)
2. Change the tab's `text` property to **Books**.
3. Change the tab's `id` property to **booksTab**.
4. Delete the nested layout panel under tab `booksTab`.
5. Repeat this and add another tab with `text` **Courses** and `id` **coursesTab**. Delete the nested layout panel under tab `coursesTab`.
6. Reposition the tab set component so that it is within the page fragment boundary (the white area) and that its left edge is flush with the fragment boundary (the `left` style attribute should be 0).
7. In the CactusBanner Outline view, select the Cactus Consortium static text component, drag it up to the `f:subview` component, and drop it on top of the subview.

The static text component is now listed *after* the tab set component in the CactusBanner Outline view. This makes the static text component render *on top of* the tab set component. When you moved the component in the Outline view, its `style` property was cleared. The component now appears in the design view in the upper-left corner (on top of the image).

8. In the design view, move the static text component back (next to) the tab set component, placing it to the right of the third tab.

Add `style` and `styleClass` attributes to the tab set component.

1. Select the tab set component. In the Properties view for property `style`, add attribute `; width: 696px` to the end. This aligns the width of the tab set component with the image hyperlink component.
2. In the tab set Properties view for property `styleClass`, specify **TabGrp**.
3. In the tab set Properties view opposite property `selected`, click the check mark for the drop down list corresponding to the tabs. Choose the first, blank entry. Even though the property sheet will display Home (`homeTab`), property `selected` *should not appear in bold*.

**Creator Tip**

*If the tab set's selected property is bold, then Creator has generated a tag in the JSP file to set it. Because you'll set the tab set's selected property in each enclosing page's prerender () method, you must not generate the corresponding JSP tag. The JSP code executes after the prerender () method, rendering the method ineffective.*



4. Select the Save All icon on the toolbar to save these changes.

### Setting the Currently Selected Tab

As you've seen from working with the tab set component in the first tab set example, the tab set displays tabs and renders them as either "selected" or "not selected." A selected tab (there can only be one selected tab) has a contrasting color and its link is inactive. It is the current tab. A non-selected tab has an active link. In this example, the action associated with clicking a non-selected tab causes page navigation. When the new page is rendered, the tab set must reflect the newly selected tab.

The most straightforward way to maintain the state of the currently selected tab is to set it in the enclosing page's prerender () method. Setting the selected tab in each page ensures that the correct tab is selected even when navigation to a new page happens through the hyperlink components in the Navigation-Panel page fragment or the image hyperlink.

As mentioned in the previous page's Creator Tip, the only caveat is you must ensure that Creator does not generate JSP code to set the tab set's selected property, since this will take precedence over the prerender () code.

1. In the Projects window under Web Pages, double-click **Page1.jsp** to bring it up in the design view.
2. Select the Java button in the editing toolbar to edit the Java source.
3. Add the following code to method prerender (). Copy and paste from **FieldGuide2/Examples/WebPageDesign/snippets/cactus2\_page1\_tabset.txt**. The added code is bold.

```
public void prerender() {
    CactusBanner cactusBanner = (CactusBanner) getBean(
        "CactusBanner");
    cactusBanner.getTabSet1().setSelected("homeTab");
    // see if user is logged in
    . . .
}
```

You'll add the same code to the `prerender()` methods in **Courses.java** (with `String "coursesTab"`) and **Books.java** (with `String "booksTab"`).

1. Bring up **Courses.java** in the Java source editor. Add the same code to the `prerender()` method.
2. Change the tab set's `setSelected()` argument to `"coursesTab"`.
3. Bring up **Books.java** in the Java source editor. Add the same code to the `prerender()` method and change the tab set's `setSelected()` argument to `"booksTab"`.
4. Select the Save All icon on the toolbar to save these changes.

### **Configure Action Method for Tabs**

The navigation rules have already been configured for this project. Because you used navigation wildcard notation in project `Cactus1`, these rules will apply to the tab set component as well. You do not need to make any adjustments to the navigation rules. However, you do need to specify the return string for each tab's action method. To do this, you'll generate action methods through the IDE and provide the code for the action event handler.

1. Bring up `CactusBanner` in the design view.
2. Double-click tab component `homeTab`. Creator generates the default `homeTab_action()` event handler and brings up **CactusBanner.java** in the Java source editor.
3. Replace the `return null` with `return "Home"` as shown.

```
public String homeTab_action() {
    // TODO: Replace with your code
    return "Home";
}
```

4. Return to the design view (click Design in the editor toolbar) and repeat steps 2 and 3 for tab component `booksTab`. Replace the `return null` with `return "Books"`.

```
public String booksTab_action() {
    // TODO: Replace with your code
    return "Books";
}
```



5. Return to the design view and repeat steps 2 and 3 for tab component `coursesTab`. Replace the `return null` with `return "Courses"`.

```
public String coursesTab_action() {  
    // TODO: Replace with your code  
    return "Courses";  
}
```

6. Make sure that the tab set's `selected` property is not set in the Properties view. If it's bold, select the first (blank) entry in the drop down list for property `selected`. The property name should no longer be bold.
7. Click the Save All icon on the toolbar to save these changes.

### **Check Pages with Modified CactusBanner**

Check and adjust the placement of the page fragments on each page. Once you've done this, deploy and run the application.

#### **Creator Tip**

---

*Since the grid panel contains a vertical line on the left border, align this border with the first tab's left edge. This creates a pleasing visual connection between the grid panel component and the tab set component. The easiest way to adjust the page is to simultaneously select the `NavigationPanel` and `CactusFooter` page fragments. Then, while holding down the Shift key, move the fragments down and to the left to make the alignment with the tab set component. By moving both fragments together, you keep their relative position constant.*

---



When you run the application, check to make sure the tab set component reflects the correct page navigation, whether you use the hyperlink components or the tab set component for navigation. Figure 7-35 shows project `Cactus2` running in a browser with page `BooksPage` rendered.

## **7.8 Key Point Summary**

This chapter explores some of Creator's tools and components that web designers use to compose web pages and visually organize their projects.

- Creator's visual editor helps you compose web pages by providing component dragging, dropping, and page positioning.

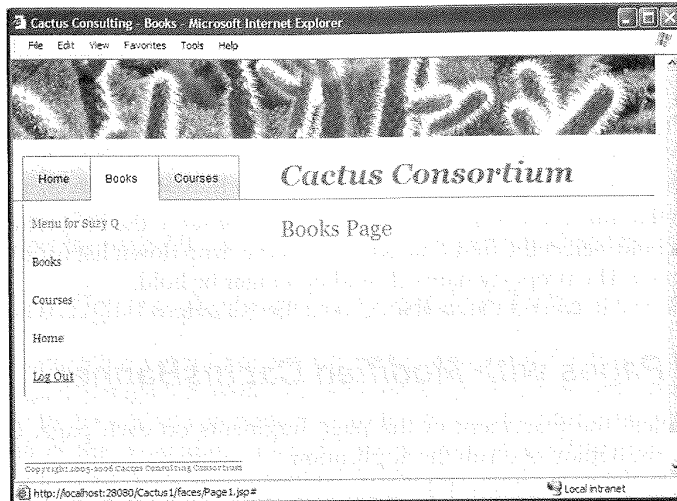


Figure 7-35 Project Cactus2 running in a browser

- By default the visual editor displays a grid that helps you align components. You can turn off the grid or change its size using the **Tools > Options > Visual Designer** menu. You can temporarily disable grid alignment by repositioning the component while holding down the Shift key.
- You can select multiple components by dragging a mouse around the target components, enclosing them in a box. You can also use **<Shift+Click>** to add components to those that are already selected.
- To align components with one another, select the target components, position the mouse over the reference component, right-click, and select **Align**. This brings up the Align context menu with choices for alignment.
- For horizontal alignment options, select Left, Center, or Right. For vertical alignment options, select Top, Middle, Bottom.
- Creator's Basic, Layout, and Composite components are rendered using *themes*. A theme is a bundled set of cascading style sheets, JavaScript files, and images that apply to the components and page. The available themes are listed in the Projects window under node Themes.
- To change the current theme, right-click a new theme selection in the Projects window and select **Set As Current Theme**. To make the new theme take effect for deployment, stop the application server and clean and rebuild the project.
- You can control the look of a component by modifying its *style* property. Property *style* accepts property-value pairs to control style attributes such as color, background color, font characteristics and page position.

Facebook's Exhibit No. 1003

Page 00282

- Creator provides a style editor to manipulate a component's `style` property. To use the style editor, click the editing box opposite property `style`.
- In addition to the `style` property, Creator also uses Cascading Style Sheets (CSS) to control the look of its components and pages. You can add style classes to a project's default style sheet, `stylesheet.css`. You can also provide your own style sheet.
- To edit the default style sheet, double-click file `stylesheet.css` in the Projects window under **Web Pages > resources**. Creator brings up the style sheet in the Style Sheet Editor.
- Apply one or more style classes to a component by specifying them in a comma separated list for the component's `styleClass` property.
- Using a style sheet with the `styleClass` property is easier than configuring a component's `style` property to achieve a uniform look.
- Creator provides several components that provide grouping and layout capabilities. The layout panel component provides the option of using a grid layout which lets you use the design view to easily position nested components.
- The grid panel component provides a cell for each nested component. Creator places each component in the next available cell. The default number of columns is one, but you can change this value in the grid panel's Properties window. The grid panel is especially useful when you want to include one or more components after a component with indeterminate sizing (such as a table that can have any number of rows).
- The anchor and hyperlink components control page scrolling. The hyperlink component jumps to a spot on the page marked by the anchor component. Jumping to an anchor does not perform a page request.
- Page Fragments are page building blocks for web applications. Typically, you use page fragments to hold parts of your web page that you'd like to standardize for a uniform look, such as page headers, standard menus, or footers.
- You can save a project as a template. When you create a new project, you can then select a saved template as a starting point.
- You can use wildcards in page navigation rules. Select the Source button in the Page Navigation editor and modify the `navigation.xml` source file directly. Wildcards simplify navigation rules by reducing the number of navigation cases you define.
- The tab set is a composite component that contains nested tabs. Under each nested tab is a layout panel component. Tabs are similar to hyperlinks in that you can specify action event handling code as well as navigation strings.
- You can use tab sets with page fragments or put a tab set and its nested tabs all on one page. You may also put a separate tab set on its own page.
- A tab set's `selected` property holds the component `id` (as a String) of the tab that is currently selected.

# ACCESSING WEB SERVICES

## Topics in This Chapter

- Google Web Services
- Adding and Testing a Web Service
- Nested Components
- Exceptions and Error Handling
- Message and Message Group Components
- Hyperlink Component
- String Validation
- Table Component and Data Providers
- Saving and Restoring Page Data

# Chapter 10

**W**eb services are software APIs that are accessible over a network in a heterogeneous environment. This network accessibility is achieved with a set of XML-based open standards such as the Web Services Description Language (WSDL), the Simple Object Access Protocol (SOAP), and Universal Description, Discovery, and Integration (UDDI). Both web service providers and clients use these standards to define, publish, and access web services.

Creator's default application server provides support for web services. Pre-installed with Creator is a Google Web Service client, which appears in the Servers window under node **Web Services > Samples > GoogleSearch**. The Google Web Service APIs provide a SOAP interface to search Google's index, accessing information and web pages from its cache. With SOAP and WSDL, Google enables clients to access these services in a variety of programming environments (including, of course, Java).

This chapter shows you how to create an application that uses the Google Web Service API. Then, you'll enhance it. After creating a project that uses web services, you'll (hopefully) exclaim, "Is that all?" because the steps are fairly simple. And that's the way technology should be when industry-wide standards are adopted. You'll see that once we drag and drop the web service onto the design canvas in Creator, we spend most of our time showing you elaborate ways to manipulate and display the data that Google returns.

## 10.1 Google Web Services

We've divided this example into several projects that incrementally build on features of the previous project. With each increment, you'll start with the project you previously built. Alternatively, you can pull up any of the projects from the Creator download and make changes to these projects.

### Note



*You must register with Google before using their web service. You'll also want to download the Google Web APIs developer's kit since it has additional documentation. Registration is free and painless. Once you register, Google will email you a key, which is required for access to their service. The Google Web Service URL is at <http://www.google.com/apis/>.*

Let's look at a summary of the projects you'll be building. In this first version, you'll submit a search query to Google's search service and display just the first result that's returned. This is equivalent to the "I'm Feeling Lucky" submit button on the Google web site. In subsequent versions, you'll add validation for the query text field and display (up to) all ten results returned. Finally, you'll add pagination so that you can obtain and display subsequent groups of results.

Figure 10-1 shows Creator's design canvas with the components you'll add for project **Google1**. The image component holds Google's recognizable logo, a button component initiates the search, and a text field holds the search string. For the results display, static text component `timeCount` displays the search time and results count, the hyperlink component and an embedded static text component display the target URL, and static text component `snippet` displays the URL's "snippet" description. You'll bind these components to the various properties of the result object and you'll nest them inside a grid panel container to more easily manipulate them as a group.

### Create a New Project

1. From Creator's Welcome Page, select button Create New Project. From the New Project dialog, under Categories select **Web** and under Projects select **JSF Web Application**. Click Next.
2. In the New Web Application dialog, specify **Google1** for Project Name and click Finish.

After creating the project, Creator comes up in the design view of the editor pane. You can now set the title.

Facebook's Exhibit No. 1003

Page 00286

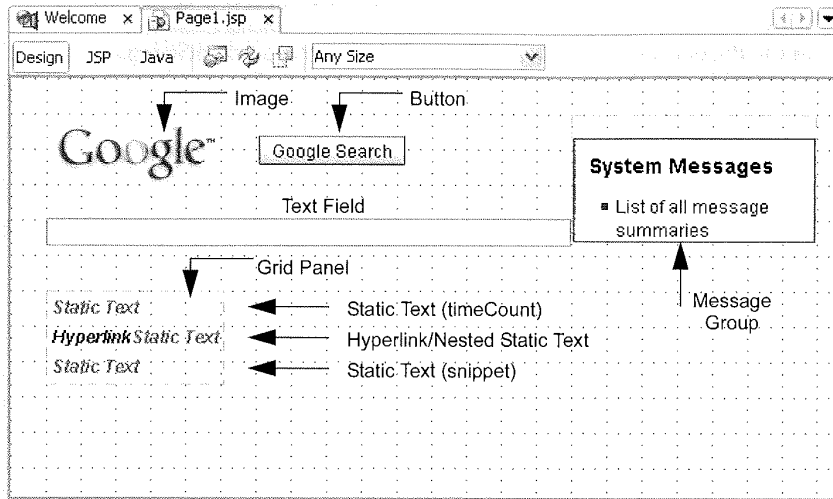


Figure 10-1 Creator's design canvas view showing project Google1's components

3. Select **Title** in the Properties window and type in the text **Google Search 1**. Finish by pressing **<Enter>**.

## Add the Google Logo

It's a nice touch to include the Google logo when building a web application with Google's search service. To add an image on the page you simply drop the image's file directly on the design canvas.

- Using a file utility program (such as Windows Explorer), drag file **FieldGuide2/Examples/WebServices/images/Logo\_40wht.gif** to the Design view and drop it on the canvas (consult Figure 10-1 for positioning).

This places the Google logo on the page (using the image component) and copies the file into your project's resources directory. You can see this in the Projects window (select **Google1 > Web Pages > resources**).

## Add a Text Field Component

You'll need a text field component to obtain the user's search query.

1. In the Basic Components palette, select component **Text Field** and drag it onto the design canvas. Place it below the Google logo.

2. Make sure it is selected and stretch it so that it's approximately 15 grid units wide.
3. In the Properties window, change its `id` property to `searchString`.
4. To provide a tooltip for this text field, type the text `Type in a search string` followed by `<Enter>` for the `toolTip` property (under Behavior).

### Add a Button Component

In this application, a button component initiates a search using Google's Web Service API.

1. In the Basic Components palette, select Button and drag it onto the design canvas. Place it to the right of the Google logo.
2. Make sure the button is still selected. Type in the text `Google Search` followed by `<Enter>`. Creator resizes the button to accommodate the longer text string, which now appears inside the button on the design canvas. This sets the button's `text` property.
3. In the Properties window, change the button's `id` attribute to `search`.
4. To provide a tooltip for the button, edit its `toolTip` property in the Properties window (under Behavior). Type in the text `Search Google for the Search String` followed by `<Enter>`.
5. Align the Google logo with the button. Select the button component in the design canvas. While pressing `<Shift>`, move the mouse to the Google logo and left-click, which simultaneously selects the logo.
6. With both components selected, make sure the mouse is over the logo and right-click. Select option `Align > Middle` from the context menu. The button will move so that it is centered vertically in relation to the logo.



#### Creator Tip

*Creator provides several ways to help you place components on the canvas. By default, components "snap to the grid lines." To adjust components without regard to the grid lines, hold the Shift key as you move the component on the canvas. You can select multiple components using the Shift key while you left-click with the mouse. Then it is possible to move the selected components as a group. Finally, make adjustments between components by selecting them and right-clicking the mouse inside the "anchor" component, as described above. The Align menu selection has several options that you can use to manipulate the selected components.*



## Add the Google Web Services

Since the Google Web Service client is preinstalled for you, you can simply drag and drop this component to add it to your project.

1. In the Servers window, expand the **Web Services > Samples > Google-Search** nodes.
2. Drag the **doGoogleSearch** node and drop it anywhere on the editor pane. Nothing appears in the design canvas; however, you will see **googleSearchClient1** and **googleSearchDoGoogleSearch1** in the Outline view for Page1, as shown in Figure 10-5 on page 460.

## Adding a Web Service to the IDE

The Creator installation process configures the Google web service as well as other web services listed under the **Web Services > Samples** node in the Servers window. Creator also provides a way to *add* a web service to the Services view. For example, here are the steps to load the Google web service client into Creator manually.

### Creator Tip

---

*Since the GoogleSearch web service client is included with Creator, you do not need to follow these steps to access it from a Creator project. We include this procedure in case you'd like to add a web service that has not been pre-configured with Creator.*

---



Before you can add a web service, you must provide the location (URL) of its Web Services Description Language (WSDL) page. This is the information that describes the particular web service's API.

We're going to step through the process to add the Google Search to Creator as an example. Once you've determined the Web Service URL, you can use these steps to add any web service.

1. Go to the Servers view.
2. Right-click on Web Services.
3. Select Add Web Service. Creator pops up the Add Web Service dialog.
4. In the URL field at the top, supply the URL of the WSDL file of the target web service. Here is the URL for Google's WSDL file.

<http://api.google.com/GoogleSearch.wsdl>

This is the location of the WSDL (Web Services Description Language) file for the Google Search Service.

5. Click Get Web Service Information. The Google web service API appears in the Web Service Information window, as shown in Figure 10–2.

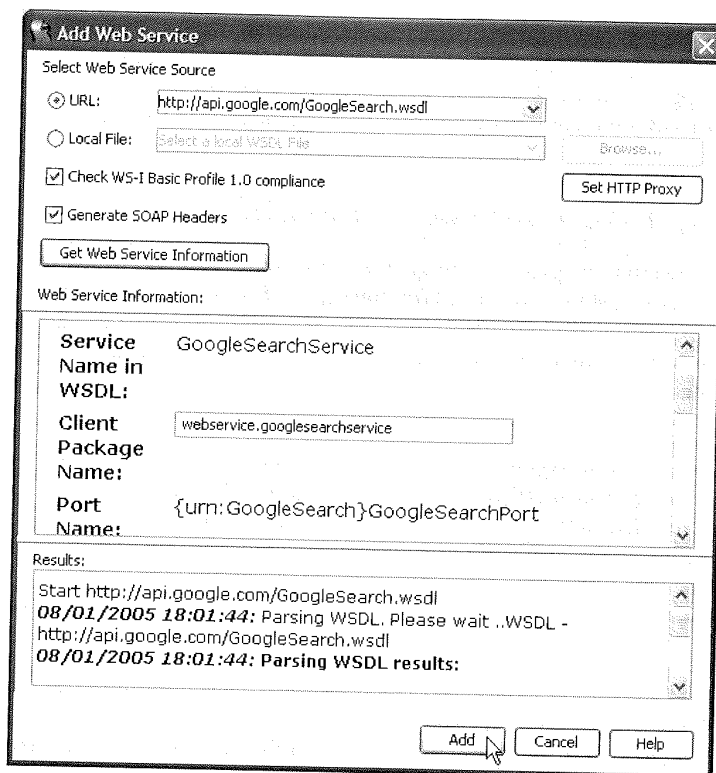


Figure 10–2 Add Web Service dialog

6. Scroll through the information in the Web Service Information window. Creator displays detailed information about the web service, including its name, the package name, port name, port display name, port address, and the methods. Information on the methods include the method names, the parameters and their types, and the return type. We'll examine the search method `doGoogleSearch()` in more detail later in this chapter.
7. Click Add. The name GoogleSearch appears under the Web Services node in the Servers window.

Once a web service is listed in the Servers view, you can select it and add it to your Creator project (as you did with GoogleSearch).

## Add Search Result Properties to Page1

The trick to easily manipulating the results of the Google search web service is make the return object accessible through the IDE. The search web service returns an object (GoogleSearchResult) that contains information that you'll access. Object GoogleSearchResult also includes an object array (resultElements) that contains specific information on the search result sites. You'll make both of these objects properties, then add an object array data provider to bind to the components on the page.

1. In the Projects window, expand the **Sources Packages > google1** nodes.
2. Right-click on **Page1.java** and select **Add > Property**. Creator pops up the New Property Pattern dialog, as shown in Figure 10-3.

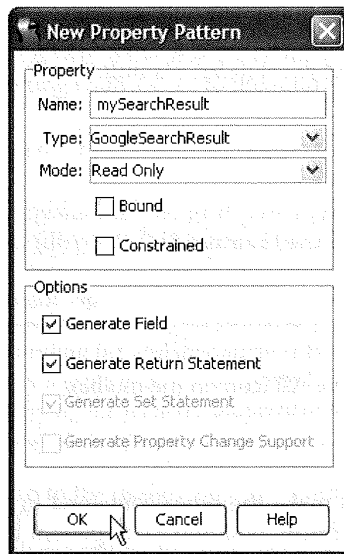


Figure 10-3 Adding property mySearchResult to Page1.java

3. For Name, specify **mySearchResult**, for Type, use **GoogleSearchResult**, and for Mode select **Read Only**. Name and Type are case sensitive, so be sure to match the capitalizations. Click OK.

4. Add a second property to Page1. For Name, specify **resultArray**, for Type specify **ResultElement[]**, and for Mode select **Read Only**.



#### Creator Tip

*Make sure that property `resultArray` is **NOT** an indexed property, but that Type includes the array notation `[]`.*

5. Select the Java label in the editing toolbar to bring up **Page1.java** in the Java source editor.
6. Scroll to the end of the file where you'll see the generated code that added properties `mySearchResult` and `resultArray`. You'll see syntax errors. Fix these using the shortcut **<Alt+Shift+F>** (fix imports), or **<Alt+Shift+I>** (import shortcut). (You must put the cursor anywhere inside **GoogleSearchResult** and **ResultElement[]** before using the **<Alt+Shift+I>** import shortcut.)
7. Add the following initialization statement to the end of method `init()` in **Page1.java**, as shown (the added code is bold). This allows you to control the visibility of the grid panel container component by binding its `rendered` property to whether or not property `mySearchResult` is null.

```
public void init() {
    . . .
    super.init();
    . . .
    // Creator-managed Component Initialization
    . . .
    mySearchResult = null;
}
```

8. Save the project files by clicking the Save All icon on the toolbar.

### Add a Data Provider

Using an object array data provider will make the types contained in `ResultElement` visible through the IDE. This, in turn, will make component binding easy.

1. Select Design in the editing toolbar to return to the design view.
2. Expand the Data Providers node in the Components palette.
3. From the Data Providers Components palette, select Object Array Data Provider and drop it on the design view. You'll see component `objectArrayDataProvider1` in the Page1 Outline view.
4. In the Properties window, change the object array data provider's `id` property to **myResultObject**.

Facebook's Exhibit No. 1003

Page 00292

5. Still in the Properties window, select the drop down opposite property `array` and select **resultArray**. This connects the data provider to the array returned in the `GoogleSearchObject`.

## Layout and Grouping with Grid Panel

You'll now add the components that will display the results of the Google Search. Because you want to control these components as a group, you'll use a Grid Panel component as a container for the display components.

1. From the Components palette, expand the Layout node, if necessary.
2. From the Layout Components palette, select Grid Panel and place it on the page below the text field component.
3. In the Properties window under Advanced, *uncheck* property `rendered`. This sets the `rendered` property to `false` and causes the grid panel to disappear from the design view.
4. Select the JSP label in the editing toolbar to bring up **Page1.jsp** in the editor pane.
5. Scroll down to the grid panel tag and locate the `rendered` property. Change its setting from `false` to the following.

```
rendered="#{not empty Page1.mySearchResult}"
```

This means that the grid panel (and all of its sub-components) will be rendered (displayed) if `Page1.mySearchResult` is not empty (`null`).

### Creator Tip

---

*By nesting the display components (static text and hyperlink components) inside a grid panel, we can control the rendering of all of these components by specifying the `rendered` property of the container component (the grid panel).*

---



6. Return to the design view by selecting the Design label in the editing toolbar. The grid panel should reappear on the design canvas.
7. Check the setting for property `rendered`. In the Properties window, locate property `rendered` and hold the mouse pointer over the cell. Creator displays a tooltip with the current setting for this property, as shown in Figure 10-4.

## Add a Static Text Component

Next, let's add a static text component to display some of the search results from Google. You'll add this component to the grid panel container.

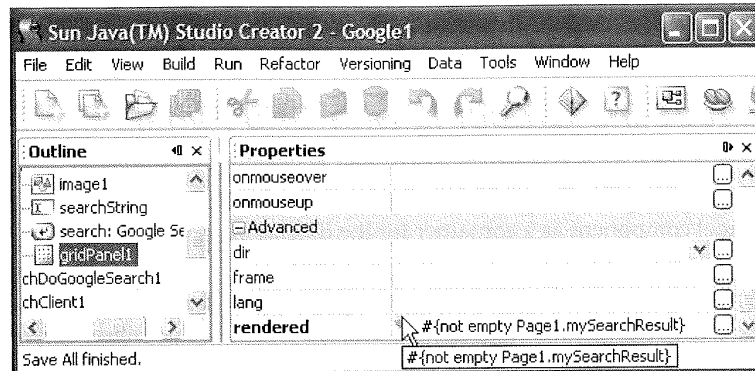


Figure 10-4 Showing the grid panel's rendered property binding expression

1. From the Basic Components palette, select Static Text. Place it on top of the grid panel. You can make it a sub-component of grid panel by dropping it on top of the grid panel component in the Outline view, or dropping it on the grid panel in the design view.
2. In the Properties window, change its `id` property to `timeCount`.

The static text component `timeCount` will display the amount of time in seconds that the search request took on Google's server, as well as the estimated number of search results found. Although Google's search returns at most ten results, this number is the estimated total (anywhere from zero to thousands).

### Using Hyperlink with a Nested Static Text

The hyperlink component allows application writers to submit a form, navigate to an external URL, or navigate to an anchor within the same page. You'll use it to hold the URL returned in the Google search results. Normally, the `text` property of the hyperlink component is sufficient to display descriptive text for its URL. However, in this situation, the text will contain embedded HTML code supplied by the Google search web service. To correctly render this, use a *nested* static text component and *uncheck* its `escape` property.

1. In the Basic Components palette, select Hyperlink and drop it onto the grid panel in the Page1 Outline view. It should appear nested under the grid panel at the same level as the `timeCount` static text component you already added. The `id` property of this component is `hyperlink1`.
2. In the Basic Components palette, select Static Text and drop it directly on top of the hyperlink component you just added. (You can drop it on top of the

component on the design canvas, or you can drop it on top of the hyperlink displayed in the Page1 Outline view.)

#### Creator Tip

---

*When you drop it on top of the hyperlink component in the design view, make sure that the hyperlink component is outlined in blue. This is an indication that you have selected it for placement and that the static text component will be nested. If you drop it onto the hyperlink component in the Outline view, the hyperlink component should be selected (white text in a blue background), making the static text component nested.*

---



3. Make sure that the nested static text component is selected. In the Properties window, change its `id` property to **nestedText**.
4. Under Data in the Properties window, *uncheck* the `escape` property. This allows correct rendering of HTML tags embedded within the text.
5. Add another static text component and drop it on top of the grid panel in the Outline view. This second static text component will display the “snippet” returned by the Google search.
6. In the Properties window, change the `id` property to **snippet**.
7. The snippet that Google returns will also contain embedded HTML tags. To display the HTML formatting correctly, *uncheck* the `escape` property in the static text’s Properties window.

### Add a Message Group to Display Errors

When you access an external web service, there is always a possibility that the access could fail. The server that provides the web service could be inaccessible, the machine that runs the web application could fail, or the access key may be incorrect. In any case, you want to know that the web services call has failed and why.

We’ll show you the code that guards against any type of failure later in this section. For now, you’ll use a Message Group component to display messages for this web application.

- From the Basic Components palette, select Message Group component and drop it onto the design canvas. Place it to the right of the text field.

The Message Group component renders messages that are tied to any component, as well as messages generated by the message routines, `info()`, `warn()`, `error()`, and `fatal()`. You can use these routines to report information back to the user from any bean (Java class) that extends class `FacesBean`.

You've finished adding the components to the page. Compare the components you have placed on the page with those shown in Figure 10-5, the Outline view for Page1.

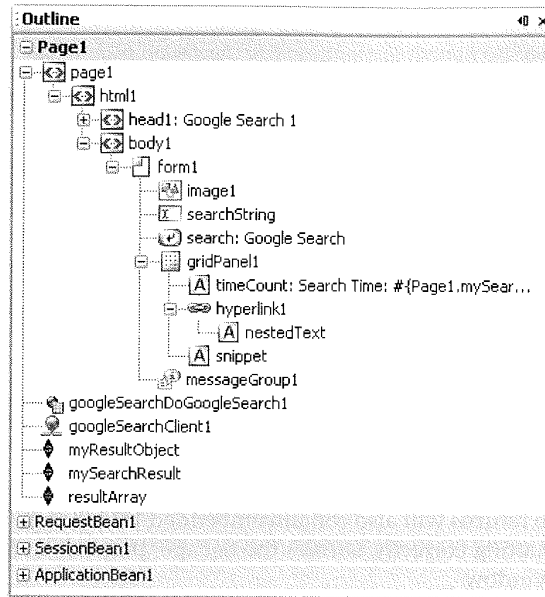


Figure 10-5 Page1 Outline view of project Google1

## Deploy and Run



### Creator Tip

*Although you haven't added the calls to the Google Web Service yet, let's build and run the web application anyway. When the application runs, the page is redisplayed when you click the Search Google button (admittedly, not much).*

To run the project, click the green arrow in the toolbar or select **Run > Run Main Project** from the main menu. The page should display the Google logo in the upper-left corner, as well as the text field and button. The grid panel and all of its nested components will not be visible since property `mySearchResult` is empty (null). You can type in a test search string, but clicking the button does not (yet) access the Google web service. It does, however, redisplay the page.

Now it's time to look at the methods in the Google web service API.



## Inspect the Web Service

The Google web service is already included in your application, so let's use Creator to learn more about it.

When you added the Google web service to your page, Creator modified the Java page bean file (**Page1.java**) to import the Google web service package, as shown here.

```
import webservice.googlesearchservice.googlesearch.  
    GoogleSearchClient;  
import webservice.googlesearchservice.googlesearch.  
    GoogleSearchDoGoogleSearch;
```

In the Navigator window Creator displays the Page1 methods (orange circle icon), the constructor (yellow diamond icon), and private variables (blue rectangle icon). Find the blue rectangle next to variable `googleSearchClient1` and double-click. In the Creator-managed code, you'll see private variable `googleSearchDoGoogleSearch1` defined as follows.

```
private GoogleSearchDoGoogleSearch googleSearchDoGoogleSearch1  
    = new GoogleSearchDoGoogleSearch();
```

This is the object that you use to make calls to Google's web service API, as follows.

```
mySearchResult = (GoogleSearchResult)  
    googleSearchDoGoogleSearch1.getResultObject();
```

### Creator Tip

---

*At this point, you will undoubtedly find Google's documentation to be helpful. A detailed description of the methods and their parameters can be found on the Google Web Site: <http://www.google.com/apis/reference.html>*

---



Table 10.1 contains a list of the parameters for initiating a search with the `googleSearchDoGoogleSearch` object. The search returns a `GoogleSearchResult` object.

**Table 10.1** doGoogleSearch() parameters

<i>Name</i>	<i>Type</i>	<i>Description</i>
key	String	Key provided to you by Google. A key is required for access to the Google service.
q	String	Search query.
start	int	Zero-based index of the first desired result.
maxResults	int	Number of results desired per query. This is at most 10.
filter	boolean	Specifies whether or not you want filtering, which helps eliminate very similar results.
restricts	String	Limits the search to a subset of the Google Web index.
safeSearch	boolean	Enables filtering of adult content.
lr	String	Language Restrict—limits the search to documents with the specified languages.
ie	String	Input Encoding—deprecated.
oe	String	Output Encoding—deprecated.

Table 10.2 contains some of the methods for return object `GoogleSearchResult`. Note that these methods are the getter form of JavaBeans object properties. Therefore you can call the getter method, or access the property using a JSF EL expression, such as the following.

```
#{Page1.mySearchResult.startIndex}
```

which evaluates to the starting index of the returned results.

**Table 10.2** GoogleSearchResult public methods

<i>Name</i>	<i>Return Type</i>	<i>Description</i>
getDirectoryCategories	Array	Array of the Directory Category items corresponding to the ODP <sup>a</sup> directory matches for this search.
getEndIndex	int	Index (1-based) of the last search result in the ResultElements array.
getEstimatedTotalResultsCount	int	Estimates of the total number of results for the query.
getResultElements	ResultElement []	Array containing the results.
getSearchComments	String	Search comments.
getSearchQuery	String	Search query you provided.
getSearchTime	double	Time it took the Google server to compute the results.
getSearchTips	String	Tips for searching.
getStartIndex	int	Index (1-based) of the first search result in the ResultElements array.
isDocumentFiltering	boolean	True if document filtering is enabled.
isEstimateIsExact	boolean	True if the total results estimate is exact.

- a. “The **Open Directory Project** is the largest, most comprehensive human-edited directory of the Web. It is constructed and maintained by a vast, global community of volunteer editors.” (See About the Open Directory Project, <http://dmoz.org>.)

Finally, each response includes an array of ResultElement objects. Some of the methods you use to access a ResultElement object are listed in Table 10.3.

**Table 10.3** ResultElement public methods

<i>Name</i>	<i>Return Type</i>	<i>Description</i>
getCachedSize	String	Size of the cached document.
getDirectoryCategory	DirectoryCategory	Name of the ODP category in which the result occurs.
getDirectoryTitle	String	Name of the result as it appears in the Open Directory.
getHostName	String	Hostname of the result.
getSnippet	String	Short description of the result page.
getSummary	String	Description of the result as it appears in the Open Directory.
getTitle	String	Page title of the result.
getURL	String	URL of the result page.
isRelatedInformationPresent	boolean	True if there are related documents to this result.

### **Testing the Google Web Service**

Creator provides a web services testing mechanism. This is a quick way to study the method, provide parameters, and look at the results. Here's how.

1. In the Servers window under **Web Services > Samples > GoogleSearch**, right-click method `doGoogleSearch()` and select **Test Method**. Creator pops up the Test Web Service Method dialog.
2. Fill in the dialog as shown in Figure 10-6. For `key`, provide your key (sent to you by Google after you register at their web site), for `q`, provide a query string (we used "I M Pei Louvre"), and for `maxResults`, use **1**. You can use default values for all of the other parameters.
3. Click Submit. A successful test displays the results in the Results window. Expand **GoogleSearchResult > ResultElement[] > ResultElement** to see the results.
4. Click Close to finish.

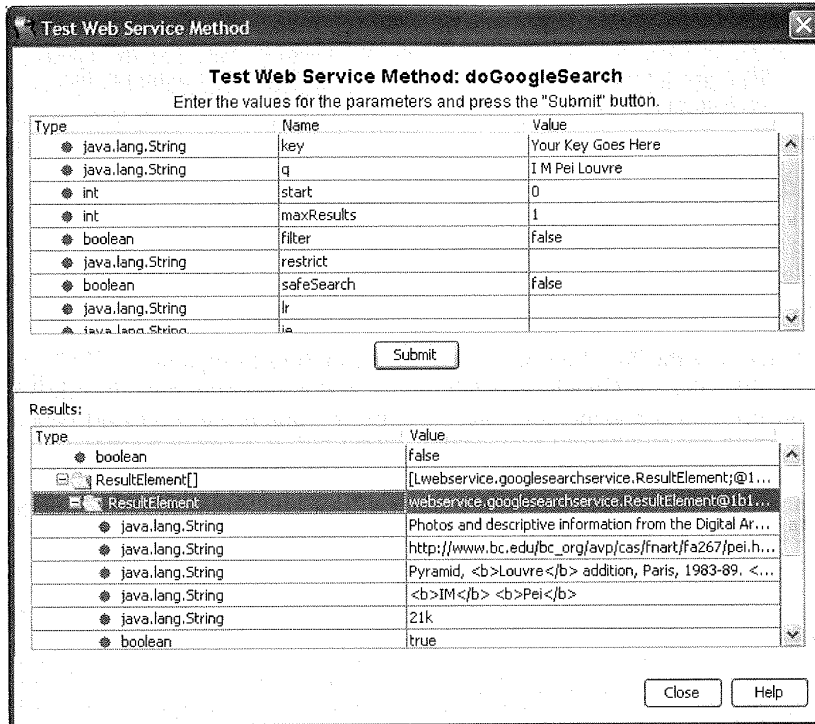


Figure 10-6 Testing Web Service Method doGoogleSearch

## Configure Web Service Call

Instead of supplying method parameters in the event handling code, you can configure the web service through the Properties window, as follows.

1. Select `googleSearchDoGoogleSearch1` in the Page1 Outline view.
2. In the Properties window, the method's arguments are all listed under the General heading. For property `key`, provide your key (the one Google sent to you) and for `maxResults`, use `10`. These are the only properties that you need to set.

**Creator Tip**

*Make sure you include your Google Web API's License Key for property key. Otherwise, your application will return an exception. You'll configure the search query (property q) later.*

## Add Event Handling Code for Button

You'll now add the Java code to access Google's search method. You'll put the code in the action method associated with the Google Search button on the page.

1. Make sure the Page1 design view is active in the editor pane.
2. Double-click the Google Search button. Creator generates default code for the button's action method, `search_action()` and brings up **Page1.java** in the Java source editor.
3. Add the following code to the `search_action()` method. From your Creator book download, copy and paste the file **FieldGuide2/Examples/Web-Services/snippets/google1\_search.txt** into the `search_action()` event handler. The added code is bold.

### Listing 10.1 Method `search_action()`

```
public String search_action() {
    try {

        mySearchResult = (GoogleSearchResult)
            googleSearchDoGoogleSearch1.getResultObject();
        resultArray = mySearchResult.getResultElements();
        myResultObject.setResultArray(
            (java.lang.Object[])getValue("#{Page1.resultArray}"));

    } catch (Exception e) {
        log("Remote Connect Failure: ", e);
        mySearchResult = null;
        error("Remote Site Failure: " + e.getMessage());
    }
    return null;
}
```

## Handling Exceptions and Error Messages

Let's examine the search button's action event code.

Because method `doGoogleSearch()` (which you invoke through the client `googleSearchDoGoogleSearch1`) throws `RemoteException`, you should include its call inside a `try` block. In this case, the accompanying catch handler will catch *any* exception, including `RemoteException`. The catch handler performs several tasks. First, it logs the error with the application server's log. You can view the log by selecting **Deployment Server** in the Servers window. Right-click and select **View Server Log**. Creator displays the log in the Output window (by default this is below the editor pane) under the tab for the application server host machine and port (ours is `localhost:24848`).

The next line sets the `Page1` property `mySearchResult` to null. Recall that you bound the `rendered` property of the grid panel to whether or not this property is empty. Since there is nothing to display when an exception occurs, it makes sense to make sure the screen is not cluttered with a previous call's results.

The last line of the catch handler is a call to method `error()`. Method `error()` posts a message to the `FacesContext`. This message is not associated with a particular component, but is a generic user message. Generic messages will be displayed by a message group component if there is one on the page. (This is why you placed a message group component on the page.) Along with method `error()`, Creator provides methods `info()`, `warn()`, and `fatal()` for reporting messages back to the user. These message reporting methods render differently on the page, depending on their severity level.

If the call to `googleSearchDoGoogleSearch1` succeeds, you store the results in variable `mySearchResult` (a property), making the results accessible to the rest of the web application. You also set property `resultArray` and the data provider's `array` property. In the next section, you'll bind the components using these `Page1` components.

## Specify Binding for the Display Components

The static text and hyperlink components that you nested inside the grid panel display portions of the result returned from the call to Google's search method. Instead of setting these components' properties inside the event handler, you can specify the bindings through the Properties window. Let's provide the bindings now. (You may want to refer to Table 10.3 on page 464, which lists the properties for the returned result object.)

### Design Note

---

*Because taking small steps is always better than attempting a giant leap, let's display only the first result on your web page. In a later section, we'll have you display all of the returned results (a maximum of ten).*

---



1. Click the Design label in the editing toolbar to return to the design view.
2. Select the static text component `timeCount`. In the Properties window for property `text`, type in the following binding expression. (Note that this expression combines literal text with expressions.) Type in the complete text on a single line and finish with **<Enter>**. The text will appear on the design canvas.

```
Search Time: #{Page1.mySearchResult.searchTime}; Approx.  
Results: #{Page1.mySearchResult.estimatedTotalResultsCount}
```

This will display the search time and the estimated total number of results returned from the search.



#### Creator Tip

*If you hold the mouse over the text property in the Properties window, the value is displayed in a tooltip. Use this to check that you entered the expressions and literal text correctly.*

Now you'll bind the display components to the object array data provider, `myResultObject`.

1. Select the hyperlink component (use the Outline view). In the Properties window, select the editing box opposite property `text`. Creator pops up a property customizer. Click button **Unset Property** to remove the default text, **Hyperlink**.
2. The hyperlink component holds the URL property of the result returned from the Google search. In the Properties window, select the editing box opposite property `url`. Creator displays the `url` property customizer.
3. Select radio button **Use binding** and tab **Bind to Data Provider**. Make sure that data provider **myResultObject** is selected, as shown in Figure 10-7. Select property **URL** in the Data field window. Click **OK**.

Creator generates the following binding expression for the hyperlink's `url` property.

```
#{Page1.myResultObject.value['URL']}
```

4. Select the nested static text component `nestedText`. This component will display the title of the returned search result. In the Properties window for property `text`, click the editing box to bring up the `text` property customer.



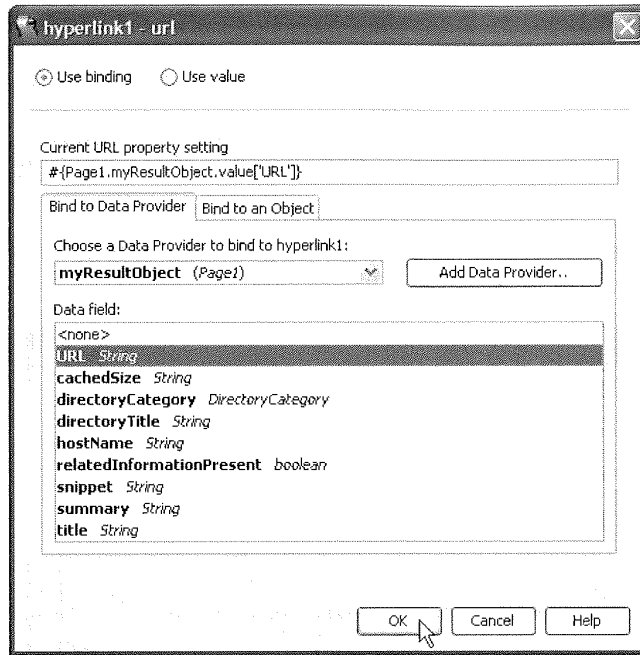


Figure 10-7 Bind url property to the object array data provider myResultObject

5. Select radio button **Use binding**, tab **Bind to Data Provider**, Data Provider **myResultObject**, and Data field **title**. Select OK. Creator generates the following expression.

```
#{Page1.myResultObject.value['title']}
```

6. Make sure that the escape property for this static text component is *unchecked* (set to false). This allows the embedded HTML elements that Google supplies to be rendered correctly on the page.
7. Select the static text component **snippet**. This component displays a short description of the returned URL. In the Properties window, bring up the customizer for property text.
8. Select radio button **Use binding**, tab **Bind to Data Provider**, Data Provider **myResultObject**, and Data field **snippet**. Select OK. Creator generates the following expression.

```
#{Page1.myResultObject.value['snippet']}
```

9. Make sure that the `escape` property for this static text component is also *unchecked* (set to false).

Finally, you'll bind the text field `searchString` to the query parameter `q` in the `googleSearchDoGoogleSearch1` object.

1. In the Design view, select text field component `searchString`. In the Properties window, bring up the customizer for property `text`.
2. Select radio button **Use binding** and tab **Bind to an Object**.
3. In the **Select binding target** window, expand `googleSearchDoGoogleSearch1` node and select property `q`, as shown in Figure 10–8. Click OK.

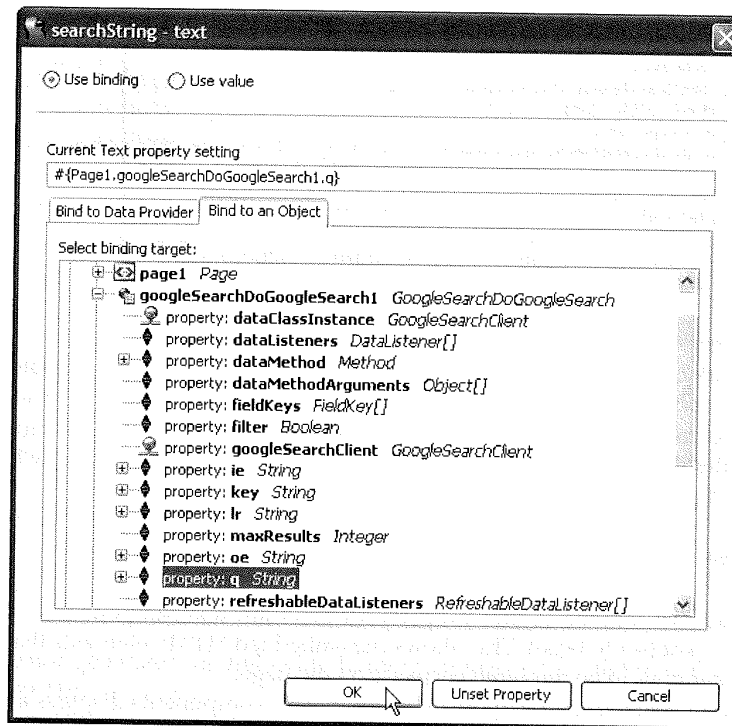


Figure 10–8 Bind `searchString` text property to property `q` (query)

## Deploy and Run

You're ready to test this initial version of the Google search web application.

- From the main menu bar, select **Run > Run Main Project** or select the green arrow on the icon bar. You can test the Google Search API by typing in various search queries. Click on the URL (displayed as the title) and go to that web page. Figure 10-9 shows a screen shot of the application.

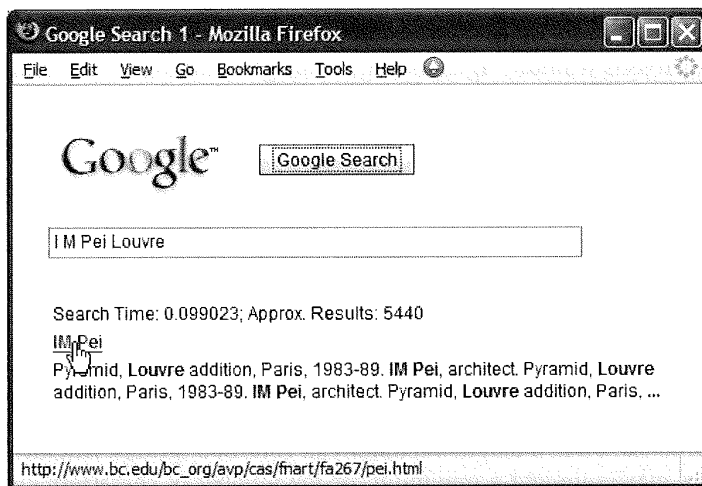


Figure 10-9 First version of the Google Web Search application

## 10.2 Validation - Project Google2

You have created a simple web application that uses a published web service. Now you're going to build on this example and enhance it in the following ways.

- Provide validation for the text field component and require that the user provide something. That is, you want to prevent a zero-length string and require a minimum length for the search string (three characters).
- Place a message component on the page to report validation errors associated with the text field component.
- Configure the message group component so that it displays global messages only.

## Copy the Project

To avoid starting from scratch, make a copy of the Google1 project. This step is optional. If you don't want to copy the project, simply skip this section and continue making modifications to Google1.

1. Bring up project Google1 in Creator, if it's not already opened.
2. From the Projects window, right-click node Google1 and select Save Project As. Provide the new name **Google2**.
3. Close project Google1. Right-click Google2 and select Set Main Project. You'll make changes to the Google2 project.
4. Expand **Google2 > Web Pages** and open **Page1.jsp** in the design view.
5. Click anywhere in the background of the design canvas of the Google2 project. In the Properties window, change the page's **Title** property to **Google Search 2**.

## Add a Validator

It's always a good idea to validate user input. Included in the components palette are a set of validators to help with this task. With text strings, two validators are of interest. First, a length validator can control a String's length. You can specify a maximum and a minimum for the number of characters input. Interestingly enough, if you want to prevent a zero-length string, you cannot use the length validator (and set the minimum to 1). Instead, you must use the component's *required* property. The *required* property is set in the Properties window for each text field component. If you check it (set it to true) and the component's value is a String, then its length must be greater than zero.

Validation in this application is important for two reasons. First, the web application developer has better control over user input and can give feedback to increase program usability. Secondly, by requiring valid input at the application's server site, you prevent access to Google's web site with an invalid search request.

For this example, let's prevent a zero-length string and set a length minimum of three characters and a maximum of 2,048 characters (this is the maximum search query allowed by Google).

**Creator Tip**

*For testing, you'll set the length minimum to 3 and its maximum to 25. Coupled with the required validator, you should get the following behavior. If the user leaves the text field empty, you'll get a message from the required validator saying that a value is required. If you type in a 1- or 2-character text value, you'll get feedback from the length validator saying that it was less than the minimum of 3. Likewise, if you type in more than 25 characters, the length validator will complain that the string was more than the maximum. Note that a zero-length string does not trigger the length validator even if the minimum is set to 1. You must use the required property of the component!*



Let's add validation to the application now.

1. Make sure Page1 is active in the design canvas.
2. Select the text field component, `searchString`.
3. In the Properties window, click the checkbox for the `required` attribute. This means the text field component cannot be empty.
4. From the Components palette, expand the Validators node, select Length Validator, and drop it on top of the `searchString` text field component. Creator sets the `validator` attribute for `searchString` to `lengthValidator1`. Component `lengthValidator1` appears in the Page1 Outline view.

You've instantiated a length validator for the text field component. Now you have to give it length boundaries: the minimum and maximum allowable.

5. Select `lengthValidator1` in the Outline view. In its corresponding Properties window, change attribute `maximum` to **25** and `minimum` to **3**.

These values are probably not the limits you'd want to use in your production application, but they're good values for testing. Once you're convinced that the application is working the way you want, set the maximum to 2048, which is the maximum imposed by Google. The advantage of using the validator instead of letting Google complain is that you save a trip to the Google server.

Note also that the private `_init()` method in the Java page bean has been modified to include the minimum and maximum settings you defined in the Properties window, as follows. (Unfold the Creator-managed Component Definition block to see the `_init()` method.)

```
lengthValidator1.setMaximum(25);  
lengthValidator1.setMinimum(3);
```

## ***Add a Message Component***

You've already placed a message group component on the page. And, if you run the web application now, the message group component will display error messages detected during validation. However, the look of the message group component is not really what you want, since validation messages aren't really "System Messages." For validation error reporting, use the message component. Like the message group component, the message component retrieves messages from the JSF context. The difference is that the message component is associated with a single component. That way, the web application designer can control exactly where on the page the error message for a specific input component appears. This is particularly useful for pages that contain many input components (such as a form that submits a whole page of personal information). Thus, when the validator sends an error message to the JSF context, it identifies the component whose input is marked invalid. A message component tied to this input component will pick up the error message and display it on the page.

1. Return to the Design view.
2. From the Basic Components palette, select Message component and drop it onto the design canvas. Place it in between the text field and the grid panel.
3. When you place the message component on the canvas, you can associate it with the text field using the mouse. Press and hold **<Ctrl+Shift>**, left-click the mouse, and drag the cursor to the `searchString` text field. This sets the message component's `for` property. In the design view, the message component now displays the text **Message summary for searchString**.

## ***Message and Message Group Components***

Go ahead and run the web application (select **Run > Run Main Project** from the main menu). Either leave the input field blank or type in less than three characters and press the Google Search button. You'll see that the application displays the error message twice: both the message component (which is tied to the text field) and the message group component (which displays all messages) display the validation error.

By default, a message group component displays all messages associated with a page, both messages tied to a specific component (such as the validation error message you just saw) and system messages (such as a problem with your Google authorization key). But when you also use message components specific to an input component, it's better to restrict a message group component to display global messages only. Global messages are those that are not tied to a specific component.

1. Make sure that Page1 is active in the design view.

2. Select the message group component.
3. In the Properties window, check property `showGlobalOnly` (set it to true). The message group component now displays the text "List of global message summaries."
4. Rerun the application and check its behavior for reporting validation errors as well as system errors. To simulate a system error, temporarily disconnect your test machine from the internet or use an incorrect access key. The Google web services call will fail. Figure 10–10 shows the application running with a validation error message (from the length validator).

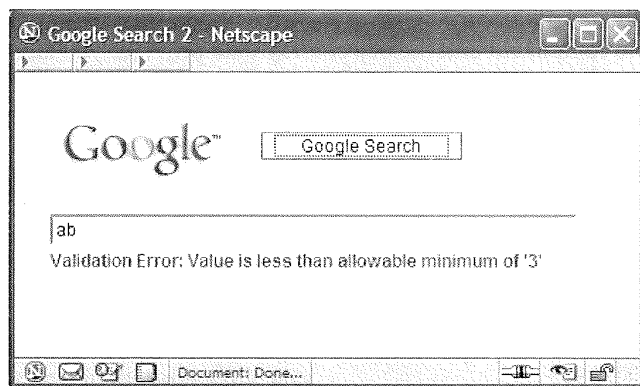


Figure 10–10 Google Web Search with input validation

#### Creator Tip

*During testing, note that when you complete a successful search and follow this with invalid input, the previous results are cleared from the page. You get this behavior because you reset property `mySearchResult` to null during the `Page1` method `init()`. JSF calls method `init()` with each page access, assuring that the page will only render results from a new, valid call to `doGoogleSearch()` method.*



## 10.3 Displaying Multiple Result Elements

The previous version of the application displays only the first result element returned from a Google search. Most of the time, the Google search returns an

array of ten results. You get the first ten results of the query if parameter `start` is set to zero and `maxResults` is set to 10. To get the next set of ten results, set `start` to 10 (instead of 0). For the third set of ten results, set `start` to 20, and so on.

Google returns the total result count with method `getEstimatedTotalResultsCount()`; you can easily determine the number of results returned by getting the `length` attribute of the `ResultElement` array. This will be at most ten. Furthermore, Google imposes a 1,000 count limit, so even if the query returns 30,000 hits, Google will give you at most 1,000 (in ten-count page increments).

In this version of our Google search application, you're going to display all (up to ten) elements of the `ResultElement` array (that is, the first *page*). You'll use Creator's Table component and the object array data provider you already configured.

Our approach is not so different from the project you just built: you use a hyperlink component to hold the `ResultElement`'s URL, a nested static text holds the result's title, and a second static text component holds the result's snippet. To make sure that the snippet information starts on its own line in the table, you'll use a static text component to hold the HTML `<br/>` tag, which forces a line break in the table cell.

Here are the enhancements that you're going to make to this project.

- Use a Table component to format and display the results from a Google search web service call.
- Use an Object Array Data Provider (component `myResultObject`) to map the `ResultElements[]` array into the table component.
- Specify binding expressions for the table title, column title, as well as each component that you'll add to the table column.

### Copy the Project

To avoid starting from scratch, make a copy of the Google2 project. This step is optional. If you don't want to copy the project, simply skip this section and continue making modifications to Google2.

1. Bring up project Google2 in Creator, if it's not already opened.
2. From the Projects window, right-click node Google2 and select Save Project As. Provide the new name **Google3**.
3. Close project Google2. Right-click Google3 and select Set Main Project. You'll make changes to the Google3 project.
4. Expand **Google3 > Web Pages** and open **Page1.jsp** in the design view.
5. Click anywhere in the background of the design canvas of the Google3 project. In the Properties window, change the page's **Title** property to **Google Search 3**.



## Add a Table Component

The table component is a convenient way to present tabular data, such as an array of objects. You'll be using a single column (to mimic the display you see from Google's web site). In each cell, you'll display the result web site's title followed by the snippet. The title is the text for the hyperlink to the result's site.

1. Make sure that Page1 is active in the design view. Close the Output window to give yourself more room to work in the design editor.
2. From the Basic Components palette, select Table and drag it to the canvas. Place it below the grid panel component that's already on the page. (You'll delete the grid panel component later. For now, leave it on the page.) You'll see a default table rendered on the design canvas and the default table data provider, `defaultTableDataProvider1`, appears in the Outline view.
3. When you place the table component on the canvas, the table's title is selected so that you can provide your own title.
4. Type in the following text to set the title.

```
Search Results (#{Page1.mySearchResult.startIndex} to  
#{Page1.mySearchResult.endIndex})
```

Type the text all on a single line and finish with **<Enter>**. The title text will appear in the table's title area.

## Configure the Table

When you place the table component on the page, creator configures it with a default data provider. You'll use the object array data provider you configured earlier instead.

1. Select the table component, right-click, and choose Table Layout from the context menu.
2. In the drop down menu for Get Data From, choose **myResultObject**. Creator displays the data fields in the Selected window.
3. Use the < (left arrow) to remove all fields except URL, snippet, and title. Click Apply.
4. Select column URL and change the component type to Hyperlink. Click Apply and OK to close the dialog.

Creator binds each of these columns to the URL, snippet, and title fields of the data provider for you. You'll need a bit more customizing to get a look that's similar to the page the Google web site builds. Look at the Page1 Outline view. You'll see the table component (`table1`), a nested table row group, and

three table column components with headings URL, snippet, and title. You'll now rearrange these components a bit.

1. From the Page1 Outline view, select the static text component under the column entitled **title** and drop it on top of the hyperlink component under the **URL** column. (This should nest component `staticText3` under component `hyperlink2`.)
2. In the Properties window for `staticText3`, change its `id` property to **nested-Text** and *uncheck* its `escape` property.
3. In the Properties window, hold the cursor over the `text` property and verify that its binding is set to the following.

```
#{currentRow.value['title']}
```

4. Now select the hyperlink component (`hyperlink2`). In the Properties window, set property `url` to the following. (By default, Creator binds the `text` property instead.)

```
#{currentRow.value['url']}
```

5. In the Properties window for the hyperlink component, select the editing box opposite property `text` to bring up the property customizer and select **Unset Property**.
6. From the Basic Components palette, select Static Text and drop it on component `tableColumn1` in the Page1 Outline view. The static text should appear at the same level as the hyperlink component.
7. In the Properties window, *uncheck* its `escape` property and set its `text` property to `<br>`. (This will provide a line break in the table cell and improve the formatting.)
8. From the Page1 Outline view, select the static text component under the column entitled **snippet** and drop it on top of component `tableColumn1`.
9. In the Properties window, change its `id` property to **snippet** and *uncheck* its `escape` property. Now hold the cursor over the `text` property and verify that its binding is set to the following.

```
#{currentRow.value['snippet']}
```

You'll now make final configuration changes to the table component.

1. In the Page1 Outline view, remove components `tableColumn2` and `tableColumn3` (there shouldn't be any nested components in these unused columns).
2. Select the table component. In the Properties view, set property `width` to **500**.

3. In the Page1 Outline view, select static text component `timeCount` and bring up the property customizer for property `text`. Copy it using `<Ctrl+C>` and click OK.
4. Now select component `tableColumn1` and bring up the property customizer for property `headerText`. Paste (use `<Ctrl+V>`) the value from `timeCount`'s `text` property. Click OK. The column's header shows the new value.
5. Select the table component. In the Properties view, *uncheck* property `rendered`. The table disappears from the design view.
6. Click button JSP to bring the the JSP source editor.
7. Change the table's `rendered` property to the following. (You might want to copy and paste from the grid panel's `rendered` property.)

```
#{not empty Page1.mySearchResult}
```

8. Return to the design view. Delete the grid panel component (and all of its nested components).
9. Move the table component up so that it is directly underneath the message component, as shown in Figure 10-11.

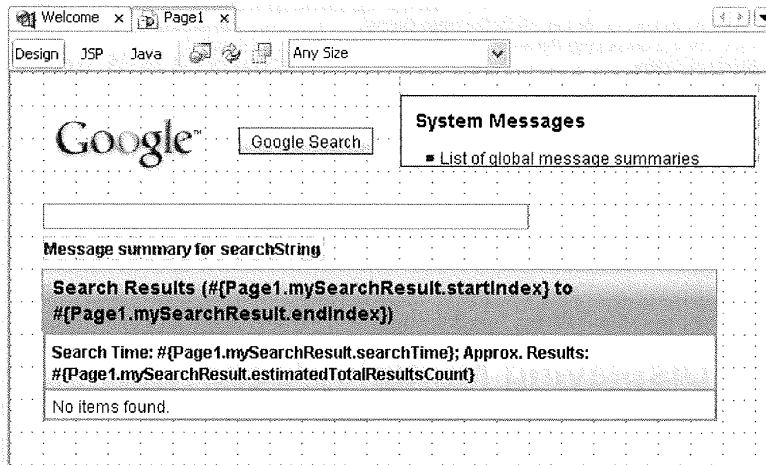


Figure 10-11 Google Search application using Data Provider and Table components



The image files you'll use are in your Creator book's **FieldGuide2** download bundle (**FieldGuide2/Examples/WebServices/images**), but if you'd like to use arrow graphics of your own, simply substitute the appropriate **.gif** or **.jpg** file.

The modifications for this project include adding two new image hyperlink components and image files for their display. You'll also install action event methods for the hyperlink components to page forward or backward through the Google search results. These additions are straightforward. What's a bit tricky is that you have to keep track of some of the parameters across page requests when you call Google. That means you can't use local variables with request scope inside **Page1.java**, since the **Page1** bean is instantiated with each page request. Therefore, you'll need to save and restore these control variables in session scope using the **Page1** methods **destroy()** and **init()**. (Ironically, **destroy()** refers to the *destruction* of **Page1** but is used to *save session state* before said destruction. To review the different types of scope for web application objects, see "Scope of Web Applications" on page 224.)

Although Creator's table component has sophisticated paging controls, these are used to page through a large data set. In this application, the maximum array size is ten. In order to access the subsequent results, you must submit a new call to the Google web service search method with a different starting index and obtain a new result array.

### **Copy the Project**

To avoid starting from scratch, make a copy of the **Google3** project. This step is optional. If you don't want to copy the project, simply skip this section and continue making modifications to **Google3**.

1. Bring up project **Google3** in Creator, if it's not already opened.
2. From the **Projects** window, right-click node **Google3** and select **Save Project As**. Provide the new name **Google4**.
3. Close project **Google3**. Right-click **Google4** and select **Set Main Project**. You'll make changes to the **Google4** project.
4. Expand **Google4 > Web Pages** and open **Page1.jsp** in the design view.
5. Click anywhere in the background of the design canvas of the **Google4** project. In the **Properties** window, change the page's **Title** property to **Google Search 4**.

### **Add an Image Hyperlink Component**

Before you add components to project **Google4**, look at Figure 10-13, the design canvas for this project. You can see the placement of the image hyperlink components (the two arrows).

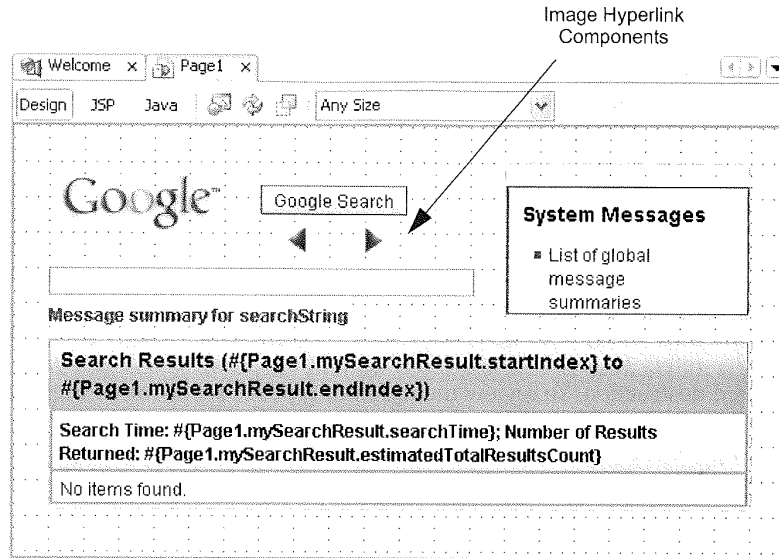


Figure 10-13 Design canvas showing component layout for project Google4

1. From the Basic Components palette, select Image Hyperlink and drop it onto the design canvas under the Google Search button.
2. In the Properties window, change its id property to **previous**.



#### Creator Tip

*You're changing the standard id that Creator uses because it's easier to keep track of the components in the Java page bean file. By using meaningful id names (such as `previous` and `next`), the associated action methods that Creator generates will have meaningful names, too.*

3. In the Properties window under Behavior, set property `toolTip` to **View the previous set of results**.
4. Make sure that the image hyperlink component is selected, right-click, and select Set Image from the context menu. Creator pops up an Image Customizer dialog that allows you to specify the URL, File, or Theme Icon for the image.
5. In the dialog, select radio button Choose File, browse to the Creator book download, and specify directory **FieldGuide2/Examples/WebServices/images** for the field labeled "Look in:", as shown in Figure 10-14.

Facebook's Exhibit No. 1003

Page 00318

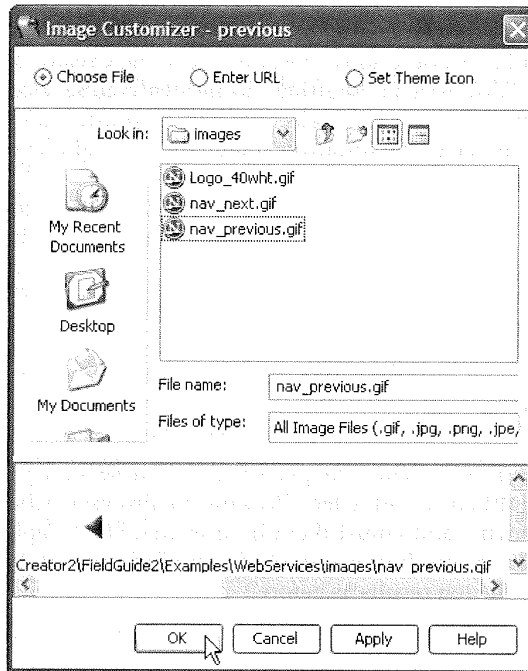


Figure 10-14 Image Customizer dialog for image component

6. Select file **nav\_previous.gif**. Click Apply. Creator displays the arrow in the dialog's Preview window and the image appears on the design canvas.
7. Click OK. Creator copies the image file to the project's **Web Pages > resources** directory.
8. In the Properties window for the image hyperlink, click the editing box opposite property text and select **Unset Property** in the customizer dialog.

### ***A Second Image Hyperlink Component***

Follow the same procedure to add a second image hyperlink component and a second image to the page.

1. From the Basic Components palette, select Image Hyperlink and drop it onto the design canvas under the Google Search button.
2. In the Properties window, change its `id` property to **next**.
3. In the Properties window under Behavior, set property `toolTip` to **View the next set of results**.

4. Make sure that the image hyperlink component is selected, right-click, and select Set Image from the context menu.
5. In the dialog, select radio button Choose File, browse to the Creator book download, and specify directory **FieldGuide2/Examples/WebServices/images** for the field labeled "Look in:". Select file **nav\_next.gif**. Click OK. Creator copies the image file to the resources folder and the right-arrow image should now appear on the design canvas.
6. In the Properties window for the image hyperlink, click the editing box opposite property *text* and select **Unset Property** in the customizer dialog.
7. Adjust the image hyperlink components so that they're aligned vertically to each other. Select the hyperlink component *next*, press and hold the Shift key, and then select the hyperlink component *previous*. With both components selected, right-click and select **Align > Middle** from the context menu.

## Deploy and Run

You might want to experiment with the placement of these newly added graphic components. Right-click and select Preview in Browser. Check the placement of the components and adjust them if necessary. Now deploy and run project **Google4**. (The arrow buttons won't do anything useful, but you should be able to display the ten results as before.)



### Creator Tip

*The arrow buttons erase the table (why?). To see the search results again, click the Google Search button. The table is cleared because clicking an arrow button generates an action event, which initiates the JSF page life cycle process. You haven't specified any action, but the system proceeds through the different life cycle phases anyway. When JSF instantiates the Page1 page bean, it invokes Page.init(), which sets mySearchResult to null and the table component is not rendered.*

## Add SessionBean1 Properties

You will soon add control variables to the **Page1.java** file. These values keep track of the current index and other controls you need for displaying more than the first page of results. To maintain these values across page requests, you add them to the SessionBean1 managed bean as *properties*. This automatically puts them in session scope. The following properties are all type Integer and mode Read/Write:

- `startIndex` - index of the first result (parameter `start`)
- `currentCount` - length of the `ResultElements[]` array



- `totalCount` - estimated total number of results for the query; used to test end conditions

Here are the steps to add properties to `SessionBean1`.

1. From the Projects window, select **Session Bean**, right-click and select **Add > Property**.
2. Creator pops up the New Property Pattern dialog.
3. For Name, specify **startIndex**, for Type, specify **Integer**, and for Mode, select the default **Read/Write**. Click OK.
4. Repeat these steps for **currentCount** and **totalCount**. Specify Type **Integer** and Mode **Read/Write** for both.
5. In the Projects window, double-click node **Session Bean**. This brings up file **SessionBean1.java** in the Java source editor.
6. Add the following code to the end of method `init()` to initialize the three properties. Copy and paste from file `FieldGuide2/Examples/WebServices/snippets/google4_session_init.txt`. The added code is bold.

```
public void init() {  
    . . .  
    startIndex = new Integer(0);  
    currentCount = new Integer(0);  
    totalCount = new Integer(0);  
}
```

### ***Specify the Action Code***

When a user clicks the right-arrow hyperlink, the web application should display the next ten results from the Google search. Conversely, clicking the left-arrow hyperlink displays the previous ten results. Because you'll make similar calls to the Google search API, you should place this code in its own method. Once you determine the correct start index, you can call this method from the action event handlers for the search button and from both hyperlinks.

1. To return to the design view for `Page1`, select the tab labeled **Page1.jsp** at the top of the editor pane.
2. In the design canvas, select the right-arrow hyperlink `next` and double-click.
3. Creator brings up the Java page bean file, **Page1.java**, and displays the generated event handler, `next_action()`.
4. To keep track of the index variables and the result count information that Google returns, you'll need integer control variables. Place these declara-

tions above method `next_action()`. Copy and paste file `FieldGuide2/Examples/WebServices/snippets/google4_variables.txt`.

```
private int startIndex = 0;
private int prevIndex = 0;
private int currentCount = 0;
private int totalCount = 0;
```

The `startIndex`, `currentCount`, and `totalCount` integer variables are saved and restored in session scope for the action handlers. To do this, you'll use the `Page1` methods `destroy()` and `init()` to save and restore the `SessionBean1` properties. Recall that method `init()` is invoked after the page is constructed and method `destroy()` is invoked after the page is rendered. (Table 6.6 on page 266 describes these life cycle methods.)

1. Add the following code to method `destroy()`. Copy and paste from `FieldGuide2/Examples/WebServices/snippets/google4_destroy.txt`. This code calls setters to store `startIndex`, `currentCount`, and `totalCount` as equivalently named properties in the `SessionBean1` object. The added code is bold.

---

#### Listing 10.2 Method `destroy()`

---

```
private void destroy() {
    getSessionBean1().setStartIndex(new Integer(startIndex));
    getSessionBean1().setCurrentCount(
        new Integer(currentCount));
    getSessionBean1().setTotalCount(new Integer(totalCount));
}
```

---

2. Next, add the following code to the end of method `init()`. Copy and paste from `FieldGuide2/Examples/WebServices/snippets/google4_init.txt`. The added code is bold.

---

#### Listing 10.3 Method `init()`

---

```
private void init() {
    . . .
    mySearchResult = null;
    startIndex = getSessionBean1().getStartIndex().intValue();
    currentCount =
        getSessionBean1().getCurrentCount().intValue();
    totalCount = getSessionBean1().getTotalCount().intValue();
}
```

---

Most of the code that resides in the `search_action()` event handler can be pulled out and placed in a method that all three action event handlers will call. Let's call this new method `doSearch()`. The difference is that the start index, which was previously hard-coded to zero, has been parameterized (`int start`). The second difference is that we're saving the total search count (`totalCount`) and the length of the `ResultElements []` array (`currentCount`).

3. To create the `doSearch()` method, use `FieldGuide2/Examples/WebServices/snippets/google4_doSearch.txt` and place it directly before the `search_action()` method (near the end of the Java page bean file). Note that this method sets the starting index value by calling method `setStart()`. It also sets `totalCount` and `currentCount` from the `mySearchResult` object.

---

**Listing 10.4 Method** `doSearch()`

---

```
public void doSearch(int start) {
    try {
        googleSearchDoGoogleSearch1.setStart(start);
        mySearchResult = (GoogleSearchResult)
            googleSearchDoGoogleSearch1.getResultObject();

        resultArray = mySearchResult.getResultElements();
        myResultObject.setArray(
            (java.lang.Object[])getValue(
                "#{Page1.resultArray}"));

        totalCount =
            mySearchResult.getEstimatedTotalResultsCount();
        currentCount =
            mySearchResult.getResultElements().length;

    } catch (Exception e) {
        log("Remote Connect Failure: ", e);
        mySearchResult = null;
        error("Remote Site Failure: " + e.getMessage());
    }
}
```

---

4. The `search_action()` method is now simpler, since all that's required is to reset the index control variables and call `doSearch()`. Copy and paste from

file `FieldGuide2/Examples/WebServices/snippets/google4_search_action.txt` to modify this method. Here's the new code.

---

**Listing 10.5** Method `search_action()`

---

```
public String search_action() {
    startIndex = 0;
    prevIndex = 0;
    doSearch(startIndex);
    return null;
}
```

---

Clicking the right-arrow hyperlink returns the next set of results from Google. To effect this return, update the `start` parameter (see Table 10.1 on page 462) of the `doGoogleSearch()` method. Also note that the code in the action handler `next_action()` is similar to the code in the above `search_action()`. You just need to check for upper limits in the index control variables.

5. Add code to the `next_action()` event handler. Copy and paste from file `FieldGuide2/Examples/WebServices/snippets/google4_next_action.txt`. Note that the index variables from the session object are automatically restored and saved through methods `init()` and `destroy()`. (The added code is bold.)

---

**Listing 10.6** Method `next_action()`

---

```
public String next_action() {
    prevIndex = startIndex;
    startIndex = startIndex + currentCount;

    if (startIndex >= totalCount || startIndex >= 1000) {
        startIndex = prevIndex;
        prevIndex -= currentCount;
    }

    doSearch(startIndex);
    return null;
}
```

---

Now let's add a `previous_action()` method to handle action events associated with the left-arrow hyperlink.

6. Return to the design canvas by selecting **Design** from the editing toolbar.

7. Select the left arrow hyperlink `previous` and double-click. This creates the event handler method in the Java page bean for you and places the cursor at the beginning of the method.
8. Add the following code to the default `previous_action()` method. Copy and paste from file `FieldGuide2/Examples/WebServices/snippets/google4_previous_action.txt`. The added code is bold.

---

**Listing 10.7** Method `previous_action()`

---

```
public String prevl_action() {  
    prevIndex = startIndex - currentCount;  
    startIndex = prevIndex;  
  
    if (startIndex <= 0) {  
        startIndex = 0;  
        prevIndex = 0;  
    }  
  
    doSearch(startIndex);  
    return null;  
}
```

---

You'll note that the structure of `previous_action()` is similar to the `next_action()` method.

### ***Deploy and Run***

Deploy and run the project. You should be able to page through multiple result sets by using the arrow graphics "right" and "left." Figure 10-15 shows the fourth page of a result set.

## **10.5 Key Point Summary**

Creator provides an easy drag and drop feature for accessing a web service through your project. You can add a web service so that it is accessible through the Servers window, test a web service method, and view the returned results.

- Web services provide a standard way to access services over a network in a heterogeneous environment.
- You can add a web service client to Creator by supplying the URL of its Web Service Description Language (WSDL) page.



Figure 10-15 The Google Web Search application displaying the third page of results

- You can test a web service by selecting one of its methods, right-clicking on the method, and selecting Test Method. Creator displays a Test Web Service Method dialog in which you supply parameter values, submit the call, and examine the results.
- You can access methods of a web service by dragging its node onto the design canvas. Web services appear in the Outline view for the page.
- The Google web service API provides a SOAP interface to search Google's index of pages.
- Initiate a search of the Google engine by dragging the doGoogleSearch web service onto your page.
- Web service methods should be invoked within a try block.
- Use image components to add graphics to your web pages.
- Use a static text component to display read-only text. Setting its `escape` attribute to `false` allows correct rendering of HTML tags. JSF dynamically sizes the output text component for you.
- Use a hyperlink component to submit a form, navigate to an external URL, or navigate to an anchor within the same page.
- Use an image hyperlink component to render an action component using an image.

- You can nest a static component within a hyperlink component and use it to display the text for the hyperlink. This allows you to render HTML tags correctly by *unchecking* the escape property.
- You can control whether or not a component is displayed through its rendered property.
- You can nest components inside a grid panel or other layout component. This allows you to easily control the rendering on these components as a group through the parent component's rendered property.
- The length validator makes sure that input is within a certain range for length. It does not check for empty fields.
- The required attribute makes sure the field is not empty.
- Validators write error messages to the JSF context. Use a message component to display error messages generated by a specific component.
- Use the message group component to display generic user messages. Set its showGlobalOnly property to true to suppress error messages that are already displayed through component-specific message components.
- You can generate user messages using one of info(), warn(), error(), and fatal() from within any FacesBean object.
- Page1 lives in request scope. You can maintain information you need to access across page requests as properties in SessionBean1, which is defined in session scope.
- Use page bean method destroy() to save session state in SessionBean1. Use page bean method init() to restore session state from SessionBean1.



Results

g on  
rvice  
and

le's

web

ly

RL,

g an

Revised  
and  
Updated for  
Creator 2

# JAVA™ STUDIO CREATOR FIELD GUIDE

LC COPYRIGHT  
0 022 368 654 0  
2nd Edition

### About the Authors

**PAUL ANDERSON**, founding member of the Anderson Software Group, specializes in making software engineering understandable. He has taught courses at IBM, AT&T, Yahoo, Hewlett-Packard, the U.S. Navy, Qualcomm, and other leading firms.

**GAIL ANDERSON**, founding member and Director of Research at Anderson Software Group, specializes in Java, UNIX/Linux, C, C++, and object-oriented design. She has developed courseware on EJB™, JSP™, servlets, JDBC, and UML.

Together, Gail and Paul Anderson have coauthored *The UNIX C Shell Field Guide*, (Prentice Hall, 1986) *Navigating C++ and Object-Oriented Design* (Prentice Hall, 1997) and *Enterprise JavaBeans Component Architecture*. (Sun Microsystems Press/Prentice Hall, 2002).

[www.prenhallprofessional.com](http://www.prenhallprofessional.com)

♻️ Text printed on recycled paper

PRESENTED BY  
PEARSON EDUCATION

### *The Insider's Guide to Sun Microsystems' Breakthrough Java Development Environment*

Java Studio Creator offers developers a remarkably productive visual environment for building, integrating, and delivering business-critical applications. **Java Studio Creator Field Guide, Second Edition** has been revised and updated for Creator 2 and includes five completely new chapters. Leading Java instructors and consultants Gail and Paul Anderson help you use Java Studio Creator to simplify and accelerate your entire development process. One step at a time, the Andersons walk you through

- Mastering Java Studio Creator's interface and workflow
- Simplifying Web development with JavaServer™ Faces components, validators, and data converters
- Controlling Web application page flow with the Page Navigation editor
- Building Creator projects with JavaBeans™ components
- Accessing Web services using XML-based open standards: a case study using Google Web Service APIs
- Using databases and data-aware components—including detailed coverage of new JDBC™ RowSets
- Customizing applications: localization, internationalization, and custom validation
- Debugging with Creator's built-in debugger

With Java Studio Creator and this book, you'll spend less time on application "plumbing"—leaving more time for the high-value tasks you really care about. Simply put, you'll get more done, faster...and have more fun doing it.

### COMPANION WEB SITE

at Anderson Software Group, Inc. [www.asgtech.com](http://www.asgtech.com)

Contains all code samples from the book

9 780132 254601 5 4 9 9 9

ISBN 0-13-225460-3  
Facebook's Exhibit No. 1003  
\$49.99 US \$66.99 CANADA  
Page 00328