

Includes  
A Complete Project

# Java Server Programming

## Java EE6 (J2EE 1.6)

# Black Book



Java Server  
Programming  
Java EE6  
(J2EE 1.6)

**KOGENT**  
Learning Solutions Inc.

Indispensable  
Comprehensive  
Reference

**dreamtech**  
PRESS



Digitized by the Internet Archive  
in 2022 with funding from  
Kahle/Austin Foundation

[https://archive.org/details/isbn\\_9788177229363](https://archive.org/details/isbn_9788177229363)

BHUSHAN M. PATIL.

# Java Server Programming Java EE6 (J2EE 1.6)

## Black Book™

Sunbeam Institute of  
Information Technology  
Anuda Chambers, 203, Shaniwar Peth,  
Near Gujar Hospital, Karad- 415 110  
Maharashtra - INDIA

Kogent Learning Solutions Inc.

Published by:



©Copyright by Dreamtech Press, 19-A, Ansari Road, Daryaganj, New Delhi-110002

Black Book is a trademark of Paraglyph Press Inc., 2246 E. Myrtle Avenue, Phoenix Arizona 85202, USA exclusively licensed in Indian, Asian and African continent to Dreamtech Press, India.

This book and the enclosed CD may not be duplicated in any way without the express written consent of the publisher, except in the form of brief excerpts or quotations for the purposes of review. The information contained herein is for the personal use of the reader and may not be incorporated in any commercial programs, other books, databases, or any kind of software without written consent of the publisher. Making copies of this book or any portion for any purpose other than your own is a violation of copyright laws.

**Limits of Liability/disclaimer of Warranty:** The author and publisher have used their best efforts in preparing this book. The author make no representation or warranties with respect to the accuracy or completeness of the contents of this book, and specifically disclaim any implied warranties of merchantability or fitness of any particular purpose. There are no warranties which extend beyond the descriptions contained in this paragraph. No warranty may be created or extended by sales representatives or written sales materials. The accuracy and completeness of the information provided herein and the opinions stated herein are not guaranteed or warranted to produce any particulars results, and the advice and strategies contained herein may not be suitable for every individual. Neither Dreamtech Press nor author shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

**Trademarks:** All brand names and product names used in this book are trademarks, registered trademarks, or trade names of their respective holders. Dreamtech Press is not associated with any product or vendor mentioned in this book.

ISBN: 978-81-7722-936-3

Edition: 2010

Printed At: Jai Durga Enterprises

# Contents at a Glance

<b>Introduction.....</b>	<b>xxxii</b>
<b>Chapter 1: Java EE 6: An Overview .....</b>	<b>1</b>
<b>Chapter 2: Web Applications and Java EE 6 .....</b>	<b>35</b>
<b>Chapter 3: Working with JDBC 4.0 .....</b>	<b>53</b>
<b>Chapter 4: Working with Servlets 3.0 .....</b>	<b>151</b>
<b>Chapter 5: Handling Sessions in Servlets 3.0 .....</b>	<b>207</b>
<b>Chapter 6: Implementing Event Handling and Wrappers in Servlets 3.0.....</b>	<b>235</b>
<b>Chapter 7: Working with JavaServer Pages (JSP) 2.1 .....</b>	<b>275</b>
<b>Chapter 8: Implementing JSP Tag Extensions.....</b>	<b>327</b>
<b>Chapter 9: Implementing JavaServer Pages Standard Tag Library 1.2 .....</b>	<b>357</b>
<b>Chapter 10: Implementing Filters.....</b>	<b>401</b>
<b>Chapter 11: Working with JavaServer Faces 2.0.....</b>	<b>427</b>
<b>Chapter 12: Understanding JavaMail 1.4 .....</b>	<b>517</b>
<b>Chapter 13: Working with EJB 3.1 .....</b>	<b>547</b>
<b>Chapter 14: Implementing Entities and Java Persistence API 2.0 .....</b>	<b>611</b>
<b>Chapter 15: Implementing Java Persistence Using Hibernate 3.5 .....</b>	<b>681</b>
<b>Chapter 16: Implementing JBoss Seam.....</b>	<b>721</b>
<b>Chapter 17: Java EE Connector Architecture 1.6.....</b>	<b>761</b>
<b>Chapter 18: Java EE Design Patterns .....</b>	<b>795</b>
<b>Chapter 19: Implementing SOA using Java Web Services .....</b>	<b>825</b>
<b>Chapter 20: Working with Struts 2.....</b>	<b>917</b>

Chapter 21: Working with Spring 3.0.....	1005
Chapter 22: Securing Java EE 6 Applications.....	1041
Chapter 23: People Management Solutions .....	1079
Chapter 23: Section A: Developing the Login Module.....	1087
Chapter 23: Section B: Developing the Profile Management Module .....	1103
Chapter 23: Section C: Developing the Recruitment Module .....	1127
Chapter 23: Section D: Developing the Attendance Management Module.....	1167
Chapter 23: Section E: Developing the Leave Management Module .....	1181
Chapter 23: Section F: Developing the Payroll Module.....	1203
Appendix A: AJAX.....	1129
Appendix B: Installing Java EE 6 SDK .....	1245
Appendix C: Working with NetBeans IDE 6.8 .....	1253
Appendix D: Implementing Internationalization.....	1281
Appendix E: Working with Facelets.....	1293
Appendix F: Working with JMS.....	1303
Glossary .....	1315
Index .....	1325
What's on the CD-ROM .....	1336

# Table of Contents

<b>Introduction.....</b>	<b>xxi</b>
<b>Chapter 1: Java EE 6: An Overview .....</b>	<b>1</b>
Evolution of Java .....	2
Starting with Java .....	4
Java Programming Language.....	4
Java Runtime Environment .....	4
Java Virtual Machine .....	5
Java Platform.....	5
Exploring Enterprise Architecture Types.....	6
The Single-Tier Architecture .....	6
The 2-Tier Architecture .....	6
The 3-Tier Architecture .....	8
The n-Tier Architecture.....	9
Objectives of Enterprise Applications .....	11
Exploring the Features of the Java EE Platform .....	13
Platform Independence.....	13
Managed Objects.....	13
Reusability .....	13
Modularity .....	14
Easier Development.....	14
Simplified EJB.....	14
Enhanced Web Services .....	14
Support for Web 2.0.....	14
Exploring the New Features of the Java EE 6 Platform.....	14
Exploring the Java EE 6 Platform .....	15
The Runtime Infrastructure .....	15
The Java EE 6 APIs.....	15
Exploring the Architecture of Java EE 6 .....	18

## Table of Contents

---

Describing Java EE 6 Containers .....	19
Container Types .....	20
Java EE 6 Container Architecture.....	20
Developing Java EE 6 Applications .....	22
Probable Java EE Application Architectures .....	23
Application Development and Deployment Roles.....	25
Application Development Process.....	25
Listing the Compatible Products for the Java EE Platform.....	28
Introducing Web Servers.....	29
Introducing Application Servers.....	29
The WebLogic Application Server.....	29
The WebSphere Application Server .....	29
The JBoss Application Server .....	29
The Glassfish Application Server.....	30
Java Database Connectivity .....	30
Java Servlet.....	31
JavaServer Pages .....	31
JavaServer Faces .....	31
JavaMail.....	31
Enterprise JavaBeans .....	31
Hibernate.....	32
Seam.....	32
Java EE Connector Architecture .....	32
Web Services .....	32
Struts.....	32
Spring.....	33
JAAS.....	33
AJAX .....	33
Summary .....	33
Quick Revise .....	33
<b>Chapter 2: Web Applications and Java EE 6 .....</b>	<b>35</b>
Exploring the HTTP Protocol .....	36
Describing HTTP Requests .....	38
Describing the HTTP Responses .....	41
Introducing Web Applications .....	41
Describing Components of a Web Application .....	42
Describing Structure / Modules of Web Applications .....	45
Describing Web Containers .....	46

Exploring Web Architecture Models .....	47
Describing the Model-1 Architecture .....	47
Describing the Model-2 Architecture .....	48
Exploring the MVC Architecture .....	49
Describing the Model Component.....	50
Describing the View Component.....	50
Describing the Controller Component.....	50
Summary .....	51
Quick Revise .....	51
<b>Chapter 3: Working with JDBC 4.0 .....</b>	<b>53</b>
Introducing JDBC.....	54
Components of JDBC.....	54
JDBC Specification .....	55
JDBC Architecture.....	55
Exploring JDBC Drivers .....	56
Describing the Type-1 Driver .....	56
Describing the Type-2 Driver (Java to Native API) .....	58
Describing the Type-3 Driver (Java to Network Protocol/ All Java Driver).....	59
Describing the Type-4 Driver (Java to Database Protocol) .....	60
Exploring the Features of JDBC.....	61
Additional Features of JDBC 3.0 .....	61
New Features in JDBC 4.0.....	62
Describing JDBC APIs .....	64
The java.sql Package.....	64
The javax.sql Package .....	66
Exploring Major Classes and Interfaces .....	68
The DriverManager Class .....	68
The Driver Interface.....	69
The Connection Interface .....	70
The Statement Interface.....	72
Exploring JDBC Processes with the java.sql Package .....	75
Understanding Basic JDBC Steps.....	75
Creating a Simple JDBC Application.....	78
Working with the PreparedStatement Interface .....	82
Working with the CallableStatement Interface .....	88
Working with ResultSets.....	95
Working with Batch Updates .....	106

## Table of Contents

Describing SQL 99 Data Types.....	110
Exploring JDBC Processes with the javax.sql Package.....	127
Using DataSource to Make a Connection .....	127
Exploring Connection Pooling .....	128
Using RowSet Objects .....	131
Working with Transactions.....	144
ACID Properties.....	144
Types of Transactions.....	145
Transaction Management .....	145
Summary .....	148
Quick Revise .....	149
<b>Chapter 4: Working with Servlets 3.0 .....</b>	<b>151</b>
Exploring the Features of Java Servlet.....	152
Servlet – A Request and Response Model .....	152
Servlet and Environment State.....	153
Security Features .....	155
HTML-Aware Servlets .....	156
HTTP-Specific Servlets.....	156
Performance Features.....	157
3-Tier Applications .....	158
Web Publishing System.....	159
Exploring New Features in Servlet 3.0 .....	159
Exploring the Servlet API.....	161
Describing the javax.servlet Package.....	161
Exploring the javax.servlet.http Package .....	163
Explaining the Servlet Life Cycle .....	165
The init() Method.....	166
The service() Method .....	166
The destroy() Method .....	167
Understanding Servlet Configuration.....	167
Creating a Sample Servlet .....	169
Exploring Directory Structure .....	170
Configuring the Servlet.....	171
Packaging, Deploying and Running the Web Application .....	171
Creating a Servlet by using Annotation .....	173
Working with ServletConfig and ServletContext Objects.....	174
Working with the HttpServletRequest and HttpServletResponse Interfaces.....	175

Using the HttpServletRequest Interface.....	175
Using the HttpServletResponse Interface .....	185
Exploring Request Delegation and Request Scope .....	190
Implementing Servlet Collaboration.....	194
Collaboration through the System Properties List.....	194
Collaboration through a Shared Object.....	195
Collaboration through Inheritance .....	202
Summary .....	204
Quick Revise .....	204
<b>Chapter 5: Handling Sessions in Servlets 3.0 .....</b>	<b>207</b>
Describing a Session.....	208
Introducing Session Tracking .....	208
Exploring the Session Tracking Mechanisms .....	209
Using Cookies.....	209
Using Hidden Form Fields .....	213
Implementing URL Rewriting.....	213
Using Secure Socket Layer.....	217
Using the Java Servlet API for Session Tracking.....	217
History of Session Tracking.....	217
Session Creation and Tracking.....	218
Creating Login Application using Session Tracking .....	228
Exploring the Directory Structure of Login Application .....	228
Building the Front-End .....	228
Creating and Managing a Session.....	229
Configuring the Login Application .....	231
Running the Login Application .....	232
Summary .....	233
Quick Revise .....	234
<b>Chapter 6: Implementing Event Handling and Wrappers in Servlets 3.0.....</b>	<b>235</b>
Introducing Events.....	236
Introducing Event Handling.....	236
Working with the Types of Servlet Events.....	237
Implementing the Servlet Context Level Events .....	237
Implementing the Servlet Session Level Events.....	246
Developing the onlineshop Web Application .....	254
Creating the JavaBeans for the onlineshop Web Application .....	254
Creating the CartContextListener Class.....	258

## Table of Contents

---

Building the Front-end of the onlineshop Web Application .....	260
Introducing Wrappers .....	266
Exploring the Need for Wrappers .....	267
Exploring the Types of Wrapper Classes.....	267
Working with Wrappers .....	268
Creating the Home HTML Page .....	269
Creating the WrapperTestServlet.java File .....	269
Creating the TestServlet.java File.....	270
Creating the MyRequestWrapper.java File .....	270
Creating the MyResponseWrapper.java File.....	271
Creating the web.xml File .....	271
Packaging, Deploying, and Running the wrapper Web Application.....	272
Summary .....	273
Quick Revise .....	273
<b>Chapter 7: Working with JavaServer Pages (JSP) 2.1 .....</b>	<b>275</b>
Introducing JSP Technology .....	276
Exploring New Features of JSP 2.1.....	276
Listing Advantages of JSP over Java Servlet.....	277
Exploring the Architecture of a JSP Page .....	277
The JSP Model I Architecture .....	277
The JSP Model II Architecture.....	278
Describing the Life Cycle of a JSP Page .....	278
The Page Translation Stage.....	278
The Compilation Stage.....	279
The Loading & Initialization Stage .....	279
The Request Handling Stage .....	279
The Destroying Stage.....	280
Working with JSP Basic Tags and Implicit Objects .....	280
Exploring Scripting Tags.....	280
Exploring Implicit Objects .....	284
Explaining Directive Tags.....	291
Working with Action Tags in JSP.....	297
Exploring Action Tags.....	297
Declaring a Bean in a JSP Page .....	305
Exploring the JSP Unified EL.....	310
Understanding the Basic Syntax of using EL .....	311

Classifying EL Expressions.....	311
Describing Tag Attribute Types.....	313
Resolving EL Expressions.....	313
Describing EL Operators.....	315
Describing EL Objects .....	317
Using Functions with EL.....	323
Summary .....	325
Quick Revise .....	325
<b>Chapter 8: Implementing JSP Tag Extensions .....</b>	<b>327</b>
Exploring the Elements of Tag Extensions.....	328
The TLD File .....	328
The taglib Directive .....	329
The Tag Handler .....	329
Exploring the Tag Extension API.....	329
The Tag Extension Interfaces.....	330
The Tag Extension Classes.....	334
Working with Classic Tag Handlers.....	344
Exploring the Life Cycle of Classic Tag Handlers .....	344
Implementing Classic Tags.....	345
Working with Simple Tag Handlers.....	349
Exploring the Life Cycle of Simple Tag Handlers .....	349
Implementing Simple Tag Handler.....	349
Working with JSP Fragments.....	352
Creating a JSP Fragment .....	352
Invoking a JSP Fragment.....	352
Exploring the JspFragment Class.....	353
Working with Tag Files .....	353
Handling Dynamic Attributes in Tag Files.....	354
Exporting Variables from a Tag File to a JSP Page .....	354
Using Attributes to Provide Names for Variables .....	355
Invoking JSP Fragments from Tag Files.....	355
Summary .....	356
Quick Revise .....	356
<b>Chapter 9: Implementing JavaServer Pages Standard Tag Library 1.2 .....</b>	<b>357</b>
Introducing JSTL .....	358
Explaining the Features of JSTL .....	358
Exploring the Tag Libraries in JSTL.....	359

## Table of Contents

---

Working with the Core Tag Library .....	359
Exploring the Tags in the Core Tag Library .....	359
Using the Core Tag Library in the coreTagApp Application.....	363
Working with the XML Tag Library .....	366
Exploring the Tags of the XML Tag Library .....	367
Using the XML Tag Library in the XMLTagApp Application.....	373
Working with the Internationalization Tag Library .....	375
Exploring the Tags of the Internationalization Tag Library .....	375
Using the Internationalization Tag Library in Web Applications .....	382
Working with the SQL Tag Library .....	389
Exploring Tags of the SQL Tag Library .....	389
Using the SQL Tag Library in the SqlTagApp Application .....	395
Working with the Functions Tag Library .....	397
Exploring the Functions Available in the Functions Tag Library .....	397
Using the JSTL Functions in the JSTLFunctionApp Application .....	398
Summary .....	400
Quick Revise .....	400
<b>Chapter 10: Implementing Filters.....</b>	<b>401</b>
Exploring the Need of Filters.....	402
Exploring the Working of Filters.....	403
Exploring Filter API.....	403
The Filter Interface .....	403
The FilterConfig Interface .....	404
The FilterChain Interface .....	405
Configuring a Filter.....	405
Configuring Filters Using Deployment Descriptor .....	405
Configuring Filters Using Annotations.....	407
Creating a Web Application Using Filters .....	407
Using Deployment Descriptor to Configure a Filter .....	407
Exploring the Directory Structure of FilterApp Application .....	411
Using Annotations to Configure a Filter .....	413
Creating a Servlet to Test the Filter .....	415
Using Initializing Parameter in Filters.....	417
Creating the MsgFilter Filter .....	417
Creating a JSP Page to Test the Filter .....	417
Configuring the Filter .....	418
Testing a Filter .....	418

Manipulating Responses .....	420
Creating the ServletOutputStreamFilter Class.....	420
Creating the MyGenericResponseWrapper Class.....	421
Creating the Filter .....	422
Creating the Servlet to Test the Filter .....	423
Configuring the Filter.....	423
Testing the FilterPrePost Filter.....	424
Discussing Issues in Using Threads with Filters .....	424
Summary .....	425
Quick Revise .....	425
<b>Chapter 11: Working with JavaServer Faces 2.0.....</b>	<b>427</b>
Introducing JSF .....	428
Explaining the Features of JSF .....	429
Exploring the JSF Architecture .....	430
Describing JSF Elements.....	432
UI Component.....	432
Renderer.....	433
Validators.....	433
Backing Beans.....	434
Converters.....	435
Events and Listeners.....	435
Message .....	437
Navigation .....	438
Exploring the JSF Request Processing Life Cycle.....	438
The Restore View Phase .....	439
The Apply Request Values Phase .....	440
The Process Validations Phase .....	440
The Update Model Values Phase .....	440
The Invoke Application Phase .....	441
The Render Response Phase .....	441
Exploring JSF Tag Libraries .....	441
JSF HTML Tags .....	442
JSF Core Tags .....	459
JSF Standard UI Components .....	470
Command Components .....	472
Data Component .....	472
Form Component .....	472
Image Component .....	473

## Table of Contents

Input Component.....	473
Message and Messages Component.....	473
Output Component .....	473
Parameter Component .....	473
Checkbox Component.....	473
SelectItem and SelectItems Component.....	473
SelectMany and SelectOne Component.....	473
ViewRoot Component.....	474
Working with Backing Beans.....	474
Using the Backing Bean Method as an Event Handler .....	476
Using Backing Bean Method as Validator .....	477
Managing Backing Beans .....	477
JSF Input Validation.....	478
Using Validator Method .....	478
Using Validators .....	479
JSF Type Conversion.....	480
Standard JSF Converters .....	481
Creating Custom Converters.....	481
Handling Page Navigation in JSF .....	482
Describing Internationalization Support in JSF.....	484
Configuring Supported Locales .....	484
Creating Resource Bundles .....	484
Accessing Localized Messages from Resource Bundle .....	485
Configuring JSF Applications.....	486
Setting web.xml.....	487
Setting the faces-config.xml File.....	488
Developing a JSF Application.....	489
Setting Development Environment .....	489
Creating JSF Pages .....	490
Creating the Employee Backing Bean.....	499
String getEmployees() .....	502
String addNew().....	502
String update() .....	502
String deleteEmployee().....	502
String getDetail() .....	502
void getEmployee(ActionEvent).....	502
Managing Employee Bean .....	503
Creating the EmployeeDB Class .....	503

Creating the EmailValidator Class.....	506
Configuring a JSF Application.....	507
Enabling JSF Servlet in the web.xml File.....	507
Navigation Rules Defined in the faces-config.xml File .....	507
Supporting Internationalization.....	509
Exploring the Directory Structure of the Application .....	510
Running the KogentPro Application .....	511
Displaying All Employees .....	512
Getting Employee Detail.....	512
Adding New Employee .....	513
Editing Employee Detail .....	513
Deleting Employee.....	514
Summary .....	514
Quick Revise .....	515
<b>Chapter 12: Understanding JavaMail 1.4 .....</b>	<b>517</b>
Introducing JavaMail.....	518
Exploring the E-Mail Protocols .....	519
MIME.....	520
Establishing Communication between an E-mail Client and E-mail Server .....	520
Exploring the JavaMail Architecture.....	521
Exploring the JavaMail API .....	521
The Session Class .....	522
The Authenticator Class.....	524
The Message Class .....	524
The MimeMessage Class.....	524
The Part Interface.....	526
The Multipart Class .....	528
The ContentType Class .....	529
The MimeBodyPart Class .....	529
The PreencodedMimeBodyPart Class .....	529
The MimeUtility Class.....	530
The InternetHeader Class.....	531
The ParameterList Class.....	531
The QuotaAwareStore Interface .....	531
The Resource Class .....	532
The Quota Class .....	532
The SharedFileInputStream Class.....	532
The SharedByteArrayInputStream Class.....	533

## Table of Contents

The ByteArrayDataSource Class .....	533
The Address Class .....	533
The Store Class .....	534
The Folder Class.....	535
The Transport Class.....	541
Working with JavaMail .....	542
Sending Mails.....	542
Reading Mails.....	544
Summary .....	545
Quick Revise .....	545
<b>Chapter 13: Working with EJB 3.1 .....</b>	<b>547</b>
Understanding EJB 3 Fundamentals.....	548
Why EJB 3?.....	548
EJB 3 – Architecture and Concepts .....	549
Features of EJB 3.....	552
Classifying EJBs .....	555
Introducing Session Beans .....	555
Conversational State.....	556
State Management of a Bean .....	556
The Stateless Session Beans .....	556
The Stateful Session Beans.....	558
Stateless versus Stateful Session Beans .....	559
Implementing Session Beans .....	560
Exploring Business Interface .....	560
Exploring Bean Class.....	560
Working with a Stateless Session Bean .....	560
Working with a Stateful Session Bean.....	567
Introducing the MDB.....	571
Characteristics of the MDB .....	571
Structure of the MDB.....	572
Life Cycle of the MDB .....	572
Implementing the MDB .....	574
Implementing the MessageDrivenBean and MessageListener Interfaces .....	574
Implementing Business Logic inside the onMessage() Method.....	574
Creating a Sample MDB Application .....	574
Packaging, Deploying, and Running the Application.....	579
Managing Transactions in Java EE Applications .....	584
Exploring Transaction Properties .....	585

Exploring Transaction Model .....	586
Explaining Distributed Transactions .....	588
Implementing Transaction Management in EJB 3 .....	589
Explaining Bean-Managed Transactions .....	590
Explaining Container-Managed Transactions .....	591
Explaining EJB 3 Timer Services.....	593
Different Types of Timers .....	593
Strengths and Limitations of EJB Timer Services .....	594
Timer Service API .....	594
Implementing EJB 3 Timer Service .....	597
Creating Timer Objects.....	598
Canceling a Timer Object .....	599
Expiring the Timer Object.....	599
Exploring EJB 3 Interceptors .....	599
Specifying Interceptors.....	600
Exploring the Life Cycle of Interceptors .....	600
Working with the Interceptor Class .....	601
Applying Interceptors through XML .....	602
Disabling Interceptors .....	602
Using the Business Method Interceptors .....	603
Using the Life Cycle Callback Methods .....	603
Specifying Default Interceptor Methods .....	604
Exploring the Life Cycle Callback Methods in an Interceptor Class .....	605
The @PreDestroy Annotation .....	605
The @PostConstruct Annotation .....	606
The @PostActivate Annotation .....	606
The @PrePassivate Annotation .....	606
Exploring the Life Cycle Callback Interceptor Methods in an MDB .....	606
Exploring the Life Cycle Callback Interceptor Methods in a Session Bean .....	608
Summary .....	609
Quick Revise .....	610
<b>Chapter 14: Implementing Entities and Java Persistence API 2.0 .....</b>	<b>611</b>
Understanding Java Persistence and EntityManager API .....	612
Introducing Entities .....	613
Specifying the @ENTITY Annotation .....	619
Specifying the @Table Annotation .....	619
Specifying the @Column Annotation .....	619
Specifying the @Enumerated Annotation .....	620

Specifying the @Lob Annotation.....	621
Specifying the @Temporal Annotation .....	621
Exploring Entity and Session Beans .....	621
Describing When To Use Entity Beans.....	621
Describing an Entity Class.....	622
Describing the Life Cycle of Entity .....	622
Entity Listeners and Callbacks .....	623
Packaging a Persistence Unit.....	624
Obtaining an EntityManager.....	625
Interacting with an EntityManager .....	626
Understanding Entity Relationship Types.....	628
The One-to-One Relationship.....	628
The One-to-Many Relationship.....	634
The Many-to-One Relationship .....	638
The Many-to-Many Relationship .....	642
Mapping Collection-Based Relationships .....	647
Understanding Entity Inheritance.....	648
Single Table Per Class Hierarchy .....	651
Separate Table Per Subclass.....	653
Single Table Per Concrete Entity Class .....	654
Understanding JPQL.....	655
JPQL Functions.....	655
JPQL Statements.....	657
The SELECT Clause .....	658
The FROM Clause .....	659
The WHERE Clause .....	659
The ORDER BY Clause.....	660
Conditional Expressions .....	660
Query API .....	663
Developing Sample Application .....	667
Exploring the Directory Structure .....	667
Creating the Web Module.....	669
Configuring Connection Pool and JDBC Resource .....	676
Summary .....	679
Quick Revise .....	679
<b>Chapter 15: Implementing Java Persistence Using Hibernate 3.5 .....</b>	<b>681</b>
Introducing Hibernate .....	682
Why Hibernate? .....	682

What is New in Hibernate 3.5.....	683
Exploring the Architecture of Hibernate.....	684
Noteworthy Interfaces of Hibernate.....	685
The Hibernate Cache Architecture .....	685
Downloading Hibernate.....	687
Exploring HQL .....	688
Need of HQL .....	688
HQL Syntax .....	688
Understanding Hibernate O/R Mapping .....	692
Working with Hibernate .....	699
Setting up the Development Environment.....	699
Creating Database Table .....	699
Writing Hibernate Configuration File, JavaBean, and Hibernate Mapping File .....	699
Implementing O/R Mapping with Hibernate.....	702
Developing a JavaBean.....	703
Developing Hibernate Configuration File .....	704
Developing Hibernate Mapping File .....	704
Creating the EmployeeData.java File .....	705
Developing Controller Component.....	707
Developing View Components .....	711
Creating the web.xml File .....	713
Exploring Directory Structure .....	714
Running the Application.....	715
Summary .....	718
Quick Revise .....	719
<b>Chapter 16: Implementing JBoss Seam .....</b>	<b>721</b>
Listing the Features of the Seam Framework .....	722
Working with the Seam Framework.....	723
Understanding Contexts .....	723
Working with Seam Components.....	724
Using Annotations .....	731
Implementing BPM and Page Flow in Seam .....	734
Stateless Navigation Model .....	735
Stateful Navigation Model.....	735
Using jPDL Pageflows .....	736
Configuring JBoss Seam .....	737
Configuring JSF in Seam .....	738
Configuring EJB components in Seam .....	739

## **Table of Contents**

Creating a Jboss Seam Application .....	739
Creating an EJB Component.....	740
Creating Views .....	743
Creating Resources .....	748
Packaging and Deploying the Seam Application .....	753
Running the Application.....	756
Summary .....	759
Quick Revise .....	760
<b>Chapter 17: Java EE Connector Architecture 1.6.....</b>	<b>761</b>
Describing the Key Concepts of the JCA.....	762
Enterprise Information Systems.....	763
Resource Manager .....	764
Resource Adapter .....	764
Managed Environment .....	764
Non-Managed Environment .....	765
Connection of an Application Client with a Resource Manager.....	765
System Contracts.....	765
Common Client Interface.....	766
Describing the Life Cycle Management of a Resource Adapter.....	767
Bootstrapping a Resource Adapter Instance .....	769
Understanding a ManagedConnectionFactory JavaBean and Outbound Communication.....	770
Understanding an ActivationSpec JavaBean and Inbound Communication.....	771
Managing the Life Cycle of a Resource Adapter .....	771
Exploring Workflow Management .....	773
Listing the Advantages of Workflow Management .....	773
Describing the Work Management Model.....	773
Describing the Work Interface.....	774
Describing the ExecutionContext Class .....	775
Describing the WorkListener Interface .....	775
Describing the WorkEvent Class .....	776
Describing the WorkAdapter Class .....	776
Exploring the Differences between JDBC and JCA.....	777
Exploring the Inbound Communication Model.....	777
Discussing a Scenario using Inbound Communication .....	778
Exploring Message Inflow from EIS to Resource Adapter .....	779
Exploring the Message Inflow to Message Endpoints .....	781
Exploring Activation Specifications (JavaBean).....	781

Exploring Administered Objects.....	783
Understanding EJB Invocation .....	784
Understanding the CCI API.....	785
The ConnectionFactory Interface .....	786
The ConnectionSpec Interface .....	787
The Connection Interface .....	787
The Interaction Interface .....	788
The InteractionSpec Interface .....	788
The LocalTransaction Interface .....	788
Exploring JCA Exceptions.....	789
The Application Exception .....	789
The System Exception .....	789
Packaging and Deploying a Resource Adapter.....	790
Understanding Directory Structure of a Resource Adapter.....	790
Packaging Considerations .....	791
Packaging a Resource Adapter .....	791
Deploying a Resource Adapter .....	792
Explaining the Deployment Descriptor for a Resource Adapter .....	792
Explaining the Role of Parties Involved in Deployment of a Resource Adapter .....	793
Summary .....	793
Quick Revise .....	793
<b>Chapter 18: Java EE Design Patterns .....</b>	<b>795</b>
Describing the Java EE Application Architecture .....	796
Introducing a Design Pattern.....	797
Discussing the Role of Design Patterns .....	797
Exploring Types of Patterns.....	797
The Front Controller Pattern .....	799
The Composite View Pattern.....	802
The Composite Entity Pattern .....	804
The Intercepting Filter Pattern .....	806
The Transfer Object Pattern .....	808
The Session Facade Pattern.....	811
The Service Locator Pattern .....	813
The Data Access Object Pattern .....	815
The View Helper Pattern .....	817
The Dispatcher View Pattern.....	819
The Service To Worker Pattern .....	821

## Table of Contents

---

Summary .....	823
Quick Revise .....	823
<b>Chapter 19: Implementing SOA using Java Web Services .....</b>	<b>825</b>
Overview of SOA .....	826
Describing the SOA Environment .....	827
The Core Layer .....	828
The Platform Layer .....	828
The Quality of Services Layer .....	829
Overview of JWS .....	830
Role of WSDL, SOAP, and Java/XML Mapping in SOA .....	831
Role of WSDL in SOA .....	831
Role of SOAP in SOA .....	831
Role of Java/XML Mapping in SOA .....	831
Exploring the JAX-WS 2.2 Specification .....	839
The Invocation Sub-Specification Category .....	840
The Serialization Sub-Specification Category .....	841
The Deployment Sub-Specification Category .....	841
Exploring the JAXB 2.2 Specification .....	841
Mapping Annotations .....	841
Binding Runtime Framework .....	841
Implementing Validation .....	841
Exploring Marshal Event Callbacks .....	841
Exploring Partial Binding .....	841
Exploring Binary Data Encoding .....	841
Exploring JAXB Binding Language .....	841
Explaining Portability .....	841
Exploring the WSEE 1.3 Specification .....	841
Port Component .....	841
Servlet Endpoints .....	841
EJB Endpoints .....	841
Simple Packaging .....	841
Handler Programming Model .....	841
Exploring the WS-Metadata 2.2 Specification .....	841
WSDL Mapping Annotations .....	851
SOAP Binding Annotations .....	851
Handler Annotations .....	851
Service Implementation Bean .....	851
Start From WSDL and Java .....	851

Automatic Deployment.....	851
Describing the SAAJ 1.3 Specification .....	851
Working with SAAJ and DOM APIs .....	851
Creating a Simple SOAP Message .....	852
Accessing the different Message Parts of a SOAP Message .....	852
Adding Content to the SOAPBody Object.....	852
Sending a Message.....	853
Retrieving the Content of a Message.....	854
Adding Content to the Header Element .....	854
Creating and Adding Attachments .....	854
Retrieving Attachments .....	855
Describing the JAXR Specification .....	855
JAXR Architecture.....	855
JAXR Client.....	856
JAXR Provider .....	856
Exploring the StAX 1.0 Specification .....	857
StAX APIs .....	857
StAX Factory Classes .....	858
Using the JAX-WS 2.2 Specification .....	859
Invoking Web Services by using JAX-WS Proxies.....	860
Implementing JAX-WS WSDL to Java Mapping .....	860
Marshalling and Unmarshalling Method Calls to SEI .....	862
Invoking a Web Service using a Proxy.....	864
Using the JAXB 2.2 Specification .....	864
Binding between XML Schema and Java Classes .....	865
Customizing JAXB Binding .....	865
Exploring an Example of JAXB 2.2 Java/XML Binding .....	869
Implementing Type Mappings with JAXB 2.2 .....	874
Implementing Type Mappings with JAXB 2.2 Annotations.....	878
Using the WSEE and WS-Metadata Specifications .....	882
Deployment using a Servlet Endpoint .....	883
Deployment using an EJB Endpoint.....	883
Deployment without Deployment Descriptors.....	884
Deployment with Deployment Descriptor .....	890
Implementing the SAAJ Specification .....	896
Implementing the JAXR Specification .....	900
Setting up a Connection .....	900
Querying a Registry.....	901

## Table of Contents

---

Manipulating Registry Objects.....	903
Implementing the StAX Specification.....	906
Reading XML Streams.....	906
Writing XML Streams.....	907
Reading an XML File using the Cursor API.....	908
Reading an XML File using the Event Iterator API.....	911
Writing an XML File using the Cursor API.....	913
Summary .....	914
Quick Revise .....	914
<b>Chapter 20: Working with Struts 2.....</b>	<b>917</b>
Introducing Struts 2 .....	918
Explaining MVC 2 Design Pattern for Struts 2.....	918
The Need for Struts 2.....	919
Processing Request in Struts 2.....	919
Exploring Relation between WebWork 2 and Struts 2.....	920
Describing Struts 2 Architecture.....	921
Exploring Struts 2 Configuration Files.....	922
Explaining Zero Configuration Applications.....	925
Exploring Struts 2 Annotations.....	926
Understanding Actions in Struts 2.....	926
Action Classes .....	926
POJO as Action.....	935
Implementing Actions in Struts 2 .....	935
Dependency Injection and Inversion of Control .....	951
The ApplicationAware Interface.....	952
The ParameterAware Interface .....	952
The ServletRequestAware Interface .....	953
The ServletResponseAware Interface .....	954
The SessionAware Interface .....	954
Preprocessing with Interceptors.....	954
What are Interceptors? .....	955
Interceptors as RequestProcessor .....	955
How to Configure Interceptors? .....	955
Stacking of Interceptors.....	956
Bundled Interceptors .....	956
Writing Interceptors .....	957
OGNL Support in Struts 2.....	959
Syntax of OGNL.....	959

Using OGNL in Struts 2 .....	960
Implementing Struts 2 Tags .....	961
Generic Tags .....	961
UI Tags .....	963
Controlling Results in Struts 2 .....	964
What is Result? .....	964
Types of Results .....	964
Configuring Results.....	965
Configuring Result Types .....	966
Performing Validation in Struts 2 .....	967
XWork Validation framework.....	967
Bundled Validators.....	967
Registering Validators.....	969
Defining Validation Rules.....	970
Custom Validators .....	970
Short-circuiting Validators .....	971
Validation Annotation.....	972
ConversionErrorFieldValidator Annotation .....	972
Validating Stuts2App Application .....	979
Internationalizing Struts 2 Applications .....	983
Describing Internationalization and Localization .....	983
Global Resource Bundle .....	985
Implementing Plugins in Struts 2 .....	986
Struts 2 Bundled Plugins .....	986
Tiles Plugin .....	987
Integrating Struts 2 with Hibernate .....	999
Setting the MySQL Database and Table.....	1000
Configuring Hibernate .....	1000
Developing Struts Hibernate Plugin .....	1002
Summary .....	1003
Quick Revise .....	1003
<b>Chapter 21: Working with Spring 3.0.....</b>	<b>1005</b>
Introducing Features of the Spring Framework .....	1006
What's New in Spring 3.0.....	1007
Exploring the Spring Framework Architecture .....	1007
Explaining the Spring Core Module.....	1008
Explaining the Spring AOP Module.....	1008
Explaining the Spring ORM Module.....	1008

## Table of Contents

---

Explaining the Spring Web MVC Module .....	1009
Explaining the Spring Web Flow Module .....	1009
Explaining the Spring DAO Module .....	1009
Explaining the Spring Application Context Module.....	1009
Exploring Dependency Injection and Inversion of Control.....	1009
Explaining DI.....	1010
Explaining IoC Container .....	1010
Exploring AOP with Spring .....	1011
Describing the AOP Concepts.....	1011
Explaining Types of Advices.....	1012
Spring AOP Capabilities and Its Goals .....	1012
Managing Transactions .....	1012
Need of Transaction Management Support .....	1013
Spring Transaction Abstraction .....	1013
Resource Synchronization with Transactions .....	1013
Declarative Transaction Management.....	1016
Programmatic Transaction Management.....	1016
Choosing between Programmatic and Declarative Transaction Managements .....	1017
Application Server-Specific Integration.....	1018
Exploring Spring Form Tag Library .....	1018
Classifying the Types of Tags.....	1018
Exploring Spring's Web MVC Framework.....	1025
Key Features of Spring Web MVC.....	1025
The DispatcherServlet Class .....	1026
Controllers .....	1027
Handler Mappings.....	1029
Views and View Resolvers .....	1029
Implementing Spring Web MVC Framework .....	1030
Creating the Controller .....	1031
Creating the Views.....	1031
Creating the Spring Configuration File.....	1032
Creating the Web Configuration File .....	1032
Exploring the Directory Structure .....	1033
Running the Application.....	1034
Testing Spring Applications .....	1034
The Unit Testing.....	1034
The Integration Testing .....	1034
Integrating Spring with Hibernate .....	1037

Integrating Struts 2 with Spring .....	1037
Configuring Spring in a Struts 2 Application.....	1038
Summary .....	1039
Quick Revise .....	1039
<b>Chapter 22: Securing Java EE 6 Applications .....</b>	<b>1041</b>
Introducing Security in Java EE 6.....	1042
Authentication.....	1042
Protection Domain.....	1043
Exploring Security Mechanisms.....	1045
Application Layer Security .....	1045
Transport Layer Security .....	1045
Message Layer Security.....	1046
Implementing Security on an Application Server.....	1046
Realm.....	1047
User.....	1047
Group .....	1047
Role .....	1048
Securing Enterprise Beans.....	1048
Using the Programmatic Security Approach .....	1049
Using Security Identity .....	1049
Securing Application Clients .....	1050
Implementing Security in Web Applications .....	1050
Using JAAS .....	1050
Using Authentication Mechanisms .....	1053
Implementing Security .....	1055
Describing Declarative Security .....	1055
Implementing Programmatic Security .....	1067
Summary .....	1077
Quick Revise .....	1078
<b>Chapter 23: People Management Solutions .....</b>	<b>1079</b>
Software Requirements .....	1080
SDLC of the Project .....	1080
Requirement Analysis .....	1080
Software Design .....	1080
Database Design.....	1082
Development .....	1085
Testing .....	1085

**Table of Contents**

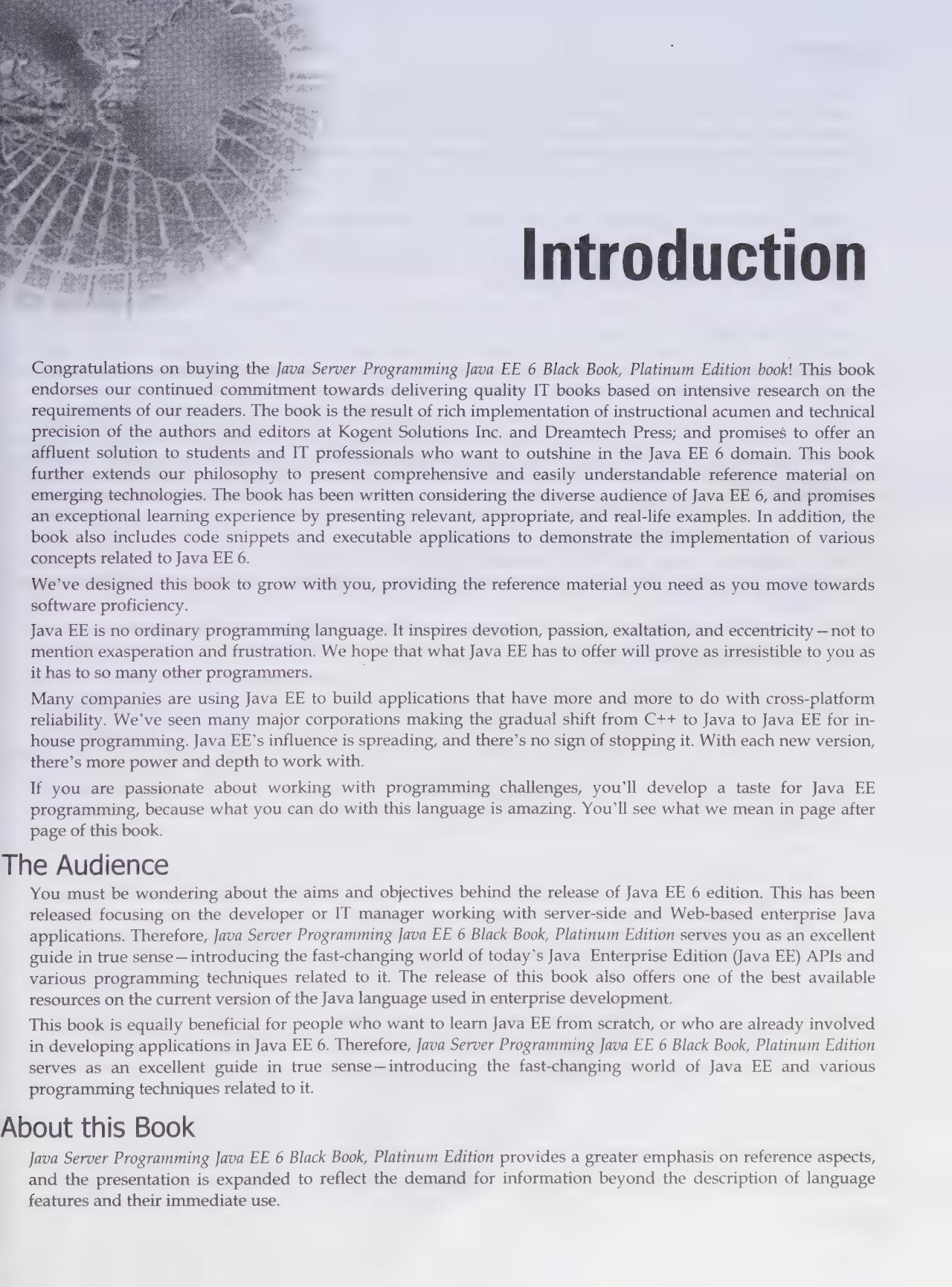
Implementation and Maintenance.....	1086
Summary .....	1086
<b>Chapter 23: Section A: Developing the Login Module.....</b>	<b>1087</b>
Designing Login User Interface.....	1088
Creating the people_user_login Servlet.....	1089
Creating the UserLoginDBObj Class .....	1092
Creating the UserLoginDBMethods Class.....	1093
Creating the people_default.jsp File.....	1095
Directory Structure of the Project.....	1098
Login and Navigating to Home Page.....	1099
Changing Password.....	1101
<b>Chapter 23: Section B: Developing the Profile Management Module .....</b>	<b>1103</b>
Implementing Logic with Servlet.....	1104
Creating the people_employee Servlet.....	1104
Creating the EmployeeObj Class .....	1108
Creating the EmployeeDBMethods Class.....	1108
Creating the GenerateId Class.....	1111
Creating Views .....	1112
Creating the employee_insert JSP Page .....	1112
Creating the employee_search JSP Page .....	1116
Creating the employee_edit JSP Page.....	1118
Creating the employee_list JSP Page.....	1121
Creating the employee_profile JSP Page.....	1123
<b>Chapter 23: Section C: Developing the Recruitment Module .....</b>	<b>1127</b>
Registering a New Applicant.....	1128
Creating the people_applicant Servlet.....	1128
Creating the ApplicantDBObj Class .....	1131
Creating the ApplicantDBMethods Class.....	1132
Creating the GenerateId Class.....	1138
Creating an Interface for Applicant Registration.....	1138
Conducting Rounds of Test .....	1149
Creating the applicant_test_dtl Servlet.....	1149
Designing JSP Views .....	1153
Working of the Recruitment Module .....	1164

<b>Chapter 23: Section D: Developing the Attendance Management Module.....</b>	<b>1167</b>
Creating the time_management Servlet .....	1168
Creating the Classes in the com.TimeManagement Package .....	1172
Creating JSP Views.....	1176
Creating the employee_daily_attendance JSP Page .....	1176
Creating the employee_daily_attendance_summary JSP Page.....	1179
<b>Chapter 23: Section E: Developing the Leave Management Module .....</b>	<b>1181</b>
Creating the leave_management Servlet.....	1182
Creating the LeaveRequest Class.....	1185
Creating the LeaveMgmtBeanMethods Class .....	1186
Creating the GenerateId Class.....	1190
Designing JSP Views.....	1190
Creating the leave_request JSP Page .....	1191
Creating the leave_request_edit JSP Page.....	1195
Creating the leave_request_reject JSP Page .....	1197
Creating the leave_request_list JSP Page .....	1200
<b>Chapter 23: Section F: Developing the Payroll Module.....</b>	<b>1203</b>
Updating Salary Statement .....	1204
Creating people_payroll Servlet.....	1204
Creating the EmpSal Class.....	1211
Creating the EmployeeAgreement Class .....	1211
Creating the PayrollBeanMethods Class.....	1211
Designing JSP Views.....	1218
Creating the employee_agreement JSP Page .....	1218
Creating the employee_agreement_edit JSP Page .....	1222
Creating the salary_search.jsp File .....	1224
Creating the salary_slip JSP Page .....	1226
<b>Appendix A: AJAX.....</b>	<b>1229</b>
<b>Appendix B: Installing Java EE 6 SDK .....</b>	<b>1245</b>
<b>Appendix C: Working with NetBeans IDE 6.8 .....</b>	<b>1253</b>
<b>Appendix D: Implementing Internationalization.....</b>	<b>1281</b>
<b>Appendix E: Working with Facelets.....</b>	<b>1293</b>
<b>Appendix F: Working with JMS.....</b>	<b>1303</b>

**Table of Contents**

---

Glossary .....	1315
Index .....	1325
What's on the CD-ROM .....	1336



# Introduction

Congratulations on buying the *Java Server Programming Java EE 6 Black Book, Platinum Edition* book! This book endorses our continued commitment towards delivering quality IT books based on intensive research on the requirements of our readers. The book is the result of rich implementation of instructional acumen and technical precision of the authors and editors at Kogent Solutions Inc. and Dreamtech Press; and promises to offer an affluent solution to students and IT professionals who want to outshine in the Java EE 6 domain. This book further extends our philosophy to present comprehensive and easily understandable reference material on emerging technologies. The book has been written considering the diverse audience of Java EE 6, and promises an exceptional learning experience by presenting relevant, appropriate, and real-life examples. In addition, the book also includes code snippets and executable applications to demonstrate the implementation of various concepts related to Java EE 6.

We've designed this book to grow with you, providing the reference material you need as you move towards software proficiency.

Java EE is no ordinary programming language. It inspires devotion, passion, exaltation, and eccentricity—not to mention exasperation and frustration. We hope that what Java EE has to offer will prove as irresistible to you as it has to so many other programmers.

Many companies are using Java EE to build applications that have more and more to do with cross-platform reliability. We've seen many major corporations making the gradual shift from C++ to Java to Java EE for in-house programming. Java EE's influence is spreading, and there's no sign of stopping it. With each new version, there's more power and depth to work with.

If you are passionate about working with programming challenges, you'll develop a taste for Java EE programming, because what you can do with this language is amazing. You'll see what we mean in page after page of this book.

## The Audience

You must be wondering about the aims and objectives behind the release of Java EE 6 edition. This has been released focusing on the developer or IT manager working with server-side and Web-based enterprise Java applications. Therefore, *Java Server Programming Java EE 6 Black Book, Platinum Edition* serves you as an excellent guide in true sense—introducing the fast-changing world of today's Java Enterprise Edition (Java EE) APIs and various programming techniques related to it. The release of this book also offers one of the best available resources on the current version of the Java language used in enterprise development.

This book is equally beneficial for people who want to learn Java EE from scratch, or who are already involved in developing applications in Java EE 6. Therefore, *Java Server Programming Java EE 6 Black Book, Platinum Edition* serves as an excellent guide in true sense—introducing the fast-changing world of Java EE and various programming techniques related to it.

## About this Book

*Java Server Programming Java EE 6 Black Book, Platinum Edition* provides a greater emphasis on reference aspects, and the presentation is expanded to reflect the demand for information beyond the description of language features and their immediate use.

The range of topics and coverage offered by this book is a superior mix of APIs; though some readers might quibble with the ordering of topics here, (it is hard to see why JDBC and Servlet begin the tour of Java EE). We know that Java EE 6 is the latest version of Java Platform, Enterprise Edition, which is several years old, and its APIs have grown by leaps and bounds. To cover the older material, the authors have been careful while highlighting what's new and improved. At each juncture, they have done a fine job of listing relevant APIs; because of which the book makes an excellent reference for everyday programming.

This volume promises to increase your productivity by its exact presentation of Web and EJB deployment (using freeware Java deployment tools) and the Glassfish V3 application server, which is used for deploying components. The full tour of deployment descriptor options for Servlets and EJBs, as provided in the book, will also be appreciated by the working Java developers.

The ways to design truly scalable and maintainable enterprise systems with Java, combined with JSPs, Servlets, and EJBs, has been presented excellently in the book. This book also provides a discussion on software patterns (such as the Front Controller pattern), illustrated with real code. You can be the master of this important emerging technology with the help of the coverage of custom tag libraries; in addition to the evolving JSP Standard Tag Library (JSTL) from Sun and Apache.

This book promises to be an almost indispensable resource for any enterprise Java developer, due to its extensive coverage of today's rich and complex Java EE 6 platform, and practical focus on real-world design and deployment. Therefore, it will serve as both a reference and tutorial to the latest in high-end Java for your next large-scale project.

This book mainly covers the following:

- ❑ Java EE container architecture and runtime services related to it
- ❑ Web component development with Servlets 3.0 and JavaServer Pages 2.2
- ❑ Comprehensive coverage about business logic components with EJB 3.1, all inclusive of session bean and JPA
- ❑ Java EE 6 technologies for various distributed development, such as RMI, JDBC and JNDI
- ❑ SOA architecture, which works with Web services covering SOAP and WSDL.
- ❑ Struts 2.2, Spring 3.0, Java Server Faces 2.0, Hibernate 3.5, and Seam framework.
- ❑ Packaging, deploying, and running all Web and Enterprise applications on Glassfish Application Server.
- ❑ The People Management System project, which helps in handling HR operations for large organizations.

There are hundreds of topics covered in this book, and each of them is explained by applications showing how it works. This book is divided into separate and easily accessible topics, each addressing a different programming issue. The following are just a few of those topics:

- ❑ EJB 3.1
- ❑ JDBC 4.0
- ❑ Web containers
- ❑ Servlets 3.0
- ❑ JSP 2.2
- ❑ JSTL
- ❑ Struts 2.2
- ❑ JSF 2.0
- ❑ Web services
- ❑ XML
- ❑ JMX
- ❑ JavaMail
- ❑ JCA 1.6
- ❑ Spring 3.0
- ❑ Seam

- Hibernate 3.5
- UML

The web tier technology chapters cover the components used in developing the presentation layer of a Java EE or stand-alone Web application:

- Java Servlet
- JavaServer Pages (JSP)
- JavaServer Pages Standard Tag Library (JSTL)
- JavaServer Faces
- Web application internationalization and localization

The Enterprise JavaBeans (EJB) technology chapters cover the components used to develop the Business logic of a Java EE application:

- Session beans
- Entity beans
- Message-driven beans
- JPA
- Hibernate

The platform services chapters cover the system services used by all the Java EE component technologies:

- Transactions
- Resource connections
- Security
- Java Message Service

Our sole intent has been to provide a book with a depth sufficient to make more than one reading rewarding to most programmers. Enjoy reading!

## How to Use This Book

In this book, most of the code is tested with the Java Standard Edition 5 and 6 SDK, and the Glassfish V3 application server providing support for the Java EE 6 platform. In few chapters, such as Hibernate, Seam, and Spring, you need some additional APIs to develop and run the applications.

## Conventions

We have used standard conventions throughout this book. This section helps you get acquainted with these conventions to ensure a superior learning experience.

Listing 4.12 provides the code for the `Servlet2Servlet.java` file (you can find this file on the CD in the `code\JavaEE\Chapter4\FirstApp\src\com\kogent` folder):

**Listing 4.12:** Displaying the Code for the `Servlet2Servlet.java` File

```
package com.kogent;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Servlet2Servlet extends HttpServlet
{
    public void doGet(HttpServletRequest request, HttpServletResponse
        response) throws IOException, ServletException
    {
        response.setContentType("text/html");
        String param = request.getParameter("value");

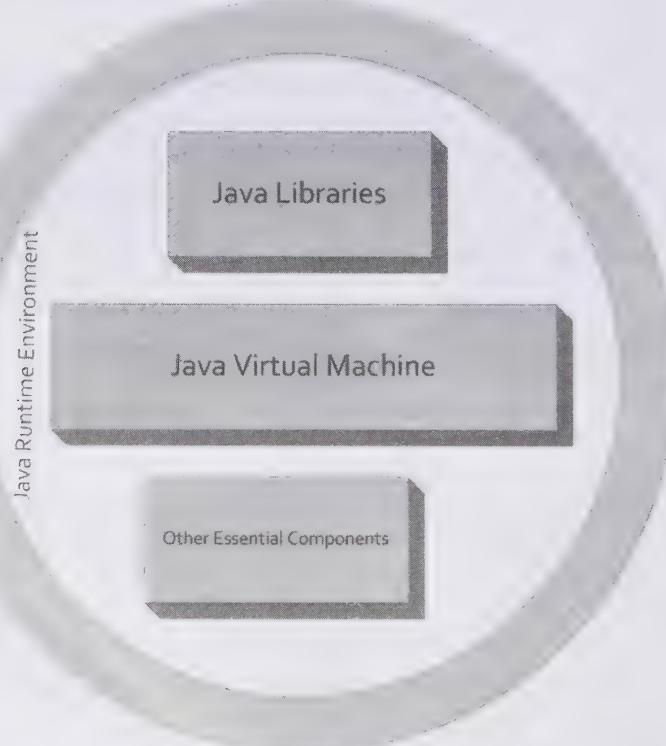
        if(param != null && !param.equals(""))
        {
            response.getWriter().println("Value is " + param);
        }
    }
}
```

```
        request.setAttribute("value", param);
        RequestDispatcher rd =
request.getRequestDispatcher("/servlet2servlet2");
        rd.forward(request, response);
        return;
    }

PrintWriter out = response.getWriter();
out.println("<html>");
out.println("<head>");
out.println("<title>Servlet #1</title>");
out.println("</head>");
out.println("<body>");
out.println("<h1>A form from Servlet #1</h1>");
out.println("<form>");
out.println("Enter a value to send to Servlet #2.");
out.println("<input name=\"value\"><br>");
out.print("<input type=\"submit\" \">");
out.println("value=\"Send to Servlet #2\"");
out.println("</form>");
out.println("</body>");
out.println("</html>");
}
}
```

The reserved words, properties, methods, events, classes, interfaces, namespaces (used in the code snippets), exceptions, etc. are shown distinctly using a different font within the text.

Each figure has a caption to help you understand the figure better.



**Figure 1.1: Displaying the Components of Java Runtime Environment**

Notes are represented in the following format:

**NOTE**

*It is recommended that the interface be as generic as possible to avoid changes at a later stage. As you know, all objects communicate with the interface and not the object itself; therefore, changes in the object and not the interface make the process relatively simple and quick.*

The tables are numbered and are placed just below their references in the chapters, as shown below:

**Table 4.2: Response Header Fields and their Values**

Header Field	Header Value
Age	Represents the estimated time since the last response generated from the server. The value of this header field is usually a positive integer.
Content-Length	Indicates the size of the message body, in decimal number of octets (8-bit bytes), sent to a recipient.
Content-Type	Refers to the MIME type corresponding to the content of an HTTP response. A browser can use this value to determine whether the content is rendered internally or launched to be rendered by an external application.
Date	Represents the date and time at which a message originated.
Location	Specifies the location of a new resource in case HTTP response codes redirect a client to such a resource. The location is specified as an absolute address.
Pragma	Specifies the implementation-specific directives that may be applied to any recipient along the request-response chain. <i>No-cache</i> , which indicates that a resource should not be cached, is the most commonly used value.
Retry-After	Indicates the tentative duration for which a service is unavailable to a requesting client. It is used with a 503 (Service Unavailable) response. A date can be returned as a value of this field. The value can also be an integer representing the number of seconds (in decimals) after the time of the response.
Server	Represents information about the server that generated the current response, as a String value.

## Other Resources

Java EE 6 comes with an immense amount of documentation—hundreds of books worth referring to. The documentations are hosted online, and you can access them using a Web browser. Of course, there are plenty of Web pages out there on Java EE 6. Here are a few:

- ❑ Sun's JNDI SDK.
- ❑ LDAP server Netscape's iPlanet Directory Server, <http://www.iplanet.com>
- ❑ JSP Standard Tag Library, <http://jakarta.apache.org/taglibs>
- ❑ IBM Web Service Toolkit, <http://www.alphaworks.ibm.com/tech/ettkws>
- ❑ Spring framework 3.0, <http://www.springframework.org>
- ❑ Hibernate 3.5, <http://www.hibernate.org/6.html>

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

## The Black Book Philosophy

“Complexity kills. It sucks the life out of developers, it makes products difficult to plan, build and test, it introduces security challenges, and it causes end-user and administrator frustration.”

Ray Ozzie – Chief Software Architect at Microsoft Corporation

Whichever language or platform you might be learning or working on, being a prolific programmer and administrator is fraught with challenges, complications, complexities, and frustrations. We at Kogent Learning Solutions, Inc. and Dreamtech Press realize this veracity seamlessly; and ensure that our **Black Book** series comes as an accomplished and dexterous solution to these complexities. The unique structure of Black Books ensures that complex topics are chunked into appropriate sections and presented in a more comprehensible manner, employing diagrams, code, and real-life examples for further simplicity. In addition, Black Books ensure a learning path replete with adequate practice avenues of the concepts learned by means of code examples, listings, executable programs and applications. Each Black Book is designed to grow with its readers, providing all the guidance and reference material they need as they move towards software proficiency. In summation, the Black Book philosophy extends beyond simple pedagogy; and seeks to decipher the practical challenges imbibed in the professional roles of its readers, whether students or professionals – that is why we call it an “Comprehensive Problem Solver”!

# Java EE 6: An Overview

## If you need an information on:

See page:

Evolution of Java	2
<b>Starting with Java</b>	4
Exploring Enterprise Architecture Types	6
Objectives of Enterprise Applications	11
Exploring the Features of the Java EE Platform	13
<b>Exploring the New Features of the Java EE 6 Platform</b>	14
Exploring the Java EE 6 Platform	15
<b>Exploring the Architecture of Java EE 6</b>	18
Describing Java EE 6 Containers	19
Developing Java EE 6 Applications	22
Listing the Compatible Products for the Java EE Platform	28
<b>Introducing Web Servers</b>	29
Introducing Application Servers	29
<b>Java Database Connectivity</b>	30
Java Servlet	31
JavaServer Pages	31
JavaServer Faces	31
<b>JavaMail</b>	31
Enterprise JavaBeans	31
<b>Hibernate</b>	32
Seam	32
<b>Java EE Connector Architecture</b>	32
Web Services	32
Struts	32
Spring	33
JAAS	33
AJAX	33

Java Enterprise Edition 6 (Java EE 6) is the latest Java platform to develop enterprise applications. Enterprise applications are applications that connect different business departments of an organization, such as Human Resource and Marketing, by using various types of user interfaces. In other words, enterprise applications are software applications that help an organization to manage its business activities. Enterprise applications can be accessed by the end users or partners of the organization through the Internet, intranet, or private networks.

The advancements in communication technology and the Internet have opened a whole new world of opportunities for business organizations. Moreover, with the growth of e-commerce, data has become a valuable asset for an organization. This change in the business environment and the increased importance of data has led to an information-based economy, which has forced many organizations to think for a solution for the smooth functioning of the most basic business practices. As a consequence, the need to simplify and restructure the implementation of business practices was felt, which in turn led to the development of Java EE.

Java EE is provided to Java programmers in the form of an integrated software package that supports a distributed computing architecture, provides definitions to package distributed components, and deploys these components. Java EE is targeted at large-scale business systems. The applications developed by using Java EE are partitioned into functional pieces, which help an organization to ensure scalability and easy maintenance of the applications. Java EE facilitates software deployment by providing standard interfaces and services. Interfaces define how various software modules interconnect with each other, and standard services define how different software modules communicate with each other in an application.

This book introduces you to the most sought after Java platform, i.e., Java EE 6. It explores the Java EE 6 platform, providing an in-depth view of its capabilities, and also introduces you to a wide range of Java EE 6 technologies. Moreover, it highlights the features and functionalities introduced in Java EE 6, which have changed the way enterprise application components are developed and configured in an enterprise application.

This chapter is divided into various sections that provide an overview of Java and Java EE 6. Section A, *Introducing Java*, deals with the history and evolution of Java and discusses the Java platform. Section B, *Enterprise Architecture*, discusses the types of enterprise architectures based on which an enterprise application is created. Section C, *Introducing the Java EE 6 Platform*, discusses Java EE 6, its features, and the various types of Java EE applications that can be developed. The next section, Section D, *Introducing the Servers for Java EE Applications*, discusses various Web and application servers used to deploy and execute Web and enterprise applications. The last section, Section E, *Snapshot of Java EE 6 Related Technologies*, provides an outline of Java EE 6 related technologies described in the book.

## **Section A: Introducing Java**

Java is a platform-independent programming language used to create secure and robust applications that may run on a single computer or may be distributed among servers and clients over a network. Java features such as platform-independency and portability ensure that while developing Java EE enterprise applications, you do not face the problems related to hardware, network, and the operating system.

Now, let's trace the evolution of Java.

### **Evolution of Java**

Before the advent of Java, C was an extremely popular language among programmers. It seemed to be the perfect programming language, combining the best elements of both low-level and high-level languages into a single programming language.

However, similar to the other programming languages, C too had its limitations. As programs written in C grew longer, they became more unwieldy and difficult to use. The reason for this was that there was no easy way to break a long C program into multiple self-contained compartments. This meant that the code in the first line of the program could interfere with the code in the last line, and the programmer had to keep the whole code in mind while writing the program.

Object Oriented Programming (OOP) provided a solution to the problem. With OOP, it is possible to break down long programs into semi-autonomous units. In other words, OOP introduced the motto *divide and conquer*. That is, you could divide a program into easily conceptualized units, which could be combined to

present the solution of a larger problem. Let's understand this with an example. Suppose you have a complex system to preserve food at low temperature, for which you perform various operations manually, such as monitoring the temperature of the food by using a thermometer. When the temperature reaches a certain high point, you start the compressor to circulate the coolant, and then start a fan to blow air over the cooling vanes. This system represents one way of preserving your food. However, there is another way as well: You can connect all these operations into an easily conceptualized unit and make them automatic, similar to a refrigerator. The larger process of preserving food is broken down into smaller processes, each of which is implemented independently. These smaller processes are:

- Monitoring the temperature of the food
- Starting the compressor when the temp reaches a specific point
- Starting a fan to blow air over the cooling vanes

When all these smaller processes are combined as one large process, you arrive at the actual solution of a refrigerator. Apart from breaking down a larger process into smaller processes, you should also note that the internal processing are hidden from your view, and all you have to do is put the food in the refrigerator and take it out when it has to be consumed. In other words, OOP also provides the abstraction feature to hide the complexity of a program or an application from its users.

Objects in OOP work similar to the small processes: they hide the programming details from the rest of the program and reduce the interdependencies that spring up in a long C program. They do this by setting up a well-defined and controllable interface that handles the connection between each object and the rest of the code. To understand the concept better, let's take an example. Suppose you have an object called *Screen*, which contains the details to handle all interactions with the screen. You can use this object in different ways depending on the purpose of using the object (in this case, the screen display). After creating the object, you can simply use the screen object instead of providing the code to handle the screen.

When the OOP feature was added to the C language, the language was renamed as C++. C++ allows programmers to deal with long programs and object oriented code. Apart from this, several other problems were solved as well. For example, supporting objects makes it easier for software manufacturers to provide a user a lot of prewritten, ready-to-use code. To create an object, you use a *class*, which serves as an object's type, similar to a variable referring to the variable type. Support for classes in the C++ language allows software manufacturers to include huge readymade class libraries that users can use to create objects easily. One of the most popular libraries of C++ classes is the Microsoft Foundation Class (MFC) library, which comes with Microsoft Visual C++.

When you write a Windows program in C, you need several pages of solid code just to display a blank window. However, by using a class in the MFC library, you can create the window of your choice easily, by creating an object of the window. The object already has built-in functionalities of the type of window you want to create, so all it takes to create the window is one line of code—the line where you create the new window object by using the selected class.

You can also use an MFC class as a *base class* for your classes, and add the functionality that you want to those classes, by using a process called *inheritance* in OOP. For example, suppose you want your window to display a menu bar. You can do this by *deriving* your own class from a plain MFC window and adding a menu bar to the class. In this way, you can build your own class just by adding a few lines of code to implement the classes that have already been created by Microsoft programmers. (You learn about OOP in depth in this book).

The introduction of class libraries and the concept of using objects were welcomed by Microsoft programmers, and C++ rapidly gained popularity. It seemed as though the perfect programming language had arrived. However, the programming environment itself was about to undergo a great change with the popularization of a new programming environment—the Internet. With the advent of the Internet, Java gained more popularity.

The first version of Java appeared in 1991, and was written in 18 months at Sun Microsystems. Java was not originally created as an Internet programming language. In fact, it was not even called *Java* but *Oak* in those days, and was used internally at Sun Microsystems.

## Starting with Java

Before you start working with Java, you must get familiar with the underlying concepts of Java as a programming language. Java refers to a combination of the following four aspects:

- Java programming language
- Java Runtime Environment (JRE)
- Java Virtual Machine (JVM)
- Java platform

Let's discuss these in detail in the following sections.

### Java Programming Language

Java programming language is an object oriented language. Its syntax is similar to the C++ language, but provides some advanced features, such as static typing of objects and a rigid system of exceptions, which require every method in the call stack to handle exceptions. Java fulfils the following primary objectives of an OOP language:

- Uses the object oriented methodology
- Allows the same program to be executed on multiple operating systems
- Contains built-in support to use computer networks
- Executes code securely from remote sources

Java is generally considered to be the most popular programming language today. It is also a widely used standard in enterprise programming.

### Java Runtime Environment

JRE is also known as Java Runtime Environment, and is included in the Java Development Kit (JDK), which is a set of programming tools required to develop Java applications. JRE consists of Java libraries, JVM, and other components that are required to run Java applications, as shown in Figure 1.1:

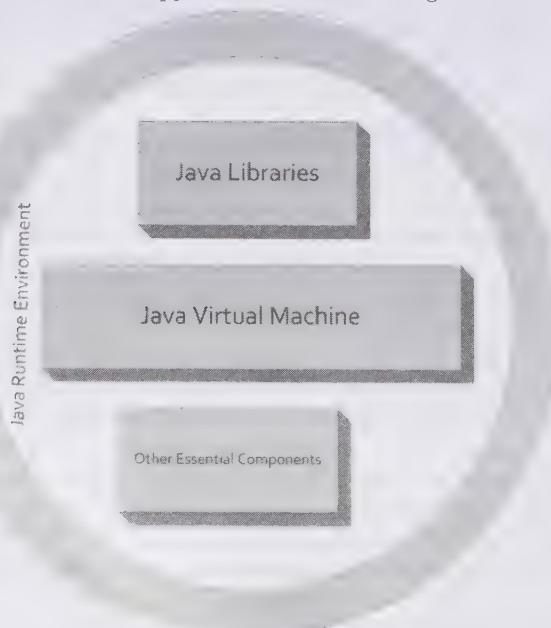


Figure 1.1: Displaying the Components of Java Runtime Environment

## Java Virtual Machine

Java itself is implemented as JVM, which is the application that executes the Java program. In other words, a JVM installed on a computer allows you to run Java programs. Therefore, Java programs do not need to be self-sufficient and do not need all the machine-level code that runs on the computer. Instead, a Java program is first *compiled* and then translated into *bytecodes*. Then, JVM reads and interprets these bytecodes to run the program. You should note that when you run a Java applet on a Web page, you are actually running a bytecode file.

JVM ensures that Java programs include less code, because all the machine-level code to run the program is already on the target computer and does not need to be downloaded. To host Java on computers, Sun Microsystems only had to redefine JVM. As programs are stored in bytecode files, the programs can run on any computer that has an installed JVM.

Since Java programs are *interpreted* by JVM, that is, executed bytecode by bytecode, interpretation can be a slow process. It was mainly for this reason that Java 2 introduced the Just In Time (JIT) compiler, which is built into JVM. The JIT compiler reads the bytecodes in sections and compiles them interactively into machine language (that is, the whole Java program is not compiled at once because Java performs runtime checks on various sections of the code). In simple terms, this means that Java programs will run faster with the new JIT compiler.

Using bytecodes also ensures that Java programs are very compact, which makes them ideal to run over the Internet. Running such programs with JVM rather than downloading full programs has another advantage, i.e., security.

## Java Platform

A platform in a computer system used to run applications created by using a programming language. For example, Sun Microsystems has provided the Java platform to run or execute Java programs or applications. In addition, Sun Microsystems provides JDK, which is used to build Web or enterprise applications in Java. For each platform, different JDKs are provided to build the required application. For example, an enterprise application in Java is developed by using Java EE SDK and is deployed as well as executed on the Java EE platform.

Over the years, the Java platform has evolved into three major editions, each addressing a distinct set of programming needs. These editions are:

- ❑ **Java Platform, Standard Edition (Java SE)**—Allows you to develop desktop and console-based applications. This edition consists of the following:
  - A runtime environment
  - A set of Application Programming Interfaces (APIs) to build a wide variety of applications comprising standalone applications, which can run on various platforms, such as Microsoft Windows, Linux, and Solaris
- ❑ **Java Platform, Enterprise Edition (Java EE)**—Allows you to build enterprise applications by using the convenient component-based approach of Java EE. In other words, various components, such as Web and EJB, help a Java developer to develop an enterprise application.
- ❑ **Java Platform, Micro Edition (Java ME)**—Enables you to build Java applications for micro-devices, which include handheld devices such as mobile phones, PDAs, and other similar devices with limited display and memory support.

Let's now explore enterprise architecture.

## Section B: Enterprise Architecture

Enterprise architecture helps in understanding the structure of an enterprise application and can be broken down into three fundamental logical layers:

- ❑ **The presentation layer**—Displays elements that store the data to be displayed to users and collects data from the users. The presentation layer is generally considered as the user interface, which includes the part

of the software that creates the controls required to design an interface for a user and validates the actions of the user.

- **The business logic layer**—Helps an application to work with and handle the processing of business logic. The logic of the application is implemented on the business layer.
- **The data storage and access layer**—Helps business applications to read and store data.

It is important to get acquainted with the concept of n-tier architecture whenever you design an enterprise application. You may be aware of the client/server system, which is based on the 2-tier architecture and has a clear separation between the data and the business logic. Apart from the two-tier architecture, the 3-tier architecture also provides separate layers for the presentation logic, the business logic, and the database. Though these architectural styles are quite common in current organizations, it is worth noting that they have emerged due to the advent of cheaper hardware platforms for clients, servers, and networking technologies.

Let's discuss these enterprise architecture types in detail in the following sections.

## Exploring Enterprise Architecture Types

Java developers select a suitable enterprise architecture type for an application according to their and the application's requirements. Suppose you want to create an application in which the presentation layer needs to implement the business logic as well. In such a case, you can select the single-tier architecture.

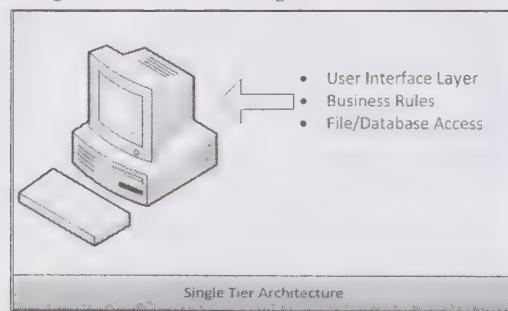
You can design enterprise architecture in many ways; however, in this section, we discuss the following types of enterprise architectures:

- The single-tier architecture
- The 2-tier architecture
- The 3-tier architecture
- The n-tier architecture

Let's now discuss these different types of architectures in the following sections.

### *The Single-Tier Architecture*

A single-tier architecture consists of the presentation logic, the business rules, and the data access layers—in a single computing architecture. Figure 1.2 shows the single-tier architecture:

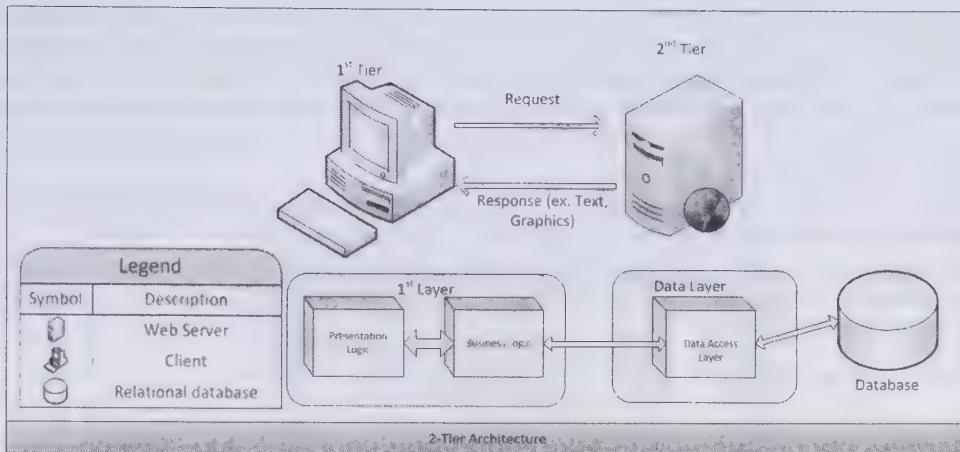


**Figure 1.2: Displaying the Single-Tier Architecture**

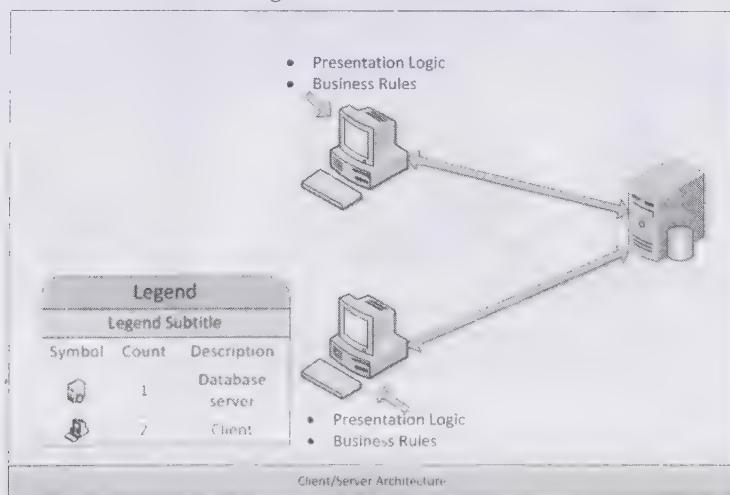
Applications created on the single-tier architecture are relatively easy to manage and implement data consistency, as data is stored at a single location. The only problem is that such applications cannot be scaled up to handle multiple users, and they do not provide an easy means of sharing data across an organization.

### *The 2-Tier Architecture*

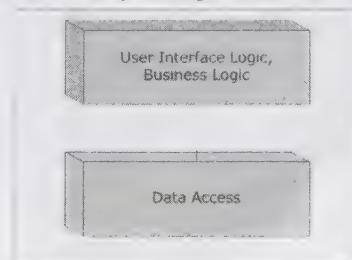
The 2-tier architecture separates the data access layer and the business logic layer. This type of architecture is generally data driven; with the application existing entirely on the local machine and the database deployed at a specific and secure location in an organization. This type of architecture is the traditional method of enterprise development. In a 2-tier application, the processing load is entrusted to the client, while the server simply controls the traffic between the application and the data access layers, as shown in Figure 1.3:

**Figure 1.3: Displaying the 2-Tier Architecture**

The main objective of the 2-tier architecture is to centralize the data so that multiple users can access a common database at a given time. Due to centralization of data, the load of executing an application is shared with the central database server. The 2-tier architecture is usually referred to as client/server, where a client communicates with a server, as shown in Figure 1.4:

**Figure 1.4: Displaying the Client/Server Architecture**

The client and server communication in 2-tier architecture is established by providing implementations for the user interface, business logic, and data access layers. Figure 1.5 shows the layers of the 2-tier architecture:

**Figure 1.5: Displaying the Client/Server Layers**

When a change occurs in a database, some users may not be ready to implement the changes, while others may insist on making the changes effective immediately. This may result in different client installations by using different versions of applications, resulting in inconsistency of the database. You may think that this problem can be solved by developing a reusable library encapsulating the business rules. In other words, a change in any rule can be simply replaced in that library and then the application can be rebuilt and redistributed. However, there are few inherent problems in this regard:

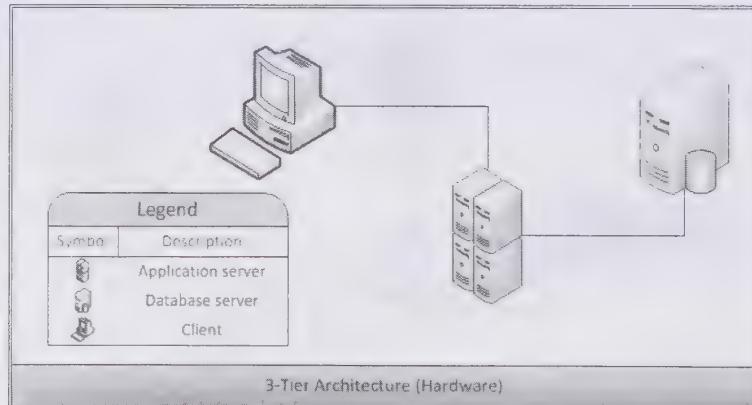
- You need to assume that all applications are created by using the same programming language
  - You need to run all applications on the same platform, leading to heavy load on a system
  - You need to recompile or reassemble the application along with the changes implemented in a new library
- The shortcomings of the 2-tier architecture led to the development of the 3-tier architecture, which we discuss in the next section.

## *The 3-Tier Architecture*

In the 3-tier architecture, an application is virtually split into three separate logical layers. The 3-tier architecture has the following layers:

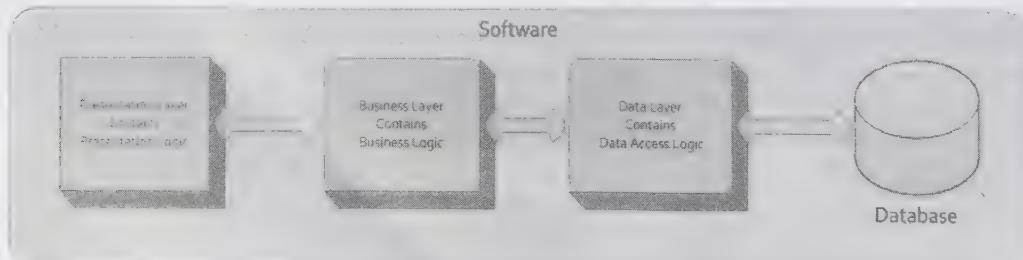
- **First tier**—Refers to the presentation layer, which consists of a Graphical User Interface (GUI) to interact with a user.
- **Middle tier**—Refers to the business layer, which consists of the business logic for an application. The middle tier represents the code that is called by a user through the presentation layer to retrieve data from the data layer.
- **Third tier**—Refers to the data layer, which contains the data access logic needed for the application.

Each of these layers has well defined interfaces, as shown in Figure 1.6:



**Figure 1.6: Displaying the 3-Tier Architecture (Hardware View)**

Figure 1.7 shows the software view of the 3-tier architecture:



**Figure 1.7: Displaying the 3-Tier Architecture (Software View)**

As the business logic and the user interface are at different layers, it adds a lot of flexibility when designing an application in the 3-tier architecture, as compared to the 2-tier architecture. By using the 3-tier architecture, multiple user interfaces can be built and deployed without changing the application logic, provided this architecture defines a clear interface to the presentation layer.

## The n-Tier Architecture

As the name suggests, there can be numerous layers in the n-tier architecture. Figure 1.8 shows an example of an n-tier architecture, which is formed by combining the 3-tier architecture with the 2-tier architecture:

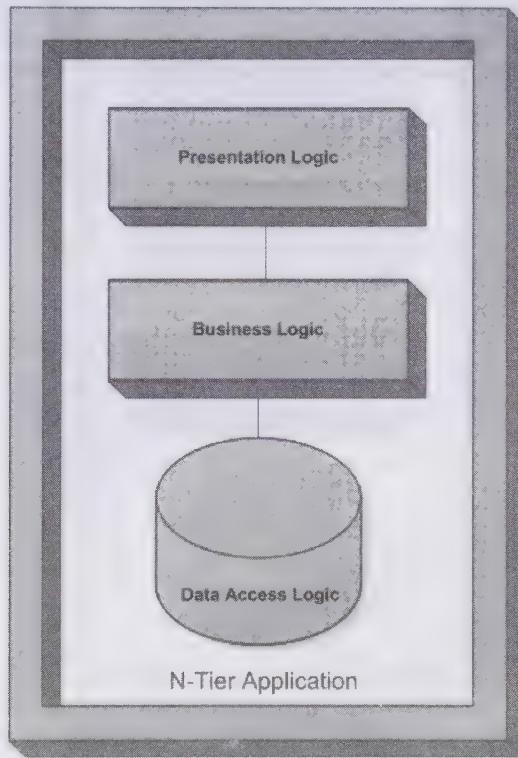


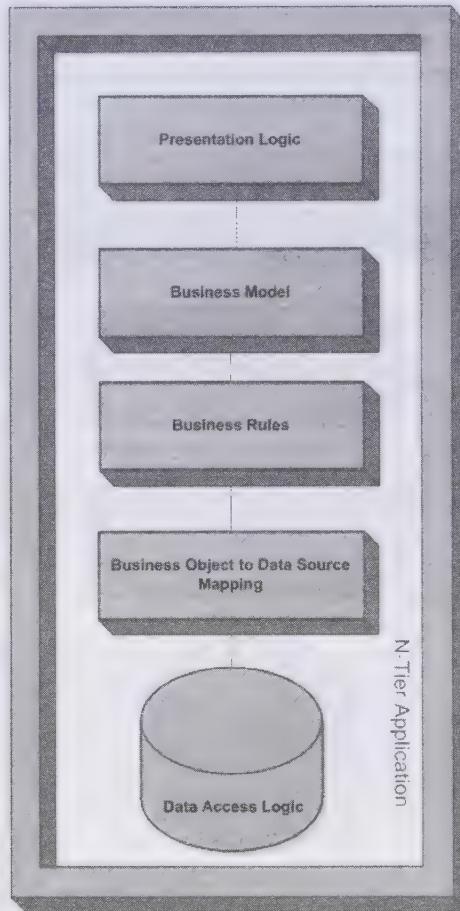
Figure 1.8: Displaying an Example of the n-Tier Architecture

In the n-tier architecture, the business logic is retrieved from the presentation layer. For example, a desktop application serves as a presentation layer representing the user interface, which can communicate with the business logic tier. It is no longer responsible for enforcing business rules or accessing databases.

The n-tier architecture supports various types of configurations to define the application layer. The n-tier architecture can be virtually distributed into the following components:

- ❑ **The user interface**—Refers to the interface between a user and a database. The interface may be a Web browser running through a firewall, a heavy desktop application, or even a wireless device.
- ❑ **The presentation logic**—Defines what will be displayed on the user interface and how a user's requests would be handled.
- ❑ **The business logic**—Models an application's business logic or rules, often by interacting with the application data.
- ❑ **Infrastructure services**—Provide additional functionalities, such as messaging and transactional support required by an application.
- ❑ **The data layer**—Stores the data of an organization.

Figure 1.9 shows the n-tier architecture:



**Figure 1.9: Displaying the Components of the n-Tier Architecture**

In an application that is deployed on a server, the business logic tier is executed on the server instead of the workstation. The business logic tier helps to bind the presentation layer to the data layer. Accessing of the business logic by multiple users may make it complex and processor-intensive. When this happens, you can scale up the server or add more servers. Scaling a single server is a lot easier and cheaper than upgrading everyone's systems.

Applications based on the n-tier architecture can be employed by using the Model-View-Controller (MVC) model. In this model, the application/business logic (Controller) controls the flow of information between a database and a user interface. Therefore, the application design is based on three functional components: Model, View, and Controller, all interacting with each other.

To convert an n-tier application into an enterprise application, you need to extend the middle-tier by allowing multiple application objects that help to create an enterprise application. Each of these application objects must have an interface that allows communication within the application objects. Consequently, with enterprise architecture, you can have multiple applications that use a common set of components across an organization to perform a desired task. This enhances the standardization of business practices by creating a single set of business functions for the organization to access. In addition, in case business rules are changed or updated, changes are required to be made in one business object only. Changes are required in the interface and subsequent objects that access the interface in rare cases only.

**NOTE**

*It is recommended that the interface be as generic as possible to avoid changes at a later stage. As you know, all objects communicate with the interface and not the object itself; therefore, changes in the object and not the interface make the process relatively simple and quick.*

The n-tier architecture has the following advantages:

- ❑ **Improved maintainability** – Allows you to create applications that are easy to maintain
- ❑ **Consistency** – Allows you to develop enterprise applications consistent in terms of component designing and their association with the layer in the architecture where they are providing functionality
- ❑ **Interoperability** – Allows you to develop highly interoperable applications, which means that you can implement components in different layers to support different technologies and platforms
- ❑ **Flexibility** – Allows you to develop and design various types of components for different layers with flexibility
- ❑ **Scalability** – Allows you to add new components without affecting the performance of an existing application, which in turn make applications more scalable

The Java EE 6 architecture is based on the n-tier architecture, which makes it very easy to build enterprise applications based on two, three, or more application layers.

Enterprise applications should meet certain objectives, which we discuss in the next section.

## Objectives of Enterprise Applications

Enterprise applications are developed based on various types of architectures described earlier. These enterprise applications should also meet certain objectives. For example, to maintain a competitive edge in the market in terms of technology and infrastructure, an organization needs to adopt new technologies to help manage business in an efficient and cost-effective manner. One place where these shifts in business practices have been felt most deeply is at the application development level. The funding and time allocated to application development is shrinking sharply, while the demand for developing solutions to complex business processes is increasing. The whole IT revolution is driven by the rapidly changing technological and economical landscape, which has created many new challenges for today's enterprise application developers.

The following objectives need to be taken into account while developing enterprise-based solutions:

- ❑ Ensuring that robust solutions and high-quality code are provided in an enterprise application so that it can perform the desired tasks efficiently. The main objective of considering robustness as a desired objective in an enterprise application is that users expect the application to be reliable and bug-free.
- ❑ Ensuring that scalability and performance is according to the expectations of the users of an enterprise application. It is expected that an enterprise application should provide sufficient scalability; i.e., the capability of an application to hold increased load with the specified resources. The scalability feature is an important consideration for Internet applications, as it is difficult to forecast the number of users and their behavior at any given time. The Java EE infrastructure helps you to meet the scalability objective. You should ensure that enterprise applications are designed so that their operations are efficiently performed in a cluster. In other words, meeting the scalability objective in enterprise applications typically requires deploying multiple server instances in a cluster. Clustering is a complex task that requires sophisticated application server functionality.
- ❑ Ensuring that object oriented design principles are implemented in an enterprise application. You know that good object oriented design practice is promoted with the use of design patterns, which serve as persistent solutions to common programming problems. Design patterns are technology and language independent. At present, a separate and distinct industry has evolved from where you can select numerous Java EE patterns.
- ❑ Avoiding complexity by finding the simplest way possible to avoid problems. While developing an enterprise application, avoid excessive complexity as this may create problems in executing the application's architecture. Sometimes, the availability of large number of components may tempt a designer to over-engineer Java EE solutions in an application, which may lead to great complexity and addition of irrelevant business requirements. However, remember that complexity only leads to additional cost

throughout the software development life cycle, which can result in various problems. On the other hand, a thorough analysis should also be made while developing an enterprise application to ensure that the application does not have a naïve and simplistic view of the requirements.

- Ensuring that clear responsibility of each component is maintained in an enterprise application. While developing an enterprise application, you should ensure that maintenance is not hindered by tightly coupled components. Maintenance is the most expensive and crucial phase of the software development life cycle process; therefore, you must consider the maintainability of software applications while designing enterprise application.
- Considering the implications of design decisions with the view to simplify the testing of an enterprise application. The main reason for simplifying the testing process of the enterprise application is because testing is an essential activity throughout the software life cycle.
- Ensuring that an enterprise application fits into an organization's long-term strategy, resulting in the promotion of reusability of the applications. Promoting reusability of applications also ensures that duplication of code is minimized (within and across various projects). Note that code reusability results from good object oriented design practice.

Depending on an enterprise application's business requirements, you may also need to meet the following objectives:

- Ensuring that an enterprise application provides support for multiple client types, such as Web applications and standalone Java GUIs, by using Swing and Java applets in an enterprise application. However, such support is often unnecessary, as Web browsers are being widely used nowadays, even for applications intended for use within an organization (ease of deployment is one of the major reasons for this support).
- Ensuring that an enterprise application is portable. The following concerns need to be considered while designing an enterprise application:
  - The extent of portability between resources, such as the databases used by an enterprise application
  - The extent of portability between application servers

None of the preceding objectives are especially new for enterprise developers; however, achieving them comprehensively and economically has always been crucial. A variety of technologies are available to address some of the preceding listed objectives.

Let's now discuss how these objectives are satisfied with the introduction of the Java EE platform, which is used to develop the Java enterprise applications.

## **Section C: Introducing the Java EE 6 Platform**

Java EE 6 is the latest version of the Java platform used to develop robust, scalable, distributive, and secure enterprise level applications. The earlier versions of Java Platform for the enterprise applications were Java 2 Platform, Enterprise Edition (J2EE), followed by specific versions, such as J2EE 1.4. The term J2EE was renamed to Java EE in version 5. Therefore, the new version of this platform has been termed Java EE 6 (formally J2EE 1.6). The previous versions of Java Platform for the enterprise applications are still referred to as J2EE 1.3, J2EE 1.4, and Java EE 5.

### **NOTE**

*Hereafter in this book, we use Java EE instead of J2EE.*

J2EE 1.4 is a powerful platform; however, you may still find it difficult to start an application with the help of this platform. At times, even small enterprise applications need a lot of code, which needs to be reduced. You can reuse the code of an application in another application, which reduces time and effort. In addition, you can encapsulate the complexity of code by providing a set of APIs to the programmers. Therefore, the main objective of Java EE is to provide Java programmers a set of classes and interfaces that reduce development time and application complexity, thereby improving the performance of the application.

Java EE can also be defined as an extension to J2EE, and is capable of meeting the requirements of different users in an organization, such as customers, suppliers, and employees, by providing various functionalities and services. For example, Java EE focuses on making the development of enterprise applications easier by

supporting annotations and Plain Old Java Objects (POJOs), reducing the need of Deployment Descriptors, enhancing Web services, and supporting technologies, such as Asynchronous JavaScript and XML (AJAX). In J2EE 1.4, programmers need to create Extensible Markup Language (XML) Deployment Descriptors to provide action mappings and execution flow to applications. However, with advances in the different versions of Java EE, the creation of XML Deployment Descriptors has become optional. Programmers can now use Java annotations in source files, which allow the Java EE application server to configure a Web or enterprise component at deployment or runtime. The annotations are used to embed program data that needs to be provided in a Deployment Descriptor.

Let's now explore the features and functionalities of the Java EE 6 platform.

## Exploring the Features of the Java EE Platform

So far, we have discussed the need of enterprise programming and briefly introduced the latest Java platform for enterprise development. There are several potential paths that you can take to implement enterprise solutions. Among these, there are two common paths, one being driven by Microsoft, with its new .NET suite, and the second offered by Sun Microsystems. You may know about the acquisition contract between Oracle and Sun Microsystems; therefore, Oracle is reviewing the product roadmap of Sun Microsystems. Apart from Microsoft and Sun, there are a host of other players in the field of enterprise solutions as well, such as BEA and International Business Machine (IBM). Although users have such a wide variety of choices, Java EE continues to be a popular choice with them. To understand the reasons for such popularity, let's look at some features of the Java EE platform. These include the following:

- Platform independence
- Managed objects
- Reusability
- Modularity
- Simplified Enterprise JavaBeans (EJB)
- Enhanced Web services
- Support for Web 2.0

Now, let's discuss these features in detail.

### *Platform Independence*

Platform independence is perhaps the most important feature of Java EE. Today, a wide range of information is available in different formats, which is spread across multiple platforms. Therefore, it is essential to adopt a programming language that can function equally well across multiple platforms. For this purpose, Java EE is used in organizations as it is a platform-independent programming language.

### *Managed Objects*

Java EE provides a managed environment for its components. Moreover, Java EE applications are container-centric. These two properties are very critical for building server-side applications, since they allow Java EE components to use the infrastructure services provided by Java EE servers.

Another important property of Java EE applications is the ease by which you can modify and control the behavior of the applications without changing their code. Initially, these properties may appear cumbersome; however, if you consider a large-scale application with numerous components interacting with each other to execute complex business processes, these properties are necessary to maintain such applications.

### *Reusability*

Reusability of code is an important aspect of programming, especially when developing and maintaining multiple, complex applications. Reusability is also important when you need to develop applications that require frequent updates. One method to achieve reusability is to segregate an application into individual components. Another method is to use the concept of object orientation to encapsulate shared functionalities. Java uses both the methods because it is an object oriented language and provides the mechanism for code reusability.

## Modularity

When you develop a complete server-side application, the program can become large and complex. As a best practice, you must break the application into various modules, with each module responsible for performing a specific task. This also makes the application much easier to maintain. Java EE provides the property to modularize an application by breaking it down into different tiers associated with individual tasks. These modules further interact with each other to execute the business logic.

## Easier Development

Java EE has simplified the way in which components are created and configured. With the use of annotations, you need to write less code in Java EE applications, which further reduces the need for a Deployment Descriptor.

## Simplified EJB

The use of POJOs has simplified EJB programming. To store the data of EJBs persistently in a database, a new persistence API has also been introduced in Java EE. In EJB 3, we do not need to create Remote and Home objects for the enterprise beans, and the use of Deployment Descriptors has also been reduced by the use of annotations and dependency injection. For more information about EJB, refer to *Chapter 13, Working with EJB 3.1*.

## Enhanced Web Services

Java EE includes the latest Web service APIs that provide a simplified and enhanced approach to design, develop, test, and deploy Web services. For more information about Web Services, refer to *Chapter 19, Implementing SOA using Java Web Services*.

## Support for Web 2.0

Web 2.0 is mainly associated with Web applications that allow you to share information, provide user-centered design, and interoperability on World Wide Web. Java EE allows you to develop applications supporting Web 2.0 with the help of technologies such as JavaServer Faces (JSF), JSP Standard Tag Library (JSTL), and AJAX. For more information about JSTL and JSF, refer to *Chapter 9, Implementing JavaServer Pages Standard Tag Library 1.2* and *Chapter 11, Working with JavaServer Faces 2.0*.

Now that you have a brief knowledge about the general features of the Java EE platform, let's explore the new features introduced in the Java EE 6 platform.

## Exploring the New Features of the Java EE 6 Platform

The Java EE 6 platform provides an environment for building enterprise applications by using a distributed multi-tiered application model. The Java EE 6 platform provides the following features:

- ❑ **Profiles and configurations**—Serves as a subset of the features of the Java EE 6 platform. Web Profile is introduced as a subset of the Java EE 6 platform, which includes the technologies used by Web application developers. However, the enterprise technologies that would not be used by Web developers are not provided in the Web Profile. With the evolution of the Java EE 6 platform, the technologies that can be considered for pruning are identified. Pruning a technology refers to the process of adding the technology as an optional component in the Java EE platform instead of a required component.
- ❑ **Enhanced extensibility on a Web tier**—Enables the use of third-party frameworks by self-registering. In other words, you can easily include and configure a third-party framework in a Web application by registering the framework with the Java EE platform. Earlier, Web developers used the third-party frameworks in their applications by registering them with the third-party vendors. They also needed to add or edit XML Deployment Descriptors. This was a tedious task; therefore, the Java EE 6 platform provides an enhanced approach to configure and register these frameworks.
- ❑ **Use of annotations**—Allows you to use annotations to define Web components, such as servlets and servlet filters. In addition, the annotations used for dependency injection have been standardized, resulting in the addition of more portability to injectable classes across frameworks. Moreover, the requirements considered while packaging a Java EE application have been simplified as Java EE 6 allows you to directly add an

enterprise bean to a Web ARchive (WAR) file. In other words, in Java EE 6, you do not need to create a Java ARchive (JAR) file while packaging an enterprise bean in an Enterprise ARchive (EAR) file.

- **Enhanced support for Web services**—Introduces the new versions of various technologies, such as Java Servlet 3.0, EJB 3.1, and Java API for RESTful Web services (JAX-RS). You learn more about the new features of each of these technologies in their related chapters in this book.

After briefly discussing the new features of the Java EE 6 platform, let's explore the Java EE 6 platform in detail.

## Exploring the Java EE 6 Platform

The Java EE 6 platform uses a distributed multi-tiered application model to develop enterprise applications. It provides a runtime infrastructure to manage and host applications. All runtime applications are located in the Java EE application server. Apart from the runtime infrastructure, a set of Java APIs are also provided by the Java EE 6 platform to build Java EE 6 applications. These APIs describe the programming model for Java EE applications.

Let's discuss the runtime infrastructure and Java EE 6 APIs in detail in the following sections.

### *The Runtime Infrastructure*

The runtime infrastructure component includes numerous services to manage enterprise applications. The Java EE architecture provides a uniform way to access platform-level services through its runtime environment. These services include messaging, security, and distributed transactions management. Therefore, you can develop Java EE 6 applications that implement complex database transactions or even Java applets across a network.

### *The Java EE APIs*

In enterprise services, you need server access to run distributed applications. These enterprise services include transaction processing, multithreading, and database access. The Java EE architecture unifies access to such services in its enterprise service APIs. A Java EE 6 application can access these APIs through containers, such as Web and EJB.

Java Platform, Standard Edition (Java SE) SDK, provides core APIs to write Java EE components and development tools, and JVM. Java EE SDK uses some of the core APIs of Java SE SDK to provide a runtime environment to the Java EE application. The following are the noteworthy Java EE 6 APIs:

- Java Servlet 3.0
- JavaServer Pages (JSP) 2.2
- JavaServer Pages Standard Tag Library (JSTL) 1.2
- JavaServer Faces (JSF) 2.0
- Enterprise JavaBeans (EJB) 3.1
- Java Persistent API (JPA) 2.0
- Java Messaging API 1.1
- Java Transaction API (JTA) 1.1
- JavaMail 1.4
- Java EE Connector Architecture (JCA) 1.6
- Java API for RESTful Web Services (JAX-RS) 1.1
- Java API for XML Registries (JAXR) 1.0

Apart from the preceding Java EE 6 APIs, a few APIs of Java SE 6 are also used in developing Java EE 6 applications. The following is a list of the noteworthy Java EE 6 APIs included in Java SE 6:

- Java Database Connectivity API 4.0
- Java Naming Directory Interface (JNDI)
- JavaBeans Activation Framework (JAF) 1.1
- Java API for XML Processing (JAXP) 1.3
- Simple Object Access Protocol (SOAP) with Attachments API for Java

- ❑ Java API for XML Web Services (JAX-WS) 2.2
- ❑ Java Architecture for XML Binding (JAXB) 2.2
- ❑ Java Authentication and Authorization Services (JAAS)

Let's discuss each of the Java EE 6 APIs as well as the Java SE 6 APIs that are included in Java EE 6.

## Java Servlet 3.0

The Java Servlet technology allows you to create HyperText Transfer Protocol (HTTP)-specific servlets, which extend the capabilities of the servers hosting applications. The servlets are accessed with the help of the request-response programming model. A new version of the Java Servlet technology is introduced in the Java EE 6 platform, i.e., Java Servlet 3.0, which has the following features:

- ❑ Provides asynchronous support; i.e., the ability to receive or send data to a client without blocking the data
- ❑ Provides the use of annotations for Web components instead of using Deployment Descriptors
- ❑ Provides support for Web framework pluggability, which simplifies the task of plugging different Web frameworks in a Web application, depending on the needs of a developer
- ❑ Provides enhancements to existing Servlet 2.5 APIs

## JavaServer Pages 2.1

The JSP technology lets you integrate Java code with Hypertext Markup Language (HTML) in a text-based document. A JSP page is a text-based document that contains:

- ❑ Static template data that can be represented in a text-based format, such as HTML, Wireless Markup Language (WML), or XML
- ❑ JSP elements that can be used to display dynamic content on a Web page

## JavaServer Pages Standard Tag Library 1.2

JSTL 1.2 provides a set of standard tags that can be used in a JSP page by embedding Java code. JSTL includes various tags to control the flow of execution of an application, to support internationalization, and to access a database by using Structured Query Language (SQL). An application using JSTL tags can be deployed on any JSP container that supports the tags.

## JavaServer Faces 2.0

JSF provides a component-based API that is used to build robust and rich user interfaces for Web applications. The components in JSF can be easily integrated to create a server-side user interface and can be developed in conjunction with Java Servlet and JSP. The JSF technology also simplifies the task of connecting user interface components to application data sources. In addition, this technology allows client-generated events to connect to event handlers on a server.

Initially, problems were faced in managing user interfaces and in developing business logic. However, the use of the JSF components has solved these problems. In JSF, you can utilize a user interface component along with the relevant renderer to produce presentation code for various devices. This implies that if a client's device changes, you only need to configure your system in such a manner that it uses the same renderer for the new client. You do not need to change the JSF code even if there is a change in the client's device. For more information about JSF refer to *Chapter 11, Working with JavaServer Faces 2.0*.

## Enterprise JavaBeans 3.1

EJB 3.1 is a component-based architecture used to develop, deploy, and manage reliable enterprise applications in production environments. An enterprise bean is a server-side piece of code with fields as well as methods to define the modules of the business logic.

An enterprise bean can be considered as a building block that can be used alone or with other enterprise beans to execute the business logic on the Java EE server conforming to the EJB architecture. Java EE provides all infrastructure services to EJB, such as Java Database Connectivity (JDBC), Java Naming Directory Interface (JNDI), Java Message Service (JMS), and Java Transaction APIs (JTA). The latest version of EJB in Java EE 6 is EJB 3.1. Compared to its previous version, EJB 2.1, the programming of EJB components has been made simpler in EJB 3.1. For more information about EJB 3.1, refer to *Chapter 13, Working with EJB 3.1*.

## Java Persistent API 2.0

JPA is an API used to manage persistence and object relational mapping in the Java EE and Java SE environments. Java EE 6 introduces JPA 2.0, which incorporates new features, such as additional functionalities for object relational mapping and support for query language and validations. For more information about JPA, refer to *Chapter 14, Implementing Entities and Java Persistence API 2.0*.

## Java Message Service API 1.1

Java EE application components can create, send, receive, and read messages by using the JMS API 1.1. The JMS API provides a common set of programming strategies and messaging concepts, which improves the productivity of programmers. JMS API 1.1 in Java EE 6 provides support for concurrent consumption of messages and distributed transactions. The term distributed transaction implies that a database establishes connections to Enterprise Information System (EIS) with the help of the Java EE connector architecture.

## Java Transaction API 1.1

JTA is used to manage distributed transactions. JTA 1.1 specifies a standard Java interface for a transaction manager to interact with the resource manager, the application server, and with transactional applications. By default, the Java EE 6 architecture provides auto commit, which is used to manage commits and rollbacks of transactions. JTA also allows you to separate an entire transaction based on the database operations performed by an application.

## JavaMail 1.4

Sometimes you may need to develop an enterprise application used to send e-mail notifications. You can develop such applications by using the JavaMail API provided by the Java EE platform. The JavaMail API provides a JavaMail service provider, which allows application components to send e-mail messages. The JavaMail API has the following two parts:

- An application-level interface** – Allows application components to send e-mail messages from Microsoft Outlook or other mail clients
- A service provider interface** – Helps application components to send e-mails messages from Gmail, Yahoo and other e-mail service providers

## Java EE Connector Architecture 1.6

JCA is used to create resource adapters that help Java EE application components to interact with the resource manager of Enterprise Information Systems (EIS). Generally, JCA is used by Java EE tool vendors and system integrators to plug resource adapters to a Java EE product. There is a different resource adapter for each type of database or EIS, since a resource adapter is specific to its resource manager.

## Java API for RESTful Web Services 1.1

The Java EE 6 platform provides a Java API to develop Web services based on the Representational State Transfer (REST) architectural style. A JAX-RS application is a Web application comprising classes and libraries that are packaged in a (WAR file). JAX-RS is introduced as an API in the Java EE 6 platform.

## Java API for XML Registries 1.0

The Java EE platform provides JAXR, which supports XML Registry and Repository standards and allows an application to access business as well as general purpose registries on the Internet.

Moreover, businesses can share information by using JAXR. XML-based groups on the Internet have developed schemas for particular kinds of XML documents, which can be used in business transactions. For example, two organizations may agree to use schemas for standard purchase and sales order forms. In this case, JAXR would facilitate both parties to access the order forms that are stored as schemas in a standard business registry.

## Java Database Connectivity API 4.0

The JDBC API helps to execute SQL commands from Java programs. You can use the JDBC API with enterprise beans, servlets, JSPs, and Java classes to interact with a database. The JDBC API contains application-level interfaces and service provider interfaces. Application-level interfaces are used by application components to

access the database, while service provider interfaces help to associate the JDBC driver with the Java EE platform.

## Java Naming Directory Interface

JNDI provides the functionality naming and directory services for various objects or resources available in the Java EE container. It provides the methods to perform standard directory operations, such as associating attributes with objects and searching for the objects in a Java EE application. Using JNDI, a Java EE application can store and retrieve any type of named Java objects.

## JavaBeans Activation Framework 1.1

JAF is a standard extension to the Java platform providing standard services to identify the type of any arbitrary information, encapsulate access to the information, identify the operations available on it, and instantiate appropriate JavaBean components to perform these operations.

## Java API for XML Processing 1.3

You can parse, transform, and validate an application with the help of JAXP. In addition, JAXP allows an application to query XML documents. This API is independent of XML processor implementation. JAXP is now a part of Java SE after the release of Java SE 6.

## SOAP with Attachments API for Java

SAAJ is a low-level API used by SOAP handlers in JAX-WS to access SOAP messages. It enables you to develop and use messages that conform to the SOAP 1.1 and SOAP 1.2 specifications. The SAAJ API can be used to write SOAP messaging applications directly without using JAX-RPC or JAX-WS.

## Java API for XML Web Services 2.2

The JAX-WS specification helps to create Web service endpoints and enables client components to access Web services. It also describes the deployment information of Web services and clients. The JAX-WS specification supports Web services that use JAXB. The JAXB API is used to bind XML data to Java objects.

## Java Architecture for XML Binding 2.2

JAXB helps to bind an XML schema with a Java object in a simple and easy way. It helps in marshalling a Java content tree into XML instance documents, and vice versa. You can also create XML schema from Java objects by using JAXB, and use JAXB independently and with JAX-WS-based services where it is used to bind data for Web service messages.

## Java Authentication and Authorization Services

JAAS is a Java-based standard Pluggable Authentication Module (PAM) framework that helps you to implement the authentication and authorization functionalities for one or multiple users in an application. This framework extends the Java platform security architecture to support user-based authorization. You should note that the JAAS framework has now been integrated with Java SE.

After getting a brief overview of the APIs of the Java EE 6 platform, let's now explore the Java EE 6 architecture.

# Exploring the Architecture of Java EE 6

The architecture of Java EE 6 is based on the multi-tier distributed application model, in which the application logic is segregated into various application components. A Java EE 6 application contains these application components, which are installed on different machines based on the number of tiers in the Java EE environment.

Java EE applications can be 3-tier or 4-tier (as shown in Figure 1.10), but they are generally considered to be 3-tier as they are located over three different locations: the client machine, the Java EE server, and the database server. This implies that the application logic is divided at different machines, such as client, Java EE server, and database or legacy at the backend. Figure 1.10 shows the Java EE application model comprising both distributed and multi-tiered architectures:

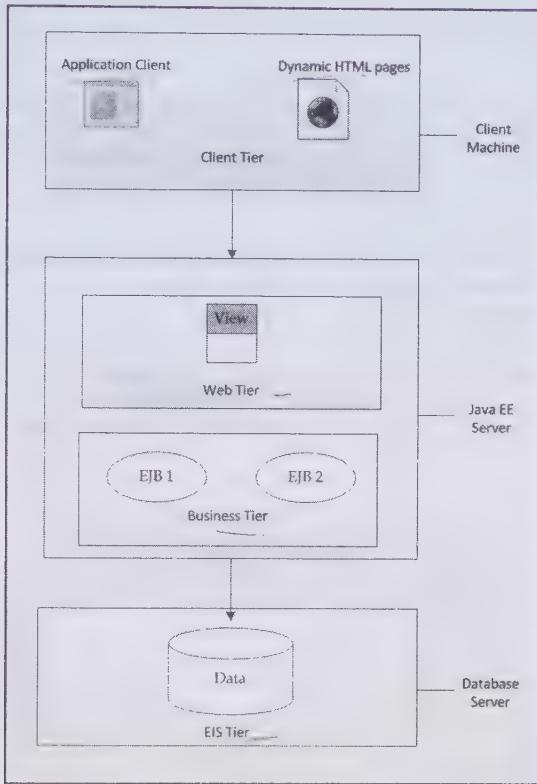


Figure 1.10: Displaying the Java EE Application Model

The 3-tier architecture and applications, shown in Figure 1.10, extend the standard 2-tier architecture.

You can see in Figure 1.10 that the middle tier that is represented by the Java EE server, has been divided into two tiers, the Web tier and the Business tier.

Let's now learn about the containers in Java EE 6.

## Describing Java EE 6 Containers

Java EE applications are comparatively easy to write due to the architecture of Java EE, which is component-based and platform-independent. The Java EE server provides various services in the form of containers for every component type. You do not have to develop these services by yourself; and are therefore, free to concentrate on solving the problems related to the implementation of the business logic. Containers are a central theme in the Java EE architecture. The Web and business components are placed in the container of an application server. These components can access the Java EE 6 infrastructure service with the help of well-defined interfaces.

A Java EE container acts as a runtime interface between the application components and the low-level platform-specific functionality that support the components. Java EE containers provide deployment, management, and execution support for application components. A feature of the Java EE platform is its support for declarative programming. Many low-level details, such as persistence, security, and multithreading issues, are declaratively specified in Deployment Descriptor. Java EE containers provide services as well as an execution environment for the components to be deployed on the server.

Different application components are installed in their respective containers during deployment; these containers act as interfaces between the components and the low-level platform-specific functionality that support the components.

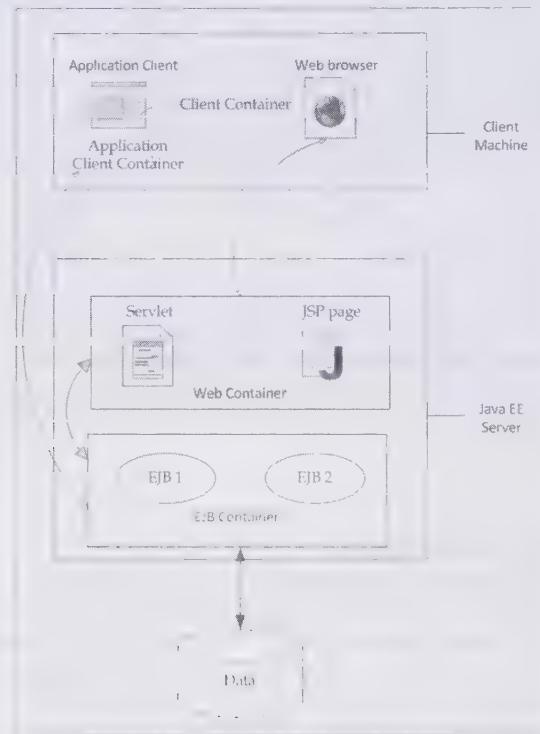
Let's now discuss Java EE 6 container types and their architecture.

## **Container Types**

During the deployment of a Java EE application, the components of the application are installed in the Java EE containers available in the Java EE server. The Java EE server is used to execute the application, which contains EJB and Web components. Java EE containers can be categorized as follows:

- ❑ **EJB container**— Allows you to execute all enterprise beans for a Java EE application. Enterprise beans and their containers run on the Java EE server.
- ❑ **Web container**— Allows you to execute all JSP pages and servlet components for a Java EE application. Web components and their containers run on the Java EE server.
- ❑ **Application client container**— Allows you to execute all application client components for a Java EE application. Application clients and their containers run on the client machine.
- ❑ **Applet container**— Allows you to execute an applet. The Applet container is a combination of the Web browser and Java Plug-in running together on the client machine.

Figure 1.11 shows the communication of the Java EE containers with each other:



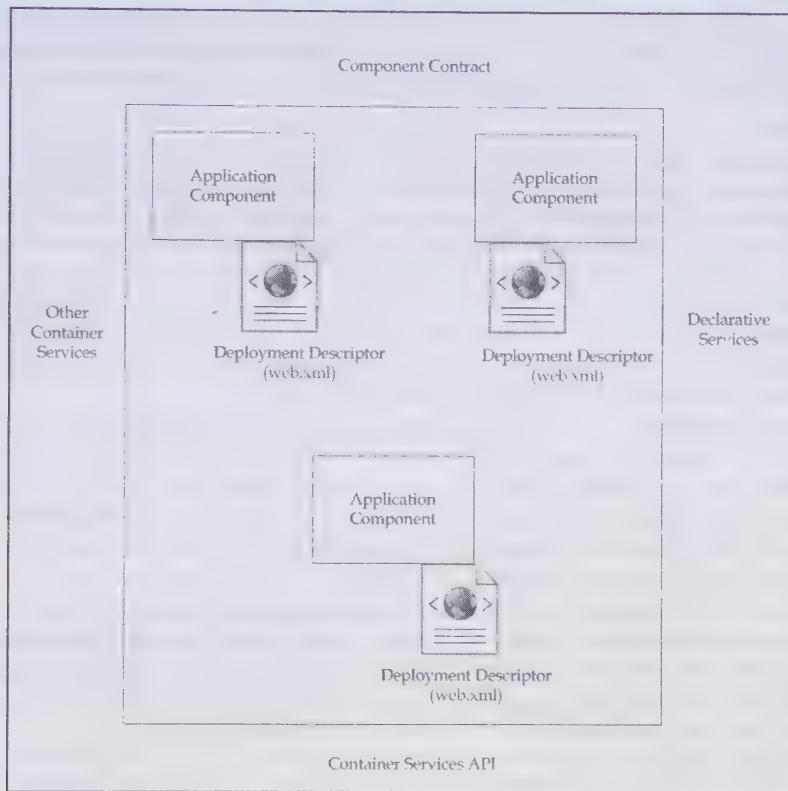
**Figure 1.11: Displaying Java EE Components and Containers**

Figure 1.11 shows the flow of communication among various the Java EE components and containers. For example, an application client located in the application client container can communicate with the EJB container to access the EJB resource.

## **Java EE 6 Container Architecture**

You must be aware by now that a container is an interface between a component and low-level platform functionality that supports the component. The architecture of a Java EE container includes different components, such as JSP, servlets, and EJBs. An archive file is used to package these components, which also includes different Deployment Descriptors.

Figure 1.12 shows the architecture of a Java EE container:



**Figure 1.12: Displaying the Java EE Container Architecture**

The Java EE container itself is divided into four parts:

- Component contract
- Container service APIs
- Declarative services
- Other container services

Let's now briefly explain these parts of a container in the following sections.

## Component Contract

Component contracts are rules, stored in the form of APIs, which application components must extend or implement. A Java EE container acts as a runtime environment to manage application components. At runtime, the instances of these application components are invoked within the JVM of the container. The container manages the life cycle of the application components. Therefore, they have to abide by certain contracts specified by the container. In other words, the container manages the location, instantiation, pooling, initialization, service invocation, and removal of application components by using the component contracts.

For example, suppose an applet is downloaded by the browser and instantiated and initialized in JVM of the browser. In other words, the applet exists in the runtime environment provided by JVM of the browser. As the container has to create, initialize, and invoke the methods on application components, they have to implement or extend certain Java interfaces or classes. Let's suppose that a Java applet is required to extend the `java.applet.Applet` class specified by the JDK. The `init()`, `start()`, `stop()`, and `destroy()` methods control the life cycle of the applet and are present only in the `java.applet.Applet` class. If the applet does not extend this class, JVM cannot call these methods.

## Container Services APIs

The container provides APIs and protocols that all application components need to use by extending or implementing them. This is how you can use service APIs, such as JDBC, JTS, JNDI, and JMS, in the container. All these APIs specified in the Java EE platform are provided as a view by the container.

## Declarative Services

The Java EE architecture allows you to use Deployment Descriptors to declare application components. These Deployment Descriptors provide services to application components and simplify application programming by allowing components and applications to be customized at packaging and deployment time. In addition, the components can use the required resources during the packaging and deploying of the application. Various elements used to customize Java EE 6 services are configured in Deployment Descriptor, which allows a Java EE container to interpose the new service before the transfer of a request to the application component. The advantage of this approach is that you can interpose new services without changing the application component.

Examples of other container services include component life cycle, resource pooling, garbage collection, and clustering. The runtime services provided by a container are as follows:

- ❑ Managing the complete life cycle of the application components, which includes creation and pooling, or destroying of the application components when they are of no use.
- ❑ Implementing resource pooling either by object pooling, which manages short lived objects, or by connection pooling, which is a technique used for sharing server resources among clients.
- ❑ Populating the JNDI namespace based on the deployment names associated with EJB components. This information is typically supplied at deployment time.
- ❑ Populating the JNDI namespace with the objects that are necessary for using container service APIs. Some of the objects include data source objects for database access, queue and topic connection factories to obtain connections to the JMS, and user transaction objects to programmatically control transactions.
- ❑ Distributing the load of incoming requests to one of the JVMs that are running on various machines. You should note that in a distributable Java EE container, multiple JVMs run on one or more host machines. Consequently, in this setup, application components can be deployed on multiple JVMs. The distribution of load depends on the type of load-balancing strategy and the type of component in use. The process of distributing load is known as clustering, and helps to improve the scalability of applications.

Let's now understand the process of developing a Java EE 6 application.

## Developing Java EE 6 Applications

The development of Java EE 6 applications is governed by various Java EE specifications. In addition, before discussing the development of Java EE 6 applications, you need to know about the Java EE technologies, such as JAX-RS, Java Servlet 3.0, and JSTL 1.2. These Java EE 6 technologies can be categorized as follows:

- ❑ Web services technologies:
  - Enterprise Web Services 1.3
  - Java API for RESTful Services (JAX-RS) 1.1
  - Java API for XML-based Web Services (JAX-WS) 2.2
  - Java API for XML-based RPC 1.1
  - Java Architecture for XML Binding (JAXB) 2.2
  - Java APIs for XML Messaging 1.3
  - Java API for XML Registries (JAXR) 1.0
  - Web Services Metadata for Java Platform
- ❑ Web application technologies:
  - Java Servlet 3.0
  - JavaServer Pages 2.2
  - JavaServer Faces 2.0

- JavaServer Pages Standard Tag Library 1.2
- Enterprise application technologies:
  - Context and Dependency Injection for Java (Web Beans 1.0)
  - Dependency Injection for Java 1.0
  - Bean Validation 1.0
  - Enterprise JavaBeans 3.1
  - Java EE Connector Architecture 1.6
  - Common Annotations for Java Platform 1.1
  - Java Message Service API 1.1
  - Java Persistence API 2.0
  - Java Transaction API 1.1
  - JavaMail 1.4
- Management and security technologies:
  - Java Authentication Service Provider Interface for Containers
  - Java EE Application Deployment 1.2
  - Java EE Management 1.1
  - Java Authorization Contract for Containers 1.3
- Java EE related specifications in Java SE:
  - Java API for XML Processing (JAXP) 1.3
  - Java Database Connectivity 4.0
  - Java Management Extensions (JMX) 2.0
  - JavaBeans Activation Framework (JAF) 1.1
  - Streaming API for XML (StAX) 1.0

In the following sections, we explore the possible architectures of Java EE 6 applications, look at some guidelines regarding application development and deployment roles, and finally, describe the application development process in detail.

## *Probable Java EE Application Architectures*

Before moving ahead to discuss the development of Java EE applications, let's briefly review some architectures that you can use to develop Java EE 6 applications. For example, you can use Applet to create a user interface, JSP to implement business logic, and a database as backend to store the user data. In other words, this section helps you to learn how different technologies can be used to build the architecture of various Java EE applications. The following are the various combinations of Java EE application architecture:

- **Applet client with JSP and database**—Represents the Java EE application architecture in which the Java applet resides at the presentation tier (client end) and communicates with the business tier. The business logic is implemented on JSP pages, and JDBC is used at the data access tier, as shown in Figure 1.13:

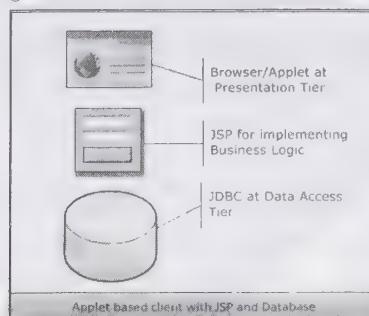
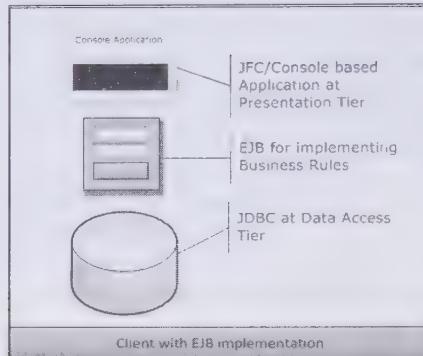


Figure 1.13: Displaying an Applet Client with JSP and Database

The Java applet provides an interactive and dynamic user interface within a Web page and accesses additional content from JSPs. In addition, JSPs access data from a database by using the JDBC API.

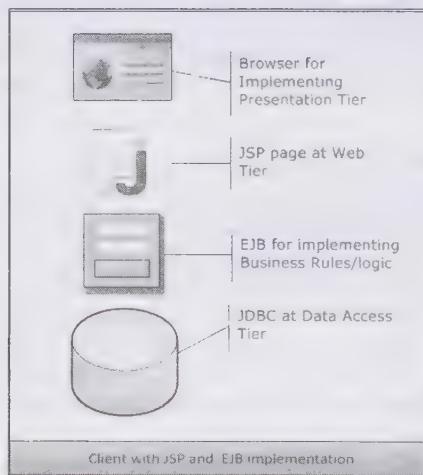
- **Application client with EJB** – Represents the Java EE application architecture in which an application client comprises the presentation tier, which communicates with the EJBs residing at the business tier. Figure 1.14 shows the Java EE application architecture in which the application client is used with EJB:



**Figure 1.14: Displaying an Application Client with EJB**

As shown in Figure 1.14, the business logic is implemented with the help of EJBs that run on separate machines.

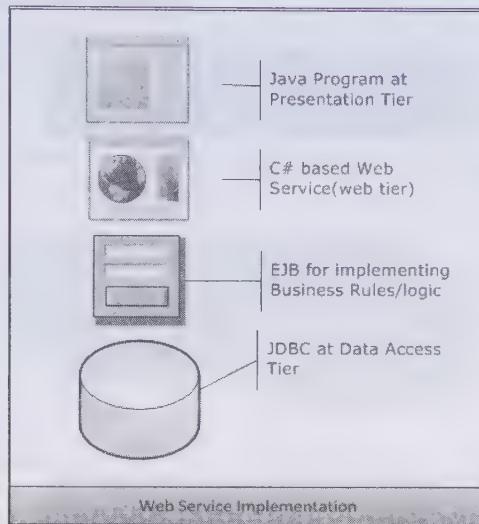
- **JSP client with EJB** – Represents the Java EE application architecture in which JSP works on the Web tier to communicate with the business tier in response to a client's requests. Figure 1.15 shows the Java EE application architecture in which JSP is at the client end and business logic is implemented by EJBs:



**Figure 1.15: Displaying the JSP Client with EJB**

In the Java EE architecture in which JSP client is with EJB, the response is generated as a Web page and is sent to the client's Web browser.

- **Web services for application integration** – Represents the Java EE application architecture in which Web components are implemented in object oriented programming languages other than Java, such as C#. Note that even though Java EE is Java-based, Web application architecture is not limited to Java components only. For example, in Figure 1.16, a client application is implemented in C# to access data from a Web service implemented in Java:



**Figure 1.16: Displaying Web Services for Application Integration**

After exploring different Java EE application architectures, let's now describe the role of different people in developing a Java EE application.

### *Application Development and Deployment Roles*

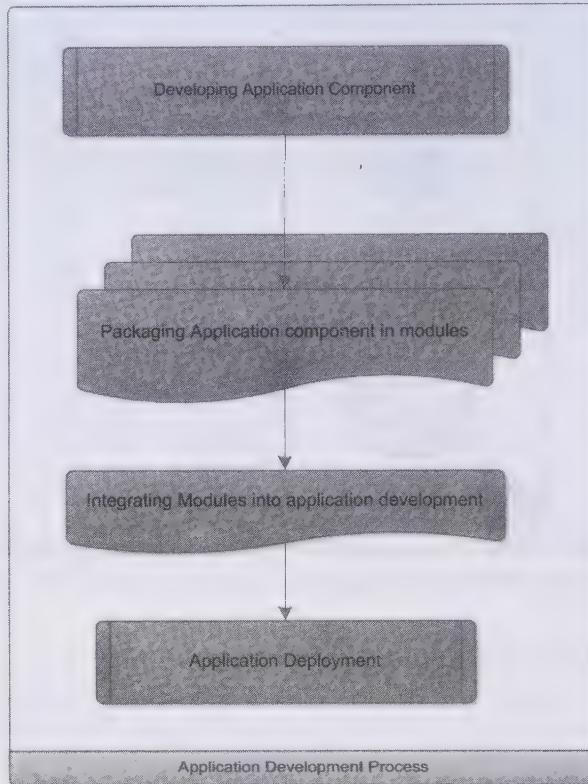
Different people play different roles in the development of a Java EE application. These roles are defined according to the Java EE specification. The reusable modules of the application help to divide the development and deployment processes of the application into distinct roles, which are as follows:

- ❑ **Java EE product providers** – Design and provide the Java EE platform, APIs, and other features in the Java EE specification. Product providers are the third party vendors that provide the operating system, the database system, the application server, or Web server. These products implement the Java EE platform according to the Java EE 6 specification.
- ❑ **Tool providers** – Provide tools to develop, assemble, and deploy Java EE applications, which are used by component providers, assemblers, and deployers.
- ❑ **Application component providers** – Provide Web components, EJBs, applets, or application clients for use in Java EE applications. The application component providers design the HTML documents and develop Web or EJB components. The people who are assigned this role can perform the following tasks:
  - Write and compile the source code
  - Specify a Deployment Descriptor for a client
  - Bundle the compiled files and Deployment Descriptor for application deployment
- ❑ **Application assemblers** – Receive application components from component providers, specify Deployment Descriptor, and package the application components into a Java EE application.
- ❑ **Application deployers and administrators** – Configure as well as deploy Java EE 6 applications and manage the environment on which the Java EE 6 applications run. The duties of an application deployer include setting transaction controls and security attributes, and specifying connections to databases.

Let's now move ahead and explore the application development process.

### *Application Development Process*

The Java EE specification describes the application development process in certain predefined steps, as shown in Figure 1.17:



**Figure 1.17: Displaying Java EE Application Development Process**

As Figure 1.17 shows, you first need to model the business rules in the form of application components and then package the application components into modules. The multiple modules are then integrated into Java EE applications, and, finally, the packaged application is deployed and installed on the Java EE platform application server(s).

Let's now understand the preceding steps in detail.

## Developing the Application Component

In Java EE, the development process involves dividing the entire application into modules, and modules into application components. This ensures efficient resource utilization as well as ease in the development of the application. After all the components have been developed, they are integrated to make a module. These modules are further integrated to create the final application. The development of different components, such as servlets, JSPs, and EJBs, has been described in more detail in the respective chapters of this book.

## Packaging Application Components into Modules

Application components of the same type are packaged into a module. A Deployment Descriptor describes the structure of each module. You can have three different types of modules in an application:

- The Web module
- The EJB module
- The Client module

### *The Web Module*

A Web module represents a Web application. Assembling servlets, JSP files, and static content such as HTML pages into a single deployable unit creates a Web module. A Web module contains the following components:

- ❑ One or more servlets, JSP files, and HTML files
- ❑ A Deployment Descriptor, stored in an (XML) file
- ❑ A WAR file that is similar to a JAR file, but contains a WEB-INF directory. The web.xml file configures the Web components used in the Web module. This file also specifies the runtime behavior of the Web components.

Web modules are created as standalone applications, or combined with other modules to create Java EE applications. A Web module is installed and run in the Web container of an Application server.

### The EJB Module

An EJB module contains one or more EJBs, and other helper classes and resources. Extensive use of annotations in EJB 3.0 has eliminated the use of Deployment Descriptors, which were required earlier when working with previous versions of Java EE.

### The Client Module

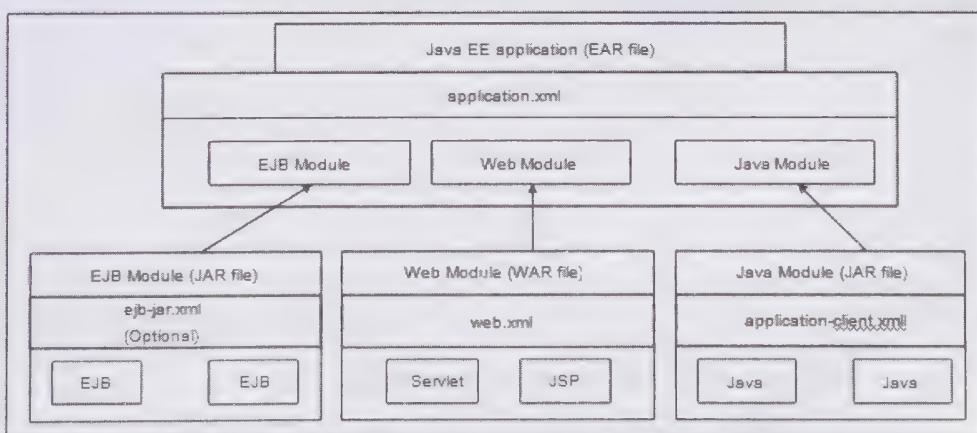
A client module is a group of Java client classes that can be directly accessed by a client. These classes are packaged into JAR files.

### Packaging Modules into Applications

A Java EE application is a combination of one or more EJB JAR files, Web application WAR files, and application client JAR files. The highest level of packaging (EAR-based packaging) is performed when one or more modules are packaged into an application. A Java EE container loads each application by using a different class-loader so that each application is isolated from the other applications. This allows multiple Java EE applications to coexist within one Java EE container.

In a Java EE application, one or more modules are composed into an EAR file. An EAR file is similar to a JAR file, except that it contains an application.xml file (located in the META-INF directory). The application.xml file provides a means of specifying which modules make up the application.

A sample composition of modules is shown in Figure 1.18:



**Figure 1.18: Displaying the Composition of Modules into Applications**

You can package a Java EE application into the EAR file by using the JAR command. In other words, you should package all the files in the EAR file that can be deployed on the Java EE application server. Any Java EE application package should contain Java EE modules and the application Deployment Descriptor.

### Deploying the Application

When you deploy a Java EE application, you install and configure the packaged modules onto a Java EE platform. The steps involved in deploying a Java EE application are as follows:

- ❑ **Installing the application**— Allows you to copy EAR files on the application server. During the deployment of the EAR file on the application server, additional implementation classes are generated with the help of a container, and finally the application is installed on the server.
- ❑ **Configuring the application**— Allows you to customize applications with application server-specific information.

Let's now discuss a few compatible products for the Java EE platform.

## **Listing the Compatible Products for the Java EE Platform**

Every Java EE technology must have an associated compatibility test suite (CTS), which verifies that a product correctly implements the standards required for delivering secure, robust, and scalable applications. For example, a servlet container must use HTTP posted form variables provided by the `HttpServletRequest` object, which is passed to the servlet. The test for this condition would be to post variables to a servlet and verify that the servlet receives all the variables with the correct values.

Sun Microsystems requires every technology that supports the Java EE platform and its specifications to pass the corresponding test suite. The tests also verify correct interoperation of the Java EE technologies, as specified by the Java EE specification.

Some of the products compatible with the Java EE platform are as follows:

- ❑ Allaire
- ❑ ATG
- ❑ Bea Systems
- ❑ Borland
- ❑ Computer Associates
- ❑ Fujitsu
- ❑ Hitachi
- ❑ HP
- ❑ IBM
- ❑ IONA
- ❑ iPlanet
- ❑ Macromedia
- ❑ NEC
- ❑ Oracle
- ❑ Pramati
- ❑ SilverStream
- ❑ Sybase
- ❑ Talarian
- ❑ Trifork

Apart from the preceding products, other third-party vendors, such as TogetherSoft, WebGain, and Borland, are committed to provide the Java EE compatibility for their products. With their experience in modeling and rapid application development, these vendors help to make the development of Java EE applications easy and quick.

Let's now explore the servers used to deploy the Java EE applications.

## **Section D: Introducing the Servers for Java EE Applications**

Depending on the application type, a developer can select the appropriate server for an application. For example, if the developer wants to deploy a Web application, the Tomcat server or the glassfish application server can be used. This section introduces the servers that can be used for Web or enterprise Java applications, which are of two types:

- ❑ Web servers
- ❑ Application servers

Note that the selection of a server depends on the type of application. A Web application is deployed on a Web server; however, an enterprise application is deployed on an application server.

Let's now briefly discuss these servers in the following sections.

## Introducing Web Servers

A Web server is a computer or virtual machine used to run Web applications. The main purpose of the Web server is to provide Web pages to clients. In the case of Web servers, a client uses a Web browser to make an HTTP request for a specific resource. You can deploy and run Java Web applications on the Tomcat server, which is an open source Web server introduced by Apache. The latest version of the Tomcat server is Tomcat 6.0. You can download the Tomcat server from the Tomcat website or using the <http://tomcat.apache.org/> link, and install it on your computer. However, before installing the Tomcat server, you must ensure that Java is also installed on the computer. By default, the Tomcat server uses port 8080 to run Web applications. After installing Tomcat, you can start or stop the Tomcat server by using its Windows service installed on your computer.

## Introducing Application Servers

An application server is a server program that provides the business logic for an application. In other words, the application server provides a GUI to run three-tier applications. The application server acts similar to an extended virtual machine used to run applications and handle database transactions. In Java Platform, the term application server sometimes refers to the Java EE platform. The following are some noteworthy application servers used to deploy Java EE applications:

- The WebLogic application server
- The WebSphere application server
- The JBoss application server
- The Glassfish Application server

### **NOTE**

*In this book, we use the Glassfish Application server to run both Web and enterprise applications.*

Let's explore each of these application servers in detail.

### *The WebLogic Application Server*

The WebLogic application server, owned by Oracle (originally BEA Systems), serves as the server software application that runs on the middle tier. It is the leading online transaction processing platform used to connect various users in a distributed computing environment. The WebLogic application server is based on the Java EE platform and includes connectors so that a client can interoperate with various server applications and EJB components. The WebLogic application server provides the functionality of resource pooling and connection sharing to developers.

### *The WebSphere Application Server*

The WebSphere application server, owned by IBM, is built by using open standards, such as Java EE, XML, and Web Services. It works with various Web servers, such as Apache HTTP Server, Netscape Enterprise Server, and Microsoft Internet Information Services. The latest WebSphere application server, version 7, has the following features:

- Provides flexible management capabilities for various WebSphere application server-based editions
- Offers the facility of managing application artifacts, irrespective of the packaging and programming models
- Simplifies the process of updating the configurations of an application with the help of a property file

### *The JBoss Application Server*

The JBoss application server, owned by JBoss, is an open source Java EE-based application server. It allows you to package, deploy, and run various Java EE applications developed by using Java EE related technologies, such

as EJB, JSP, JSF, and Hibernate. JBoss AS 5.1 is the latest version of the JBoss application server and can operate as a Java EE 6 application server.

## **The Glassfish Application Server**

The Glassfish application server, owned by Sun Microsystems, is a platform for server-side applications and Web services. It is developed as the Glassfish open source project and is also known as Sun Java System Application Server. Glassfish application server version 3 is used to package, deploy, and run Java EE 6 applications.

The server used to run applications in this book is Glassfish v3. Therefore, ensure that the Glassfish v3 application server is installed on your computer before running the applications of this book. The process of downloading and installing Glassfish v3 (formerly known as Sun Java System Application Server) is discussed in *Appendix F, Installing Java EE 6 SDK Update 3*.

This ends the discussion on the servers used to package, deploy, and run Java EE 6 applications. Let's now briefly explore all the Java EE 6 related technologies that has been discussed in the succeeding chapters of this book.

## **Section E: Snapshot of Java EE Related Technologies**

Java EE is a set of coordinated technologies that reduces the complexity of developing, deploying, and managing multi-tier, server-centric enterprise applications. These technologies help to build stable and secure Web and enterprise applications. This book discusses the following Java EE related technologies, which are used to create, deploy, and manage enterprise applications:

- Java Database Connectivity
- Java Servlet
- JSP
- JavaServer Faces
- JavaMail
- Enterprise JavaBeans
- Hibernate
- Seam
- Java Connector Architecture
- Web Services
- Struts
- Spring
- JAAS
- AJAX

Now let's discuss each of these technologies in detail.

### **Java Database Connectivity**

The JDBC technology helps establish connections between Java applications and databases such Oracle, MySQL, and MS SQL Server 2005. Many Web or enterprise applications need to interact with databases to store client-specific information. The JDBC technology provides the JDBC API, which is implemented in Java-based applications to enable the applications to access data from databases.

For example, large organizations have numerous chunks of employee and client information. These organizations need to store the information in a database so that the information can be managed, processed, and used efficiently. Web or enterprise applications need to interact with backend databases to retrieve and manipulate data. This led to the evolution of JDBC, which provides the JDBC API to enable Java-based applications to connect to databases.

*Chapter 3, Working with JDBC 4.0*, discusses JDBC 4.0 and the JDBC API with the help of various examples, ensuring better comprehension of the various classes, interfaces, and methods of the API.

## Java Servlet

Java Servlet, a Java platform technology, provides a component-based, platform-independent approach to create Web applications. While developing Web or enterprise applications, a servlet uses the JDBC API to access databases. Java EE provides Servlet 3.0 with new features, such as annotations, to develop a servlet class. *Chapter 4, Working with Servlets 3.0*, describes the Servlet API and the Servlet context.

In a Web application, client details such as logging information need to be maintained across multiple Web pages to implement session handling. Earlier, other mechanisms, such as URL rewriting, hidden form fields, and cookies, were used for session handling. Java Servlet provides an API to implement the session handling mechanism to retain a client's state.. *Chapter 5, Handling Sessions in Servlets 3.0*, elucidates the advantages and disadvantages of various session handling mechanisms provided by Java Servlet.

## JavaServer Pages

The JSP technology is an extension of the Java Servlet technology, and has been created to support embedding of Java code in HTML pages. JSP makes it easier to combine static content with dynamic content, ensuring an easy approach to build Web applications. JSP enables you to embed Java code in an HTML page by using scriptlets. The focus of Java EE 6 has been to simplify the development of Web or enterprise applications by using Java annotations. JSP 2.1 extends this impetus of Java and allows the use of annotations for dependency injection on JSP tag handlers and context listeners.

*Chapter 8, Implementing JSP Tag Extensions*, elaborates on various topics related to JSP, such as classic and simple JSP tag handlers and the Tag Extension API. The chapter also includes examples and applications to provide you a better understanding of implementing classic and simple tag handlers.

## JavaServer Faces

While developing a Web application, a developer designs a UI to interact with clients, develops the business logic to process data, and implements navigation rules to be followed when a client accesses the application. Before the introduction of JSF, programmers had to manually provide the code to define common tasks such as validating user inputs and converting user input strings into Java objects to build Web applications. Consequently, this diverted the attention of the programmers from the business logic of the applications. However, JSF has helped simplify the complexity of code for programmers and enabled them to handle these tasks easily through its APIs and tags. In addition, JSF also helps programmers to design rich UIs with its components. As a consequence, the programmers are now able to concentrate on developing and implementing the business logic, which is an essential part of any application.

*Chapter 11, Working with JavaServer Faces 2.0* shows you how to create the standard UI components by using the JSF core and HTML tags.

## JavaMail

JavaMail is introduced to implement the functionality of sending and receiving mails in an enterprise application. JavaMail provides the JavaMail API, which contains the classes and interfaces required to develop mail and messaging applications. The JavaMail API provides a platform and protocol-independent framework to send and receive mails. *Chapter 12, Understanding JavaMail*, discusses JavaMail 1.4 and also shows you how to create an application to send and receive mails. This application also helps you to understand the implementation of various classes and interfaces of the JavaMail API.

## Enterprise JavaBeans

Initially, Java programmers used JavaBeans to implement the business logic while developing an enterprise application. JavaBeans help to create UI components, but do not provide services such as transaction management and security, which are required to develop robust and secure enterprise applications. This led to the evolution of EJB, which extends Java components and provides services that help in enterprise application development.

However, EJB has proved to be a complex technology for programmers, as they need to create the Home and Remote interfaces. In EJB 3, programmers do not need to create the Remote interfaces. In addition, EJB 3 also introduces Java Persistence, a standard API used to manage persistence and Object Relational Mapping (ORM). Chapter 14, *Implementing Entities and Java Persistence API 2.0*, discusses the Java Persistence API and Java Persistence Query Language (JPQL), which has simplified the task of creating Java queries.

## Hibernate

Hibernate is an open source framework that enables a developer to work easily with relational databases in an enterprise application. In addition to the Java Persistence API, Hibernate provides the feature of mapping Java classes to database classes. Hibernate also generates Hibernate Query Language (HQL) calls, which automate the process of handling result sets. Hibernate can be used in both standalone Java applications and Java EE applications using Java Servlet and EJB. Chapter 15, *Implementing Java Persistence Using Hibernate 3.5*, describes the Hibernate API used to generate HQL queries. The chapter also includes an application to demonstrate the implementation of Hibernate 3.5.

## Seam

Seam is a framework that integrates the JSF and EJB technologies. JSF helps to create UI components of an enterprise application. However, creating UI components by using JSF requires large volumes of coding. Due to this, programmers have to focus more on the presentation layer of an enterprise application, when they should be focusing on the business logic. Using Seam allows a developer to focus on the business logic of the application.

Chapter 16, *Implementing JBoss Seam 3*, discusses the Seam context and components, and provides an application to demonstrate the implementation of Seam.

## Java EE Connector Architecture

As more and more businesses adopt e-business strategies, integration with existing enterprise information systems (EISs) is becoming the key to success. Organizations with successful e-businesses need to integrate their existing EISs with new Web applications. They also need to extend the reach of their EISs to support business-to-business (B2B) transactions.

Java Connector Architecture (JCA) defines the standard to integrate different EISs. Before JCA was defined, no specification for the Java platform addressed the problem of providing a standard architecture for integrating heterogeneous EISs. Most EIS vendors and application server vendors used non-standard vendor-specific architectures to provide connectivity between application servers and EISs. With the advent of JCA, businesses can now look forward to easy and simplified integration of their EISs to implement B2B transactions. Chapter 17, *Java EE Connector Architecture 1.6*, discusses JCA, EIS resources, and EIS services in detail.

## Web Services

As a result of major IT developments over the last few years, most people and companies now use broadband connections to connect to the Internet. The applications implementing Web services were introduced to simplify the process of accessing the resources on the Internet.

The applications that implement Web services are simple applications that run on a Web browser. These applications are built based on Web browser standards and can generally be accessed by any browser on any platform. Web services are composed of modularized services that can be registered, searched, and called over the Internet. Web services generally use SOAP to transfer data over the Internet.

Chapter 19, *Implementing SOA using Java Web Services*, discusses the Web Service specifications of Java EE 6, such as JAXB, JAX-WS, and JAXR.

## Struts

The tasks of creating the model, view, and controllers of a Web application have been simplified by the evolution of the Struts framework. Struts provide various tags, which have simplified the implementation of the view component. Instead of providing code to implement the functionality of a view, Struts provides various tags. You can use these tags to simplify the implementation of various functionalities, such as validation.

*Chapter 20, Working with Struts 2*, describes the Struts 2 API, and demonstrates how to create a simple login application by using the Struts 2 API.

## Spring

JavaBeans, which were used to implement business logic, did not provide services such as state and transaction management. Therefore, EJB was introduced to solve these problems with the help of EJB containers. The EJB container provides state management and transaction services during runtime. However, programmers find it difficult to create enterprise applications by using EJB as they have to create the Home and Remote interfaces. Moreover, running EJB-based enterprise applications are increasingly difficult for Java programmers. Therefore, Spring was introduced to simplify the task of the programmers. Spring, an open source application development framework, is a collection of sub-frameworks, also called layers, such as Spring AOP, Spring ORM, Spring Web Flow, and Spring Web Model View Controller.

*Chapter 21, Working with Spring 3.0*, discusses Spring in detail and includes an application developed by using the Spring Web MVC framework.

## JAAS

Java EE provides JAAS to implement security in Web and enterprise applications. Security refers to the process of implementing authentication and authorization to access the Web resources of an application. Security can be implemented programmatically or declaratively, as discussed in *Chapter 22, Securing Java EE 6 Applications*. The chapter includes various applications to reveal the implementation of programmatic, form-based, and declarative security.

## AJAX

Earlier, while accessing a Web page, clients had to wait for a long time after refreshing a page or clicking a button. While refreshing a Web page, the browser first sends the request to the Web server, which then performs the required operations. After this, the response is provided to the browser. While performing these operations, the client cannot do anything other than waiting for the response to be provided by the server. However, now, all this has changed. AJAX, with the help of JavaScript, provides asynchronous client and server interaction. The AJAX engine handles a client's request instead of sending the request to the server. Moreover, clients can perform other tasks while a request is being processed, since AJAX enables asynchronous client and server communication. Appendix D, *AJAX*, discusses the AJAX architecture and explains in detail how AJAX handles client requests by using the AJAX engine.

Let's now recap the main points of the chapter in a short summary.

## Summary

The chapter has been divided into five sections: A, B, C, D, and E. Section A has discussed the evolution of Java from other programming languages such as C and C++. It has introduced Java as a programming language, a platform, and a virtual machine used to compile Java applications. Section B has discussed the different types of enterprise architectures; i.e., single tier, 2-tier, 3-tier, and n-tier architecture, based on which enterprise applications are created. It has also discussed the objectives of an enterprise application. Section C has primarily focused on Java EE 6 and discussed its features, APIs, architecture, and containers. This section has also discussed the compatible products for the Java EE platform and provided various types of architectures for developing an enterprise application. Section D has described Web and application servers used to run Web and Java EE applications, respectively. Section E has provided an overview of Java EE related technologies that are covered in the book and the need for each technology.

## Quick Revise

### Q1. What is Java?

Ans. Java is a platform-independent programming language used to create secure and robust applications that may run on a single computer or may be distributed among servers and clients over a network.

**Q2. What is Java EE application?**

Ans. A Java EE application is a collection of Java EE modules. These modules can be Web components or enterprise components. The Web components can be servlets and JSPs and enterprise components can be EJBs, connectors, and other application clients. All these Java EE modules are combined to form an integrated enterprise application. A single archive file with a .ear extension is created to package a Java EE application.

**Q3. Define a servlet.**

Ans. A servlet is a simple Java program, which is executed on the server to generate dynamic content by using the Java technology. The dynamic content is normally an HTML document, and may also be an XML document as well. A servlet is capable of maintaining a state of a user in a session-based application. A Web container in a Web server is responsible for the life cycle management of a servlet.

**Q4. What is a 3-tier architecture?**

Ans. A 3-tier architecture is basically a client/server architecture in which the user interface, business logic, and data handling code are developed and maintained as independent modules. These three tiers are named the Presentation tier (user interface), the Application tier (business logic), and the Data tier (data access and storage).

**Q5. What is an EJB?**

Ans. EJB refers to Enterprise JavaBean, a distributed application component model. Applications created with EJB are highly scalable, transactional, and secure. An EJB is maintained throughout its life cycle by the EJB container.

**Q6. Name the technologies included in the Java EE platform.**

Ans. The key technologies that make the Java EE platform are Java Servlet, JSP, EJB, JCA, JMX, JAXR, JMS, JNDI, JTA, JDBC, the Deployment API, and J2EE Authorization Contract for Containers.

**Q7. Differentiate between Java, Standard Edition, Java, Enterprise Edition, and Java Micro Edition.**

Ans. The three Java editions can be differentiated as follows:

- Java, Standard Edition is a set of APIs used to develop a variety of applications, including standalone applications, applets, and client applications
- Java, Enterprise Edition is used to develop server-side applications with a component-based approach
- Java, Micro Edition is used to build applications for micro devices, such as mobile phones

**Q8. A messaging standard that allows Java EE application components to create, send, and receive messages is .....**

- A. JMS      B. JAAS      C. JDBC      D. JNDI

Ans. The correct option is A

**9. List the three modules of an enterprise application developed by using Java EE 6.**

Ans: The three modules of an enterprise application are as follows:

- The Web module
- The EJB module
- The Client module

**10. Explain the two tasks involved in deploying an application.**

Ans. The two tasks involved in deploying an application are as follows:

- Installing the application:** Refers to the process in which EAR files are copied on the application server, additional implementation classes are generated with the help of the container, and finally the application is installed on the server
- Configuring the application:** Refers to the process in which the application is customized with application server-specific information.

# 2

# Web Applications and Java EE 6

## *If you need an information on:*

Exploring the HTTP Protocol	36
Introducing Web Applications	41
Describing Web Containers	46
Exploring Web Architecture Models	47
Exploring the MVC Architecture	49

A Web application is a collection of multiple Web components that interact with each other to execute specific business logic. You can use Web browsers to access Web applications over a network, such as the Internet or intranet. Web applications perform various operations, such as accepting input data and dynamically generating one or more Web documents as responses. The Web documents generated as responses are displayed to a user in standard formats, such as Hypertext Markup Language (HTML), which are supported by common Web browsers.

In the Java EE platform, Web components such as Java Servlet or JavaServer Pages (JSPs) provide dynamic capabilities for a Web server. Servlets are Java classes that dynamically process requests and generate the desired responses. JSP pages are text-based documents that are executed as servlets and are used for creating static content. Web components process requests and perform other required operations by the help of a runtime platform known as a Web container. The Web components access various services, such as request dispatching and security, with the help of a Web container. For example, when a Web component is accessed by a user, the authenticity of the user is identified with the help of the Web container. In addition, the request sent by a user is controlled by the Web container.

While deploying an application on the Web container, you need to configure its various aspects, such as resource to a url-pattern, initialization parameters, and session details. The configuration information is maintained in an Extensible Markup Language (XML) file, called Web application Deployment Descriptor. However, in the Java EE 6 platform, the use of annotations has minimized the task of providing configuration information in Deployment Descriptors. You should note that Web components can also be integrated in an application with the help of annotations.

This chapter introduces Web components, Web Modules, and Web containers. As Web applications mostly use Hyper Text Transfer Protocol (HTTP) for data transfer, the chapter also provides a detailed knowledge about HTTP request and response. Further, a detailed description of the Web application is provided, including its benefits, components, and structure. Towards the end, the chapter discusses various Web application models and also explores the Model, View, and Controller components.

Let's start our discussion by exploring the role of HTTP in transferring information on the Internet.

## Exploring the HTTP Protocol

HTTP is a stateless, application-level communication protocol used to transfer information on the Internet. The main aim of HTTP is to send and receive user information over a network. As an application-level protocol, HTTP also defines the types of requests that clients can send to a server and the responses that the server sends back to the clients. Each request has a Uniform Resource Locator (URL), which is a String that identifies a Web component or a static object, such as an HTML page or an image file. Therefore, most of the Java EE Web clients use HTTP to browse a URL and communicate over a network. If the communication between the client and server in a Web application is HTTP-based, the Web server converts the HTTP client request to an HTTP request object. Next, the HTTP request object is delivered to the Web component that is identified by the request URL. Further, the Web component sends the HTTP response object, which is then converted into the HTTP response by the application server and finally sent to the client.

In this section, let's explore the working of HTTP and the concepts related to it, such as HTTP requests and responses.

### .H2 Processing HTTP Requests

Let's try to understand the underlying nature of HTTP requests and working of the HTTP protocol over a network with the help of an example. Consider a scenario where a user wants to visit a URL, say [www.google.com/intl/en/privacy.html](http://www.google.com/intl/en/privacy.html). Now, the process of sending the request and retrieving the response from the specified URL would be as follows:

- ❑ The user makes a request for the specific URL
- ❑ The Web browser establishes a connection with the [www.google.com](http://www.google.com) server by locating its IP address from a DNS server
- ❑ The Web browser then sends a request to the server to retrieve the `/intl/en/privacy.html` file

- The server responds with the information about the requested HTML page and the contents of the index.html file

This process of sending a request and receiving a response from the server is known as transaction. An HTTP session, i.e. the process of maintaining the client's state, remains open till a single transaction is completed. After the transaction is completed, the session is automatically closed. For example, if you want to access the <http://www.google.com> URL, the HTTP session would remain open until the server is connected to the requested URL to access the requested Web page.

Let's use the telnet command line interface to remotely access the [www.google.com](http://www.google.com) server. By default, telnet connects on server socket 23.

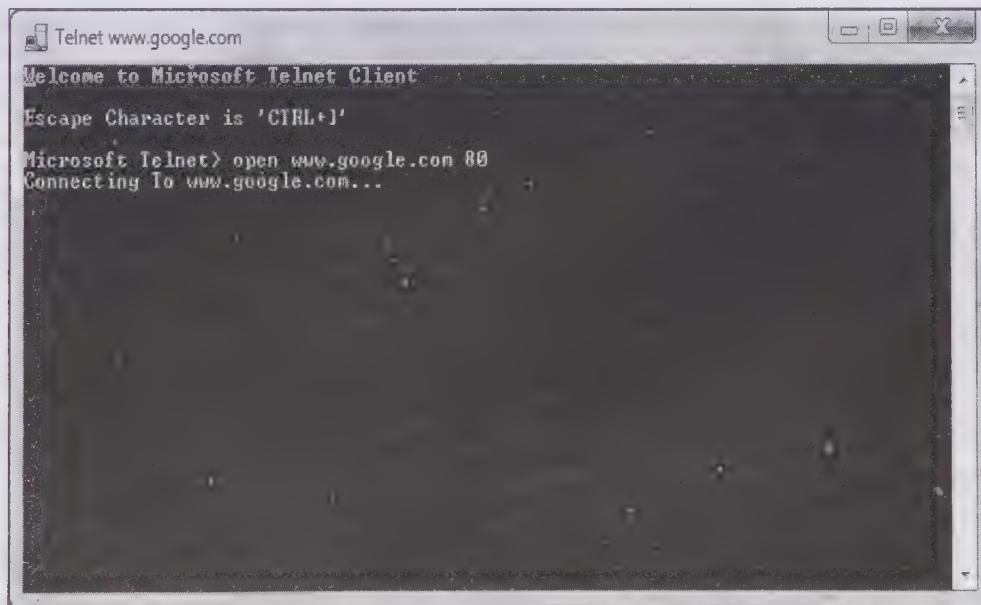
### **NOTE**

*Telnet is a network protocol used on the Internet to establish a connection with remote servers. In Windows 7, by default, the Telnet Windows feature is disabled. You first need to enable the Telnet Windows feature from the Windows Features dialog box, which is displayed by accessing the Turn Windows Features on or off option from Control Panel. Next, type telnet in the Search for programs and files text box of the Start menu to access the telnet client console.*

The following command establishes a connection with the [www.google.com](http://www.google.com) server by using Telnet Client :

```
open www.google.com 80
```

Telnet, a client-server protocol, looks up the IP address of [www.google.com](http://www.google.com) and connects to it on server socket 80. The console establishes a connection with the server and the relevant message is displayed, as shown in Figure 2.1:



**Figure 2.1: Displaying the Connection with the google.com Server**

After the connection is established, press the ENTER key until the cursor reaches a blank space.

Now, you can make a request to access the /intl/en/privacy.html file. The following command shows the use of the HTTP GET method to access the requested page:

```
GET /intl/en/privacy.html HTTP/1.1
```

After this request has been sent to the server, the server replies by sending a response header along with the resource requested.

Figure 2.2 shows the output that is echoed to the console after a short duration:

```

HTTP/1.1 200 OK
Content-Type: text/html
Last-Modified: Thu, 25 Mar 2010 09:42:43 GMT
Date: Wed, 14 Apr 2010 09:22:45 GMT
Expires: Wed, 14 Apr 2010 09:22:45 GMT
Cache-Control: private, max-age=0
Vary: Accept-Encoding
X-Content-Type-Options: nosniff
Server: gfe
Transfer-Encoding: chunked
1000
<!DOCTYPE html>
<html lang="en">
    <meta charset="utf-8">
    <title>Google Privacy C...
enter</title>
    <link rel="stylesheet" href="http://www.google.com/css/privacy.css">
<h1><a href="/"></a> Privacy Center</h1>
    <ul id="nav">
        <li>

```

**Figure 2.2: Displaying the Output of the Request Made**

In Figure 2.2, the code provided after Content-Type is written in HTML, so that you can recognize the text. The HTTP sessions only last for one transaction and after that they are closed.

Now, after discussing the process of sending requests and receiving responses, let's discuss the various methods of HTTP requests in detail.

## Describing HTTP Requests

An HTTP request is a class consisting of HTTP style requests, request lines, request methods, request URL, header fields, and body content. HTTP 1.1 defines the following request methods to be used with user requests:

- GET
- HEAD
- POST
- PUT
- DELETE
- OPTIONS
- TRACE

Now let's discuss each of these in detail.

### The **GET** Method

You can use the GET method to access the static resources, such as HTML documents and images. Apart from the static resources, the GET method also allows you to retrieve the dynamic information by using query parameters in the request URL. For instance, we can send a parameter name with the URL, such as `http://www.domain.com?name-Tim`. In this example, Tim is the dynamic information sent by including the name parameter in the request URL. The Web server can then access this dynamic information through the name parameter.

The GET method contains a Request-URI, which is used to retrieve information from the request. URI stands for Uniform Resource Identifier. If the Request-URI refers to the data-producing process, the produced data is sent as an entity in the response. However, if the Request-URI refers to the source text of the process, the text is returned as the output. The usage of the GET methods can be changed according to the request made by the user. If the user request includes the If-Modified-Since, If-Unmodified-Since, If-Match, If-None-Match, or If-Range header fields, the GET method is converted to a conditional GET method, in which case the conditional header fields are defined. The desired entity is transferred to the client only if the condition specified in the

conditional header field is fulfilled. You should note that the conditional GET method allows cached entities to be refreshed, resulting in the reduction of unnecessary network usage.

If the GET method associated with a request contains the range header field, it is called a partial GET method. If the required data already exists with the client, it is not transferred by the partial GET method, which leads to the reduction of network usage. In this type of method, only a part of the entity is transferred; therefore, the name partial GET method.

## The HEAD Method

Sometimes, a client might only need to view the header of a response (Content-Type or Content-Length). In such cases, the client can use the HEAD request method to retrieve the header. The HEAD method is similar to the GET method, except that the server does not return a message body in response to the HEAD method.

The HEAD request method can be used to retrieve the meta information of the entity requested by the user. The meta information in the HEAD request method must be identical to the information sent in the response to a GET request. Sometimes, the response to a HEAD request is cacheable. It indicates that the information contained in the response may be used to update a previously cached entity from that resource. In case of any modifications, if the new field value indicates that the cached entity differs from the current entity, the cached entity needs to be updated.

Based on the type of the request, the client application specifies the resource that is required as a part of the request. To understand this concept better, consider a scenario where a user provides the `http://java.sun.com/index.jsp` URL in the address bar of a Web browser. The Web browser sends the GET request to the `java.sun.com` Web server to locate the `index.jsp` resource. You should note that a URI should specify a resource name in the HTTP lexicon. In other words, apart from the domain name in URL, a specific resource name should also be provided. For example, in our case, URL is `http://java.sun.com/index.jsp`, which contains `index.jsp` as the resource name and `java.sun.com` as the Web server. The `index.jsp` file is located in the document root of the Web server serving the `java.sun.com` domain.

## The POST Method

The POST method is generally used in cases where a large amount of information needs to be sent to a Web server. This method allows you to access dynamic resources and provides the following functionalities:

- Defines annotations of existing resources
- Posts a message to a mailing list or newsgroup
- Defines a block of data, such as the result of submitting a form, to a data-handling process
- Extends a database through an append operation

The function of the POST method is dependent on the Request-URI. The posted entity is a part of that URI in the same way as a file is a part of a directory containing it or a news article is a part of a newsgroup to which it is posted.

The 200 (OK) or 204 (No Content) response status is generated if the response provided by the POST method does not contain the resource identified by a URI. If the resource that is requested by a user is created on the origin server, the 201 (Created) response status is provided. The 201 (Created) response contains a new resource describing the request status and a location header.

A POST request allows the encapsulation of multi-part messages into the request body. For example, you can use POST requests to upload text or binary files from the server. POST requests can also be used in applets to send serializable Java objects, or even raw bytes, to the Web server. In addition, POST requests offer a wider choice compared to GET requests in terms of the contents of a request.

Table 2.1 lists the differences between the GET and POST requests:

**Table 2.1: Differences between the GET and POST Methods**

<b>Parameter</b>	<b>GET</b>	<b>POST</b>
Transmission of request parameters	Transmits the request parameters in the form of a query string that is appended to the request URL.	Transmits the request parameters within the body of the request.
URL size	Limits you to a maximum of 256 characters in the URL, excluding the number of characters in the actual path.	Does not limit the size of the URL for submitting name/value pairs, as they are transferred in the header and not in the URL.
Purpose	Allows you to retrieve data.	Allows you to retrieve, save, or update data. It also allows you to order a product or send e-mail messages.
Caching ability	Considers the request as cacheable.	Does not consider the request to be cacheable.
Type of resource	Allows a user to access the static resources, such as HTML documents.	Allows a user to transfer the information according to the request made by the client. The POST method is used to transfer large and complex information to the server.

### The **PUT** Method

The PUT method allows you to store an entity in the specified Request-URI. You should note that if the Request-URI in the URL does not reference the name of an existing resource, the Web server creates the resource with the specified name. Apart from creating a new resource, if any modification is done to an existing resource, the 200 (OK) or 204 (No Content) response codes are generated. These response codes depict that the modification in the existing resource is done successfully. If a new resource is created on the request of a user, the 201(Created) response code is generated, informing the user about the successful creation of the resource. However, if either the new resource is not created properly or the modifications are not done appropriately in the existing resource, the corresponding response codes are generated, reflecting the nature of the problem.

Various URIs can identify a single resource. For example, a resource might have a URI for identifying all logged in users, which is separate from the URI identifying each login user. In this case, a PUT request on a general URI may result in various other URIs that are being defined by Web server.

### The **DELETE** Method

You can delete a resource, if it is no longer needed, by using the DELETE method. While deleting a resource, you should provide the name of the resource to be deleted in the Request-URI. You should note that if the response contains the status of deletion of a resource, the 200 (OK) response code is generated, depicting that the resource has been successfully deleted. If the response is 202 (Accepted), it specifies that the resource has not yet been deleted. Similarly, if the response code is 204 (No Content), it specifies that the resource has been deleted but the response does not include an entity. Responses to the DELETE method are not cacheable. This method permanently deletes the files from the cache as well.

### The **OPTIONS** Method

The OPTIONS method requests the information related to various communication options available on request and response. You should note that the responses to the OPTIONS method are not cacheable. The capabilities of a Web server and the options or requirements related to a resource can be determined with the help of the OPTIONS method. If an entity body is included in the OPTIONS method, the media type should be provided by the content-type field.

If an asterisk (\*) is the Request-URI, the OPTIONS method is applied generally to the server instead of being applied to a specific resource. As the communication options of a server mainly depend on the requested resource, the \* request is useful as a ping or no-op type of method. It only allows the client to test the capabilities of the server; for example, the \* request can be used to test a proxy for HTTP/1.1 compliance. If the

Request-URI does not contain an asterisk, the OPTIONS method applies only to the options that are available while communicating with that resource.

## The TRACE Method

The TRACE method is used to invoke a remote application layer associated with a request message. A TRACE request must not include an entity. A client uses the TRACE method to diagnose or test the input received at the other end of the request chain.

The value in the header field of the TRACE method acts as a trace of the request chain. For the client's benefit, the Max-Forwards header field limits the length of the request chain, which is useful for testing a chain of proxies forwarding messages in an infinite loop.

## Describing the HTTP Responses

When an HTTP request is received by a server, the status of the response is displayed with some meta information about the response. This meta information constitutes the part of the response header. The server also sends the content that corresponds to the resource specified in the request, except for the case of the HEAD request. The following code snippet provides the meta information about the response:

```
Response = Status-Line
          *{([ general-header
              | response-header
              | entity-header ) CRLF );
          CRLF
          [ message-body ] }
```

The response that is to be displayed to the user contains a status line. A status line is generally the first line of the response header. The status line provides the information related to protocol versions, as well as the status codes, depending upon the responses. The response-header field allows the Web server to pass additional information of the response to the client. These headers inform the server about further access to the resources requested by the Request-URI.

If a client sends a request to a URL, for example `http://Java.sun.com/index.html`, the browser receives the content from the `index.html` files as a part of the request. The content header of the response include the Content-Type, Date, and Expires fields. These fields are used to set time-related data in a Web application. For example, the Expires header field of a page can be set to the date specified in the Date header field to indicate the browsers that they should not cache the page.

To indicate the type of content in request and response bodies, the response header fields use Multi-Purpose Internet Mail Extensions (MIME). MIME types include `text/html` and `image/gif` content types. In these content types, the first part of the header indicates the type of data, such as text and image, while the second part indicates the standard extension, such as `html` for text and `gif` for image. MIME is used to facilitate the exchange of different kinds of data files. At the beginning of each transmission, HTTP servers use MIME headers. Then, the information in MIME headers is used by the browsers to decide how to parse and render the content. MIME header is also used by browsers while transmitting data in the request body and deciding the type of data being sent. For example, the default MIME type is `application/x-www-form-urlencoded`, which is used for encoding in POST requests.

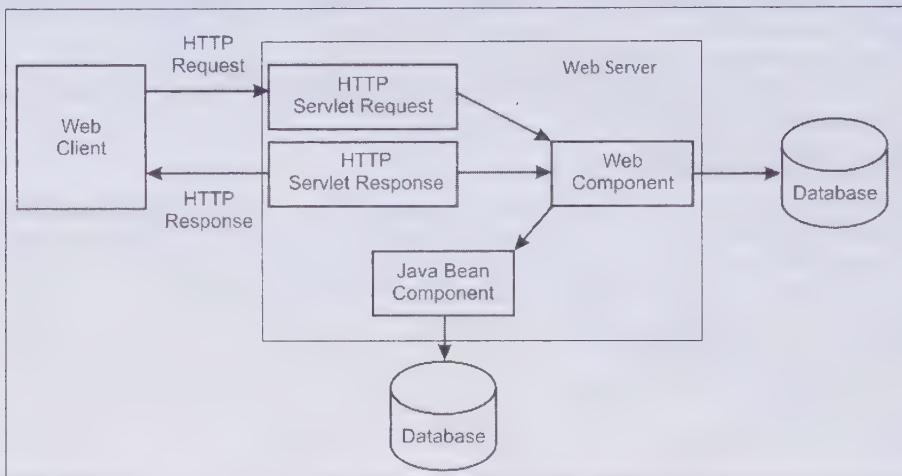
Let's now explore Web applications.

## Introducing Web Applications

A Web application is a server-side application that contains Web components (servlets and JSP pages), HTML/XML documents, and other resources in a directory structure or archived format, known as a Web Archive (WAR) file. A Web application is generally located at a central server, which provides services to various clients. In a Web server, the Web components can be conveniently distributed as Java Archive (JAR) files with the `.war` extension.

Web applications are of the following two types:

- ❑ **Presentation-oriented**—Generates dynamic content and Web pages by using various markup languages, such as HTML and XML. A presentation-oriented application generally serves as a client for the service-oriented Web applications.
  - ❑ **Service-oriented**—Implements a Web service and is invoked by presentation-oriented Web applications.
- Figure 2.3 shows the interaction between a Web client and a database:



**Figure 2.3: Displaying Request Processing in a Web Application**

Figure 2.3 depicts that:

- ❑ A Web server converts the request into an HttpServletRequest object.
- ❑ The HttpServletRequest object is sent to a Web component to establish a communication with JavaBeans components or a database for generating the dynamic content.
- ❑ The HttpServletResponse object is generated by the Web component to pass the request to the other Web component to retrieve the data from the database. Finally, the HttpServletResponse object displays the data to the client as the response generated by the Web server.

The development of Web application requires the following support:

- ❑ **A programming model and an API support**—Specify how to develop applications.
- ❑ **Server-side runtime support**—Includes support for applicable network services, and a runtime for executing the applications.
- ❑ **Deployment support**—Includes support for installing the application on the server. It also deploys the configuration details of the application components, such as specifying initialization parameters and specifying any database.

Let's now explore the components of a Web application.

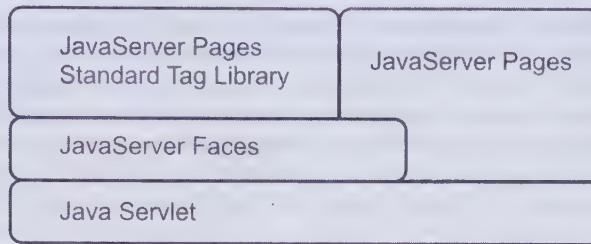
### *Describing Components of a Web Application*

A Web application consists of Web components, such as Java Servlet and JSPs. Earlier, Web-based applications used to face maintenance problems, which were solved by using the Model/View/Controller (MVC) paradigm for building user interfaces.

An application based on the MVC paradigm contains three components, Model, View, and Controller. A View is created by using JSP and the logic to manage requests is implemented in servlets, which serve as Controller. Moreover, the existing business rules in an application are referred to as Model.

With the introduction of Java Servlet and JSP technologies, the new technologies have also been introduced to build interactive Web applications.

Figure 2.4 shows the components of a Web application:



**Figure 2.4: Displaying the Components of a Web Application**

Now, let's discuss about the components of a Web application in detail.

## Java Servlet

A Java Servlet is used to extend the capabilities of servers in a client-server programming model and is considered the building block for developing Web applications. Java Servlet extends the functionality of a Web server. Although servlets can respond to any type of requests, they are commonly used in the Web applications hosted by Web servers. Similar to applets, servlets are also executed on the Web servers. In addition, servlets are portable platforms that are used to provide dynamic content, irrespective of the Web server in use. You should note that a browser-based application that calls a servlet does not need support for Java, as the output can be of any other type, such as HTML or XML.

Servlets are based on Java. This feature allows servlets to function on any platform that has a Java Virtual Machine (JVM) and a Web server that supports servlets. In a general Web application, if we make any change in the code of a servlet, it is necessary to recompile and redeploy that servlet so that the change is reflected in the application. Servlets can communicate directly with existing enterprise resources by using generic APIs, such as Java Database Connectivity (JDBC). As a result, you can simply and quickly develop the applications. In addition, developers can use servlets to extend the functionalities of a Web application similar to any Java application. For example, a Controller servlet can be extended to become a secure Controller. The functionalities of the original Controller can also be retained along with new security features.

The performance of servlets is better as compared to the Common Gateway Interface (CGI) scripts. This is because a CGI script needs to be loaded in various processes for every request. On the other hand, a servlet, once loaded in the memory, can be run multiple times on a single lightweight thread. Apart from this advantage, a servlet can also maintain or pool various connections to databases, resulting in less time consumption to process requests. Servlets can also directly access the parameters passed with the HTTP request because these parameters are considered as objects. However, in CGI-based applications, a form posts the parameters that are converted into environment properties to be used in programs.

## JavaServer Pages

JSP pages are used to create static content in the form of text-based documents. Apart from the static content, you can provide the dynamic content in the JSP pages. The JSP technology, introduced by Sun Microsystems, provides a rapid approach of developing Web pages and allows you to reuse the code based on the component-based architecture.

Let's look at how the JSP technology was introduced. You should note that both Java Servlet and JSP technologies have been introduced to achieve specific tasks. The Java Servlet technology was developed to serve as a mechanism to:

- ❑ Accept the client's requests from the Web browsers
- ❑ Get the enterprise data from the application tier or databases
- ❑ Perform application logic on the data
- ❑ Format the data for presentation in the browser

Initially, the Web designers could not preview the look and feel of an HTML page until runtime. It was also very difficult to locate the appropriate sections of code in a servlet when data or its display format changed. In addition, if the code of a servlet was modified, Web designers had to recompile and reload the servlets into the

Web server, provided that the presentation and business logic were implemented in a single servlet. With the introduction of the JSP technology, all these problems have been solved.

JSP provides a mechanism to specify the mapping from a JavaBeans component to the HTML (or XML) presentation format. A Web designer uses graphical development tools, such as JSP standard tag library (JSTL) and JavaServer Faces (JSFs) to create and view the content. JSF is also used to specify where data from the Enterprise JavaBeans (EJB) or enterprise information system tiers is displayed. Nowadays, the JSP technology allows you to use custom tags to format data of a JSP page dynamically; whereas, earlier you need to provide the Java code to format the data. Moreover, JavaBeans are used as components in a JSP page to implement the application logic.

The JSP technology allows the Web developers as well as designers to work efficiently and effectively. The Web designers can use the JavaBeans components or custom tags (provided by Web developers) to design user interfaces. Similarly, the main role of Web developers is to implement the logic in the application; therefore, the knowledge of designing the user interface is not required.

### JSTL and JSF

As explained in the previous subsection, the JSP technology is used to develop simple Web pages to provide dynamically generated content. Two more components, JSTL and JSF, can be used in a JSP page to simplify the code provided in a JSP page. JSTL is a set of libraries used by JSP for implementing JSTL tags; whereas, the JSF technology is a framework to build user interfaces for Web applications.

JSF, with a well-defined programming model, enables developers to quickly and easily build Web applications by using reusable User Interface (UI) components in a page. JSF allows the Web applications to manage all the complexities of creating a UI on the server.

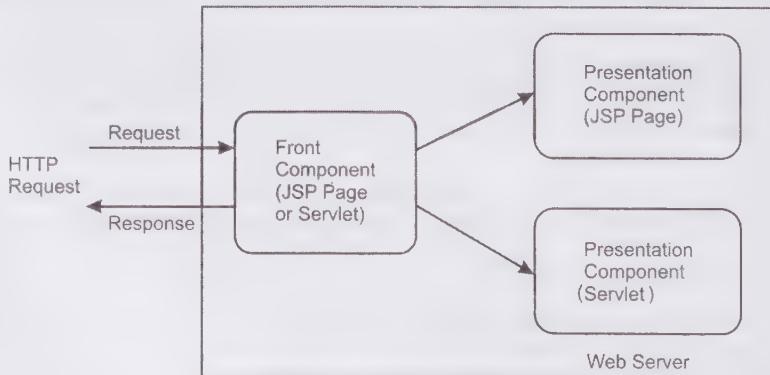
When a client sends a request to access a Web component, the component is processed in the server to generate the desired output for the client's request. The servlets used in the application are capable of handling complex logic processing, navigation paths between screens, and enterprise data.

You can reduce Java code within JSP pages by using custom tags and JavaBeans components in a Web application.

Web components of a JSF application are of the following types:

- **Presentation components** – Generates the HTML or XML response. A JSP page, JSF page, or servlet can be a presentation component. You should note that in a JSP page, the reusable custom tags or presentation logic can be implemented.
- **Front components** – Handles or converts the HTTP request into a form of an application.

The basic mechanism of using the presentation and front components is illustrated in Figure 2.5:



**Figure 2.5: Displaying Roles of Web Components**

As shown in Figure 2.5, firstly the front component accepts a request from a user. Next, the front component identifies the appropriate presentation component to forward the request. Further, the request is processed and the response is returned to the front Controller. Finally, the response is forwarded to the user.

Apart from the presentation and front components, Deployment Descriptor is an important part of Java EE 6 Web applications, as it helps in configuring the Web components. The following are the reasons of defining Deployment Descriptor:

- ❑ **Initializing parameters for servlets and Web applications**— Allow you to reduce the amount of coding required to define initializing parameters within the Web applications. For example, if a servlet requires access to a database, you can provide the login and password to access the database in Deployment Descriptor. In addition, you can configure the servlets and its initialization parameters used in a Web application in Deployment Descriptor.
- ❑ **Servlet/JSP definitions**—Include the name of the servlet or JSP, the class of the servlet or JSP, and the description of the servlet, which is optional. Each servlet/precompiled JSP used in a Web application should be defined in Deployment Descriptor.
- ❑ **Servlet/JSP mappings**—Provide information that is used by Web containers to map incoming requests to servlets and JSPs.
- ❑ **Specifying MIME types**—Allow you to specify the MIME types for each Web application in Deployment Descriptor.
- ❑ **Security**—Allows you to manage access control for an application by using Deployment Descriptor, for example, authentication and authorization details for users.

Let's now explore the structure or modules of Web applications.

## *Describing Structure /Modules of Web Applications*

As we have learned earlier, a Web application comprises static resource, such as files, images, Web components, helper classes, and libraries. The Web container enhances the capabilities of Web components and makes them easier to develop. In addition, the Web container provides the services, such as security and transaction management, for the Web components.

You should note that the process of creating, packaging, deploying, and running a Web application varies from the traditional standalone Java classes. The process to create, deploy, and execute a Web application can be summarized in the following steps:

- ❑ Develop the code for the Web component
- ❑ Develop Deployment Descriptor for a Web application
- ❑ Compile the Web components, along with the helper classes that are referenced by the components
- ❑ Package the Web application into a deployable unit
- ❑ Deploy the Web application on a Web container
- ❑ Access a URL referencing the Web application

In the Java EE 6 architecture, the Web resources are the Web components as well as files containing static Web content. The collection of Web resources creates a Web module, which is the smallest deployable unit. In addition to these, a Web module may contain the following files:

- ❑ **A public directory**—Contains tag files, which are implementations of Tag libraries
- ❑ **A WEB-INF/web.xml file**—Refers to the Web application Deployment Descriptor
- ❑ **A WEB-INF/classes directory**—Contains Server-side classes—servlets, utility classes, and JavaBeans components
- ❑ **A WEB-INF/lib directory**—Contains JAR Archives of libraries called by Server-side classes

The root of the application is the public area, excluding the WEB-INF directory. All the HTML files used in the application must be kept under this directory. Any files under the public area can be used by the Web container.

The application specific subdirectories, such as package directories, can be created under the /WEB-INF/classes location.

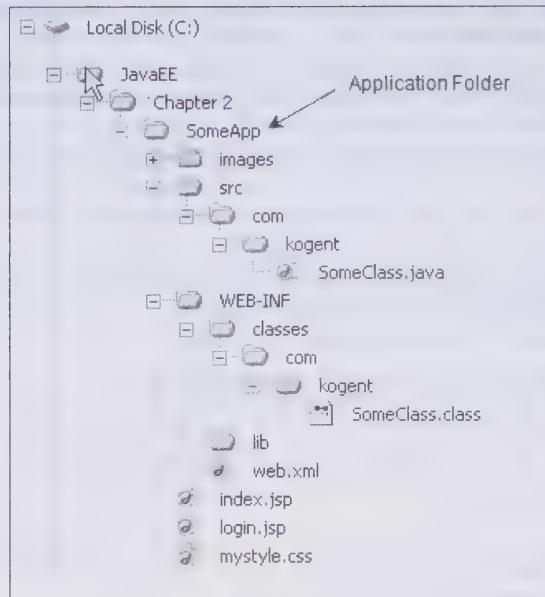
The Web application modules can be deployed and packaged in two ways:

- ❑ As an unpacked file structure
- ❑ As a JAR file, known as a WAR file

The use and composition of a WAR file is different from a JAR file. The WAR file is created by using a .war extension and can be deployed into any Web container that conforms to the Java Servlet specification.

You should note that a runtime Deployment Descriptor, web.xml, is included in the WAR file, and needs to be deployed on the application server. The web.xml file is located in the /WEB-INF folder of the application directory.

Figure 2.6 shows the common directory structure that has been followed throughout the book:



**Figure 2.6: Displaying the Directory Structure of a Web Application**

Figure 2.6 shows the structure of the Web application to be deployed in the Java EE 6 application server. The directory structure, defined in Figure 2.6, is considered as common directory structure throughout the book. To follow the structure, you need to create a folder named Java EE under any public area. Next, you need to create the folders on the basis of chapter numbers that contain chapter-specific applications.

In Figure 2.6, SomeApp is the application folder in which all the files of the application are placed. The SomeApp folder also contains some other folders, such as image, src and WEB-INF. The image folder stores all the image files that are used in the application. The src folder stores all the Java files used in the application. The WEB-INF folder is used to store the web.xml file, along with two sub-folders, classes and lib. The classes folder is used to store the .class files; whereas, the lib folder is used to store the executable JAR files required in the application. All the JSP and HTML pages, along with the WEB-INF directory, are stored in the root directory of an application. All the Java source files in the application are defined under a common package named com.kogent. After understanding the directory structure of the Java EE 6 applications, let's learn about the Web container, which serves as an interface to deploy Web applications.

## Describing Web Containers

A Web container refers to the platform that holds the Web components and provides platform-specific functionality to these components. To execute a Web component, you need to assemble it in the Web module and deploy it in the Web container. Alternatively, a Web container serves as a runtime environment for a Web application. The Web application runs within a Web container of a Web server. In addition, a Web container is used to provide Web components associated with the Web application.

Few Web servers may also be used to provide additional services, such as security, concurrency, transaction, and secondary storage. The EJB container is used to provide these additional services. A Web server does not need to be located on the same machine as the EJB server. Java EE 6 application components use the protocols and methods available in the container to access the functionality provided by the server.

The Java EE 6 application server supports the following types of Web containers:

- ❑ **Web containers in a Java EE 6 application server**—Refer to the built-in Web containers in Java EE 6 supported application servers, such as BEA's WebLogic server, Borland's Inprise application server, Netscape's iPlanet application server, and IBM's WebSphere application servers.
- ❑ **Web containers built into Web servers**—Refer to Web containers that are integrated with the Web server. Few examples of such containers are Java WebServer from Sun Microsystems and Jakarta Tomcat from Apache project.
- ❑ **Web container in a separate runtime**—Refers to Web servers and Web containers that are configured separately. The examples include Apache and Microsoft IIS, which require a separate Java runtime to actually run the servlets and Web server plug-in to integrate the Java runtime with the Web server. The communication between the Web server and Web container is managed by the plug-in.

Let's now learn about the various Web architecture models, such as Model-1 and Model-2 architectures.

## Exploring Web Architecture Models

Earlier, different programmers used to implement the same functionality in an application in different ways. Therefore, the logic and execution flow of an application used to vary from one programmer to another. This led to the problem of understanding the execution flow of the application, which was resolved with the introduction of development models. These models provide a standard way of flow control in an application and describe the technologies used in different parts of the application. A development model facilitates the design process by separating the code according to the functions performed by different parts of an application.

Two types of development models are used for Web applications in Java. These models are classified based on different approaches used to develop Web applications, which are:

- ❑ Model-1 architecture
- ❑ Model-2 architecture

Let's learn about these models in detail.

### Describing the Model-1 Architecture

The Model-1 architecture (Figure 2.7) was the first development model used to develop Web applications. This model uses JSP to design applications, which is responsible for all the activities and functionalities provided by the application. Applications using the Model-1 architecture contain a number of JSP pages, with each page providing different functionality and view to different users.

Figure 2.7 shows the Model-1 architecture:

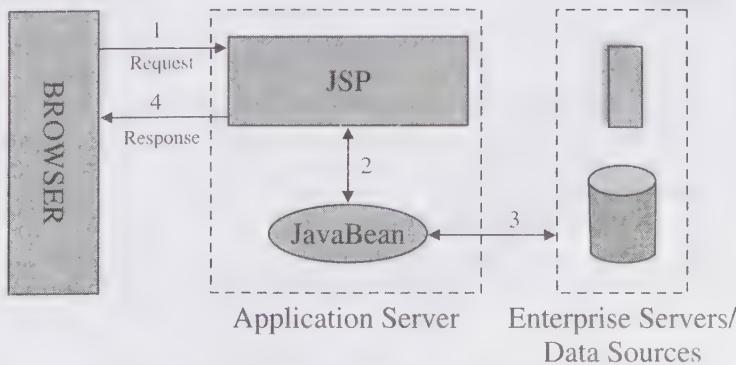


Figure 2.7: Displaying the Model-1 Architecture

In the Model-1 architecture (Figure 2.7), Web applications are developed by mixing the business and presentation logic. In this model, JSP pages receive HTTP requests, which are then transferred to the data layer by JavaBeans. After the requests are serviced, JSPs send HTTP responses back to the client. A JSP page not only contains display elements, but also retrieves HTTP parameters, calls the business logic, and handles the HTTP session.

The Model-1 architecture is page-centric and only suitable for small Web applications. Web applications implementing this type of architecture contain a series of JSP pages, where a user needs to navigate from one page to another. Consequently, the Model 1 architecture is not suitable for large Web applications because of these and some other limitations.

Despite its simple structure and easy to learn features, the Model-1 architecture was not successful for designing large projects because this model architecture:

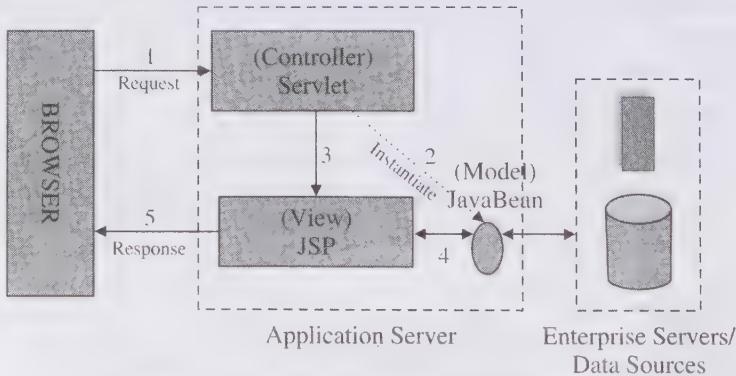
- Leads to inflexibility and difficulty in maintaining applications. A single change in one page may cause changes in other pages, leading to unexpected results.
- Involves the developer at both the page development and the business logic implementation stages. There is no provision for the division of labor between the page designer and the business logic developer.
- Increases the complexity of a program with the increase in the size of the JSP page; therefore, it is difficult to trace the flow of control and debug the program.
- Increases the effort required for maintaining JSP pages in an application.

Let's now understand the second type of architecture, that is, Model-2.

### *Describing the Model-2 Architecture*

The drawbacks in the Model-1 architecture led to the introduction of a new model, called Model-2. The Model-2 architecture was targeted at overcoming the drawbacks of Model-1 and helping developers to design more powerful Web applications. As it came after the advent of Model-1, it was named Model-2 to recognize it as a part of the Model-1 series.

Figure 2.8 shows the Model-2 architecture:



**Figure 2.8: Displaying the Model-2 Architecture.**

In the Model-2 architecture, the JSP and Java Servlet technologies were used together to develop a Web application. Servlets handle the control flow while JSPs handle HTML page creation. In due course, the approach of using JSPs and servlets together evolved as the Model-2 architecture. In this type of design model, the presentation logic is separated from the business logic.

So far, Model-2 is the most successful development model used to develop Web applications. It not only overcomes the limitations of the Model-1 architecture, but also provides new features that have their own advantages. Some of these advantages are as follows:

- Allows use of reusable software components to design business logic
- Offers great flexibility to the presentation logic, which can be modified without affecting the business logic

- Allows each software component to perform a different task, making it easy to design an application by simply embedding these components in the application

The Model-2 architecture resembles the classical MVC architecture. Let's now discuss the MVC architecture in detail next.

## Exploring the MVC Architecture

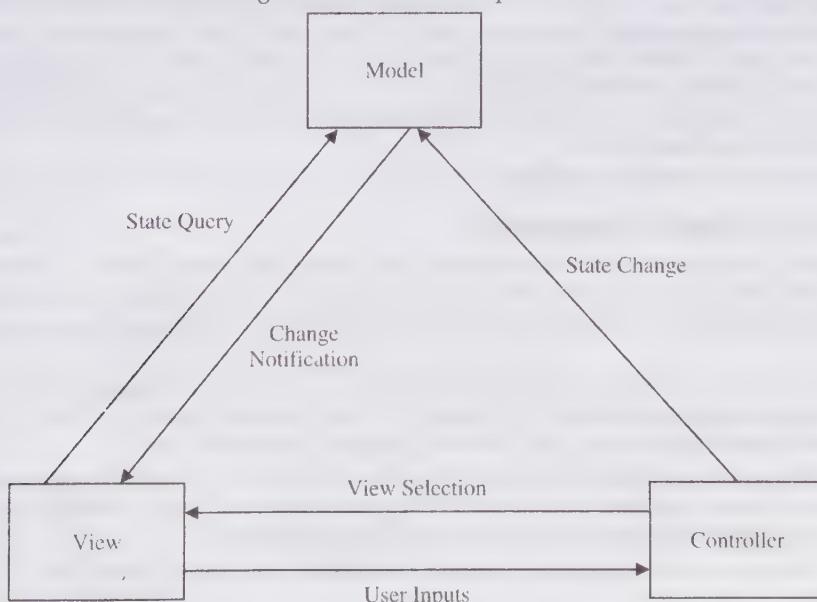
Today, a Web application may need to interact with different types of clients having different types of user interfaces. A Web application may need to represent an HTML view to a client, a Wireless Markup Language (WML) view to a wireless client, Java Foundation Classes (JFC)/Swing view to the administrator, and an XML view to some other type of client.

When a Web application needs to support only one type of client with one type of interface, it is a good idea to combine client-specific data and business logic with the interface-specific logic. However, if this type of approach is used for Web applications that require interaction with multiple types of clients, it would require different applications to support each type of client. This leads to the repetition of non-interface specific code in the interface-specific code, which further leads to increased effort in designing, debugging, and maintaining Web applications.

The solution to this problem is to use the MVC architecture while designing Web applications. The idea behind the MVC architecture is to separate the core business model functionality of the Web application from the presentation and control logic of the interface-specific application. With this arrangement, it is possible to have multiple views of the same data model of the Web application. Under MVC architecture, data (Model) and user interface (View) are separated from each other so that any change in the user interfaces does not affect the data in the database, and vice versa.

A Web application can ensure a coherent interaction with users by allowing the three components of MVC--Model, View, and Controller--to communicate with each other.

Figure 2.9 shows the interaction among the various MVC components:



**Figure 2.9: Displaying the Interaction among the MVC Architecture Components**

Figure 2.9 shows that View retrieves the data of an application by using the query methods of the Model. In addition, whenever a user sends a request, it passes through the Controller. Then, the Controller intercepts the request from the View and passes it to the Model for appropriate action. MVC architecture allows separation

of the business logic, data, and the presentation logic. In addition, Controller, which is a servlet, stores all the business logic. A View, normally a JSP page, contains all the code for presentation. A Model is implemented by using a pure Java or bean class. The different components in the application, such as Controller, Model, and View, are reusable and make applications highly scalable.

Let's learn more about each of the components of the MVC architecture next.

### **Describing the Model Component**

The Model component displays the data on which an application is based. In a Web application, JavaBeans hold the data needed by a Web application to process user queries. Events sent to Web Controller serves the basis for all modifications to the data. The Model component represents the data and the business logic of the application, and is not associated with the presentation logic of the application. The Model component does not provide the specification for data access logic. It performs its interfacing with other components by using a set of public methods.

An application may have many states that need to be stored somewhere. These states are encapsulated by the Model component of the application. All the functionalities and features of the application are provided by the Model component. The application behaves according to the business logic of the application, implemented by the Model component. Any change in the state of the Model component state is notified immediately; and this change is reflected in all Views.

### **Describing the View Component**

A View provides the Graphical User Interface (GUI) for Model. A user interacts with the application through View, which displays the information based on Model and allows the user to alter data. A Model can have multiple Views. The information provided by a Model has different meanings for different users and is interpreted by them in different ways. A View is responsible for displaying the information to users in a form that is understandable to them.

Model manages the database, the content of which can be changed and updated frequently. Any change in the database must be reflected to the user as soon as possible. Therefore, a View communicates with a Model to get information, if any change takes place in the database. Users who want to modify the content of a Model do not communicate with it directly. Instead, they communicate with the Model through a View, which communicates with the Controller regarding the user input. The Controller then makes the required changes in the Model and also notifies the other associated Views.

Let's now describe the Controller component.

### **Describing the Controller Component**

The Controller component controls all the Views associated with a Model. When a user interacts with the View component and tries to update the Model, the Controller component invokes various methods to update the Model. The Controller of an application also controls the data flow and transfers the requests between Model and View.

The behavior of a Web application is determined by the behavior of its various MVC components. The interaction among the various components is managed by the Controller, which, by controlling the flow among these components, determines the way the application performs the intended tasks.

Let's understand how the Controller performs its role when a user needs to change the data stored in a Model. If the user wishes to change this data, the user sends a request to the Controller, which consults the Model to update all the Views. The following steps are followed to change the data:

1. The user sends a request through an interface provided by the View, which passes the request to the Controller.
2. The Controller receives the input request.
3. The Controller processes the request according to the Controller logic, and if access to the Model is not required, the process moves to step 5.

4. The Model is accessed and modified, if required. It then needs to notify all the associated Views regarding the modification.
5. The View presents a user interface according to the modified or original Model, as the case may be.
6. The View remains idle after the current interaction and waits for the next interaction to begin.

This process is repeated again for every new request.

Let's now summarize the concepts described in the chapter.

## Summary

The chapter has discussed various methods of HTTP request, such as GET, POST, PUT, and DELETE. In addition, you have also learned about the meta information displayed to a user in the form of HTTP response. Further, various Web components of a Web application, such as JSP and servlets have been described. Moreover, the chapter has explored the different types of Web containers. Towards the end, you have learned about various Web application models and the MVC architecture that is used nowadays to develop a Web application.

The next chapter explains how Web applications can access data from relational databases using JDBC.

## Quick Revise

- Q1.** The two Web components are .....  
 A. Servlets and web.xml      B. JSP and web.xml  
 C. JSP and servlets      D. Servlets and Tag Libraries
- Ans. C
- Q2.** What is used to map an incoming request to servlet/JSP?  
 A. web.xml      B. JSP page  
 C. URL pattern      D. Servlets
- Ans. A
- Q3.** HTTP is a communication ..... used to transfer information on the Internet.  
 A. protocol      B. component  
 C. resource      D. request
- Ans. A
- Q4.** Which of the following status codes is displayed to represent the non existence of a resource on the Web server?  
 A. 404      B. 401  
 C. 500      D. 503
- Ans. A
- Q5.** Which of the following components provides services such as security, concurrency, and life-cycle management of the Web components in a Web application?  
 A. Web container      B. Web application  
 C. JSP      D. Servlets
- Ans. A
- Q6.** What is a Web container?  
 Ans. A container that manages the execution of JSP pages and servlets is known as a Web container. It serves as a runtime environment for a Web application.
- Q7.** What is web.xml?  
 Ans. The web.xml file is a Deployment Descriptor, which stores the configurations details of a Web application. The configuration detail includes details of all the servlets and filters, and the URL mapping for each of them.

**Q8. Define JSTL.**

Ans. JSTL is a set of tag libraries that help us in embedding logic into a JSP page without inserting any Java code in the page.

**Q9. Explain HTTP.**

Ans. HTTP is a stateless application-level communication protocol. It is a request-response protocol; i.e. an HTTP client requests for a particular service from the HTTP Server and the Server generates a response for that client.

**Q10. Explain the MVC architecture.**

Ans. The MVC architecture allows the separation of business logic, data, and presentation logic. In this architecture, the Controller, which is a servlet, stores all the Business logic. The View, normally a JSP page, contains all the code for presentation. The Model, or the data part, is implemented by using pure Java or bean classes. The different objects in the application, such as Model, View, and Controller, are reusable and make Web applications highly scalable.

# 3

# Working with JDBC 4.0

**If you need an information on:**

**See page:**

Introducing JDBC	54
Exploring JDBC Drivers	56
Exploring the Features of JDBC	61
Describing JDBC APIs	64
Exploring Major Classes and Interfaces	68
Exploring JDBC Processes with the <code>java.sql</code> Package	75
Exploring JDBC Processes with the <code>javax.sql</code> Package	127
Working with Transactions	144

Enterprise applications that are created using the Java EE technology need to interact with databases to store application-specific information. For example, search engines use databases to store information about the Web pages and job portals use databases to store information about the candidates and employers who access the Web sites to search and advertise jobs on the Internet. Interacting with database requires database connectivity, which can be achieved by using the Open Database Connectivity (ODBC) driver. This driver is used with Java Database Connectivity (JDBC) to interact with various types of databases, such as Oracle, MS Access, MySQL, and SQL Server. JDBC is an Application Programming Interface (API), which is used in Java programming to interact with databases. JDBC works with different database drivers to connect to different databases.

This chapter focuses on JDBC, which is used to provide database connectivity to enterprise applications. In this chapter, you first learn about JDBC drivers as well as the features of JDBC 3.0 and 4.0 versions. You also learn about the JDBC APIs that provide various classes and interfaces to develop a JDBC application. Next, the use of `java.sql` and `javax.sql` packages in JDBC implementation is described in detail. Towards the end, you learn to work with transactions in the JDBC application.

## Introducing JDBC

JDBC™ is a specification from Sun Microsystems that provides a standard abstraction (API / protocol) for Java applications to communicate with different databases. It is used to write programs required to access databases. JDBC, along with the database driver, is capable of accessing databases and spreadsheets. JDBC can also be defined as a platform-independent interface between a relational database and the Java programming language. The enterprise data stored in a relational database can be accessed with the help of JDBC APIs. The JDBC API allows Java programs to execute SQL statements and retrieve results. The classes and interfaces of JDBC allow a Java application to send requests made by users to the specified Database Management System (DBMS). Instead of allowing the drivers to target a specific database, the users can specify the name of the database used to retrieve the data.

The following are the characteristics of JDBC:

- Supports a wide level of portability.
- Provides Java interfaces that are compatible with Java applications. These providers are also responsible for providing the driver services.
- Provides higher level APIs for application programmers. The JDBC API specification is used as an interface for the application and DBMS.
- Provides JDBC API for Java applications. The JDBC call to a Java application is made by the SQL statements. These statements are responsible for the entire communication of the application with the database. The user can send any type of SQL queries as requests to a database.

## Components of JDBC

JDBC has four main components through which it can communicate with a database. These components are as follows:

- The JDBC API** – Provides various methods and interfaces for easy and effective communication with the databases. It also provides a standard to connect a database to a client application. The application-specific user processes the SQL commands according to his need and retrieves the result in the `ResultSet` object. The JDBC API provides two main packages, `java.sql`, and `javax.sql`, to interact with databases. These packages contain the Java SE and Java EE platforms, which conform to the write once run anywhere (WORA) capabilities of Java.
- The JDBC DriverManager** – Loads database-specific drivers in an application to establish a connection with the database. It is also used to select the most appropriate database-specific driver from the previously loaded drivers when a new connection to the database is established. In addition, it is used to make database-specific calls to the database to process the user requests.
- The JDBC test suite** – Evaluates the JDBC driver for its compatibility with Java EE. The JDBC test suite is used to test the operations being performed by JDBC drivers.

- ❑ The **JDBC-ODBC bridge**—Connects database drivers to the database. This bridge translates JDBC method calls to ODBC function calls, and is used to implement JDBC for any database for which an ODBC driver is available. The bridge for an application can be availed by importing the sun.jdbc.odbc package, which contains a native library to access the ODBC features.

## **JDBC Specification**

With the emergence of JDBC 4.0, various changes, such as support for Binary Large Object (BLOB) and Character Large Object (CLOB) have been introduced in JDBC API.

The specifications that are available in different versions of JDBC are as follows:

- ❑ **JDBC 1.0**—Provides basic functionality of JDBC.
- ❑ **JDBC 2.0**—Provides JDBC API in two sections, the JDBC 2.0 Core API and the JDBC 2.0 Optional Package API.
- ❑ **JDBC 3.0**—Provides classes and interfaces in two Java packages, java.sql and javax.sql. JDBC 3.0 is a combination of JDBC 2.1 core API and the JDBC 2.0 Optional Package API. The JDBC 3.0 specification provides performance optimization features and improves the features of connection pooling and statement.
- ❑ **JDBC 4.0**—Provides the following advance features:
  - Auto loading of the Driver interface
  - Connection management
  - ROWID data type support
  - Annotation in SQL queries
  - National Character Set Conversion Support
  - Enhancement to exception handling
  - Enhanced support for large objects

JDBC 4.0 is the new and advance specification used with Java EE 5 and the same version of JDBC is followed in Java EE 6.

## **JDBC Architecture**

A JDBC driver is required to process the SQL requests and generate results. JDBC API provides classes and interfaces to handle database-specific calls from users. Some of the important classes and interfaces defined in JDBC API are as follows:

- ❑ DriverManager
- ❑ Driver
- ❑ Connection
- ❑ Statement
- ❑ PreparedStatement
- ❑ CallableStatement
- ❑ ResultSet
- ❑ DatabaseMetaData
- ❑ ResultSetMetaData
- ❑ SqlData
- ❑ Blob
- ❑ Clob

The **DriverManager** in the JDBC API plays an important role in the JDBC architecture. It uses some database-specific drivers to effectively connect enterprise applications to databases.

Figure 3.1 demonstrates the simple JDBC architecture:

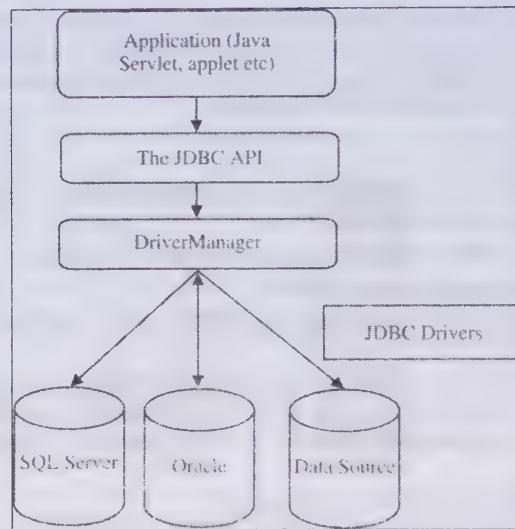


Figure 3.1: Displaying the Architecture of JDBC

As shown in Figure 3.1, the Java application that needs to communicate with a database has to be programmed using JDBC API. The JDBC driver (third-party vendor implementation) supporting data source, such as Oracle, and SQL, has to be added in the Java application for JDBC support, which can be done dynamically at run time. The dynamic plugging of the JDBC drivers ensures that the Java application is vendor independent. In other words, if you want to communicate with any data source through JDBC, you need a JDBC driver that intelligently communicates with the respective data source. Currently, there are more than 220 JDBC drivers available in the market, which are designed to communicate with different data sources.

Some of the available drivers are pure Java drivers and are portable for all the environments; whereas, others are partial Java drivers and require some libraries to communicate with the database. You need to understand the architectures of all the four types of drivers to decide which driver to use to communicate with the data source.

Let's now learn about the JDBC drivers in detail.

## Exploring JDBC Drivers

The different types of drivers available in JDBC are listed in Table 3.1:

Table 3.1: Types of JDBC Drivers

JDBC Driver Types	Description
Type-1 Driver	Refers to the Bridge Driver (JDBC-ODBC bridge)
Type-2 Driver	Refers to a Partly Java and Partly Native code driver (Native-API Partly Java driver)
Type-3 Driver	Refers to a pure Java driver that uses a middleware driver to connect to a database (Pure Java Driver for Database Middleware )
Type-4 Driver	Refers to a Pure Java driver (Pure), which is directly connected to a database

Now let's discuss each of these drivers in detail.

### Describing the Type-1 Driver

The Type-1 driver acts as a bridge between JDBC and other database connectivity mechanisms, such as ODBC. An example of this type of driver is the Sun JDBC-ODBC bridge driver, which provides access to the database through the ODBC drivers. This driver also helps the Java programmers to use JDBC and develop Java applications to communicate with existing data sources. This driver is included in the Java2 SDK within the

`sun.jdbc.odbc` package. This driver converts JDBC calls into ODBC calls and redirects the request to the ODBC driver. The architecture of the Type-1 driver is shown in Figure 3.2:

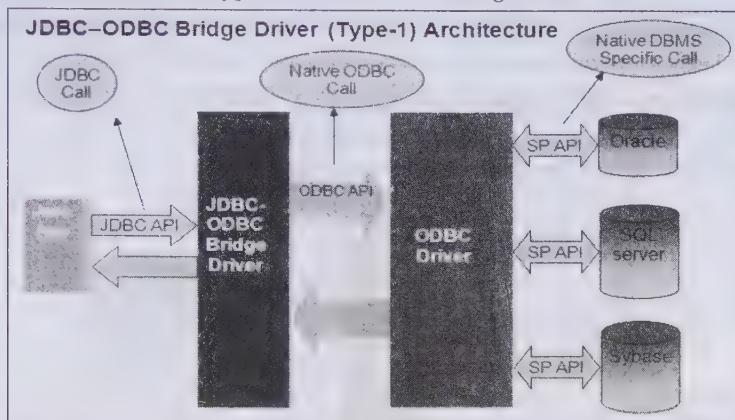


Figure 3.2: Displaying the Architecture of the JDBC Type-1 Driver

Figure 3.2 shows the architecture of the system that uses the JDBC-ODBC bridge driver to communicate with the respective database. In Figure 3.2, SP API refers to the APIs used to make a Native DBMS specific call. Figure 3.2 shows the following steps that are involved in establishing connection between a Java application and data source through the Type-1 driver:

1. The Java application makes the JDBC call to the JDBC-ODBC bridge driver to access a data source.
2. The JDBC-ODBC bridge driver resolves the JDBC call and makes an equivalent ODBC call to the ODBC driver.
3. The ODBC driver completes the request and sends responses to the JDBC-ODBC bridge driver.
4. The JDBC-ODBC bridge driver converts the response into JDBC standards and displays the result to the requesting Java application.

The Type-1 driver is generally used in the development and testing phases of Java applications.

### Advantages of the Type-1 Driver

Some advantages of the Type-1 driver are as follows:

- ❑ Represents single driver implementation to interact with different data stores
- ❑ Allows us to communicate with all the databases supported by the ODBC driver
- ❑ Represents a vendor independent driver

### Disadvantages of the Type-1 Driver

Some disadvantages of the Type-1 driver are as follows:

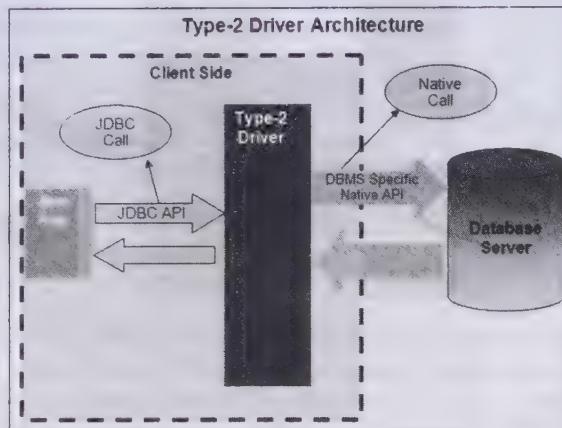
- ❑ Decreases the execution speed due to a large number of translations
- ❑ Depends on the ODBC driver; and therefore, Java applications also become indirectly dependent on ODBC drivers
- ❑ Requires the ODBC binary code or ODBC client library that must be installed on every client
- ❑ Uses Java Native Interface (JNI) to make ODBC calls

The preceding disadvantages make the Type-1 driver unsuitable for production environment and should be used only in case where no other driver is available. The Type-1 driver is also not recommended when Java applications are required with auto-installation applications, such as applets.

## Describing the Type-2 Driver (Java to Native API)

The JDBC call can be converted into the database vendor specific native call with the help of the Type-2 driver. In other words, this type of driver makes Java Native Interface (JNI) calls on database specific native client API. These database specific native client APIs are usually written in C and C++.

The Type-2 driver follows a 2-tier architecture model, as shown in Figure 3.3:



**Figure 3.3: Displaying the Architecture of the JDBC Type-2 Driver**

As shown in Figure 3.3, the Java application that needs to communicate with the database is programmed using JDBC API. These JDBC calls (programs written by using JDBC API) are converted into database specific native calls in the client machine and the request is then dispatched to the database specific native libraries. These native libraries present in the client are intelligent enough to send the request to the database server by using native protocol.

This type of driver is implemented for a specific database and usually delivered by a DBMS vendor. However, it is not mandatory that Type-2 drivers have to be implemented by DBMS vendors only. An example of Type-2 driver is the Weblogic driver implemented by BEA Weblogic. Type-2 drivers can be used with server-side applications. It is not recommended to use Type-2 drivers with client-side applications, since the database specific native libraries should be installed on the client machines.

### Advantages of the Type-2 Driver

Some advantages of the Type-2 driver are as follows:

- ❑ Helps to access the data faster as compared to other types of drivers
- ❑ Contains additional features provided by the specific database vendor, which are also supported by the JDBC specification

### Disadvantages of the Type-2 Driver

Some disadvantages of the Type-2 driver are as follows:

- ❑ Requires native libraries to be installed on client machines, since the conversion from JDBC calls to database specific native calls is done on client machines
- ❑ Executes the database specific native functions on the client JVM, implying that any bug in the Type-2 driver might crash the JVM
- ❑ Increases the cost of the application in case it is run on different platforms

### Examples of the Type-2 Driver

Some examples of the Type-2 driver are as follows:

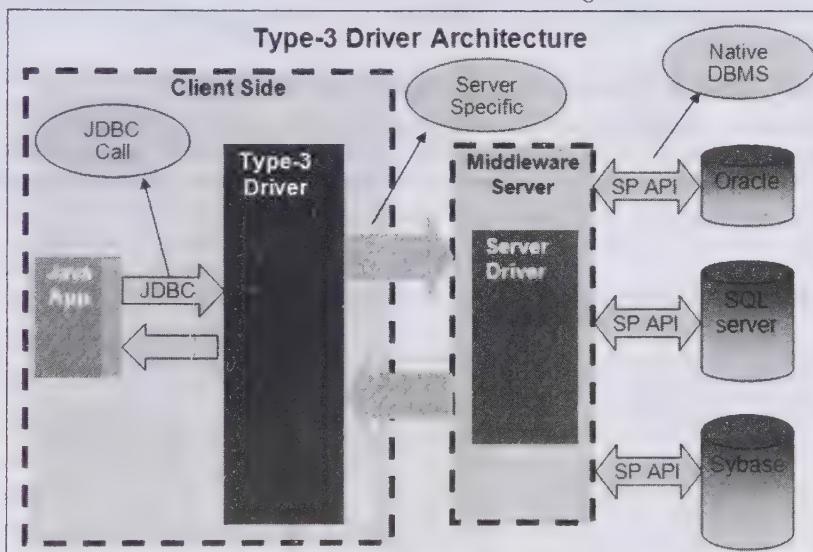
- ❑ **OCI (Oracle Call Interface) Driver** – Communicates with the Oracle database server. This driver converts JDBC calls into Oracle native library calls.

- ❑ **Weblogic OCI Driver for Oracle**—Makes JNI calls to Weblogic library functions. The Weblogic OCI driver for Oracle is similar to the Oracle OCI driver.
- ❑ **Type-2 Driver for Sybase**—Converts JDBC calls into Sybase dblib or ctlib calls, which are native libraries to connect to Sybase.

## Describing the Type-3 Driver (Java to Network Protocol/All Java Driver)

The Type-3 driver translates the JDBC calls into a database server independent and middleware server-specific calls. With the help of the middleware server, the translated JDBC calls are further translated into database server specific calls.

The Type-3 drivers follow the 3-tier architecture model, as shown in Figure 3.4:



**Figure 3.4: Displaying the Architecture of the JDBC Type-3 Driver**

As shown in Figure 3.4, a JDBC Type-3 driver listens for JDBC calls from the Java application and translates them into middleware server specific calls. After that, the driver communicates with the middleware server over a socket. The middleware server converts these calls into database specific calls. These types of drivers are also known as *net-protocol fully Java technology-enabled* or *net-protocol* drivers.

The middleware server can be added in an application with some additional functionality, such as pool management, performance improvement, and connection availability. These functionalities make the Type-3 driver architecture more useful in enterprise applications. Type-3 driver is recommended to be used with applets, since this type of driver is auto downloadable.

## Advantages of the Type-3 Drivers

Some advantages of the Type-3 driver are as follows:

- ❑ Serves as a all Java driver and is auto downloadable.
- ❑ Does not require any native library to be installed on the client machine.
- ❑ Ensures database independency, because a single driver provides accessibility to different types of databases.
- ❑ Does not provide the database details, such as username, password, and database server location, to the client. These details are automatically configured in the middleware server.
- ❑ Provides the facility to switch over from one database to another without changing the client-side driver classes. Switching of databases can be implemented by changing the configurations of the middleware server.

## Disadvantage of the Type-3 Driver

The main disadvantage of the Type-3 driver is that it performs the tasks slowly due to the increased number of network calls as compared to Type-2 drivers. In addition, the Type-3 driver is also costlier as compared to other drivers.

## Examples of the Type-3 Drivers

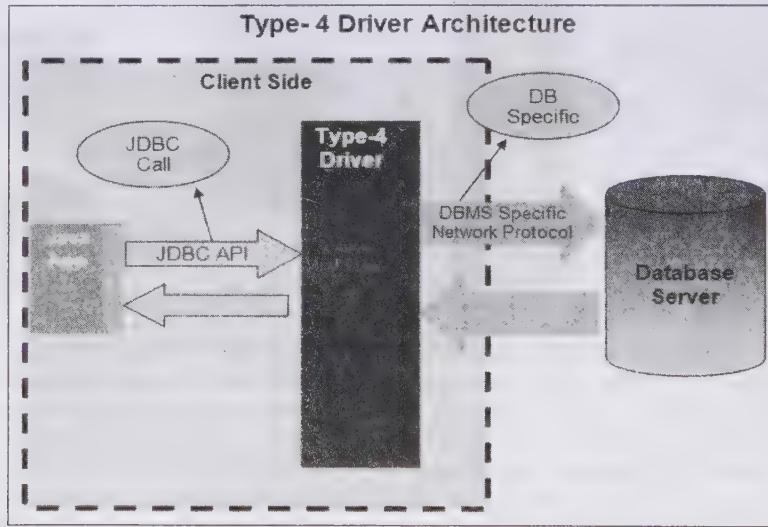
Some examples of the Type-3 driver are as follows:

- ❑ **IDS Driver**—Listens for JDBC calls and converts them into IDS Server specific network calls. The Type-3 driver communicates over a socket to IDS Server, which acts as a middleware server.
- ❑ **Weblogic RMI Driver**—Listens for JDBC calls and sends the requests from the client to the middleware server by using the RMI protocol. The middleware server uses a suitable JDBC driver to communicate with a database.

## Describing the Type-4 Driver (*Java to Database Protocol*)

The Type-4 driver is a pure Java driver, which implements the database protocol to interact directly with a database. This type of driver does not require any native database library to retrieve the records from the database. In addition, the Type-4 driver translates JDBC calls into database specific network calls.

The Type-4 drivers follow the 2-tier architecture model, as shown in Figure 3.5:



**Figure 3.5: Displaying the Architecture of the JDBC Type-4 Driver**

As shown in Figure 3.5, the Type-4 driver prepares a DBMS specific network message and then communicates with database server over a socket. This type of driver is lightweight and generally known as a thin driver. The Type-4 driver uses database specific proprietary protocols for communication. Generally, this type of driver is implemented by DBMS vendors, since the protocols used are proprietary.

You can use the Type-4 driver when you want an auto downloadable option for the client-side applications. In addition, it can be used with server-side applications.

## Advantages of the Type-4 Driver

Some advantages of the Type-4 driver are as follows:

- ❑ Serves as a pure Java driver and is auto downloadable
- ❑ Does not require any native library to be installed on the client machine
- ❑ Uses database server specific protocol
- ❑ Does not require a middleware server

## Disadvantage of the Type-4 Driver

The main disadvantage of the Type-4 driver is that it uses database specific proprietary protocol and is DBMS vendor dependent.

## Examples of the Type-4 Driver

Some examples of the Type-4 driver are:

- Thin Driver for Oracle from Oracle Corporation
- Weblogic and Mssqlserver4 for MS SQL server from BEA systems

## Exploring the Features of JDBC

JDBC 3.0 specification provides several features and procedures that can be used by Java database programmers. The core packages, along with the additional features, are present in the JDBC 3.0 version. Let's explore these features in detail next.

### Additional Features of JDBC 3.0

The features introduced in JDBC 3.0 are as follows:

- The JDBC metadata API**—Includes the instance of the ParameterMetaData interface to describe the parameter properties and their types used in the PreparedStatement interface.
- Named parameters**—Updates the CallableStatement object so that users can access the parameters by using the names rather than the indexes of the parameters.
- Changes to data types**—Include several new and modified data types. Few data type changes made in the JDBC 3.0 specification are:
  - **Large objects (BLOB, CLOB, and REF)**—Allow you to update the BLOB, CLOB, and REF type values in a database. Two new data types, BOOLEAN and DATALINK, have been introduced in JDBC 3.0.
  - **ResultSet values**—Updates the values of the ResultSet and ARRAY types available.
  - **New data types**—Include two new data types, java.sql.Types.DATALINK and java.sql.Types.BOOLEAN. These data types update the SQL data types with the same name. The DATALINK data type is capable of accessing the external resources; whereas, the BOOLEAN data type is equivalent to the BIT type. The value of the DATALINK data type is accessed by using the getURL() method, and the respective value of the boolean data type is accessed by using the getBoolean() method. These two methods take an instance of the ResultSet interface associated with the application.
  - **Access to the auto-generated keys**—Helps access the values of the auto-generated keys. You need to specify Statement.RETURN\_GENERATED\_KEYS or Statement.NO\_GENERATED\_KEYS in the execute() method to access the values of the auto-generated keys. The values for the auto-generated keys can be accessed in ResultSet. The ResultSet contains the values for the auto-generated keys and the getGeneratedKeys() instance method is used to access the values of the auto-generated keys.
- Connector relationship**—Maintains the connection between JDBC and J2EE (Java EE). The connector architecture provides a set of connectors through which the enterprise applications connect to JDBC. This connection provides a resource adapter, which is used to connect JDBC to remote systems. The JDBC API provides three main service providers to define the connector architecture, which are as follows:
  - **ConnectionPoolDataSource**—Refers to an interface provided by the JDBC API. The ConnectionPoolDataSource interface is used to connect the applications to JDBC DataSource and backend systems.
  - **XADatasource**—Refers to a feature of JDBC 2.0 API Optional package. XADatasource provides transactional support to enterprise applications for accessing the resources.
  - **Security Management**: Maintains the security mechanism for enterprise applications.
- ResultSet functionality**—Requires the programmer to close all the connections and results manually in JDBC programming. JDBC 3.0 supports the functionality of cursor holdability to ensure that the Connection

and ResultSet objects are closed. You need to maintain the following two constants to maintain the ResultSet holdability within an application:

- **HOLD\_CURSOR\_OVER\_COMMIT**—Ensures that ResultSet objects are open till a commit operation is performed
  - **CLOSE\_CURSOR\_AT\_COMMIT**—Ensures that ResultSet objects are closed after a commit operation is performed
- **Returning multiple results**—Refers to a feature of the JDBC 3.0 specification to provide the Statement interface, which can access multiple results simultaneously. The Statement interface includes a new method in JDBC API to access multiple results. The new method added to the JDBC API is an overloaded form of the `getMoreResults()` method. It includes an integer flag that is used to specify the behavior of ResultSets. The flags included in the JDBC API are as follows:
- **CLOSE\_ALL\_RESULTS**—Closes all the previously opened ResultSets by calling the `getMoreResults()` method
  - **CLOSE\_CURRENT\_RESULT**—Closes the current ResultSet object by calling the `getMoreResults()` method
  - **KEEP\_CURRENT\_RESULT**—Retains the current ResultSet object by using the `getMoreResult()` method
- **Connection pooling**: Allows you to maximize the performance of enterprise applications in the JDBC 3.0 specification.

Table 3.2 describes the properties of connection pooling:

<b>Table 3.2: Properties of Connection Pooling</b>	
<b>Property Name</b>	<b>Description</b>
maxStatements	Specifies the maximum number of statements that the connection pool can keep open
initialPoolSize	Specifies the number of physical connections that the pool should keep open while being initialized
minPoolSize	Specifies the minimum number of physical connections that can remain in the pool while it is being initialized
maxPoolSize	Specifies the maximum number of physical connections that can remain in the pool while it is being initialized
maxIdleTime	Specifies the time duration within which an unused pool should remain open prior to the closing of the connection
propertyCycle	Specifies the time interval, in seconds, that a pool should wait for the property policy

- **PreparedStatement pooling**—Allows you to compile the commonly used SQL statements to improve the performance of the statement. The PreparedStatement pooling is needed to increase the lifetime of the PreparedStatement object. The concept of the PreparedStatement pooling comes from the connection pooling mechanism.
- **Using Savepoints**—Add the most exciting features to JDBC 3.0 specifications. Transactions in a database ensure that the persisted data remains in a consistent state. However, sometimes the data of a current transaction might be rolled back. A Savepoint is an intermediate point within a transaction at which a transaction may be rolled back.

Now, let's discuss about the new features that have been added to JDBC 4.0.

## New Features in JDBC 4.0

Many new and advanced functionalities were introduced in JDBC 4.0. JDBC 4.0 includes the enhanced features of JDBC, which are mentioned as follows:

- **Auto loading of the JDBC driver class**—Provides auto loading of the JDBC drivers instead of loading them explicitly. In the previous versions of JDBC, you had to use the `Class.forName()` method to load the driver in a database. In JDBC 4.0, when the `getConnection()` method is called in an application, the `DriverManager` object automatically loads a driver in the database.
- **Connection management enhancement**—Allows the database programmers to establish a new connection by specifying the host name and an available port number. This can be done by using a set of parameters to maintain a standard connection. Connection management enhancement also adds some methods to the pre-existing interfaces, such as `Connection` and `Statement`.
- **Support for RowId**—Adds the `RowId` interface to the JDBC 4.0 specification to provide support for the `ROWID` data type. `RowId` is useful in tables where multiple columns do not have a unique identifier.
- **Dataset implementation of SQL using annotations**—Introduces the concept of annotation while using SQL, which ultimately results in fewer lines of code. The annotations are used along with the queries. The query results can be bound to the Java classes to speed up the processing of the query output. The JDBC 4.0 specification provides the following two main annotations:
  - **The SELECT annotation**—Retrieves query specific data from a database. You can use the `SELECT` annotation in a `SELECT` query within a Java class. The attributes of the `SELECT` annotation are described in Table 3.3:

**Table 3.3: Attributes of the SELECT Annotation**

Name	Type	Description
Sql	String	Specifies a simple SQL <code>SELECT</code> query.
Value	String	Represents the value specified for the <code>sql</code> attribute.
Table name	String	Specifies the name of the table created in a database.
Readonly, connected, scrollable	boolean	Indicates whether <code>DataSet</code> is <code>ReadOnly</code> or <code>Updatable</code> . It also indicates whether or not <code>DataSet</code> is connected to a back-end database. In addition, it indicates whether or not it is scrollable when the query is used in a connection.
allColumnsMapped	boolean	Indicates whether or not the column names used in the annotations are mapped to the corresponding fields in <code>DataSet</code> .

- **The UPDATE annotation**—Updates the queries used in database tables. The `UPDATE` annotation must include the `SQL annotation type` to update the fields of a table.
- **SQL exception handling enhancements**—Introduces certain enhancements to the `SQLException` class, which are as follows:
  - **New exception subclasses**—Provide new classes as enhancement to `SQLException`. The new classes that are added to the `SQLException` exception class include `SQL non-transient exception` and `SQL transient exception`. The `SQL non-transient exception` class is called when an already performed JDBC operation fails to run, unless the cause of the `SQLException` exception is corrected. On the other hand, the `SQL transient exception` class is called when a previously failed JDBC operation succeeds after retry.
  - **Casual relationships**—Support the Java SE chained exception mechanism by the `SQLException` class (also known as `Casual facility`). It allows handling multiple SQL exceptions raised in the JDBC operation.
  - **Support for the for-each loop**—Implements the chain of exceptions in a chain of groups by the `SQLException` class. The `for-each` loop is used to iterate on these groups.
  - **SQL XML support**—Introduces the concept of XML support in SQL DataStore. Some additional APIs have been added to JDBC 4.0 to provide this support.

## Describing JDBC APIs

JDBC API is a part of the JDBC specification and provides a standard abstraction to use JDBC drivers. The JDBC API provides classes and interfaces that are used by Java applications to communicate to databases. The JDBC driver communicates with a relational database for any requests made by a Java application by using the JDBC API. The JDBC driver not only processes the SQL commands, but also sends back the result of processing of these SQL commands. In addition, the JDBC API can be used to access the required data from all the database types, such as SQL Server, Sybase, and Oracle. A programmer does not need to write different programs to access the data from the database. The JDBC API satisfies the *write once and run anywhere* behavior of Java. Therefore, JDBC is used largely to access data from various data sources.

The JDBC API is based upon the X/Open Call Level Interface (CLI) specification and SQL standard statements. This is also the basic standard for ODBC. The JDBC API is a part of the Java Standard Edition (Java SE) of Java platform and is available to Java platform Enterprise Edition (Java EE) as well.

The JDBC 4.0 API specification is used to process and access data sources by using Java. The API includes drivers to be installed to access the different data sources. The API is used with SQL statements to read and write data from any data source in a tabular format. This facility to access data from the database is available through the `javax.sql.RowSet` interface. JDBC 4.0 API is mainly divided into the following two packages:

- ❑ `java.sql`
- ❑ `javax.sql`

These two packages are included in J2SE and are even available to the J2EE platform.

Now, let's discuss them in detail.

### *The `java.sql` Package*

The `java.sql` package is also known as the JDBC core API. This package includes the interfaces and methods to perform JDBC core operations, such as creating and executing SQL queries. The `java.sql` package consists of the interfaces and classes that need to be implemented in an application to access a database. The developer uses these operations to access the database in an application. The classes in the `java.sql` package can be classified into the following categories based on different operations:

- ❑ Connection management
- ❑ Database access
- ❑ Data types
- ❑ Database metadata
- ❑ Exceptions and warnings

Let's discuss these categories in detail.

### Connection Management

The connection management category contains the classes and interfaces used to establish a connection with a database.

Table 3.4 describes the classes and interfaces of the connection management category:

**Table 3.4: Classes and Interfaces of Connection Management**

Class/Interface	Description
<code>java.sql.Connection</code>	Creates a connection with a specific database. You can use SQL statements to retrieve the desired results within the context of a connection.
<code>java.sql.Driver</code>	Creates and registers an instance of a driver with the <code>DriverManager</code> interface.
<code>java.sql.DriverManager</code>	Provides the functionality to manage database drivers.
<code>java.sql.DriverPropertyInfo</code>	Retrieves the properties required to obtain a connection.
<code>java.sql.SQLPermission</code>	Sets up logging stream with <code>DriverManager</code> .

## Database Access

SQL queries are executed to access the application-specific data after a connection is established with a database. The interfaces listed in Table 3.5 allow you to send SQL statements to the database for execution and read the results from the respective database:

**Table 3.5: Interfaces of the Database Access Category**

Interface	Description
java.sql.CallableStatement	Executes stored procedures.
java.sql.PreparedStatement	Allows the programmer to create parameterized SQL statements.
java.sql.ResultSet	Abstracts the results of executing the SELECT statements. This interface provides methods to access the results row-by-row.
java.sql.Statement	Executes SQL statements over the underlying connection and access the results.
java.sql.Savepoint	Specifies a Savepoint in a transaction.

The `java.sql.PreparedStatement` and `java.sql.CallableStatement` interfaces extend the `java.sql.Statement` interface.

## Data Types

In the JDBC API, various interfaces and classes are defined to hold the specific types of data to be stored in a database. For example, to store the BLOB type values, the `Blob` interface is declared in the `java.sql` package.

Table 3.6 describes the classes and interfaces of various data types in the `java.sql` package:

**Table 3.6: Classes and Interfaces for Data Types in the `java.sql` Package**

Class/Interface	Description
java.sql.Array	Provides mapping for ARRAY of a collection.
java.sql.Blob	Provides mapping for the BLOB SQL type.
java.sql.Clob	Provides mapping for the CLOB SQL type.
java.SQLDate	Provides mapping for the SQL type DATE. Although, the <code>java.util.Date</code> class provides a general-purpose representation of date, the <code>java.sql.Date</code> class is preferable for representing dates in database-centric applications, as the type maps directly to SQL DATE type. Note that the <code>java.sql.Date</code> class extends the <code>java.util.Date</code> class.
java.sql.Nclob	Provides mapping of the Java language and the National Character Large Object types. The <code>Nclob</code> interface allows you to store the values of the character string up to the maximum length.
java.sql.Ref	Provides mapping for SQL type REF.
java.sql.RowId	Provides mapping for Java with the SQL RowId value.
java.sql.Struct	Provides mapping for the SQL structured types.
java.sql.SQLXML	Provides mapping for the SQL XML types available in the JDBC API.
java.sql.Time	Provides mapping for the SQL type TIME, and extends the <code>java.util.Date</code> class.
java.sql.Timestamp	Provides mapping for the SQL type TIME and extends the <code>java.util.Date</code> class.
java.sql.Types	Holds a set of constant integers, each corresponding to a SQL type.

In addition to the data types mentioned in Table 3.6, the JDBC API provides certain user-defined data types (UDT) available in JDBC API. The UDTs available in the `java.sql` package are listed in Table 3.7:

**Table 3.7: Classes and Interfaces for UDT in the java.sql Package**

Class/Interface	Description
java.sql.SQLData	Provides a mapping between the SQL UDTs and a specific class in Java.
java.sql.SQLInput	Provides methods to read the UDT attributes from a specific input stream. The input stream contains a stream of values depicting the instance of the SQL structured or SQL distinct type.
java.sql.SQLOutput	Writes the attributes of the output stream back to the database.

JDBC API also provides some default data types that are associated with a database. The default types include the DISTINCT and DATALINK types. The DISTINCT data type maps to the base type to which the base type value is mapped. For example, a DISTINCT value based on a SQL NUMERIC type maps to a java.math.BigDecimal type. A DATALINK type always represents a java.net.URL object of the URL class defined in the java.net package.

## Database Metadata

The metadata interface is used to retrieve information about the database used in an application. JDBC API provides certain interfaces to access the information about the database used in the application. These metadata interfaces are described in Table 3.8:

**Table 3.8: Classes and Interfaces of Database MetaData**

Class/Interface	Description
java.sql.DatabaseMetaData	Obtains the database features. This interface is used by driver vendors to ensure that a user is aware of the capabilities of a database and the JDBC driver used along with the database.
java.sql.ParameterMetaData	Allows access to the database types of parameters in prepared statements.
java.sql.ResultSetMetaData	Provides methods to access metadata of ResultSet, such as the names of columns, their types, the corresponding table names, and other properties.

## Exceptions and Warnings

JDBC API provides classes and interfaces to handle the unwanted exceptions raised in an application. The API also provides classes to handle warnings related to an application.

Table 3.9 describes the classes for exception handling:

**Table 3.9: Classes for Exception Handling**

Classes	Description
java.sql.BatchUpdateException	Updates batches.
java.sql.DataTruncation	Identifies data truncation errors. Note that data types do not always match between Java and SQL.
java.sql.SQLException	Represents all JDBC-related exception conditions. This exception also embeds all driver and database-level exceptions and error codes.
java.sql.SQLWarning	Represents database access warnings. Instead of catching the SQLWarning exception, you can use the appropriate methods on java.sql.Connection, java.sql.Statement, and java.sql.ResultSet to access the warnings.

Let's now briefly discuss the JDBC extension APIs (javax.sql) available in JDBC API.

## The *javax.sql* Package

The javax.sql package is also called as the JDBC extension API, and provides classes and interfaces to access server-side data sources and process Java programs. The JDBC extension package supplements the java.sql package, and provides the following support:

- DataSource**
- Connection and statement pooling
- Distributed transaction
- Rowsets

## DataSource

The `java.sql.DataSource` interface represents the data sources related to the Java application.

Table 3.10 describes the interfaces of the `DataSource` interfaces provided by the `javax.sql` package:

**Table 3.10: Interfaces for DataSource**

Interface	Description
<code>javax.sql.DataSource</code>	Represents the <code>DataSource</code> interface used in an application
<code>javax.sql.CommonDataSource</code>	Provides the methods that are common between the <code>DataSource</code> , <code>XADatasource</code> and <code>ConnectionPoolDataSource</code> interfaces

## Connection and Statement Pooling

The connections made by using the `DataSource` objects are implemented on the middle-tier connection pool. As a result, the functionality to create new database connections is improved. The classes and interfaces available for connection pooling in the `javax.sql` package are listed in Table 3.11:

**Table 3.11: Classes and Interfaces for Connection Pooling**

Class/Interface	Description
<code>javax.sql.ConnectionPoolDataSource</code>	Provides a factory for the <code>PooledConnection</code> objects.
<code>javax.sql.PooledConnection</code>	Provides an object to manage connection pools.
<code>javax.sql.ConnectionEvent</code>	Provides an <code>Event</code> object, which offers information about the occurrence of an event.
<code>javax.sql.ConnectionEventListener</code>	Provides objects used to register the events generated by the <code>PooledConnection</code> object.
<code>javax.sql.StatementEvent</code>	Represents the <code>StatementEvents</code> interface associated with the events that occur in the <code>PooledConnection</code> interface. The <code>StatementEvents</code> interface is then sent to the <code>StatementEventListeners</code> instance, which is registered with the instance of the <code>PooledConnection</code> interface.
<code>javax.sql.StatementEventListener</code>	Provides an object that registers the event with an instance of <code>PooledConnection</code> interface.

## Distributed Transaction

The distributed transaction mechanism allows an application to use the data sources on multiple servers in a single transaction. JDBC API provides certain classes and interfaces to handle distributed transactions over the middle-tier architecture, as listed in Table 3.12:

**Table 3.12: Classes and Interfaces for Distributed Transaction**

Class/Interface	Description
<code>javax.sql.XAConnection</code>	Provides the object that supports distributed transaction over middle-tier architecture
<code>javax.sql.XADatasource</code>	Provides a factory for the <code>XAConnection</code> objects

## Rowsets Object

A `RowSet` object is used to retrieve data in a network. In addition, the `RowSet` object is able to transmit data over a network. JDBC API provides the `RowSet` interface, with its numerous classes and interfaces, to work with tabular data, as described in Table 3.13:

**Table 3.13: Classes and Interfaces for RowSet**

Class/Interface	Description
javax.sql.RowSetListener	Receives notification from the RowSet object on the occurrence of an event.
javax.sql.RowSetEvent	Provides the event object, which is generated on the occurrence of an event on the RowSet object
javax.sql.RowSetMetaData	Provides information about the RowSet object associated with a database
javax.sql.RowSetReader	Populates disconnected RowSet objects with rows of data
javax.sql.RowSetWriter	Implements the RowSetWriter object, which is also called RowSet writer
javax.sql.RowSet	Retrieves data in a tabular format

## Exploring Major Classes and Interfaces

You have already learned about the classes and interfaces of the `java.sql` and `javax.sql` packages. Among these classes and interfaces discussed in the preceding sections, some noteworthy classes and interfaces play an important role in providing JDBC implementations in a Java application, which we explore in this section. You can establish a database connection by using the classes and interfaces of JDBC, such as `DriverManager` and `Driver`. These classes and interfaces allow you to load a driver, create a connection, and retrieve or update data in a database.

Let's explore the following major classes and interfaces in detail:

- ❑ The `DriverManager` class
- ❑ The `Driver` interface
- ❑ The `Connection` interface
- ❑ The `Statement` interface

### *The `DriverManager` Class*

`DriverManager` is a non-abstract class in JDBC API. It contains only one constructor, which is declared private to imply that this class cannot be inherited or initialized directly. All the methods and properties of this class are declared as static. The `DriverManager` class performs the following main responsibilities:

- ❑ Maintains a list of `DriverInfo` objects, where each `DriverInfo` object holds one `Driver` implementation class object and its name
- ❑ Prepares a connection using the `Driver` implementation that accepts the given JDBC URL

Table 3.14 describes the methods of the `DriverManager` class:

**Table 3.14: Methods of the `DriverManager` Class**

Method	Description
<code>public static void deregisterDriver(Driver driver) throws SQLException</code>	Drops a driver from the list of drivers maintained by the <code>DriverManager</code> class.
<code>public static Connection getConnection(String url)</code>	Establishes a connection of a driver with a database. The <code>DriverManager</code> class selects a driver from the list of drivers and creates the connection.
<code>getConnection(String url, Properties info)</code>	Establishes a connection of a driver with a database on the basis of the URL and info passed as parameters. URL is used to load the selected driver for a database. The info parameter provides information about the string/value tags used in the connection.
<code>getConnection(String url, String username, String password)</code>	Establishes a connection of a driver with a database. The <code>DriverManager</code> class selects a driver from the list of drivers and creates the connection. Along with URL, it takes two more parameters, <code>username</code> and <code>password</code> . The <code>username</code> parameter specifies the user for which the connection is

**Table 3.14: Methods of the DriverManager Class**

<b>Method</b>	<b>Description</b>
public static driver getDriver(String url)	being made, and the password parameter represents the password of the user.
public static enumeration getDrivers()	Locates the requested driver in the DriverManager class. The url parameter specifies the URL of the requested driver.
public static int getLoginTimeout()	Accesses a list of drivers present in a database.
public static getLogStream()	Specifies the maximum time a driver needs to wait to log on to a database.
public static getLogWriter()	Returns the logging or tracing PrintStream object.
public static void println(String message)	Returns the log writer.
public static void registerDriver(Driver driver)	Prints a message used in a log stream.
public static void setLoginTimeout(int seconds)	Registers a requested driver with the DriverManager class.
public static void setLogStream(PrintStream out)	Sets the maximum time that a driver needs to wait while attempting to connect to a database.
public static void setLogWriter(PrintWriter out)	Sets the logging or tracing PrintStream object.
public static void setLogWriter(PrintWriter out)	Sets the logging or tracing PrintWriter object.

## The Driver Interface

The `Driver` interface is used to create connection objects that provide an entry point for database connectivity. Generally, all drivers provide the `DriverManager` class that implements the `Driver` interface and helps to load the driver in a JDBC application. The drivers are loaded for any given connection request with the help of the `DriverManager` class. After the `Driver` class is loaded, its instance is created and registered with the `DriverManager` class.

Table 3.15 describes all the methods provided in the `Driver` interface:

**Table 3.15: Methods of the Driver Interface**

<b>Methods</b>	<b>Description</b>
public boolean acceptsURL(String url)	Checks whether the format of the given URL is according to the driver or not. In other words, it checks the subprotocol and extra information of the URL.
public connection connect(String url, Properties info)	Establishes connectivity with a database. The url parameter specifies the JDBC URL that describes the database details to which the driver is to be connected. The info parameter specifies the information of the tag/value pair used in the driver.
public int getMajorVersion()	Accesses the major version number of the driver.
public int getMinorVersion()	Retrieves the minor version number of the driver.
public DriverPropertyInfo[] getPropertyInfo(String url, Properties info)	Retrieves the properties of the driver included in a database.
public boolean jdbcCompliant()	Determines whether the driver is JDBC compliant or not. The true value of the boolean data type represents that the driver is JDBC compliant; else, this method returns false.

## The Connection Interface

The `Connection` interface is a standard type that defines an abstraction to access the session established with a database server. JDBC driver provider must implement the `Connection` interface. The `Connection` type of object (an instance of the class that implements the `Connection` interface) represents the session established with the data store.

The `Connection` interface provides methods to handle the `Connection` object.

Table 3.16 describes the methods present in the `Connection` interface:

**Table 3.16: Methods of the Connection Interface**

Methods	Description
<code>public void clearWarnings() throws SQLException</code>	Clears all the warnings for a <code>Connection</code> object. This method throws the <code>SQLException</code> exception when an error occurs.
<code>public void close() throws SQLException</code>	Closes a connection and releases the connection object associated with the connected database. It also releases the JDBC resources associated with the connection.
<code>public void commit() throws SQLException</code>	Commits the changes made in the previous commit/rollback and releases any database locks held by the current <code>Connection</code> object.
<code>public Statement createStatement() throws SQLException</code>	Creates the <code>Statement</code> object to send SQL statements to the specified database. This method takes no argument; therefore, it can be executed by using the <code>Statement</code> object.
<code>public Statement createStatement(int resultSetType, int resultSetConcurrency)</code>	Creates a <code>Statement</code> object, which is used to load the SQL statements to the specified database. The <code>ResultSet</code> object generated by this <code>Statement</code> object is of the mentioned type and concurrency.
<code>public Statement createStatement(int resultSetType, int resultSetConcurrency, int resultSetHoldability)</code>	Creates an object with the mentioned type, concurrency, and holdability.
<code>public boolean getAutoCommit()</code>	Retrieves the auto-commit mode for the current <code>Connection</code> object.
<code>public String getCatalog()</code>	Gets the name of the current catalog used in the current <code>Connection</code> object.
<code>public int getHoldability()</code>	Gets the current holdability of the <code>ResultSet</code> object created by using a <code>Connection</code> object.
<code>public DatabaseMetaData getMetaData()</code>	Gets the <code>DatabaseMetaData</code> object containing the metadata information. You should ensure that the database must be connected with a connection object.
<code>public int getTransactionIsolation()</code>	Provides the transaction isolation level of the connection object related to a database.
<code>public Map getTypeMap()</code>	Gets a map object related to a connection object.
<code>public SQLWarning getWarnings()</code>	Retrieves any warning associated with a connection object.
<code>public boolean isClosed()</code>	Specifies whether or not a database connection object is closed.
<code>public boolean isReadOnly()</code>	Specifies whether or not a connection object is read-only.
<code>public String nativeSQL(String sql)</code>	Allows you to convert the SQL statements passed to the connection object into the systems native SQL grammar.
<code>public CallableStatement prepareCall(String sql)</code>	Creates a <code>CallableStatement</code> object to call database stored procedures.
<code>public CallableStatement prepareCall(String sql, int resultSetType, int resultSetConcurrency)</code>	Creates a <code>CallableStatement</code> object that generates the <code>ResultSet</code> object of the specified type and concurrency.

**Table 3.16: Methods of the Connection Interface**

<b>Methods</b>	<b>Description</b>
public CallableStatement prepareCall(String sql, int resultSetType, int resultSetConcurrency, int resultSetHoldability)	Creates a CallableStatement object that generates the ResultSet object of the specified type, concurrency, and holdability.
public PreparedStatement prepareStatement(String sql)	Creates a PreparedStatement object to send the SQL statements over a connection.
public PreparedStatement prepareStatement(String sql, int autoGeneratedKeys)	Creates a PreparedStatement object that retrieves auto-generated keys.
public PreparedStatement prepareStatement(String sql, int[] columnIndexes)	Creates a PreparedStatement object that retrieves auto-generated keys by using a given array.
public PreparedStatement prepareStatement(String sql, int resultSetType, int resultSetConcurrency)	Generates a PreparedStatement object that generates the ResultSet object with the given type and concurrency.
public PreparedStatement prepareStatement(String sql, int resultSetType, int resultSetConcurrency, int resultSetHoldability)	Generates a PreparedStatement object that generates the ResultSet object with the given type, concurrency, and holdability.
public PreparedStatement prepareStatement(String sql, String[] columnNames)	Creates a PreparedStatement object that retrieves the auto-generated keys. The columnNames parameter of PreparedStatement is an array containing the names of the columns that contain the auto-generated keys in the target table.
public void releaseSavepoint(Savepoint savepoint)	Releases the savepoint associated with the connection object of the current transaction.
public void rollback()	Rolls back all the transactions and releases any database locks that are currently done by the connection object.
public void rollback(Savepoint savepoint)	Removes all the changes made by the connection object after a savepoint object is created.
public void setAutoCommit(boolean autoCommit)	Sets the current transaction to the connections auto-commit mode.
public void setCatalog(String catalog)	Sets the given catalog name for current Connection object's database.
public void setHoldability(int holdability)	Changes the holdability of the current connection object.
public void setReadOnly(boolean readOnly)	Sets the connection to the read-only mode to optimize the specified database.
public Savepoint setSavepoint()	Creates an unnamed savepoint in the current transaction and returns the savepoint associated with the previous transactions.
public Savepoint setSavepoint(String name)	Creates a savepoint with the name specified in the current transaction. It returns the new savepoint object.
public void setTransactionIsolation(int level)	Checks the transaction isolation level of the specified connection object.
public void setTypeMap(Map map)	Installs the TypeMap object as the current type map for the current connection.

The `Connection` interface also provides certain constants that can be used to handle connection transactions. Table 3.17 describes the constants available in the `Connection` interface:

**Table 3.17: Constants of the Connection Interface**

<b>Constants</b>	<b>Description</b>
public static final int TRANSACTION_NONE	Indicates that connection transactions are not supported in the current transaction object.
public static final int TRANSACTION_READ_COMMITTED	Prevents a transaction from reading a row with uncommitted changes. It is only used to read non-repeatable rows in a table.
public static final int TRANSACTION_READ_UNCOMMITTED	Indicates that non-repeatable and phantom reads are allowed in a transaction. It allows a row to be changed during a transaction. The changed row can be read by other transactions before the changes in the row are committed.
public static final int TRANSACTION_REPEATABLE_READ	Prevents non-repeatable reads and simultaneous transactions in a single row.
public static final int TRANSACTION_SERIALIZABLE	Prevents reading non-repeatable rows in a table.

Let's now learn about the Statement interface.

## The Statement Interface

The `Statement` interface defines a standard abstraction to execute the SQL statements requested by a user and return the results by using the `ResultSet` object. The `Statement` object contains a single `ResultSet` object at a time. It is possible that the data reading done with the help of one `ResultSet` object is interleaved with the reading done by the other. In such a case, each `ResultSet` object must be generated by different `Statement` objects. The `execute()` method of all the statements implicitly closes the current `ResultSet` object (if it is open) of a statement. The `Statement` interface provides specific methods to execute and retrieve the results from a database. The `PreparedStatement` interface provides the methods to deal with the `IN` parameters; whereas, the `CallableStatement` interface provides methods to deal with the `IN` and `OUT` parameters.

The `Statement` interface also provides certain methods that are used with a database. These methods are described in Table 3.18:

**Table 3.18: Methods of the Statement Interface**

<b>Methods</b>	<b>Description</b>
public void addBatch(String sql)	Adds the SQL commands to the existing list of commands for the <code>Statement</code> object. These commands are executed in a batch by calling the <code>executeBatch()</code> method.
public void cancel()	Cancels the statement, if the data sources do not support the statement.
public void clearBatch()	Clears all the commands listed in the batch of the <code>Statement</code> interface.
public void clearWarnings()	Clears the warnings that are generated on the <code>Statement</code> object. You should note that after the execution of the <code>clearWarnings()</code> method, the <code>getWarnings()</code> method returns null, provided a new warning is not generated for this <code>Statement</code> object.
public void close()	Closes the <code>Statement</code> object. Therefore, it releases its control from the database and connection.
public boolean execute(String sql)	Executes the SQL commands that may return multiple result sets along with one or more update counts.
public boolean execute(String sql, int autoGeneratedKeys)	Executes the SQL commands that may return multiple result sets along with one or more update counts. It also indicates whether a driver or the auto-generated keys are available for the retrieval.

**Table 3.18: Methods of the Statement Interface**

Methods	Description
public boolean execute(String sql, int[] columnIndexes)	Executes the SQL commands that may return multiple result sets along with one or more update counts. It also indicates the driver about the availability of the auto-generated keys in an array. The array contains the list of the indexes and the tables containing the auto-generated keys.
public boolean execute(String sql, String[] columnNames)	Executes the SQL commands that may return multiple result sets along with one or more update counts. It also indicates the driver about the availability of the auto-generated keys in an array. The array contains the name of the columns in the target table that contains the auto-generated keys.
public int[] executeBatch()	Executes the SQL commands in a batch. The method returns the update count as an integer greater or equal to 0 after the successful execution of the batch statements. The integer array is used to represent the array of the SQL commands listed in the batch.
public ResultSet executeQuery(String sql)	Executes a SQL command and returns a single ResultSet.
public int executeUpdate(String sql)	Executes the SQL Data Definition Language (DDL) statements, such as INSERT, UPDATE, and DELETE.
public int executeUpdate(String sql, int autoGeneratedKeys)	Executes the SQL statements and notifies the driver about the availability of the auto-generated keys. The auto-generated keys are helpful to retrieve data from the database.
public int executeUpdate(String sql, int[] columnIndexes)	Executes the SQL statements on the basis of the SQL query and column index passed as an argument. This method also notifies the driver about the availability of the auto-generated keys. The auto-generated keys are helpful to retrieve data from the database. The array index of the auto-generated keys indicates the indexes and tables that contain the auto-generated keys.
public int executeUpdate(String sql, String[] columnNames)	Executes the SQL statements and notifies the driver about the availability of the auto-generated keys. These keys are responsible for data retrieval from the database. The array index of the auto-generated keys indicates the columns of the target table that contains the auto-generated keys.
public Connection getConnection()	Retrieves an object of Connection type, which is used to maintain the connection of a Java application with a database.
public int getFetchDirection()	Retrieves the direction of the rows from the database tables that are generated from the ResultSet object. The fetch direction for a Statement object can be set with the help of the setFetchDirection() method. If the fetch direction is not set, the fetch direction is implementation specific.
public int getFetchSize()	Gets the number of rows of default fetch size from the current ResultSet object.
public ResultSet getGeneratedKeys()	Gets the auto-generated keys created by executing the Statement object.
public int getMaxFieldSize()	Gets the maximum number of bytes that can be returned for the column values.
public int getMaxRows()	Provides the maximum number of rows in a ResultSet produced by the Statement object.
public boolean getMoreResults()	Navigates to the next result in the ResultSet object. It is also used to close the currently opened result set.
public int getMoreResults(int current)	Navigates to the next result in the object of the statement. It deals with the ResultSet object by using the instructions specified in the given flag.

**Table 3.18: Methods of the Statement Interface**

<b>Methods</b>	<b>Description</b>
public int getQueryTimeout()	Provides the number of seconds the driver has to wait to execute the statements.
public ResultSet getResultSet()	Gets the current ResultSet object generated by the Statement object.
public int getResultSetConcurrency()	Gets the concurrency of the ResultSet object generated by the Statement object.
public int getResultSetHoldability()	Gets the holdability of the ResultSet object generated by the Statement object.
public int getResultSetType()	Retrieves the result set type for the ResultSet object.
public int getUpdateCount()	Retrieves the current result set as an update count. The value returned by this method is either a positive or negative value, indicating the number of records that have been updated in a result set.
public SQLWarning getWarnings()	Gets the warnings generated on the Statement object.
public void setCursorName(String name)	Sets the cursor name to the given string. The cursor name is used by the Statement objects to execute this method.
public void setEscapeProcessing(boolean enable)	Sets the escape processing on or off.
public void setFetchDirection(int direction)	Sets the direction for the driver to process the rows in the ResultSet object.
public void setFetchSize(int rows)	Sets the number of rows that should be fetched from the database.
public void setMaxFieldSize(int max)	Sets the maximum number of bytes for the ResultSet object to store binary values.
public void setMaxRows(int max)	Sets the maximum number of rows that a ResultSet can contain.
public void setQueryTimeout(int seconds)	Sets the number of seconds a driver needs to wait for executing the Statement object.

The Statement interface also comprises few constants. Table 3.19 describes the constants available in the Statement interface:

**Table 3.19: Constants of Statement Interface**

<b>Constants</b>	<b>Description</b>
public static final int CLOSE_ALL_RESULTS	Closes all the open ResultSet objects. All the ResultSet objects should be closed before calling the getMoreResults() method.
public static final int CLOSE_CURRENT_RESULT	Indicates that the current ResultSet connected with the specified database must be closed before calling the getMoreResults() method.
public static final int EXECUTE_FAILED	Indicates the occurrence of errors while executing a batch statement.
public static final int KEEP_CURRENT_RESULT	Indicates that the current Resultset should not be closed before calling the getMoreResults() method.
public static final int NO_GENERATED_KEYS	Indicates that the generated keys should not be made available for retrieval.
public static final int RETURN_GENERATED_KEYS	Indicates that the generated keys should be made available for retrieval.
public static final int SUCCESS_NO_INFO	Indicates that a batch statement has been executed successfully.

Table 3.19 shows all the required fields in the Statement interface. These are used by a database to communicate with an application.

The Statement object is created after the connection to the specified database is made. This object is created by using the `createStatement()` method of the Connection interface, as shown in the following code snippet:

```
Connection con = DriverManager.getConnection (url, "username", "password");
Statement stmt = con.createStatement();
```

Now let's discuss how the `java.sql` package is used to implement database connectivity in an application.

## Exploring JDBC Processes with the `java.sql` Package

The `java.sql` package is used by a Java application to communicate with a database. The JDBC application-specific code should be written within an application that has to communicate with the database. There are some basic steps to use JDBC in a Java application. Let's now discuss the basic steps involved in using JDBC in an application. The following heads help you to understand how JDBC implementations are provided in a Java application by using the `java.sql` package:

- Basic JDBC steps
- Simple JDBC application
- PreparedStatement interface
- CallableStatement interface
- ResultSets
- Batch updates
- Advance data types

Now, let's discuss each of them in detail.

### Understanding Basic JDBC Steps

To establish a connection with a database and retrieve the desired results, you need to perform various steps. For example, you need to register a driver with the `DriverManager` object, obtain a connection, and execute SQL queries.

Figure 3.6 shows the basic steps involved in using JDBC to write a database program in Java:

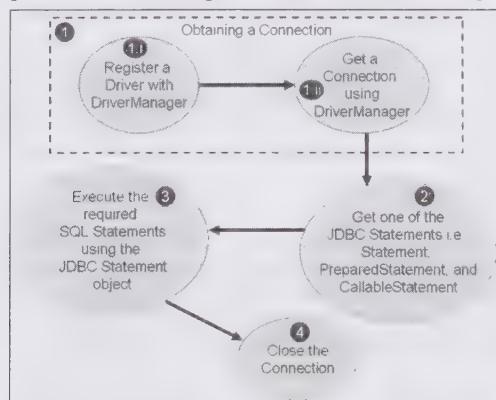


Figure 3.6: Showing Basic Steps to use JDBC

Figure 3.6 shows the following broad steps that need to be performed to implement JDBC in Java application:

1. Obtaining a connection
2. Creating a JDBC Statement object
3. Executing SQL statements
4. Closing the connection

Let's discuss each of them in detail.

## Obtaining a Connection

To obtain an object of the `Connection` class, you need to first register a driver with the `DriverManager` class by invoking the `registerDriver()` method, setting the `System` property, or invoking the `Class.forName()` method. Then, the connection is obtained by using the `java.sql.DriverManager` class.

You need to perform the following steps to obtain a connection using the `DriverManager` class:

1. Register a `Driver` object with `DriverManager`
2. Establish a connection using `DriverManager`

Now, let's discuss each of these steps in detail.

### *Registering a Driver object with DriverManager*

Registering a driver with the `DriverManager` class makes the registered driver available to the `DriverManager` class, so that the `DriverManager` object can use it to establish a connection with the database. When a driver is registered with the `DriverManager` class, it creates the `DriverInfo` object to maintain the driver details and stores these details in a class variable of the `java.util.Vector` type.

You can register the driver by using any one of the following three approaches:

- ❑ Invoke the `registerDriver()` method, which is a static method declared in the `DriverManager` class. The `java.sql.Driver` type of object is passed as an argument to the `registerDriver()` method. The following code snippet shows how to register the `Driver` object with `DriverManager`:  
`DriverManager.registerDriver (new sun.jdbc.odbc.JdbcOdbcDriver());`
- ❑ Invoke the `Class.forName(<driver class name>)` method, which is used to load the driver class explicitly. According to the JDBC specifications, a static code block should be provided in every JDBC driver implementation class. This code block passes the object of the driver implementation class through the `registerDriver()` method. The following code snippet shows how to register the `Driver` object with `DriverManager` by using the `Class.forName()` method:  
`Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");`  
 Where the `sun.jdbc.odbc.JdbcOdbcDriver` class contains the following code:  
`public class JdbcOdbcDriver extends ... {`  
`static { DriverManager.registerDriver (new sun.jdbc.odbc.JdbcOdbcDriver()); }`  
`}`

In the preceding code snippet, observe that the result is similar to using the `registerDriver()` method.

### NOTE

*It is recommended to call the `newInstance()` method on the `Class` object, which is returned by the `Class.forName` method as some of the JVMs do not call the static initializers until an instance of the class is created.*

- ❑ Set the `System` property, where the name of the property is `jdbc.drivers`. The value of the `System` property can be mapped to one or more driver implementation class names, where ':' character is used as a delimiter.

The following code snippet shows registering the driver with the `DriverManager` class:

```
System.setProperty ("jdbc.drivers", "sun.jdbc.odbc.JdbcOdbcDriver");
Use the above method in our application, or while executing the application using a Java command, we can set system properties using the -D option of java command, example:
java -Djdbc.drivers=sun.jdbc.odbc.JdbcOdbcDriver MyJdbcEx1
```

Note that in the JDBC 4.0 specifications, the `getConnection()` method of the `DriverManager` class has been enhanced to support the Java Standard Edition Service Provider mechanism. With this feature, the JDBC 4.0 Driver must include the `META-INF/services/java.sql.Driver` file. Therefore, when using JDBC 4.0 driver, you do not need to perform this step; that is, explicitly registering a Driver with `DriverManager`.

### *Establishing a Connection using DriverManager*

You can now establish connection with a database after registering the driver with the `DriverManager` class. To create a connection, invoke any one of the following methods of the `DriverManager` class:

- ❑ `getConnection (String url)`

- ❑ `getConnection (String url, String username, String password)`
- ❑ `getConnection (String url, Properties info)`

In the preceding methods, `<url>` is a JDBC URL, which represents a unique name used to identify the driver and obtain the connection. The JDBC URL even contains additional information, such as username and password, required to establish the connection. The syntax of the JDBC URL is as follows:

```
jdbc: <sub protocol> : <info>
```

In the preceding syntax:

- `Jdbc`—Represents the protocol in the JDBC URL
- `<sub protocol>`—Specifies the vendor specific name of the driver used to create the connection
- `<info>`—Takes additional information required to establish the connection, such as the database name and port number, which vary from one driver to another

The following code snippet shows some JDBC URLs:

```
For Type-1 driver, i.e. JDBC-ODBC Bridge Driver, the JDBC URL is:  
jdbc: odbc: SuchitaDSN.  
For Oracle Type-2 driver:  
String dbName = "kogent";  
String oracleURL = "jdbc:oracle:oci8:@" + dbName;  
  
//oracleURL = "jdbc:oracle:oci8:@kogent"  
  
For Oracle Type-4 driver:  
String host = "localhost";  
String dbName = "kogent";  
int port = 1521;  
String oracleURL = "jdbc:oracle:thin:@" + host + ":" + port + ":" + dbName;  
//oracleURL = "jdbc:oracle:thin:@192.168.1.123:1521:XE""
```

When the `getConnection()` method is invoked, it checks if any one of the drivers registered with the `DriverManager` class recognizes the given JDBC URL. If a driver accepts the URL, that driver is used by `DriverManager` to establish the connection with the DBMS located by the given JDBC URL. Consequently, if no driver accepts the URL, the `DriverManager` class throws the `java.sql.SQLException` exception to the application.

## Creating a JDBC Statement Object

You can execute the SQL statements only after creating the JDBC Statement object. The utility objects available to execute SQL statements are `Statement`, `PreparedStatement`, and `CallableStatement`.

Invoke the `createStatement()` method on the current `Connection` object to create the `Statement` object. The following code snippet shows how to create the `Statement` object using the `createStatement()` method:

```
Statement stmt = connection.createStatement();
```

## Executing SQL Statements

After the `Statement` object is created, it can be used to execute the SQL statements by using the `execute()`, `executeUpdate()`, or `executeQuery()` methods. The `executeQuery()` method is only used in the `SELECT` statement. For other database operations, such as `INSERT`, `UPDATE`, and `DELETE`, the `executeUpdate()` method is used to execute statements. The following code snippet shows how to execute a SQL statement:

```
//Using executeQuery()  
String query = "SELECT col1, col2, col3 FROM table_name";  
ResultSet results = stmt.executeQuery(query);  
//Using executeUpdate()  
String query= "INSERT into table_name values (value1, value2, ..., value n)";  
int count = stmt.executeUpdate(query);
```

If the statement produces a `ResultSet` object after executing the SQL statements, the `ResultSet` instance is used to retrieve the result. The `next()` method is invoked on the `ResultSet` object to navigate through a row at a time. The following code snippet shows the use of the `ResultSet` object within a connection:

```
while(results.next())  
{  
    System.out.println(results.getString(1) + " " +
```

```

        results.getString(2) + " " +
        results.getString(3));
    }
}

```

## Closing the Connection

You need to close the connection and release the session after executing all the required SQL statements and obtaining the corresponding results. This can be done by calling the `close()` method of the `Connection` interface. The following code snippet shows how to close a connection:

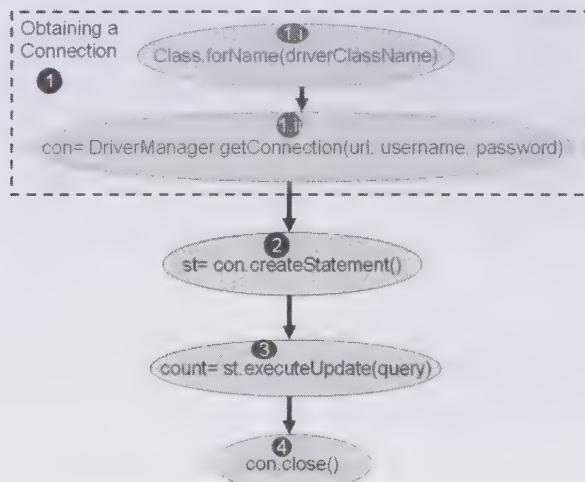
```
connection.close();
```

Now let's create a simple application to implement JDBC APIs.

## Creating a Simple JDBC Application

Let's now learn to create a simple JDBC application that inserts a record in a database table. In our case, we are inserting the record of a student in the `students` table of the Oracle data source. To insert the details of a student, you first need to establish a connection with the database and then execute the insert query.

Figure 3.7 displays how to use JDBC to obtain a connection and communicate with the database:



**Figure 3.7: Creating and Using Connection**

The steps shown in Figure 3.7 describe how to get a connection and execute the SQL statements. The following are the basic steps to use JDBC to connect to the data store and execute a simple SQL query:

1. Obtain the connection
2. Get the utility objects, such as `Statement`, `PreparedStatement`, and `CallableStatement`, to execute SQL statements
3. Execute the required SQL statements
4. Close the connection

Now, let's try to understand the concept better by creating a simple application, `BasicJDBCExample`. In this application, let's create the `JDBCExample1.java` file, which demonstrates the basic steps to access a database using JDBC.

Listing 3.1 shows the code for the `JDBCExample1.java` file (you can find this file in the `JavaEE\Chapter3\BasicJDBCExample` folder on the CD):

**Listing 3.1: Showing the JDBCExample1.java File**

```

package com.kogent.jdbc;

import java.sql.*;
public class JDBCExample1 {
}

```

```

public static void main(String args[])
throws SQLException, ClassNotFoundException {

    String driverClassName="sun.jdbc.odbc.JdbcOdbcDriver";
    String url="jdbc:odbc:XE";
    String username="scott";
    String password="tiger";

    String query = "insert into students values (101, 'Kumar')";

    //Load driver class
    Class.forName (driverClassName);

    // obtain a connection
    Connection con=DriverManager.getConnection(url, username, password);

    // Obtain a Statement
    Statement st=con.createStatement();
    //Execute the Query
    int count=st.executeUpdate (query);
    System.out.println ("Number of rows effected by this query = "+count);
    // Closing the connection as our requirement with connection is
    //completed
    con.close();

} //main
} //class

```

Listing 3.1 shows the uses of JDBC components in a simple application. The application uses the JDBC Type-1 driver (JDBC-ODBC Bridge Driver) to connect to the database. You must import the `java.sql` package to provide the basic SQL functionality and use the classes of the package. All the methods used by the application are wrapped in the `java.sql` package.

## Configuring the Application

You need to configure an application before running it. The following steps need to be performed to configure a JDBC application:

1. Create a table in a database as per your requirement
2. Configure the data source name of the database to use the JDBCExample1 application to connect to the database

Let's learn to perform the preceding steps next.

### Creating a Table

The JDBCExample1 application uses a table named `students`. The `students` table can be created by using the `CREATE` table command. The following code snippet shows how to create the `students` table in a database:

```

create table <table name> (
    <column_name1> <type>,
    <column_name2> <type>,
    ...
    ...
    <column_namen> <type>);

```

Example:

```

create table students (
    stdid number(3),
    stdname varchar2(30));

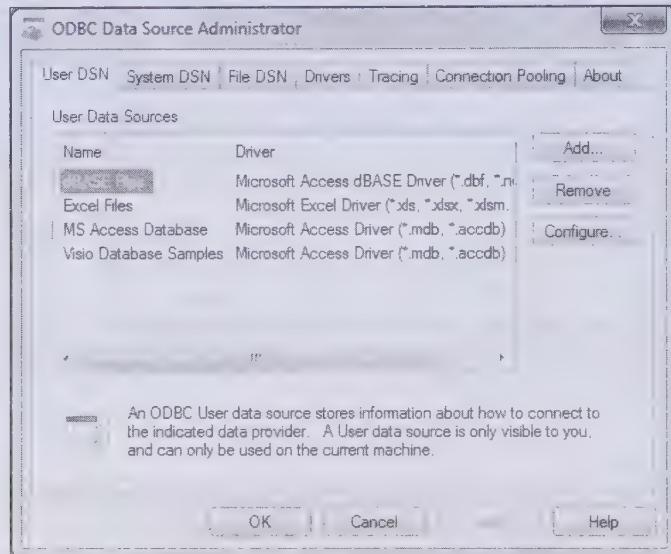
```

### Creating a Database Source Name

The code of the `JDBCExample1.java` file, given in Listing 3.1, uses the Type-1 (Jdbc-Odbc Bridge Driver) Type-1 driver to connect to the database, which requires a Data Source Name (DSN) to connect to the database.

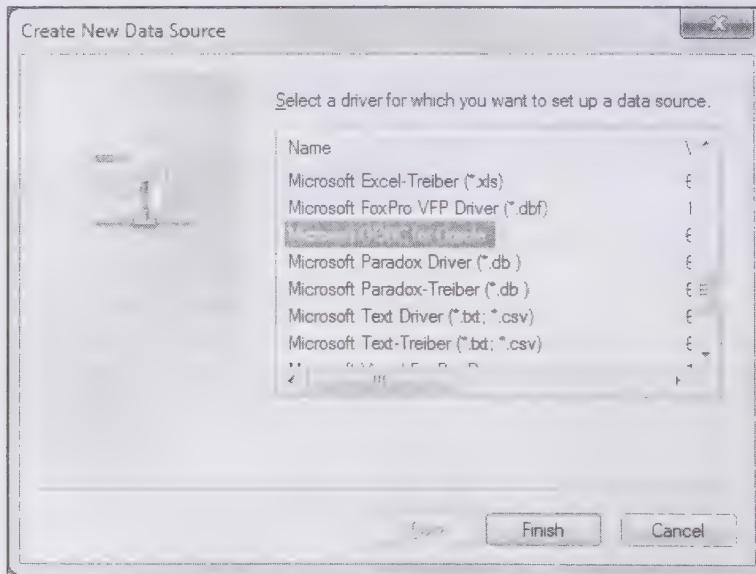
Perform the following steps to create a DSN in Windows 7:

- Select Control Panel → System and Security → Administrative Tools → Data Sources (ODBC) from the Start menu of your desktop. The ODBC Data Source Administrator dialog box appears, as shown in Figure 3.8:



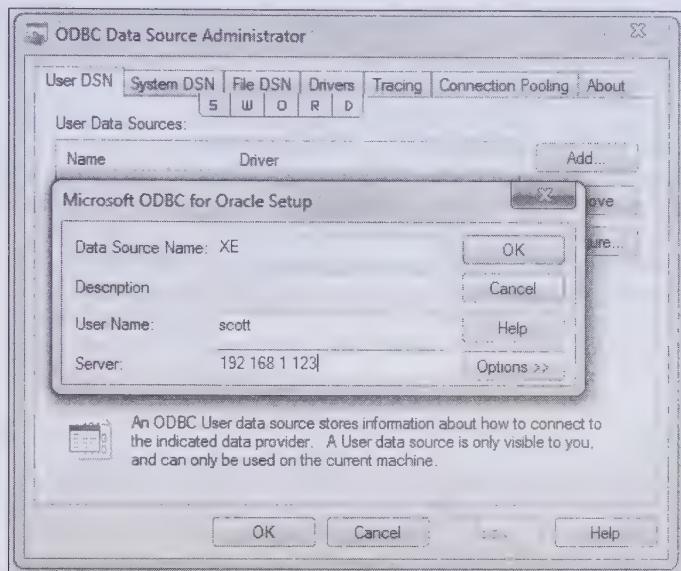
**Figure 3.8: Displaying the ODBC Data Source Administrator Screen**

- Click the Add button to add the data source to which the driver is to be connected. The Create New Data Source dialog box appears, as shown in Figure 3.9:



**Figure 3.9: Creating a New Data Source**

- Select the required driver. In our case, we have selected Microsoft ODBC for Oracle, as we want to connect to the Oracle database.
- Click the Finish button (Figure 3.9) to open the Microsoft ODBC for Oracle Setup dialog box, as shown in Figure 3.10:



**Figure 3.10: Displaying the Microsoft ODBC for Oracle Setup Dialog Box**

- Enter the details in the following fields (Figure 3.10):

- Data Source Name** – Specifies the name that the application uses within the JDBC URL. In our case, we have specified XE as the Data Source Name.
- Description** – Specifies a brief description about the DSN. This field is optional.
- User Name** – Specifies the database user name (optional). In our case, the user name is scott.
- Server** – Represents the host String that is required if the oracle database server is installed on a different computer. In our case, the IP of the server is 192.168.1.123.

- Click the OK button to create the DSN.

After creating the DSN, you can compile the Java source file by using Command Prompt. To open Command Prompt, select Start→All Programs→Accessories→Command Prompt. Command Prompt opens, where you can execute javac command to compile the source file and java command to run the .class file. Figure 3.11 shows the compilation and execution of the JDBCExample1.java file:

The screenshot shows a Windows Command Prompt window titled 'C:\Windows\system32\cmd.exe'. The command line shows the path 'D:\Java\JavaEE\Chapter 1\Basic JDBC\JDBCExample1\src\com\scott\'. The user runs the command 'java com.scott.JDBCExample1'. The output shows the application running and printing 'Number of rows effected by this query = 1'. The command prompt then returns to the user.

**Figure 3.11: Executing the Application**

After executing the JDBCExample1 class, a record is updated in the students table of the Oracle database. You can verify the updation of the record by opening the Run SQL Command Line window and connecting to the Oracle server.

You should ensure that the Oracle client is installed on your system. In our case, we are using Oracle 10g client edition. You can open the Run SQL Command Line window by selecting Start→All Programs→Oracle Client 10g Express Edition→Run SQL Command Line. The Run SQL Command Line window opens. Now, you should enter the username and password to log on to the Oracle database server. In our case, we have executed the following command to log on to the Oracle 10g database:

```
connect scott/tiger@192.168.1.123
```

In the preceding command, scott is the username and tiger is the password of the Oracle 10g server. In addition, 192.168.1.123 is the IP address of the machine on which the Oracle 10g server is installed. After executing the preceding command, you are connected to the Oracle 10g database. Now, enter the **select \* from students** command at the Run SQL Command Line prompt. You find that a record has been inserted into the students table, as shown in Figure 3.12:

The screenshot shows a terminal window titled "Run SQL Command Line". The window displays the following text:

```
SQL*Plus: Release 10.2.0.1.0 - Production on Sat Sep 24 13:32:14 2010
Copyright (c) 1982, 2005, Oracle. All rights reserved.

SQL> connect scott/tiger@192.168.1.123
Connected.
SQL> select * from students;
   STUDID  STUDNAME
        101 Kumar
SQL>
```

**Figure 3.12: Showing the Output of BasicJDBCExample in Run SQL Command Line**

Now let's discuss the PreparedStatement interface in detail.

### Working with the PreparedStatement Interface

The PreparedStatement interface, is subclass of the Statement interface, can be used to represent a precompiled query, which can be executed multiple times. Let's now first understand the difference between the execution process of a Statement object and the PreparedStatement object to execute a JDBC query.

Next, you learn about the setXX() methods and the advantages as well as disadvantages of the PreparedStatement interface. You also learn how to implement the PreparedStatement interface to execute the SQL query.

### Comparing the Execution Control of the Statement and PreparedStatement

When a Statement object is used to execute a query (that is, calling any one of the execute methods), the query is processed as follows:

1. The executeXXX() method is invoked on the Statement object by passing the SQL statement as parameter.
2. The Statement object submits the SQL statement to the database.
3. The database compiles the given SQL statement.
4. An execution plan is prepared by the database to execute the SQL statements.
5. The execution plan for the compiled SQL statement is then executed. Now, if the SQL statement is a data retrieval statement, such as the SELECT statement, the database caches the results of the SQL statement in the buffer.
6. The results are sent to the Statement object.
7. Finally, the response is sent to the Java application in the form of ResultSet.

Figure 3.13 displays the entire execution flow of the Statement object:

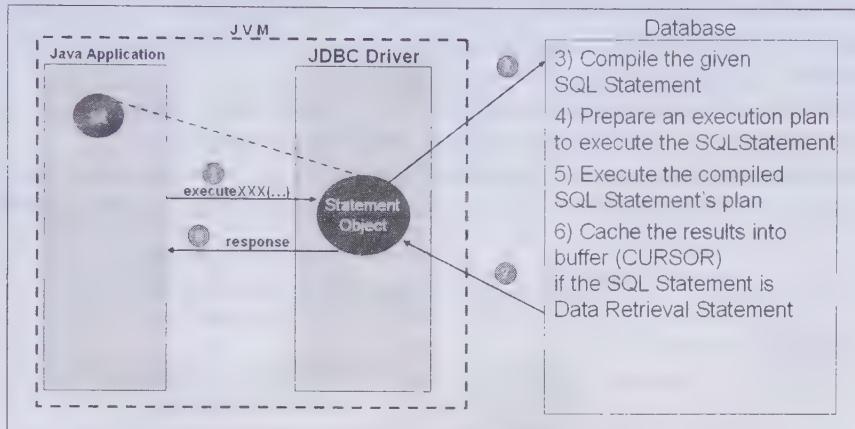


Figure 3.13: Displaying the Process Flow of the Statement Object

In Figure 3.13, the `st` element represents the Statement object reference. Compilation of a query includes syntax checking, name validation, and pseudo code generation. After a query is validated, the query optimizer prepares for the execution of the query and then returns what it considers to be the best alternative. The SQL statement needs to be executed each time it is requested.

It is not necessary to compile the SQL statement and prepare execution plan to execute a statement multiple times. DBMSs are designed to store the execution plans and execute them multiple times, if required. Consequently, the processing time of the DBMS is optimized. These stored execution plans of the SQL statements are known as pre-compiled SQL statements. DBMS intelligently maintains the compiled queries and provides a unique identity for the prepared execution plan, which the client uses to execute the same query next time. JDBC specifications support the use of this feature provided by DBMS. The PreparedStatement interface is designed specifically to support this feature.

PreparedStatements are pre-compiled; therefore, their execution is much faster as compared to the Statement objects included in an application. PreparedStatement is a subclass of the Statement interface; therefore, it inherits all the properties of the Statement interface. The execute methods do not take any parameter while using the PreparedStatement object.

You should keep in mind the following points while using the PreparedStatement interface:

- ❑ A PreparedStatement object must be associated with one connection.
- ❑ A PreparedStatement object represents the execution plan of a query, which is passed as parameter while creating the PreparedStatement object.
- ❑ After the connection on which the PreparedStatement object was created is closed, PreparedStatement is implicitly closed.
- ❑ When a PreparedStatement object is used to execute a query (that is, calling any one of the execute methods), the query is processed as follows:
  - The `prepareStatement()` method of the connection object is used to get the object of the PreparedStatement interface
  - The connection object submits the given SQL statement to the database
  - The database compiles the given SQL statement
  - An execution plan is prepared by the database to execute the SQL statements
  - The database stores the execution plan with a unique ID and returns the identity to the Connection object
- ❑ The Connection object prepares a PreparedStatement object, initializes it with the execution plan identity, and returns the reference of the PreparedStatement object to the Java application.

- ❑ The setXXX() methods of the PreparedStatement object are used to set the parameters of the SQL statement it is representing.
  - ❑ The executeXXX() method of the PreparedStatement object is invoked to execute the SQL statement with the parameters set to the PreparedStatement object
  - ❑ The PreparedStatement object delegates the request sent by a client to the database
  - ❑ The database locates and executes the execution plan with the given parameters
  - ❑ Finally, the result of the SQL statements is sent to the Java application in the form of ResultSet
- Figure 3.14 explains the flow of execution when PreparedStatement is used to execute SQL statements:

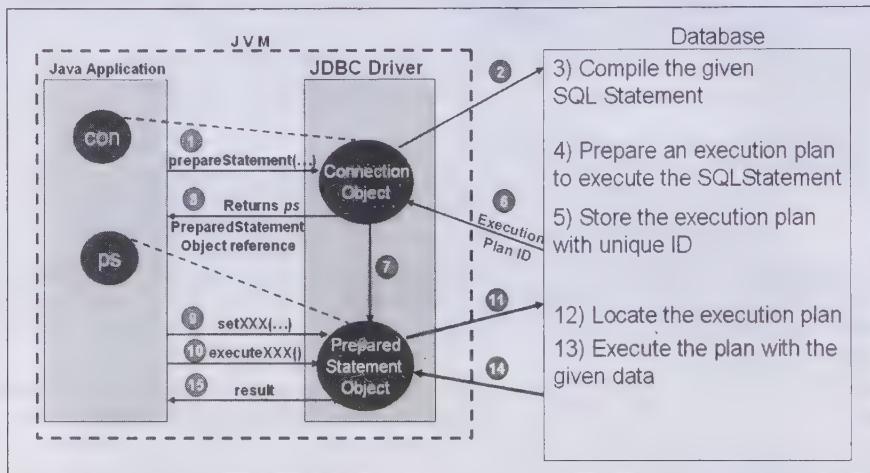


Figure 3.14: Showing Steps Involved in Using the PreparedStatement Object

In Figure 3.14, the `con` and `ps` elements represent the references of the Connection and PreparedStatement objects, respectively.

### Describing the setXXX Methods of the PreparedStatement Interface

You need to set the value of each placeholder ('?') parameter that is used inside the query string before executing a PreparedStatement object. The values for these placeholder parameters are provided at runtime to the SQL queries used within the PreparedStatement object. The values of this parameter can be set by using setXXX() methods.

Table 3.20 describes the setXXX() methods of the PreparedStatement interface:

Table 3.20: Methods Available in the PreparedStatement Interface

Method	Description
<code>setArray(int i, Array x)</code>	Sets the values of parameters to the given array object
<code>setAsciiStream(int parameterIndex, InputStream x, int length)</code>	Sets the values of the PreparedStatement parameter according to the given input stream, specified in the method
<code>setBigDecimal(int parameterIndex, BigDecimal x)</code>	Sets the values of the parameter by using the values specified in the <code>java.math.BigDecimal</code> value
<code>setBinaryStream(int parameterIndex, InputStream x, int length)</code>	Sets the binary values for the parameters used in the PreparedStatement object
<code>setBlob(int i, Blob x)</code>	Sets an integer value to the specified Blob object
<code>setBoolean(int parameterIndex, boolean x)</code>	Sets the boolean values for the parameters used in the PreparedStatement object

**Table 3.20: Methods Available in the PreparedStatement Interface**

<b>Method</b>	<b>Description</b>
setByte(int parameterIndex, byte x)	Sets the byte values for the parameters used in the PreparedStatement object
setBytes(int parameterIndex, byte[] x)	Sets the byte values in an array for the parameters used in the PreparedStatement object
setCharacterStream(int parameterIndex, Reader reader, int length)	Sets the character values for the PreparedStatement parameters and also specifies the length of the characters
setClob(int i, Clob x)	Sets an integer value to the specified Clob object
setDate(int parameterIndex, Date x)	Sets the PreparedStatement parameter with a java.sql.Date value
setDate(int parameterIndex, Date x, Calendar cal)	Sets the PreparedStatement parameter with a java.sql.Date value and also uses the calendar object to set the value of the parameter
setDouble(int parameterIndex, double x)	Sets the value of the parameter to the Java double value
setFloat(int parameterIndex, float x)	Sets the value of the parameter to the Java float value
setInt(int parameterIndex, int x)	Sets the value of the parameter to the Java int value
setLong(int parameterIndex, long x)	Sets the value of the parameter to the Java long value
setNull(int parameterIndex, int sqlType)	Sets the NULL values for the parameters of the specified sqlType
setNull(int paramIndex, int sqlType, String typeName)	Sets the NULL values for the parameters of the specified sqlType and typeName
setObject(int parameterIndex, Object x)	Sets the value of the parameter by using the given object value
setObject(int parameterIndex, Object x, int targetSqlType)	Sets the value of the parameter by using the given object value
setObject(int parameterIndex, Object x, int targetSqlType, int scale)	Sets the value of the parameter by using the given object value
setRef(int i, Ref x)	Sets the values of the parameters to the REF (<structured-type>) value
setShort(int parameterIndex, short x)	Sets the value of the parameter to the Java short value
setString(int parameterIndex, String x)	Sets the value of the parameter to the Java String value
setTime(int parameterIndex, Time x)	Sets the value of the parameter to the java.sql.Time value
setTime(int parameterIndex, Time x, Calendar cal)	Sets the value of the parameter to the java.sql.Time value by using the calendar object
setTimestamp(int parameterIndex, Timestamp x)	Sets the value of the parameter to the java.sql.TimeStamp value
setTimestamp(int parameterIndex, Timestamp x, Calendar cal)	Sets the value of the parameter to the java.sql.TimeStamp value by using the calendar object
setURL(int parameterIndex, URL x)	Sets the value of the parameter to the java.net.URL value

## Advantages and Disadvantages of Using a PreparedStatement Object

The advantages of using a PreparedStatement object are as follows:

- ❑ Improves the performance of an application as compared to the Statement object that executes the same query multiple times. The PreparedStatement object performs the execution of queries faster by avoiding the compilation of queries multiple times.

- Inserts or updates the SQL 99 data type columns, such as BLOB, CLOB, or OBJECT, with the help of setXXX methods.
- Provides a programmatic approach to set the values. In other words, the value of each parameter provided in a SQL query is passed separately by using the PreparedStatement object, unlike the Statement object.

The main disadvantage of PreparedStatement is that it can represent only one SQL statement at a time, i.e. you cannot execute more than one statement by a single PreparedStatement.

## Using the PreparedStatement Interface

The following are some of the situations when you should use PreparedStatement in a JDBC application:

- When a single query is being executed multiple times
- When a query consists of numerous parameters and complex types (SQL 99 types)

PreparedStatements are used to increase the efficiency and reduce the execution time of a query. An instance of PreparedStatement must be created to execute a precompiled SQL statement. Follow these broad-level steps to use the PreparedStatement interface:

1. Create a PreparedStatement object
2. Provide the values of the PreparedStatement parameters
3. Execute the SQL statements

Let's discuss each of these steps in detail.

### *Creating a PreparedStatement Object*

The prepareStatement(String) method of the Connection object is used to create the PreparedStatement object. The Connection object is used to access the PreparedStatement object, where the query supplied in the prepareStatement() method can contain zero or more question marks ('?', known as parameters). The values of question mark parameters can be set after the query is compiled.

The following code snippet shows how to create the PreparedStatement object in a connection:

```
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con= DriverManager.getConnection (url, "user", "password");
String query="insert into mytable values (?,?,?)";
//Step1: Get PreparedStatement object
PreparedStatement ps=con.prepareStatement (query);
```

In the preceding code snippet, the con parameter is the Connection object. This object is used to call the prepareStatement() method to obtain the PreparedStatement object. In the preceding code snippet, ps is the PreparedStatement object created by using the con object.

### *Providing the Values of the PreparedStatement Parameters*

You need to set the values of the question mark placeholders after creating the PreparedStatement object. The values of the question marks can be set by using the setXXX() methods. For example, if the question mark indicates the value of an integer data type, you can use the.setInt() method for the particular parameter. If you have a parameter of the Java string, you can call the.setString() method to set the value of the parameter. Note that these values should be set before prepared statements are executed.

In PreparedStatement, there is a setXXX method for each data type declared in Java. The setXXX method takes two arguments. The first argument indicates the parameter index and the second argument indicates the value of the parameter. Note that the parameter index starts from 1.

The following code snippet shows how to set the values of the question mark parameter:

```
//Step2: setting values for the parameters
ps.setString(1,"abc1");
ps.setInt(2,38);
ps.setDouble(3,158.75);
```

### *Executing the SQL Statements*

You can execute the precompiled SQL statements by using the execute(), executeUpdate(), or executeQuery() methods of the PreparedStatement interface. The result of these methods is same as that of the respective methods in the Statement interface.

The following code snippet shows how to execute the SQL statements:

```
ps.setString(1,"abc1");
ps.setInt(2,38);
ps.setDouble(3,158.75);
//Step3: Executing the SQL statements
int n = ps.executeUpdate(); // n is the number of rows or tables
that are being updated
```

Listing 3.2 demonstrates the use of PreparedStatement in an application. In Listing 3.2, the PreparedStatement object is used to execute the INSERT statement (you can find the PreparedStatementEx1.java file in the code\JavaEE\Chapter3\PreparedStatement folder on the CD):

**Listing 3.2:** Showing the PreparedStatementEx1.java File

```
package com.kogent.jdbc;
import java.sql.*;
/**
 * @author Suchita
 */
public class PreparedStatementEx1 {
    public static void main(String s[]) throws Exception {
        Class.forName("oracle.jdbc.driver.OracleDriver").newInstance();
        Connection con= DriverManager.getConnection (
        "jdbc:oracle:thin:@192.168.1.123:1521:XE","scott","tiger");
        String query="insert into mytable values (?,?,?)";
        //Step1: Get PreparedStatement
        PreparedStatement ps=con.prepareStatement (query);
        //Step2: set parameters
        ps.setString(1,"abc1");
        ps.setInt(2,38);
        ps.setDouble(3,158.75);
        //Step3: execute the query
        int i=ps.executeUpdate();
        System.out.println("record inserted count:"+i);
        //To execute the query once again
        ps.setString(1,"abc2");
        ps.setInt(2,39);
        ps.setDouble(3,158.75);
        i=ps.executeUpdate();
        System.out.println("query executed for the second time count: "+i);
        con.close();
    }
}
```

//main  
}//class

Listing 3.2 uses the PreparedStatement object along with the connection object. The setXXX() methods are used to set the values of the arguments. The preceding listing sets the values of the integer, string, and double data types. The executeUpdate() method used in Listing 3.2 retrieves the number of rows affected by executing the SQL statement.

The output of Listing 3.2 is shown in Figure 3.15:

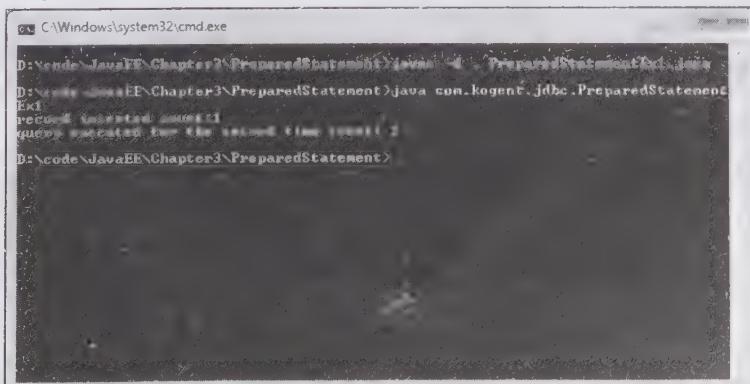


Figure 3.15: Displaying the Output of the PreparedStatementEx1.java File

After learning about the PreparedStatement interface, let's now proceed to learn about the CallableStatement interface.

## Working with the CallableStatement Interface

The CallableStatement interface extends the PreparedStatement interface and also provides support for both input as well as output parameters. The CallableStatement interface provides a standard abstraction for all the data sources to call stored procedures and functions, irrespective of the vendor of the data source. This interface is used to access, invoke, and retrieve the results of SQL stored procedures, functions, and cursors. Stored procedures let you write queries that are quick to run and easy to invoke. It is often easier to update an application by altering or making a few changes in the stored procedures. Functions are similar to procedures; however, the major difference between a function and procedure is that a function always returns a scalar value. You can also use cursors with CallableStatement to retrieve a ResultSet from a database.

Let's now demonstrate the use of CallableStatement with stored procedures, functions, and cursors.

### Describing Stored Procedures

A stored procedure is a subroutine used by applications to access data from a database. Stored procedures are called by using the CallableStatement interface in Java. The procedures called by the CallableStatement object are the database programs that contain the database interface. A stored procedure has the following properties:

- Contains input, output, or both these parameters
- Returns a value through the OUT parameter after executing the SQL statements
- Returns multiple ResultSets when required

Stored procedures are generally a group of SQL statements that allows you to make a single call to a database. The SQL statements in a stored procedure are executed statically for better performance. A stored procedure encapsulates the values of the following types of parameters:

- IN – Refers to the parameter whose value cannot be overwritten and referenced by a stored procedure
- OUT – Refers to the parameter whose value can be overwritten; however, cannot be referenced by a stored procedure
- IN OUT – Refers to the parameter whose value can be overwritten and referenced by the stored procedure

The following code snippet shows how to create or replace a stored procedure:

```
Create or [Replace] Procedure procedure_name
    [(parameter [, parameter])]
    IS
        [Declarations] BEGIN
            executables
            [EXCEPTION exceptions]
        END [Procedure_name]
```

### Using the CallableStatement Interface

In Java, the CallableStatement interface is used to call the stored procedures and functions. Therefore, the stored procedure can be called by using an object of the CallableStatement interface. The broad-level steps to use the CallableStatement interface in an application are:

1. Creating the CallableStatement object
2. Setting the values of the parameters
3. Registering the OUT parameters type
4. Executing the procedure or function
5. Retrieving the parameter values

Let's discuss these in details:

#### Creating the CallableStatement Object

The first step to use the CallableStatement interface is to create the CallableStatement object. The CallableStatement object can be created by invoking the prepareCall (String) method of the Connection object. The syntax to call the prepareCall method in an application is:

```
{call procedure_name(?, ?, ...)} // calling the prepareCall  
method with parameters.  
  
{call procedure_name}// with no parameter
```

### *Setting the Values of the Parameters*

You need to set the values of the IN and IN OUT type parameters in the stored procedure after creating the CallableStatement object. The values of these parameters can be set by calling the setXXX() method of the CallableStatement interface. The setXXX() method is used to pass the values to the IN, OUT, and IN OUT parameters. The values for a parameter can be set by using the following syntax:

```
setXXX (int index, XXX value)
```

### *Registering the OUT Parameters Type*

The OUT or IN OUT parameter used in a procedure represented by CallableStatement must be registered to collect the values of the parameters after the stored procedure is executed. You can register the parameters by invoking the registerOutParameter() method of the CallableStatement interface. This method defines the type of parameter used in the CallableStatements interface. The parameters can be registered by using the following syntax:

```
registerOutParameter (int index, int type)
```

### *Executing the Procedure or Function*

After registering the OUT parameter type, you need to execute the procedure. The execute() method of the CallableStatement interface is used to execute the procedure and does not take any argument.

### *Retrieving the Parameter Values*

You need to retrieve the OUT or IN OUT type parameter values of the stored procedure after executing the stored procedure. You can use the getXXX() method of the CallableStatement interface to retrieve the parameter values of the procedure.

After you have retrieved the results, repeat the steps if you want to execute the same procedure again with different parameter values. After performing all tasks associated with the database connection, it is a good practice to invoke the close() method on the CallableStatement object.

## **An Example of Using the CallableStatement Interface**

As learned earlier, you can use the CallableStatement interface to execute a stored procedure with the IN and OUT parameters. In this section, you first learn to implement the CallableStateement interface to execute a stored procedure that accepts the IN parameters. In other words, we create the createAccount stored procedure that needs IN parameters for execution, which are provided by using the CallableStatement interface in an application.

Later, the CallableStatement interface is used with the OUT parameter. In other words, the getBalance stored procedure is created, which provides the balance of an account holder as the output to the application invoking the stored procedure.

### *Executing a Stored Procedure with the IN Parameter*

Let's now create an application to call a stored procedure using the CallableStatement interface. You can find this application on the CD in the code\JavaEE\Chapter3\callablestatement folder.

First, create two tables called bank and personal\_details. In addition, create a procedure named createAccount by using SQL queries, as shown in the following code snippet:

```
Create table bank (  
    Accno number,  
    Name varchar2(20),  
    Bal number(10,2),  
    Acctype number  
);  
Create table personal_details (  
    Accno number,  
    address varchar2(20),
```

```
    phno number
);
```

The preceding code snippet shows that the `createAccount` procedure can be used to insert data into database tables.

The following code snippet shows the SQL query to create the `createAccount` procedure:

```
create or replace procedure createAccount (accnumber number, actype number,
acname varchar2, amt number, addr varchar2, phno number) is
begin
insert into bank values (accnumber, acname, amt, actype);
insert into personal_details values ( accnumber, addr, phno);
end;
/
```

In the preceding code snippet, the values in the `bank` and `personal_details` tables are inserted by using the `createAccount` procedure.

Figure 3.16 shows the output of executing the preceding code snippets at the Run SQL Command Line prompt:

The screenshot shows the Oracle SQL\*Plus environment with the following session history:

```
SQL*Plus: Release 10.2.0.1.0 Production on Sat Apr 24 14:03:46 2010
Copyright (c) 1992, 2005, Oracle. All rights reserved.
SQL> connect scott/tiger@192.168.1.123
Connected.
SQL> Create table bank (
2   Accno number,
3   Acname varchar2(20),
4   Amnt number(10,2),
5   Actype number,
6   Addr varchar2(20),
7   Phno number
8 );
Table created.
SQL> Create table personal_details (
2   Accno number,
3   Address varchar2(20),
4   Phno number
5 );
Table created.
SQL> Create or replace procedure createAccount (
2   accnumber number,
3   actype number,
4   acname varchar2,
5   amt number,
6   addr varchar2,
7   phno number) is
8 begin
9   insert into bank values (accnumber, acname, amt, actype);
10  insert into personal_details values ( accnumber, addr, phno);
11 end;
Procedure created.
SQL>
```

**Figure 3.16: Showing the Creation of Table and Stored Procedure**

The tables and procedures created in the Oracle 10g database, as shown in Figure 3.16, are used in Listing 3.3 to call the `createAccount` stored procedure by using `CallableStatement`. You can see the use of the `IN` parameter in Listing 3.3. The commented line (`//Step2: set IN parameters`) in Listing 3.3 shows the use of the `IN` parameter to work with `CallableStatement` (you can find the `CallableStatementEx1.java` file in the `code\JavaEE\Chapter3\callablestatement` folder on the CD):

### Listing 3.3: Showing the Code for the `CallableStatementEx1.java` File

```
package com.kogent.jdbc;
import java.sql.*;
/**
 * @author Suchita
 */
public class CallableStatementEx1 {
    public static void main(String s[]) throws Exception {
        Class.forName("oracle.jdbc.driver.OracleDriver").newInstance();
        Connection con=DriverManager.getConnection(

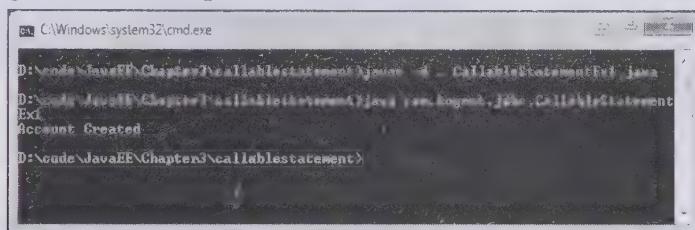
```

```

"jdbc:oracle:thin:@192.168.1.123:1521:XE","scott","tiger");
// Get CallableStatement
CallableStatement cs= con.prepareCall ("{call
createAccount (?,?,?,?,?,?)}");
// set IN parameters
cs.setInt(1, 103);
cs.setInt(2, 9);
cs.setString(3, "Neeraj");
cs.setDouble(4, 10000);
cs.setString(5, "Delhi");
cs.setInt(6, 123456789);
// register OUT parameters
//In this procedure example we don't have OUT parameters
//executing the stored procedure
cs.execute();
System.out.println("Account Created");
con.close();
}//main
}//class

```

The output of Listing 3.3 is shown in Figure 3.17:



**Figure 3.17: Showing the Output of CallableStatementEx1.java**

If CallableStatement uses the OUT parameter to work with the stored procedure, you need to register the OUT parameter using the `registerOutParameter()` method of the CallableStatement interface.

#### *Executing a Stored Procedure with the OUT Parameter*

In this section, let's create an application that calls a stored procedure named `getBalance()` by using the CallableStatement interface. First, create a procedure named `getBalance()`, as shown in the following code snippet:

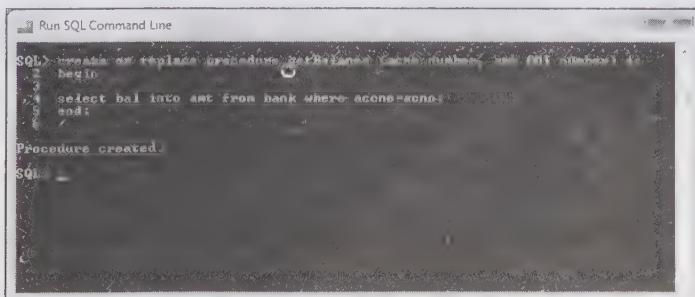
```

create or replace procedure getBalance (acno number, amt OUT number) is
begin
    select bal into amt from bank where accno=acno;
end;
/

```

In the preceding code snippet, the OUT parameter is used to hold the value (balance) retrieved by executing the SQL query.

Figure 3.18 shows the `getBalance` procedure created by using the SQL editor:



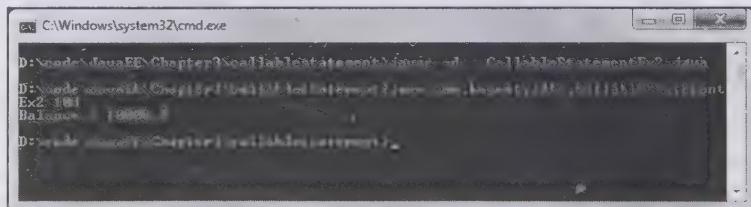
**Figure 3.18: Creating a Procedure by Using the OUT Parameter**

Let's now see how to execute the getBalance stored procedure with the OUT parameter of CallableStatement. Listing 3.4 shows the use of the OUT parameter with the stored procedure (you can find the CallableStatementEx2.Java file in the code\JavaEE\Chapter3\callablestatement folder on the CD):

**Listing 3.4:** Showing the Code for the CallableStatementEx2.Java File

```
package com.kogent.jdbc;
import java.sql.*;
/**
 * @author Suchita
 */
public class CallableStatementEx2 {
public static void main(String s[]) throws Exception {
    Class.forName("oracle.jdbc.driver.OracleDriver").newInstance();
    Connection con=DriverManager.getConnection(
        "jdbc:oracle:thin:@192.168.1.123:1521:XE","scott","tiger");
    CallableStatement cs= con.prepareCall("{call getBalance(?,?)}");
    cs.setInt(1, Integer.parseInt(s[0]));
    cs.registerOutParameter(2, Types.DOUBLE);
    cs.execute();
    System.out.println("Balance : "+ cs.getDouble(2));
    con.close();
} //main
} //class
```

Figure 3.19 displays the output of Listing 3.4:



**Figure 3.19: Showing the Output of CallableStatementEx2 by Using the OUT Parameter**

In the next subsection, let's discuss how to call functions using CallableStatements.

### Calling Functions using CallableStatements

Most of the databases provide support for the numeric, string, time, date, system, and conversion functions. These functions are used in SQL statements to return scalar values stored in a database. The scalar functions supported by a DBMS must also be supported by the database drivers used in the application. The user can access these functions by calling the metadata methods.

Table 3.21 describes the scalar function types supported by Oracle:

**Table 3.21: Scalar Functions and their Uses**

Function Type	Use
Numeric Functions	Operate on numeric data types, such as greatest(), least(), round(), trunc(), length(), and lower()
String Functions	Operate on the string data types, such as Char(), concat(), insert(), and length()
Time & Date Functions	Access all the time and date related information from a database
System functions	Retrieve the information about the DBMSs used in an application
Conversion Functions	Convert the data type of a given value into the required type

In addition to these pre-defined functions, DBMS has a feature to create user-defined functions. User-defined functions can be used within Data Manipulation Language (DML) queries; however, it is not recommended to use DML queries within a function. The user-defined functions can be used in the following situations:

- In the column names of a SELECT statement
- In the WHERE clause as a condition
- In the value clause of an INSERT statement
- In the SET clause of an UPDATE statement

The following syntax shows how to create a user-defined function:

```
Create [OR Replace] FUNCTION function_name [(parameter [, parameter])] 
RETURN return_datatype
IS/AS
[Declaration_section]
BEGIN
    executable_section
    [Exception exception_section]
END [function_name];
```

The procedure to call a function in an application is the same as that of procedures. The syntax to invoke a function in JDBC (String argument of the prepareCall method) is as follows:

```
{call ?:=function_name(?, ?, ...)} // with string parameters
{call ?:=function_name} // with no parameter
```

Listing 3.5 shows the use of a user-defined function in an application by using CallableStatement (you can find the CallableStatementEx3.java file in the code\JavaEE\Chapter3\callablestatement folder on the CD):

**Listing 3.5:** Showing the Code for the CallableStatementEx3.java File

```
package com.kogent.jdbc;

import java.sql.*;
import java.util.*;
/**
 * @author Suchita
 */
public class CallableStatementEx3 {
public static void main(String s[]) throws Exception {

    Properties p=new Properties();
    p.put("user","scott");
    p.put("password","tiger");

    oracle.jdbc.driver.OracleDriver
    od=new oracle.jdbc.driver.OracleDriver();
    Connection con=od.connect ("jdbc:oracle:thin:@192.168.1.123:1521:XE",p);

    CallableStatement cs=con.prepareCall ("{call ?:=getBalanceF(?)}");
    cs.registerOutParameter (1, Types.DOUBLE);
    cs.setInt(2,Integer.parseInt(s[0]));
    cs.execute();
    System.out.println(cs.getDouble(1));
    con.close();

} //main
} //class
```

Listing 3.5 executes a user-defined function, getBalanceF(), by using the CallableStatement object, which is used to access the function from the Oracle database. The desired output of the function is then displayed to the user.

Figure 3.20 shows the creation of the getBalanceF() function in the application:

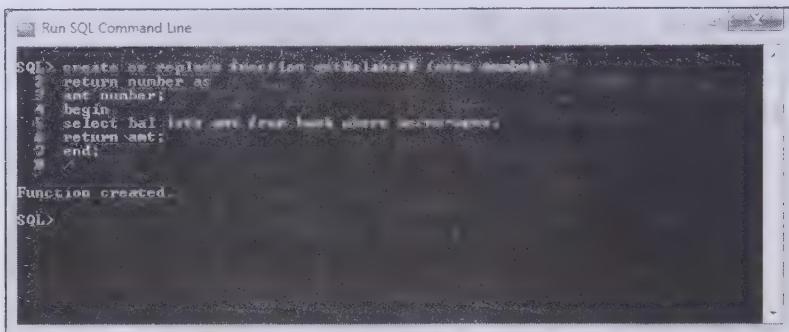


Figure 3.20: Creating a User-Defined Function

Figure 3.21 displays the output of CallableStatementEx3:



Figure 3.21: Showing Output of CallableStatementEx3 by Using Function

All the features discussed so far are used to retrieve a single record from a database. You can use a cursor to retrieve a ResultSet containing multiple records from the database. Let's discuss about the use of cursors in CallableStatements to retrieve the ResultSet object.

## Using Cursors in CallableStatements

A cursor allows you to iterate through the rows in a ResultSet. In other words, a cursor defines the run time execution environment for a query. You can open the cursor to execute the queries in that environment and read the output of the query from the cursor.

The syntax to create a cursor is shown in the following code snippet:

```
create or replace package package_name as
  TYPE type_name IS REF CURSOR;
END;
```

Cursors are used to retrieve ResultSet from a database through CallableStatement. Listing 3.6 shows the use of cursors to get the ResultSet object to access multiple records from a database (you can find the CallableStatementEx4.java file in the code\JavaEE\Chapter3\callablestatement folder on the CD):

### Listing 3.6: Showing the Code for the CallableStatementEx4.java File

```
package com.kogent.jdbc;
import java.sql.*;
import java.util.*;
public class CallablestatementEx4 {
    public static void main(String s[]) throws Exception {
        Properties p=new Properties();
        p.put("user","scott");
        p.put("password","tiger");
        oracle.jdbc.driver.oracleDriver
        od=new oracle.jdbc.driver.OracleDriver();
        Connection con=od.connect("jdbc:oracle:thin:@192.168.1.123:1521:XE",p);
        CallableStatement cs=
        con.prepareCall("{call ?:=getAccountDetails(?)}");
        cs.registerOutParameter(1, oracle.jdbc.OracleTypes.CURSOR);
        cs.setInt(2,Integer.parseInt(s[0]));
```

```

        cs.execute();
        ResultSet rs=(ResultSet) cs.getObject(1);
        while (rs.next()){
            System.out.print(rs.getInt(1)+"\t");
            System.out.print(rs.getString(2)+"\t");
            System.out.println(rs.getDouble(3));
        }
    }//while
    con.close();
}//main
}//class

```

Listing 3.6 uses the cursors and functions in the Oracle 10g database to access the ResultSet object representing all the accounts of the given account\_type.

Figure 3.22 displays the creation of the cursor and functions associated with Listing 3.6:

The screenshot shows the Oracle SQL Command Line interface. The user has run the following SQL code:

```

SQL> create or replace package mypack as
  TYPE mycursor is REF CURSOR;
end;
/
Package created.

SQL> create or replace function getAccountDetails (actype number)
 2 return mypack.mycursor as
 3 myresult mypack.mycursor;
4 begin
5   open myresult for
6   select accno,name,bal from bank where actype=actype;
7   return myresult;
8 end;
/
Function created.

```

Figure 3.22: Creating a Cursor and Functions

Figure 3.23 shows the output of the CallableStatementEx4 class created in Listing 3.6:

The screenshot shows a Windows command prompt window titled 'Run SQL Command Line'. The user has run the following command:

```

C:\Windows\system32\cmd.exe
D:\code\JavaEE\Chapter3\callablename>java -cp . -Djava.net.preferIPv4Stack=true CallableStatementEx4
D:\code\JavaEE\Chapter3\callablename>java -cp . -Djava.net.preferIPv4Stack=true CallableStatementEx4
103    Neeraj  10000.0
D:\code\JavaEE\Chapter3\callablename>

```

Figure 3.23: Showing the Output of the CallableStatementEx4.java File

## Working with ResultSets

A ResultSet is an interface provided in the `java.sql` package, and is used to represent data retrieved from a database in a tabular format. It implies that a ResultSet object is a table of data returned by executing a SQL query. A ResultSet object encapsulates the resultant tabular data obtained when a query is executed. A ResultSet object holds zero or more objects, where each of the objects represents one row that may span over one or more table columns. You can obtain a ResultSet object by using the `executeQuery` or `getResultSet` method of a statement. Some of the important points related to a ResultSet are as follows:

- ❑ ResultSets follow the iterate pattern.
- ❑ A ResultSet object is associated with a statement within a connection.
- ❑ You can obtain any number of ResultSets using one statement; however, only one ResultSet can be opened at a time. When you try to open a ResultSet using a statement that is already associated with an opened ResultSet, the existing ResultSet is implicitly closed.
- ❑ ResultSet is automatically closed when its associated statement is closed.

## Describing the Methods of ResultSets

The `java.sql.ResultSet` interface provides certain methods to work with `ResultSet` objects. The methods available in the `ResultSet` interface are used to move the cursor throughout the `ResultSet` and read the data.

Table 3.22 describes some of the most commonly used methods in the `ResultSet` interface:

**Table 3.22: Methods of the `java.sql.ResultSet` Interface**

Methods	Description
<code>absolute(int row)</code>	Moves the cursor to the specified row in the <code>ResultSet</code> object.
<code>afterLast()</code>	Places the cursor just after the last row in the <code>ResultSet</code> object.
<code>beforeFirst()</code>	Places the cursor before the first row in the <code>ResultSet</code> object.
<code>cancelRowUpdates()</code>	Cancels all the changes made to the rows in the <code>ResultSet</code> object.
<code>clearWarnings()</code>	Clears all warning messages on a <code>ResultSet</code> object.
<code>close()</code>	Closes the <code>ResultSet</code> object and releases all the JDBC resources connected to it.
<code>deleteRow()</code>	Deletes the specified row from the <code>ResultSet</code> object and the database.
<code>first()</code>	Moves the cursor to the first row in the <code>ResultSet</code> object.
<code>getArray()</code>	Retrieves the value of the specified column from the <code>ResultSet</code> object.
<code>getAsciiStream()</code>	Retrieves a specified column in the current row as a stream of ASCII characters.
<code>getXXX()</code>	Retrieves the column values of the specified types from the current row. The type can be any of the Java predefined data types, such as <code>int</code> , <code>long</code> , <code>byte</code> , <code>character</code> , <code>string</code> , <code>double</code> , or large object types.
<code>getDate()</code>	Retrieves the specified column from the current row in the <code>ResultSet</code> object. The object retrieved is of the <code>java.sql.Date</code> type in the Java programming language.
<code>getDate(String columnName, Calendar cal)</code>	Retrieves the specified column from the current row in the <code>ResultSet</code> object. The object retrieved is of the <code>java.sql.Date</code> type.
<code>getFetchDirection()</code>	Specifies the direction (forward or reverse) in which the <code>ResultSet</code> object retrieves the row from a database.
<code>getFetchSize()</code>	Retrieves the size of the associated <code>ResultSet</code> object.
<code>getMetaData()</code>	Retrieves the number, type, and properties of the <code>ResultSet</code> object.
<code>getObject(int columnIndex)</code>	Retrieves a specified column in the current row as an object in the Java programming language on the basis of the column index value passed as a parameter.
<code>getObject(int i, Map map)</code>	Retrieves a specified column as an object on the basis of the column number and <code>Map</code> instance passed as parameters.
<code>getObject(String columnName)</code>	Retrieves a specified column in the current row as an object on the basis of the column name passed as a parameter.
<code>getObject(String colName, Map map)</code>	Retrieves a specified column in the current row as an object on the basis of the column name and <code>Map</code> instance passed as parameters.
<code>getRow()</code>	Retrieves the current row number associated with the <code>ResultSet</code> object.
<code>getStatement()</code>	Retrieves the <code>Statement</code> object associated with the <code>ResultSet</code> object.
<code>getTime(int columnIndex)</code>	Retrieves the column values as a <code>java.sql.Time</code> object on the basis of column index passed as an integer parameter.

**Table 3.22: Methods of the java.sql.ResultSet Interface**

<b>Methods</b>	<b>Description</b>
getTime(int columnIndex, Calendar cal)	Retrieves the column values as a java.sql.Time object on the basis of column index as well as the cal object of the Calender class passed as parameters.
getTime(String columnName)	Retrieves the column values as a java.sql.Time object on the basis of column name passed as a String value.
getTime(String columnName, Calendar cal)	Retrieves the column values as a java.sql.Time object on the basis of String value of column name as well Calender object cal as parameters.
getTimestamp(int columnIndex)	Retrieves the column values as a java.sql.Timestamp object on the basis of the column index passed as a parameter.
getTimestamp(int columnIndex, Calendar cal)	Retrieves the column values as a java.sql.Timestamp object on the basis of the column index and the cal object of the Calendar class passed as parameters.
getTimestamp(String columnName)	Retrieves the column values as a java.sql.Timestamp object on the basis of the column name passed as a parameter.
getTimestamp(String columnName, Calendar cal)	Retrieves the column values as a java.sql.Timestamp object on the basis of the column name and the cal object of the Calendar class passed as arguments.
getType()	Retrieves the type of the ResultSet object used in a connection.
getWarnings()	Retrieves the warning reported on the ResultSet object.
insertRow()	Inserts the specified row and content into the ResultSet object and database.
isAfterLast()	Specifies whether the cursor of the ResultSet object is at the end of the last row.
isBeforeFirst()	Specifies whether the cursor is before the first row in the ResultSet object or not.
isFirst()	Specifies whether the cursor is on the first row or not.
isLast()	Detects whether the cursor is on the last row of the ResultSet object or not.
last()	Moves the cursor to the first row in the ResultSet object. The method returns true if the cursor is positioned on the first row, and false if the ResultSet object does not contain any rows.
moveToCurrentRow()	Moves the cursor to the current row in the ResultSet object.
moveToInsertRow()	Moves the cursor to the inserted row in the ResultSet object.
next()	Moves the cursor forward one row. The method returns true if the cursor is positioned on a row and false if the cursor is positioned after the last row.
previous()	Moves the cursor backward one row. The method returns true if the cursor is positioned on a row and false if the cursor is positioned before the first row.
refreshRow()	Refreshes the current row associated with the ResultSet object with the recent updates.
relative(int rows)	Moves the cursor to a relative number of rows or columns specified in the method.
rowDeleted()	Retrieves whether the row has already been deleted or not.
rowInserted()	Determines whether the current row has an insertion or not.
rowUpdated()	Retrieves whether the current row has been updated or not.

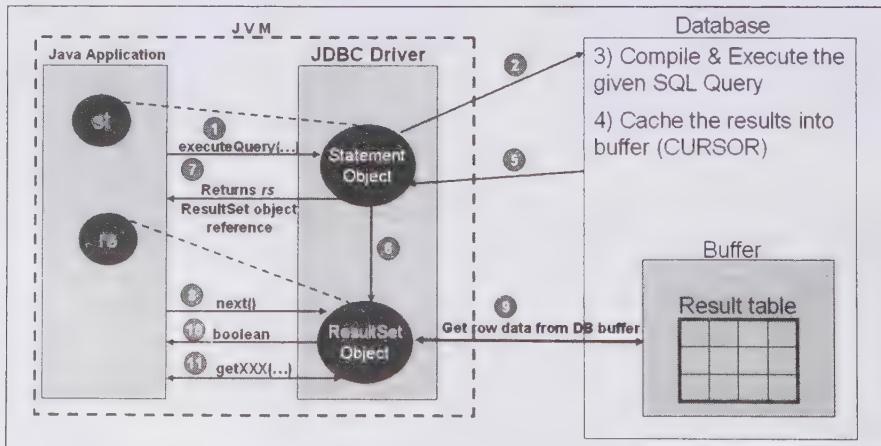
**Table 3.22: Methods of the java.sql.ResultSet Interface**

Methods	Description
setFetchDirection(int direction)	Sets the direction of the ResultSet object.
setFetchSize(int rows)	Sets the size of the ResultSet object.
updateArray()	Updates the column in the ResultSet object with a java.sql.Array value.
updateXXX()	Updates the column values of the current row of the specified type. The type can be any of the Java predefined data types, such as int, long, byte, character, string, double, and the large object types.
updateRow()	Updates the current row with new content.
wasNull()	Reports whether the last column has a SQL null value or not.
updateNull(String columnName)	Updates a specific column with a NULL value.
updateObject(int columnIndex, Object x)	Updates the specific column with an Object value.
updateTime(int columnIndex, Time x)	Updates the time value with a java.sql.Time value.
updateTimestamp(int columnIndex, Timestamp x)	Updates the time value with a java.sql.Timestamp value.
getConcurrency()	Retrieves the concurrency mode of the ResultSet object.
getCursorName()	Retrieves the SQL cursor name used by the ResultSet object.

## Using ResultSets

After obtaining a ResultSet object, you can use a Resultset to read the data (ResultSet content) encapsulated in it. Figure 3.24 shows the process flow involved in getting ResultSet from the Statement object and reading the data from the ResultSet object. The st and rs parameters represent the Statement and ResultSet object references, respectively.

Figure 3.24 shows the ResultSet operations:

**Figure 3.24: Explaining the ResultSet Operations**

Note that for every `next()` method invoked, the JDBC driver may not necessarily get the data row from the database buffer. This means that after every step 8, shown in Figure 3.24, there may not always be a step 9. Instead, the JDBC driver can get multiple rows of data at a time and buffer it on the client side. The buffering of data on the client size depends on the fetch size set for the ResultSet object. The fetch size of a ResultSet can be set by using the `setFetchSize(int)` method of ResultSet.

You can retrieve data from a ResultSet in two simple steps:

- Move the cursor position to the required row
- Read the column data using the getXXX methods

Let's discuss these steps in detail.

### Moving the Cursor Position

While obtaining data from a ResultSet, the cursor is initially placed before the first row, i.e. beforeFirst(). You can use the next() method of ResultSet to move the cursor position to the next record in the ResultSet. When the cursor is moved to the next record, it returns a boolean value indicating whether or not any record is available in the ResultSet. The next() method returns true if it successfully positions the cursor on the next row; otherwise, it returns false.

#### NOTE

JDBC 2.0 also introduces some other methods in ResultSet to move the cursor position, provided the ResultSet is of the scrollable type. The ResultSet generated is forward by default; therefore, you can iterate through it only in the forward direction from the first to the last row.

### Reading the Column Values

After moving the cursor to the respective row, you can use the getter methods of ResultSet to retrieve the data from the row where the cursor is positioned. Getter methods of ResultSet are overloaded, that means, there are two getter methods for each of the JDBC type. One of these two methods takes column index of type int as an input, where column index starts with 1; and the other method takes column name of the String type. You should note that the column names that are passed to getter methods are not case sensitive. If the same column is present more than once in a select list, the first instance of the column is to be returned.

Note that the column index supplied to the getXXX methods is the index that starts with 1, where the index numbers are given based on the resulted tabular data, and not on the source table that is queried.

For example, suppose a table of students contains two columns, stdid, and stdName. Now, if you obtain a ResultSet for the select stdName, and stdid from the students query, the column index 1 locates the stdName; whereas, index 2 locates stdid. The ResultSet interface has the getXXX method for all the basic and predefined complex types.

When the getter methods of ResultSet are invoked, the JDBC driver attempts to convert the requested column value into the respective Java type and returns the Java value. However, if it fails to convert the column value into its respective Java type, it throws the SQLException exception and describes it as a conversion error.

Figure 3.25 shows the exceptions thrown for the Oracle Thin driver:

```

C:\Windows\system32\cmd.exe
D:\eclipse\JavaEE\Chapter1\ResultSet2>javac -cp .;lib/ojdbc6.jar Main.java
159:   exception in thread "main" java.sql.SQLException: Fail to convert to int
internal representation
        at oracle.jdbc.driver.DatabaseError.throwSqlException(DatabaseError.java
:112)
        at oracle.jdbc.driver.DatabaseError.throwSqlException(DatabaseError.java
:141)
        at oracle.jdbc.driver.DatabaseError.throwSqlException(DatabaseError.java
:209)
        at oracle.jdbc.driver.CharConversionException.throwSqlException(CharConversionException.java
:132)
        at oracle.jdbc.driver.OracleResultSetImpl.next(OracleResultSetImpl.java
:251)
        at oracle.jdbc.driver.OracleResultSetImpl.getLocate(OracleResultSetImpl.java
:245)
D:\eclipse\JavaEE\Chapter1\ResultSet2>

```

Figure 3.25: Showing an Example of SQLException

Figure 3.25 shows the error message when the JDBC driver fails to convert the SQL type to the Java type. In our case, we have created the GetData.java file in which the column value is of String type and we have used getInt() method to retrieve the column value. The allowable mappings for the various SQL Types to Java types under the JDBC specification are described in Table 3.23:

**Table 3.23: JDBC and Java Data Types**

<b>JDBC Type</b>	<b>Java Type</b>
CHAR	String
VARCHAR	String
LONGVARCHAR	String
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
BIT	boolean
BOOLEAN	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int

In Java, the value of a column can be retrieved in the form of an object in a ResultSet by using the getObject() method.

The getObject() method of ResultSet uses the conversions as described in Table 3.24:

**Table 3.24: Showing the Conversion of JDBC to Java Object Type**

<b>JDBC Type</b>	<b>Java Object Type</b>
CHAR	String
VARCHAR	String
LONGVARCHAR	String
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
BIT	boolean
BOOLEAN	boolean
TINYINT	Integer
SMALLINT	Integer
INTEGER	Integer
BIGINT	Long
REAL	Float
FLOAT	Double
DOUBLE	Double
BINARY	byte[]
VARBINARY	byte[]
LONGVARBINARY	byte[]

**Table 3.24: Showing the Conversion of JDBC to Java Object Type**

JDBC Type	Java Object Type
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp
DISTINCT	Object type of underlying Type
CLOB	java.sql.Clob
BLOB	java.sql.Blob
ARRAY	java.sql.Array
STRUCT	java.sql.Struct or java.sql.SQLData
REF	java.sql.Ref
DATALINK	java.net.URL
JAVA_OBJECT	Underlying Java class
ROWID	java.sql.RowId
NCHAR	String
NVARCHAR	String
LONGNVARCHAR	String

Let's now look at some examples of using ResultSet.

### Retrieving All the Rows in a Table

As already explained, you can retrieve rows from a table by using the ResultSet object. Let's now understand how you can retrieve all the rows of the mytable table. A row in the mytable table can store data of different types, such as a string, integer, and floating-point number.

Listing 3.7 shows how you can retrieve all the rows from the mytable table (you can find the GetAllRows.java file in the code\JavaEE\Chapter3\Resultset folder on the CD):

**Listing 3.7: Showing the Code for the GetAllRows.java File**

```
package com.kogent.jdbc;
import java.sql.*;
/**
 * @author Suchita
 */
public class GetAllRows {

    public static void main(String args[]) throws
        SQLException, ClassNotFoundException {

        //Get Connection
        Connection con=prepareConnection();

        // Obtain a Statement
        Statement st=con.createStatement();
        String query = "select * from mytable";

        //Execute the Query
        ResultSet rs=st.executeQuery (query);

        System.out.println ("COL1\tCOL2\tCOL3");
        while (rs.next()){
            System.out.print (rs.getString ("COL1") + "\t");
            System.out.print (rs.getString ("COL2") + "\t");
            System.out.println (rs.getString ("COL3"));
        }
    }
}
```

```

        System.out.print (rs.getInt ("COL2") + "\t");
        System.out.println (rs.getInt("COL3"));
    }//while
    con.close();
}//main
public static Connection prepareConnection()
    throws SQLException,
    ClassNotFoundException {
    String driverClassName="oracle.jdbc.driver.OracleDriver";
    String url="jdbc:oracle:thin:@192.168.1.123:1521:XE";
    String username="scott";
    String password="tiger";

    //Load driver class
    Class.forName (driverClassName);

    // Obtain a connection
    return DriverManager.getConnection (url, username, password);
}//prepareConnection
}//class

```

In Listing 3.7, the `next()` method is used to move the cursor position in the forward direction. The application throws an exception if the user tries to move the cursor in the backward direction from the relative position of the cursor.

#### NOTE

The column names used with the `getXXX` methods of `ResultSet` are not the actual table column names; instead, they are the column names of the table that would be created as a `ResultSet`. For instance, if you use a query to select `col1` as `c1`, `col2` as `c2`, and `col3` as `c3` from the `mytable` table, the column names that you need to use in these `getXXX` methods are `c1`, `c2` and `c3` and not `col1`, `col2` and `col3`.

In Listing 3.7, we have obtained the data by using column names. However, we can also obtain the data by using column numbers. Use the following code snippet in place of the code with column names (as shown in Listing 3.7) to obtain the data using column numbers:

```

while (rs.next()) {
    System.out.print (rs.getString (1) + "\t");
    System.out.print (rs.getInt (2) + "\t");
    System.out.println (rs.getDouble (3));
}//while

```

#### NOTE

Using the column index form of the `getXXX()` methods is more efficient than the column name form, because the driver does not have to deal with the extra steps of parsing the column name, finding it in the select list, and then turning it into a number.

Compiling and running the application shown in Listing 3.7 gives the output, as shown in Figure 3.26:

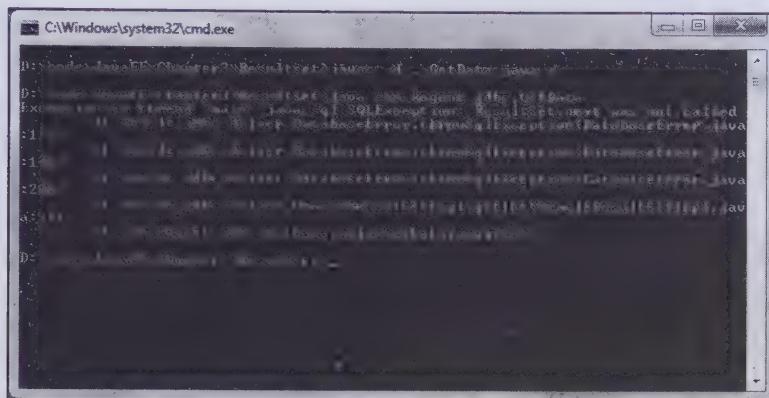
```

C:\Windows\system32\cmd.exe
D:\code\JavaEE\Chapter3\ResultSet>javac -d . GetAllRows.java
D:\code\JavaEE\Chapter3\ResultSet>java GetAllRows
abc1 15 34
D:\code\JavaEE\Chapter3\ResultSet>

```

Figure 3.26: Showing the Output of `GetAllRows.java`

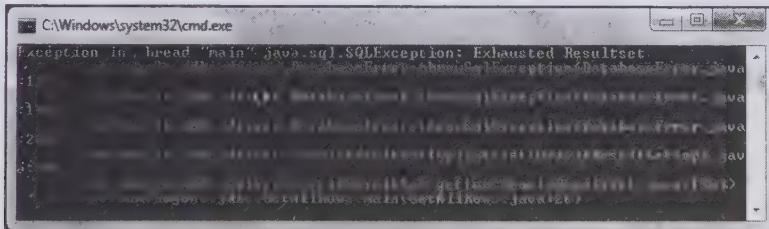
If we try to read the column values (without calling the `next()` method on `ResultSet`) obtained after executing the query, an exception is raised, as shown in Figure 3.27:



**Figure 3.27: Showing the Output Without Calling the next() Method**

We have created the `GetData.java` file in which the `next()` method on the `ResultSet` instance is not invoked; therefore, the `SQLException` exception is generated, as shown in Figure 3.27. If you see an exception as shown in Figure 3.27, it implies that you have attempted to read the data from the `ResultSet` immediately after obtaining it, without first calling the `next()` method. Note that even if you retrieve only one record (that is, one row), you still need to call the `next` row before reading column values. In such a case, the `rs.next()` method is used.

If you attempt to retrieve/read the column values even after the last record, the `SQLException` exception is raised. For example, if in Listing 3.7, you try to call the `getXXX` method after the while loop, an exception is raised as shown in Figure 3.28:



**Figure 3.28: Showing the Output of `GetAllRows.java` Accessing `ResultSet` after Last Record**

Therefore, if the `SQLException` exception is raised, you must check the control of your application to ensure that it does not read the data when the position of the `ResultSet` cursor is after the last record.

## Retrieving a Particular Column Using `ResultSet`

Apart from retrieving all the columns from a table, you can also retrieve data of a particular column from the `ResultSet`. Listing 3.8 shows how to retrieve data of the `col1` and `col2` columns from the `mytable` table (you can find the `GetData.java` file in the `code\JavaEE\Chapter3\Resultset` folder on the CD):

**Listing 3.8: Showing the Code for the `GetData.java` File**

```
package com.kogent.jdbc;

import java.sql.*;
/**
 * @author Suchita
 */
public class GetData {
    public static void main(String args[]) throws SQLException,
    ClassNotFoundException {
        //Get Connection
        Connection con=prepareConnection();

        // Obtain a Statement
        Statement st=con.createStatement();
```

```

String query = "select col3, col1 from mytable";
    //Execute the Query
ResultSet rs=st.executeQuery(query);

while (rs.next()){
    System.out.print (rs.getInt(1)+ "\t");
    System.out.println (rs.getString(2));
}
}//while
}//main

public static Connection prepareConnection()throws
SQLException, ClassNotFoundException {
    String driverClassName="oracle.jdbc.driver.OracleDriver";
    String url="jdbc:oracle:thin:@192.168.1.123:1521:XE";
    String username="scott";
    String password="tiger";

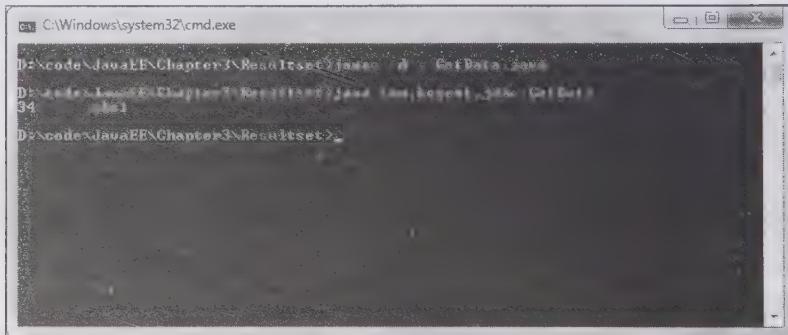
    //Load driver class
    Class.forName(driverClassName);

    // Obtain a connection
    return DriverManager.getConnection(url,username,password);
}//prepareConnection
}//class

```

In Listing 3.8, the SELECT statement is used to retrieve the data of the col1 and col3 columns from the mytable table, and the next () method is used to move the cursor position in the forward direction.

The output of Listing 3.8 is shown in Figure 3.29:



**Figure 3.29: Showing the Output of GetData**

You can change the query in Listing 3.8, as shown in the following code snippet:

```

query = "select col1, col3 from mytable";
to
query = "select col1 as c1, col3 as c3 from mytable";

```

Now, in the getXXX methods of ResultSet, you pass c1 and c3 instead of COL1 and COL3, respectively, as shown in the following code snippet:

```

System.out.print (rs.getString ("c1") + "\t");
System.out.println (rs.getInt("c3"));

```

The following code snippet shows the internal implementation of column name version of the getXXX method in a ResultSet:

```

public String getString(String s){
    int index=findColumn(s);
    return getString(index);
}

```

In the preceding code snippet, the findColumn(s) method of ResultSet returns the index number of the first found column, where the column name matches with the specified string column name.

The following are the possible exceptions that might be raised while executing this application:

- ❑ ClassNotFoundException, as shown in Listing 3.8.
- ❑ SQLException, if the column names used in the SQL query are not correct, as shown in Figure 3.30:

```

C:\Windows\system32\cmd.exe
D:\code\JavaEE\Chapter3\Resultset>java com.kogent.jdbc.GetData
Exception in thread "main" java.sql.SQLException: ORA-00904: "COL": invalid identifier
        at oracle.jdbc.driver.DatabaseError.throwSqlException(DatabaseError.java:112)
        at oracle.jdbc.driver.T4CTTIoer.processError(T4CTTIoer.java:450)
        at oracle.jdbc.driver.T4CTTIoer.processError(T4CTTIoer.java:394)
        at oracle.jdbc.driver.T4C8Oall.receive(T4C8Oall.java:523)
        at oracle.jdbc.driver.T4CStatement.doQuery(T4CStatement.java:218)
        at oracle.jdbc.driver.T4CStatement.executeQuery(T4CStatement.java:79)
        at oracle.jdbc.driver.OracleStatement.executeQuery(OracleStatement.java:1038)
        at oracle.jdbc.driver.OracleStatement.executeQuery(OracleStatement.java:1133)
        at oracle.jdbc.driver.OracleStatement.executeQuery(OracleStatement.java:1273)

```

**Figure 3.30: Showing the SQLException when Column Name is Incorrect**

In this case, verify that the column names used in the query are correct.

- ❑ The SQLException exception can be raised if the column types used with the getXXX methods of ResultSet are incorrect.

Figure 3.31 shows the SQLException exception that is raised while executing the GetData class, in which the value of a field is not internally converted into int:

```

C:\Windows\system32\cmd.exe
D:\code\JavaEE\Chapter3\Resultset>java com.kogent.jdbc.GetData
Exception in thread "main" java.sql.SQLException: Fail to convert to internal representation
        at oracle.jdbc.driver.DatabaseError.throwSqlException(DatabaseError.java:112)
        at oracle.jdbc.driver.T4CTTIoer.processError(T4CTTIoer.java:394)
        at oracle.jdbc.driver.T4C8Oall.receive(T4C8Oall.java:523)
        at oracle.jdbc.driver.T4CStatement.doQuery(T4CStatement.java:218)
        at oracle.jdbc.driver.T4CStatement.executeQuery(T4CStatement.java:79)
        at oracle.jdbc.driver.OracleCommonAccessor.getDouble(OracleCommonAccessor.java:106)
        at oracle.jdbc.driver.OracleResultSetImpl.getDouble(OracleResultSetImpl.java:111)
        at com.kogent.jdbc.GetData.main(GetData.java:24)
D:\code\JavaEE\Chapter3\Resultset>

```

**Figure 3.31: Showing the SQLException with Incorrect GetXXX() Method**

If the exception shown in Figure 3.31 is raised, you should check the column names used with the getXXX methods of ResultSet. Note that the preceding two exceptions are different, and to programmatically differentiate these exceptions and write supplementary code snippets, you need to depend on the SQL State and Error Code, which are vendor dependent. For example, if you observe the exception message shown in Figure 3.30, you find the exception message as ORA-00904. In this exception message, 00904 represents the error code. You use the getErrorCode() method of SQLException to obtain only the exception (error) code in an application.

- ❑ Instead of using column names with the getXXX methods of ResultSet, you can use column index. However, if improper column index is used, you can encounter the same exception as shown in Figure 3.31. In this case, verify that the column indexes used with the getXXX methods are correct.

You can use the preceding example to create a query for particular rows. In this case, you need to change the query, as shown in the following code snippet:

```

String query= "select * from mytable where COL1='Suchita'";
    Or
String query= "select * from mytable where COL2=36";
    Or
String query= "select * from mytable where COL2>=36";

```

## Working with Batch Updates

The batch update option allows you to submit multiple DDL/DML operations to a data source to process data simultaneously. Submitting multiple DDL/DML queries together, rather than submitting them individually, improves the performance of the query execution time. The Statement, PreparedStatement, and CallableStatement objects can be used to submit batch updates. It implies that the Statement, PreparedStatement, and CallableStatement objects are capable of keeping track of batches to be processed so that all the batches can be submitted together for processing. This feature has been introduced in the JDBC 2.0 specifications.

### Using Batch Updates with the Statement Object

Using the batch updates option with the Statement object allows you to submit a set of heterogeneous DDL/DML commands as a single unit (batch) to the underlying data source. When the Statement object is created using the `createStatement()` method of the Connection interface, it is associated with an empty batch. An application can use the `addBatch(String)` method to add a statement to the batch. After all the statements have been added to the batch, the application can invoke the `executeBatch()` method, if the batch needs to be submitted for processing. However, if the application does not submit the batch, it can invoke the `clearBatch()` method on the Statement object to remove all the statements.

#### *Describing the Batch Update Methods*

The following methods have been added in the Statement interface to support batch update:

- ❑ **addBatch (String)**—Adds one SQL statement to a batch. Only DDL and DML commands that return a simple update count can be added to the batch.
- ❑ **int [] executeBatch()**—Submits a batch to the underlying data source. When the batch is submitted to the data source, the statements in a batch are executed in the sequence in which they have been added to the batch.
- ❑ **clearBatch()**—Clears the batch before submitting it for processing. If the batch is executed successfully, the `executeBatch()` method returns an array of integer whose length is equal to the number of statements in the batch, and each element in the batch represents the respective statements update count. If the value of any element in this array is equal to `Statement.SUCCESS_NO_INFO`, it indicates that the statement has been executed successfully but the number of rows affected is unknown. In case a statement in a batch fails to be executed and produces a result set, further processing of the batch depends on the JDBC driver. In this case, the JDBC driver may still continue executing the batch or may terminate it. However, in most cases, the JDBC driver terminates the batch processing. Irrespective of the fact that the driver is implemented or not, if the batch fails to execute, the `executeBatch()` method throws `BatchUpdateException`. After the `executeBatch()` method is executed, the JDBC driver resets the batch.
- ❑ **The `java.sql.BatchUpdateException`**—Refers to an exception that is raised if the batch fails to execute. It is a subclass of `java.sql.SQLException`, which uses the `getUpdateCounts()` method of the current object and returns the `int` array, whose value can be:
  - **Less than the size of the batch**—Denotes that the driver has terminated the batch after the first failure of the execution of a query. Therefore, if the length of an array is `n`, it means that the first `n` statements in the batch have been executed successfully.
  - **Equal to the size of the batch**—Denotes that the driver has continued the batch execution process even after the batch has failed to execute. In this case, the value of each element in the array specifies the update count. If the array value pertains to the statement that has failed to execute, the array value becomes equal to the `Statement.EXECUTE_FAILED` field.

#### *Example of Using Batch Updates with the Statement Object*

Let's now look at an example of using batch updates with the Statement object. Let's create an application, called BatchUpdate, containing the `BatchUpdateEx1.java` file, which is used to perform batch updates. Listing 3.9 shows the code for the `BatchUpdateEx1.java` file (you can find the `BatchUpdateEx1.java` file in the `code\JavaEE\Chapter3\BatchUpdate` folder on the CD):

**Listing 3.9:** Showing the Code for the BatchUpdateEx1.java File

```

package com.kogent.jdbc;
import java.sql.*;
import java.util.*;
import java.io.*;
/**
 * @author Suchita
 */
public class BatchupdateEx1 {
    public static void main(String s[]) throws Exception {
        Driver d= (Driver)(Class.forName("oracle.jdbc.driver.OracleDriver").newInstance());
        Properties p=new Properties();
        p.put("user","scott");
        p.put("password","tiger");
        Connection con=d.connect("jdbc:oracle:thin:@192.168.1.123:1521:XE",p);
        Statement st=con.createStatement();
        //statement1
        st.addBatch("insert into emp(empno,sal,deptno) values("+s[0]+",1000,10)");
        //statement2
        st.addBatch("update emp set sal=2000 where empno="+s[0]);
        //statement3
        st.addBatch("insert into emp(empno,sal,deptno) values(202,1000,10)");
        //statement4
        st.addBatch("insert into emp(empno,sal,deptno) values(203,1000,10)");
        try {
            int[] counts=st.executeBatch();
            System.out.println("Batch Executed Successfully");
            for (int i=0;i<counts.length;i++){
                System.out.println("Number of records effected by statement"+(i+1)+": "+counts[i]);
            }//for
        }//try
        catch(BatchUpdateException e){
            System.out.println("Batch terminated with an abnormal condition");
            int[] counts=e.getUpdateCounts();
            System.out.println("Batch terminated at statement"+ (counts.length+1));
            for (int i=0;i<counts.length;i++) {
                System.out.println("Number of records effected by the statement"+(i+1)+" : "+counts[i]);
            }//for
        }//catch
        con.close();
    }//main
}//class

```

Listing 3.9 demonstrates how to perform batch updates using the Statement object. It also shows that the SQL statements added to the batch are executed in the order in which they have been added to the batch. In addition, it shows how to get update counts by using BatchUpdateException.

Figure 3.32 shows the output of Listing 3.9:

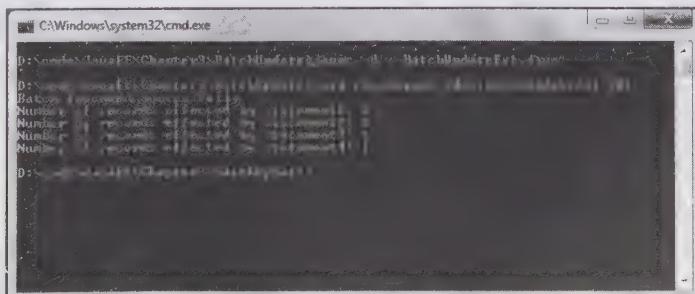
**Figure 3.32:** Showing the Output of BatchUpdateEx1.java

Figure 3.32 shows the successful execution of the batch used in Listing 3.9. You can run the preceding example again with argument value 203 to understand how the BatchUpdateException exception functions, as shown in Figure 3.33:

```
C:\Windows\system32\cmd.exe
D:\code\JavaEE\Chapter3\BatchUpdate>java -cp .;BatchUpdateEx1
Batch terminated with an abnormal condition
Batch terminated at statement
D:\code\JavaEE\Chapter3\BatchUpdate>
```

**Figure 3.33: Showing the Output of BatchUpdateEx1.java with a Different Parameter**

In Figure 3.33, a message specifying the termination of the batch execution is displayed as you try to insert a record with empno 203, which already exists (empno column of emp table is set with primary key constraint) in the database.

After learning to use batch updates with the `Statement` object, let's now learn how to implement batch updates using the `PreparedStatement` object.

## Using Batch Updates with the PreparedStatement Object

Using the batch updates feature with the `PreparedStatement` object is a bit different as compared to the `Statement` object. You can relate various input parameter values to a `PreparedStatement` object by using batch updates. The `PreparedStatement` interface provides various methods to support batch updates:

- ❑ `addBatch()`—Adds a set of input parameter values to a batch
- ❑ `int [] executeBatch()`—Executes a batch of statements in the specified data source
- ❑ `clearBatch()`—Clears the batch before submitting it for execution

Let's create an application, called `BatchUpdate`, to understand the concept better. In this application, you need to create the `BatchUpdateEx2.java` file, which is used to perform batch updates by using the `PreparedStatement` object.

The code for `batchUpdateEx2.java` is shown in Listing 3.10 (you can find the `BatchUpdateEx2.java` file on the CD in the `code\JavaEE\Chapter3\BatchUpdate` folder):

**Listing 3.10: Showing the Code for the BatchUpdateEx2.java File**

```
package com.kogent.jdbc;
import java.sql.*;
import java.util.*;
import java.io.*;
/**
 * @author Suchita
 */
public class BatchupdateEx2 {
    public static void main(String s[]) throws Exception {
        Driver d= (Driver) ( Class.forName("oracle.jdbc.driver.OracleDriver").newInstance());
        Properties p=new Properties();
        p.put("user","scott");
        p.put("password","tiger");
        Connection con=d.connect("jdbc:oracle:thin:@192.168.1.123:1521:XE",p);
        PreparedStatement ps= con.prepareStatement("insert into emp(empno,sal,deptno)
values(?, ?, ?)");
        ps.setInt(1,301);
        ps.setDouble(2,1000);
        ps.setInt(3,10);
        ps.addBatch();

        ps.setInt(1,302);
```

```

ps.setDouble(2,2000);
ps.setInt(3,10);
ps.addBatch();
try {
    int counts[] = ps.executeBatch();
    System.out.println("Batch Executed Successfully");
    for (int i=0;i<counts.length;i++){
        System.out.println("Number of records effected by statement"+(i+1)+":"
                           +counts[i]);
    }//for
}//try
catch(BatchUpdateException e){
    System.out.println("Batch terminated with an abnormal condition");
    int[] counts=e.getUpdateCounts();
    System.out.println("Batch terminated at statement"+ (counts.length+1));
    for (int i=0;i<counts.length;i++) {
        System.out.println("Number of records effected by the statement"+
                           (i+1)+":"+counts[i]);
    }//for
}//catch
con.close();
}//main
}//class

```

The example shown in Listing 3.10 demonstrates how to perform batch updates using `PreparedStatement` and how to get update counts from `BatchUpdateException`. It also shows that the SQL statements added to the batch are executed in the order in which they have been added.

Figure 3.34 shows the output of Listing 3.10:

```

D:\nodes\Java\Chapters\3\BatchUpdate>java com.kogent.jdbc.BatchUpdateEx2
Batch Executed Successfully
Number of records effected by statement1:-2
Number of records effected by statement2:-2
D:\nodes\Java\Chapters\3\BatchUpdate>

```

**Figure 3.34: Showing the Output of BatchUpdateEx2.java**

When you execute Listing 3.10, SQL statements specified in the batch are executed and the emp table is modified. Figure 3.35 shows the content of the emp table after the batch updates have been performed:

ENAME	SAL	DEPNO
SA	2400	10
MILLER	2400	10
BLAKE	2400	10

**Figure 3.35: Showing the Content of the emp Table**

**NOTE**

In Figure 3.34, update counts are shown as -2; whereas, the records are inserted successfully (Figure 3.35). In such cases, the update count value is equal to Statement.SUCCESS\_NO\_INFO, which indicates that the statement has been executed successfully but the number of rows affected is unknown.

## Describing SQL 99 Data Types

SQL-I999 specifies the SQL-I999 object model that adds UDTs to SQL. There are two types of UDTs: distinct and structured. A distinct type is based on a built-in data type, such as integer and a structured type has an internal structure, such as address that might contain the details of street, state, and postal code attributes.

The data types available in SQL-I999 types are as follows:

- BLOB data type
- CLOB data type
- Struct data type
- Array data type
- REF data type

All these types are packaged in the `java.sql` package, which provides the classes and interfaces to hold these objects. Let's describe these UDTs available in SQL-I999 types in detail.

### Describing the BLOB Data Type

A BLOB is a built-in data type used to store binary large objects, such as images, audios, or multimedia clips, as column values in a database table. The `java.sql` package provides the `Blob` interface to represent BLOB values. BLOB values can be implemented by using the SQL locator. This locator indicates that a `Blob` object contains a pointer to point to SQL BLOB values in a database. `Blob` objects provide logical pointers to the binary large objects rather than copies of the objects. Most of the databases process only one data page into the memory at a time; i.e., the whole BLOB does not need to be processed and stored in memory just to access the first few bytes of the `Blob` object. The lifetime of the `Blob` object is based on the lifetime of a transaction as well as the database in use.

The `Blob` interface provides various methods to store and retrieve BLOB values in an application.

Table 3.25 describes the methods provided by the `Blob` interface:

**Table 3.25: Methods of the Blob Interface**

Method	Description
<code>public InputStream getBinaryStream()</code>	Retrieves the BLOB value, stored by the <code>Blob</code> object, as a stream.
<code>public byte[] getBytes(long pos, int length)</code>	Retrieves all or some portion of the BLOB values stored by the <code>Blob</code> object.
<code>public long length()</code>	Returns the number of bytes of the BLOB values taken by the <code>Blob</code> object.
<code>public long position(Blob pattern, long start)</code>	Returns the byte position of the BLOB value designated by the <code>Blob</code> object.
<code>public long position(byte[] pattern, long start)</code>	Returns the position of the BLOB value in an array of bytes designated by the <code>Blob</code> object.
<code>public OutputStream setBinaryStream(long pos)</code>	Retrieves the stream used to write the BLOB value.
<code>public int setBytes(long pos, byte[] bytes)</code>	Writes the BLOB value in an array of bytes designated by the <code>Blob</code> object, starting at position <code>pos</code> , and returns the number of bytes written. The position and the number of bytes to be written must be specified in this method.

**Table 3.25: Methods of the Blob Interface**

Method	Description
public int setBytes(long pos, byte[] bytes, int offset, int len)	Sets all or part of the specified byte array to the BLOB value designated by the Blob object and returns the number of bytes written to the BLOB value.
public void truncate(long len)	Truncates the BLOB value represented by the Blob object.

Now let's use these methods to store BLOB values into the database. The following heading describes the following tasks:

- Store BLOB values into the database
- Read BLOB values

Now, let's discuss each of them in detail.

#### *Storing BLOB values*

The Blob interface of JDBC does not provide any database-independent mechanism to construct a Blob instance; and therefore, you need to either write your own implementation or depend on the implementation of the driver vendor. If you are working with a previous version of JDBC 4.0, you can use the setBinaryStream (...) method of the PreparedStatement and CallableStatement interfaces to construct a Blob instance as an InputStream of the specified length. The constructed Blob instance is passed as parameter to the setBlob() method of the the PreparedStatement and CallableStatement interfaces to store BLOB data in the database. Let's create an application called Blob to understand the concept better. This application contains a Java file named InsertBlobEx.java, which is used to store BLOB values, as shown in Listing 3.11 (you can find the InsertBlobEx.java file in the code\JavaEE\Chapter3\Blob folder on the CD):

**Listing 3.11: Showing the Code for the InsertBlobEx.java File**

```
package com.kogent.jdbc;
import java.sql.*;
import java.util.*;
import java.io.*;
/**
 * @author Suchita
 */
public class InsertBlobEx
{
    public static void main(String s[]) throws Exception
    {
        Driver d= (Driver) ( Class.forName(
            "oracle.jdbc.driver.OracleDriver").newInstance());
        Properties p=new Properties();
        p.put("user","scott");
        p.put("password","tiger");
        Connection con=d.connect("jdbc:oracle:thin:@192.168.1.123:1521:XE",p);
        PreparedStatement ps= con.prepareStatement(
            "insert into personaldetails(empno,photo) values(?,?)");
        ps.setInt(1,Integer.parseInt(s[0]));
        File f=new File("MyImg101.gif");
        FileInputStream fis= new FileInputStream(f);
        ps.setBinaryStream(2,fis, (int)f.length());
        int i=ps.executeUpdate();
        System.out.println("Record inserted successfully, count : "+i);
        con.close();
    }//main
}//class
```

You should create the personaldetails table before executing the code shown in Listing 3.11. The following code snippet shows the command used to create the personaldetails table in the Oracle database:

```
create table personaldetails(empno number, photo BLOB);
```

When you execute the code given in Listing 3.11, an image is inserted into the Oracle database. The image value is stored into the database by using the `setBinaryStream()` method of the `PreparedStatement` interface. Figure 3.36 shows the output of the `InsertBlobEx` class:

```
D:\code\JavaEE\Chapter3\blob>java -cp .;lib/* InsertBlobEx
D:\code\JavaEE\Chapter3\blob>java -cp .;lib/* InsertBlobEx
Record inserted successfully - count = 3
D:\code\JavaEE\Chapter3\blob>
```

**Figure 3.36: Displaying the Output of the `InsertBlobEx` Class**

In the earlier versions of JDBC 4.0, the `Blob` interface did not provide any database-independent mechanism to construct the `Blob` instance; therefore, to solve this problem, JDBC 4.0 APIs provide a `createBlob()` method in the `java.sql.Connection` interface. The `createBlob` method allows to create a `Blob` object to which the bytes can be set and passed as a parameter into the `setBlob()` method of the `PreparedStatement` and `CallableStatement` interfaces.

The following code snippet creates a `Blob` object, `b`, in JDBC 4.0:

```
Connection con= ... //obtain the connection
Blob b=con.createBlob(); //creates an empty Blob (blob object with no bytes)
b.setBytes(1, data); //here data is a byte[]
Now, the above created Blob object can be used with setBlob() method
```

After learning how to store the value of a `Blob` object in a database using the `getBinaryStream()` method, let's now learn how to retrieve the value of the `Blob` object from a database.

#### *Reading a BLOB value*

You can retrieve a BLOB value from a database by using the `Blob` object. Let's create an application called `Blob`, which contains a .java (`ReadBlobEx.java`) file used to read BLOB values to understand the concept better. Listing 3.12 shows the code of the `ReadBlobEx.java` file (you can find the `ReadBlobEx.java` file in the `code\JavaEE\Chapter3\Blob` folder on the CD):

#### **Listing 3.12: Showing the `ReadBlobEx.java` File**

```
package com.kogent.jdbc;
import java.sql.*;
import java.util.*;
import java.io.*;
/**
 * @author Suchita
 */
public class ReadBlobEx {
    public static void main(String s[]) throws Exception {
        Driver d= (Driver) (Class.forName(
            "oracle.jdbc.driver.OracleDriver").newInstance());
        Properties p=new Properties();
        p.put("user","scott");
        p.put("password","tiger");
        Connection con=d.connect(
            "jdbc:oracle:thin:@192.168.1.123:1521:XE",p);
```

```

Statement st=con.createStatement();
ResultSet rs=st.executeQuery( "select * from personaldetails");

while (rs.next()) {

    int empno=rs.getInt(1);
    InputStream is=rs.getBinaryStream(2);

    FileOutputStream fos=new FileOutputStream("MyImg"+empno+".gif");
    int i=is.read();

    while (i!=-1){
        fos.write(i);
        i=is.read();
    }//while
}//while
System.out.println("Image's retrived");
con.close();
}//main
}//class

```

Listing 3.12 uses the getbinaryStream() method provided by the Blob interface to retrieve BLOB values (the inserted image in this case). Figure 3.37 shows the output of the ReadBlobEx class:

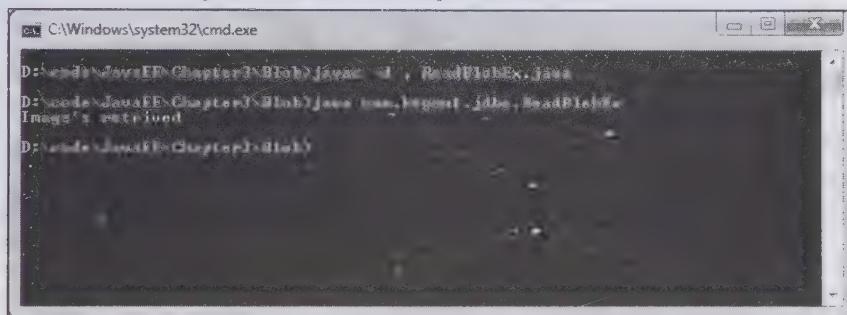


Figure 3.37: Displaying the Output of the ReadBlobEx Class

## Describing the CLOB Data Type

CLOB is a built-in data type used to store large amount of textual data. It can also be referred as a collection of data stored as a single entity in a DBMS. CLOB stores the values of large character objects as a column value of a row in a database. The java.sql package provides the Clob interface to represent the CLOB values. A Clob object contains a SQL locator to point to the CLOB data in a database. Similar to the Blob object, the lifetime of the Clob object is based on the lifetime of a transaction and the database in use.

The Clob interface provides various methods to store and retrieve CLOB values in a database, as described in Table 3.26:

Table 3.26: Methods of the Clob Interface

Method	Description
public InputStream getAsciiStream()	Retrieves a CLOB value designated by the Clob object as well as data stream.
public Reader getCharacterStream()	Retrieves the CLOB value as the java.io.Reader object.
public String getSubString(long pos, int length)	Retrieves a copy of the substring specified in the method. The CLOB value must be designated by the Clob object.
public long length()	Retrieves the number of characters from the CLOB value designated by the Clob object.
public long position(Clob searchstr, long start)	Retrieves the position of the character from the CLOB value by starting from the value of the start parameter.

**Table 3.26: Methods of the Clob Interface**

Method	Description
public long position(String searchstr, long start)	Retrieves the character position in the CLOB value where the searchstr String appears. The searchstr String represents the String to be searched in the CLOB value.
public OutputStream setAsciiStream(long pos)	Retrieves the stream to be written into the CLOB value. The starting position of the stream must be specified by the pos parameter of the method. In addition, the CLOB value must be designated by the Clob object.
public Writer setCharacterStream(long pos)	Retrieves the stream used to write the CLOB value, starting from the position specified by the pos parameter of the method.
public int setString(long pos, String str)	Writes the specified string, passed as the str parameter, into the CLOB value at the specified position, pos.
public int setString(long pos, String str, int offset, int len)	Writes the specified string of the len length into the CLOB value, starting from a specified position.
public void truncate(long len)	Truncates the CLOB value for length of len characters, associated with the Clob object.

The `java.sql.Clob` interface provides a logical pointer to the character large object rather than a copy of the large object. Let's now discuss how to retrieve CLOB values from a database and how to store these values in the database.

Now let's understand them in detail.

#### *Storing CLOB Values*

Similar to the Blob interface, the Clob interface provides no database-independent mechanism to construct the Clob instance, so you need to either write your own implementation or depend on the implementation of the vendor. If you are working with a previous version of JDBC 4.0, you can use the `setCharacterStream(...)` method of the `PreparedStatement` and `CallableStatement` interfaces to construct a Clob instance as a `ReaderObject` of specified length. You can store the CLOB data in a database by passing the `Clob` instance as a parameter to the `setClob()` method of the `PreparedStatement` and `CallableStatement` interfaces.

Let's create an application called `Clob` to understand the concept better. This application contains the `InsertEmployeeProfile.java` file to store CLOB values.. Listing 3.13 shows the `InsertEmployeeProfile.java` file (you can find the `InsertEmployeeProfile.java` file in the code\JavaEE\Chapter3\Clob folder on the CD):

**Listing 3.13:** Showing the Code for the `InsertEmployeeProfile.java` File

```
package com.kogent.jdbc;
import java.sql.*;
import java.util.*;
import java.io.*;
/**
 * @author Suchita
 */
public class InsertEmployeeProfile {
    public static void main(String s[]) throws Exception {
        Driver d= (Driver) (Class.forName(
            "oracle.jdbc.driver.OracleDriver").newInstance());
        Properties p=new Properties();
        p.put("user","scott");
        p.put("password","tiger");
        Connection con=d.connect("jdbc:oracle:thin:@192.168.1.123:1521:XE",p);
        PreparedStatement ps= con.prepareStatement(
            "insert into empprofiles values(?,?)");
        ps.setInt(1,Integer.parseInt(s[0]));
        File f=new File(s[1]);
        FileReader fr= new FileReader(f);
        ps.setCharacterStream(2,fr, (int)f.length());
    }
}
```

```

    int i=ps.executeUpdate();
    System.out.println("Record inserted successfully , count : "+i);
    con.close();
}//main
}//class

```

The user needs to create a table (empprofiles), which contains the employee profile to store the employee details into the database by using the CLOB value. In other words, to execute Listing 3.13, you first need to create the empprofiles table in the Oracle database, as shown in the following code snippet:

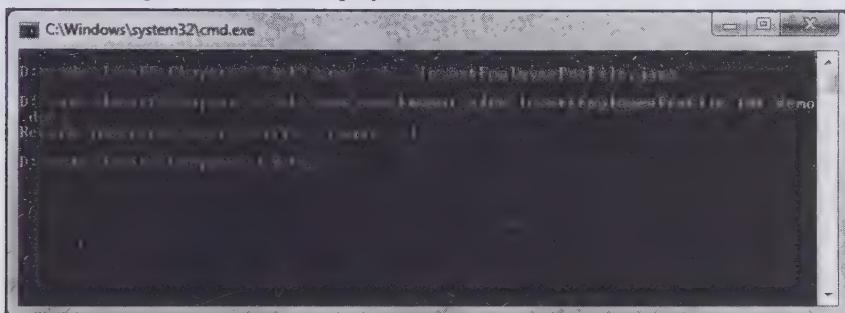
```

create table empprofiles (
    empno number,
    profile CLOB );

```

Listing 3.13 shows how to store a CLOB value into the database by using the `setCharacterStream()` method provided by the `PreparedStatement` interface. We are storing a word document in the Oracle database. The document contains all the details of a particular employee.

Figure 3.38 shows the output of the `InsertEmployeeProfile` class:



**Figure 3.38: Displaying the Output of the InsertEmployeeProfile Class**

The JDBC 4.0 APIs provide the `createClob()` method in `java.sql.Connection`. The `createClob` method allows you to create an empty `Clob` object. The byte data to the empty `Clob` object can be added or set by invoking the `setString()` or other relevant methods depending on the type of the byte data that you want to add to the object. The following code snippet shows how to create a `Clob` object:

```

Connection con=... //obtain the connection
Clob b=con.createClob(); //creates an empty Clob (Clob object with no bytes)
b.setString(1, data); //where data is a String
Now, the above created Clob object can be used with setBlob() method

```

After learning how to store CLOB values into a database by using the `getCharacterStream()` method, let's learn to retrieve CLOB values from the database.

#### Reading CLOB Values

The `Clob` interface provides the `getClob()` method to access the CLOB values stored in a database. You can also retrieve CLOB values from a database by using the `Clob` object.

Let's create an application called `Clob` to retrieve CLOB values. In this application, you need to create the `GetEmployeeProfile.java` file, as shown in Listing 3.14 (you can find the `GetEmployeeProfile.java` file in the `code\JavaEE\Chapter3\Clob` folder on the CD):

**Listing 3.14:** Showing the Code for the `GetEmployeeProfile.java` File

```

package com.kogent.jdbc;

import java.sql.*;
import java.util.*;
import java.io.*;
/**
 * @author Suchita
 */
public class GetEmployeeProfile {
    public static void main(String s[]) throws Exception {

```

```

Driver d= (Driver) Class.forName(
    "oracle.jdbc.driver.OracleDriver").newInstance();
Properties p=new Properties();
p.put("user","scott");
p.put("password","tiger");
Connection con=d.connect("jdbc:oracle:thin:@192.168.1.123:1521:XE",p);
Statement st=con.createStatement();
ResultSet rs=st.executeQuery(
    "select profile from empprofiles where empno="+s[0]);
while (rs.next()) {
    Reader r=rs.getCharacterStream(1);
    FileWriter fw=new FileWriter("Profileof"+s[0]+".doc");
    int i=r.read();
    while (i!=-1){
        fw.write(i);
        i=r.read();
    }
}
System.out.println("Profile retrieved");
con.close();
}
}

```

Listing 3.14 is used to access the details of the employee by using the `getCharacterStream()` method of the `ResultSet` interface.

Figure 3.39 shows the output of the `GetEmployeeProfile` class:

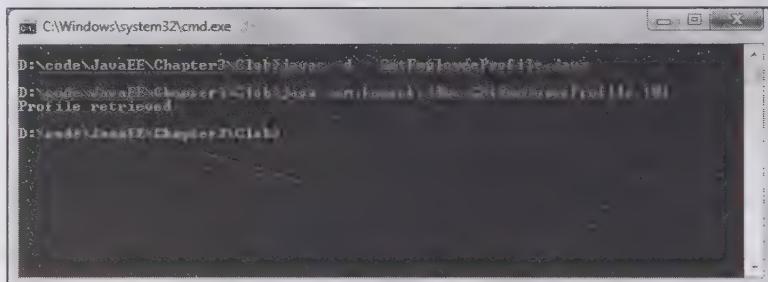


Figure 3.39: Displaying the Output of the `GetEmployeeProfile` Class

#### **NOTE**

The `Blob` and `Clob` objects can persist even after the transaction in which they are created is complete. Moreover, these objects may persist for a long time in case of lengthy transactions. This results in shortage of resources for the application using these objects. To overcome this problem, JDBC 4.0 provides the `free()` method of `java.sql.Blob` and `java.sql.Clob`, which you can use to release the `Blob` and `Clob` objects when they are not required by the application.

## Describing the Struct (Object) Data Type

Most of the databases now enable you to create Struct data types (also known as structured type), which are used to define complex data types. This is required in case you want to create a UDT in a database. For example, you might need to create a UDT to represent the address of an employee in a single column. The following code snippet shows the syntax to create a structured type in a database:

```
create type <name> as OBJECT (<variable name> <type>, ...);
```

After creating a structured type, you can reference it to create the required UDT. The following code snippet shows the example of creating a UDT:

```

create type empaddress as OBJECT (
    flatno number,
    street varchar2(20),
    city varchar2(15),
    state varchar2(10),
    pincode number
);

```

The preceding code snippet creates a structured type named empaddress, which can store the values of the flatno and pincode fields of type number, and the street, city, and state fields of type varchar2. It also shows how to create a table with the empaddress type column and insert record into that table.

After learning to create a structured type, let's now learn how to store and retrieve the values of structured types. JDBC provides two approaches to store and retrieve the values of structured types:

- A UDT in Java to represent the database object type
- The java.sql.Struct interface

Let's learn about these in detail next.

#### *Using User-Defined Object Types in Java to Represent Database Object Types*

JDBC 2.0 specification includes support for UDT by providing various methods in the PreparedStatement, CallableStatement, and ResultSet interfaces of JDBC API.

Table 3.27 shows the methods to support UDT:

**Table 3.27: Methods Supporting UDT along with their Interfaces**

Method	Interface
setObject (int parameterindex, Object o)	java.sql.PreparedStatement
setObject (int parameterindex, Object o, int targetSqltype)	java.sql.PreparedStatement
setObject (int parameterindex, Object o, int targetSqltype, int scale)	java.sql.PreparedStatement
getObject (int columnindex)	java.sql.ResultSet
getObject (int columnindex, java.util.Map m)	java.sql.ResultSet
getObject (String columnName)	java.sql.ResultSet
getObject (String columnName, java.util.Map m)	java.sql.ResultSet
getObject (int parameterindex)	java.sql.CallableStatement
getObject (int parameterindex, java.util.Map m)	java.sql.CallableStatement
getObject (String parameterName)	java.sql.CallableStatement
getObject (String parameterName, java.util.Map m)	java.sql.CallableStatement

In a JDBC application, UDTs must conform to the following rules:

- They should be declared as public non-abstract classes.
- They should be subtypes of the java.sql.SQLData interface. The java.sql.SQLData interface declares the following methods:
  - **String getSQLTypeName()** – Returns the fully qualified name of the SQL UDT represented by the Struct object. This method is called by the JDBC driver to retrieve the name of the UDT instance, which is mapped to this instance of the java.sql.SQLData interface.
  - **void readSQL (SQLInput stream, String typeName)** – Populates the current Struct object with data read from a database. This method generally reads each statement of the SQL type from the given input stream. This is done by calling a method of the SQLInput interface to read the data in the order they appear in the SQL definition of the type. It then assigns the data to appropriate fields of the Struct object. The JDBC driver initializes the input stream with a type map before calling this method, which is used by the appropriate SQLInput reader method on the stream.
  - **void writeSQL (SQLOutput stream)** – Writes the current object to the specified SQLOutput stream, which converts it back to its SQL value in the data source. The implementation of the method generally writes each element of the SQL type to the given output stream. This is done by calling a method of the SQLOutput interface to write each item in the order they appear in the SQL definition of the type.
- They should have a no argument constructor.

Let's create an application called `SQLDataInterface` to understand the concept better. In this application, you need to create the `EmployeeAddress.java` file, which is used to implement the `SQLData` interface to represent the `empaddress` type created in the preceding code snippet.

Listing 3.15 shows the content of the `EmployeeAddress.java` file (you can find the `EmployeeAddress.java` file in the `code\JavaEE\Chapter3\SQLDataInterface` folder on the CD):

**Listing 3.15:** Showing the Code for the `EmployeeAddress.java` File

```
package com.kogent.jdbc;

import java.sql.*;
/**
 * @author Suchita
 */
public class EmployeeAddress implements SQLData {
    public EmployeeAddress(){}
    public void writeSQL(SQLOutput so) throws SQLException {
        so.writeInt(fno);
        so.writeString(street);
        so.writeString(city);
        so.writeString(state);
        so.writeInt(pin);
    }//writeSQL
    public void readSQL(SQLInput si, String name) throws SQLException{
        fno=si.readInt();
        street=si.readString();
        city=si.readString();
        state=si.readString();
        pin=si.readInt();
        typename=name;
    }//readSQL
    public String getSQLTypeName()
    {return typename;}
    public void setFlatno(int i){fno=i;}
    public void setStreet(String s){street=s;}
    public void setCity(String s){city=s;}
    public void setState(String s){state=s;}
    public void setPin(int i){pin=i;}
    public void setTypeNames(String s){typename=s;}
    public int getFlatno(){return fno;}
    public String getStreet(){return street;}
    public String getCity(){return city;}
    public String getState(){return state;}
    public int getPin(){return pin;}
    String street,city,state, typename;
    int fno,pin;
}//class
```

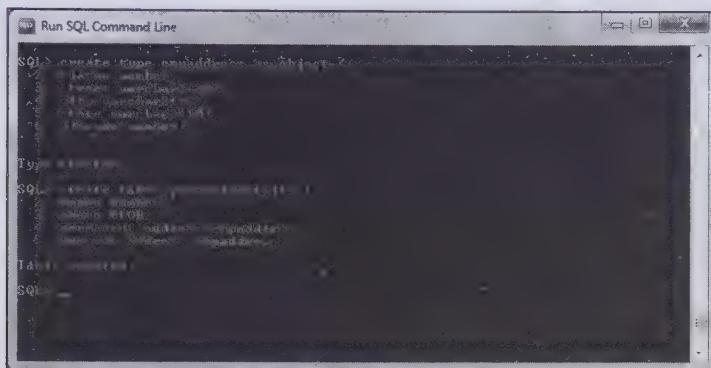
Listing 3.15 shows the JDBC UDT to represent the `empaddress` type that holds the values of the `flatno`, `street`, `city`, `state`, and `pin` fields.

#### *Implementing the `java.sql.Struct` Interface*

Now, let's understand the Struct data type by creating an application called `EmployeeAddress`. In this application, you need to create a .java (`InsertPersonalDetails.java`) file that stores an `EmployeeAddress` object in the Oracle database. You can copy the `EmployeeAddress.java` file in the `EmployeeAddress` application directory. The application is available on the CD in the `code\JavaEE\Chapter3\EmployeeAddress` folder. You need to perform the following steps to implement the `EmployeeAddress` application:

- ❑ Create an object type named `empaddress` and a database table named `personaldetails` in the Oracle database
- ❑ Create a java file `InsertPersonalDetails.java`, which inserts the object of the `EmployeeAddress` class in the database Oracle

Let's start creating an object type and a database table. Figure 3.40 shows the SQL commands to create the empaddress object type and personaldetails table in the Oracle database using the Run SQL Command Line prompt of Oracle:



**Figure 3.40: Creating Tables using Run SQL Command Line**

After creating the object type and table, you need to create a java file, `InsertPersonalDetails.java`, which inserts the object of the `EmployeeAddress` class into the Oracle database. The `InsertPersonalDetails.java` file is shown in Listing 3.16 (you can find the `InsertPersonalDetails.java` file in the code\JavaEE\Chapter3\EmployeeAddress folder on the CD):

**Listing 3.16:** Showing the Code for the `InsertPersonalDetails.java` File

```
package com.kogent.jdbc;

import java.sql.*;
import java.util.*;
import java.io.*;
/**
 * @author Suchita
 */
public class InsertPersonalDetails {

    public static void main(String s[]) throws Exception {
        Driver d= (Driver) ( Class.forName(
            "oracle.jdbc.driver.OracleDriver").newInstance());

        Properties p=new Properties();
        p.put("user","scott");
        p.put("password","tiger");
        Connection con=d.connect("jdbc:oracle:thin:@192.168.1.123:1521:XE",p);
        PreparedStatement ps= con.prepareStatement(
            "insert into personaldetails(empno,photo,permanent_address) values(?, ?, ?)");
        /*
        Here we consider Present Address is same as Permanent Address, so we want to
        insert null in place of Present Address
        */
        ps.setInt(1,7934);
        File f=new File("MyImage.gif");
        FileInputStream fis= new FileInputStream(f);
        ps.setBinaryStream(2,fis, (int)f.length());

        EmployeeAddress addr=new EmployeeAddress();
        addr.setFlatno(106);
        addr.setCity("Hyd");
        addr.setStreet("SRN");
        addr.setPin(5000049);
        addr.setState("AP");
        addr.setTypeName("EMPADDRESS");
    }
}
```

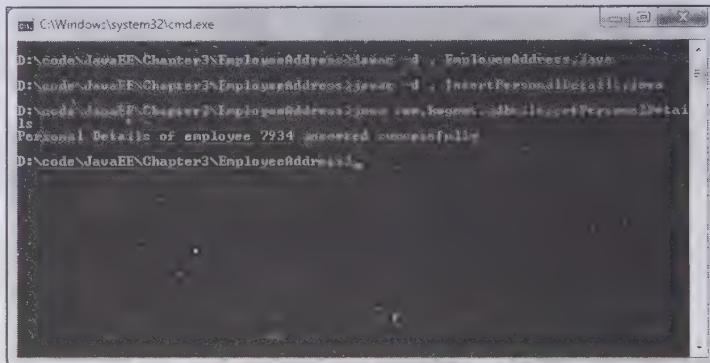
```

        ps.setObject(3,addr);
        int i=ps.executeUpdate();
        System.out.println("Personal details of employee 7934 inserted successfully");
        con.close();
    } //main
}//class

```

Listing 3.16 uses the `setObject()` method of the `PreparedStatement` interface to store the `EmployeeAddress` object into the Oracle database.

After creating all the required files, let's execute the `InsertPersonalDetails.java` file, as shown in Figure 3.41:



**Figure 3.41: Showing the Output of the InsertPersonalDetails.java File**

Figure 3.41 shows the output of Listing 3.16, which inserts a record into the `personaldetails` table.

#### NOTE

*To update the Object type, you can use the same `setObject()` method as used in Listing 3.16.*

Let's now learn how to retrieve the object type value by creating an application that contains the `GetEmployeeAddress.java` file. Listing 3.17 shows the `GetEmployeeAddress.java` file (you can find the `GetEmployeeAddress.java` file in the `code\JavaEE\Chapter3\EmployeeAddress` folder on the CD):

**Listing 3.17: Showing the Code for the GetEmployeeAddress.java File**

```

package com.kogent.jdbc;
import java.sql.*;
import java.util.*;
import java.io.*;
/**
 * @author Suchita
 */
public class GetEmployeeAddress {
    public static void main(String s[]) throws Exception {
        Driver d= (Driver) (Class.forName(
            "oracle.jdbc.driver.OracleDriver").newInstance());
        Properties p=new Properties();
        p.put("user","scott");
        p.put("password","tiger");
        Connection con=d.connect("jdbc:oracle:thin:@192.168.1.123:1521:XE",p);
        Statement st=con.createStatement();
        ResultSet rs=st.executeQuery(
            "select permanent_address from personaldetails where empno="+s[0]);
        if (rs.next()){
            HashMap map=new HashMap();
            map.put("EMPADDRESS", EmployeeAddress.class);
            EmployeeAddress addr=(EmployeeAddress)rs.getObject(1,map);
            System.out.println("Employee Found Address:");
            System.out.println("Flatno : "+addr.getFlatno());
            System.out.println("Street : "+ addr.getStreet());
        }
    }
}

```

```

        System.out.println("Pin    : "+addr.getPin());
    } //if
    con.close();
} //main
} //class

```

Listing 3.17 shows the code to retrieve the object type value from the Oracle database and represent it as the EmployeeAddress type of object in Java.

Figure 3.42 shows the output of Listing 3.17:

```

C:\Windows\system32\cmd.exe
D:\code\JavaEE\Chapter3\EmployeeAddress>java -cp .;con.kogent.jdbc.GetEmployeeAddress
7934
Employee Name: Purnima
Flatno : 106
Street : SRN
Pin   : 5000049
D:\code\JavaEE\Chapter3\EmployeeAddress>

```

**Figure 3.42: Showing the Output of the GetEmployeeAddress.java File**

Figure 3.42 shows the output of Listing 3.17 that retrieves the object type value from the Oracle database by using the EmpAddress type.

In addition to UDTs, JDBC 2.0 includes a built-in type, `java.sql.Struct`, which represents the SQL structured type. A Struct object contains values for each attribute associated with the Struct data type. By default, an instance of Struct is valid until the application has a reference of its instance. The Struct interface provides certain methods to work with the Struct objects.

Table 3.28 describes the methods of the Struct interface:

**Table 3.28: Methods of the Struct Interface**

Method	Description
<code>public Object[] getAttributes()</code>	Retrieves the structured type attributes and ordered values. Struct values are represented by the Struct object.
<code>public Object[] getAttributes(Map map)</code>	Retrieves the structured type attributes and ordered values in an array. Struct values are represented by the Struct object.
<code>public String getSQLTypeName()</code>	Retrieves the SQL type name and SQL type of the SQL Structured type associated with the Struct object.

The Struct types can be used with JDBC programs to communicate with a database. Listing 3.18 shows how to use the Struct UDTs in a database (you can find the `GetEmployeeAddressUsingStruct.java` file in the `code\JavaEE\Chapter3\Struct` folder on the CD):

**Listing 3.18: Showing the Code for the GetEmployeeAddressUsingStruct.java File**

```

package com.kogent.jdbc;

import java.sql.*;
import java.util.*;
import java.io.*;
/**
 * @author Suchita
 */

```

```

public class GetEmployeeAddressUsingStruct {
    public static void main(String s[]) throws Exception {
        Driver d= (Driver) Class.forName(
            "oracle.jdbc.driver.OracleDriver").newInstance();
        Properties p=new Properties();
        p.put("user","scott");
        p.put("password","tiger");
        Connection con=d.connect("jdbc:oracle:thin:@192.168.1.123:1521:XE",p);
        Statement st=con.createStatement();
        ResultSet rs=st.executeQuery(
            "select permanent_address from personaldetails where empno='"+s[0]+');

        if (rs.next()){
            System.out.println("Employee Found: Address");
            Struct struct=(Struct)rs.getObject(1);
            Object addr[]=struct.getAttributes();
            System.out.println("Flatno : "+addr[0]);
            System.out.println("Street : "+ addr[1]);
            System.out.println("Pin : "+addr[4]);
        }
        con.close();
    }
}

```

The output of Listing 3.18, in which we have used the Struct UDT, is shown in Figure 3.43:

```

C:\Windows\system32\cmd.exe
D:\>java -cp D:\Java\bin;D:\Java\lib\*.jar GetEmployeeAddressUsingStruct
D:\>java -cp D:\Java\bin;D:\Java\lib\*.jar GetEmployeeAddressUsingStruct
t 7733
Employee Found: Address
Flatno : 106
Street : 8th Main
Pin : 560001
D:

```

Figure 3.43: Showing the Output of the GetEmployeeAddressUsingStruct.java File

### Describing the Array Data Type

Array, one of the SQL 99 data types, offers you the facility to include an ordered list of values within a column. The `java.sql` package provides a `java.sql.Array` interface to store the values of the array types. The array object can be implemented by using a SQL locator, which indicates that the array object contains a logical pointer to locate the array value in a database. Since array objects contain UDTs, you need to create a custom mapping between the Class object for the class implementing the `SQLData` interface and the UDTs. You need to perform the following steps to create a custom mapping:

- Create a class that implements the `SQLData` interface. The methods of the `SQLData` interface are used by the data type that need custom mapping.
- Define a Map type that contains the SQL types for UDTs and the classes that implement the `SQLData` interface.

The array interface provides some methods to create custom mapping between the classes and UDTs. These methods are described in Table 3.29:

**Table 3.29: Methods of the Array Interface**

Method	Description
public Object getArray()	Retrieves the content of the array object. Array values must be designated by the array objects.
public Object getArray(long index, int count)	Retrieves a portion of the array value specified by the index. The array value must be designated by the array object.
public Object getArray(long index, int count, Map map)	Retrieves a portion of the array value specified by the index. It also specifies the number of elements that you can access. The array value must be designated by the array object.
public Object getArray(Map map)	Retrieves the content of the SQL array value. The array value is designated by the array object.
public int getBaseType()	Retrieves the JDBC elements present in an array. The array value must be designated by the array object.
public String getBaseTypeName()	Retrieves the name of the SQL elements in an array. The array value must be designated by the array object.
public ResultSet getResultSet()	Retrieves the SQL ResultSet elements present in an array. The array value must be designated by the array object.
public ResultSet getResultSet(long index, int count)	Retrieves the sub array elements, starting at the index of the array. The array value must be designated by the array object.
public ResultSet getResultSet(long index, int count, Map map)	Retrieves the sub array elements, starting at the index of the array. The sub array also contains a count of the elements. The array value must be designated by the array object.
public ResultSet getResultSet(Map map)	Retrieves the SQL array elements stored in the specified Map instance.

The Array type contains more than one value of the same data type. The syntax to create an array type in the database is as follows:

`create Type <type name> as VARRAY(<length>) of <type>`

To insert a record by using the Statement interface, you do not need to use the `java.sql.Array` interface. Instead, you can execute the preceding query by using the `executeUpdate()` method. You can use the `setArray()` method of the PreparedStatement interface to bind an array object as a parameter to a statement. However, in earlier versions of JDBC, the Array interface did not provide any database-independent mechanism to construct an array instance. In such cases, you need to either write your own implementation or depend on the implementation of the driver vendor.

Listing 3.19 shows how to use the SQL array types with the PreparedStatement objects (you can find the `InsertEmpPassportDetails.java` file in the `code\JavaEE\Chapter3\Arrays` folder on the CD):

**Listing 3.19: Showing the Code for the InsertEmpPassportDetails.java File**

```
package com.kogent.jdbc;

import java.sql.*;
import java.util.*;
import oracle.sql.*;
/**
 * @author Suchita
 */
public class InsertEmpPassportDetails {
    public static void main(String s[]) throws Exception {
        Driver d= (Driver) ( Class.forName(
                "oracle.jdbc.driver.OracleDriver").newInstance());
        Properties p=new Properties();
```

```

p.put("user","scott");
p.put("password","tiger");

Connection con=d.connect(
"jdbc:oracle:thin:@192.168.1.123:1521:XE",p);

PreparedStatement ps=con.prepareStatement(
"insert into emppassportDetails values(?, ?, ?)");

ps.setInt(1,7934);
ps.setString(2,"12345A134");

String s1[]={ "v1", "v2", "v3", "v4", "v5" };

ArrayDescriptor ad=ArrayDescriptor.createDescriptor("VISA_NOS",con);
ARRAY a=new ARRAY(ad,con,s1);

ps.setArray(3,a);
int i=ps.executeUpdate();
System.out.println("Row Inserted, count : "+i);
con.close();

}//main
}//class

```

Listing 3.19 uses the SQL array types to insert the array values into an array. To insert the array values in the array, you need to create the array type in the database, so that the values inserted from the application through the array type can be stored in the array. The array type for the Array application is the emppassportDetails table with the columns. The following code snippet shows how to create the emppassportDetails table:

```

create table emppassportDetails (
empno number, passportno varchar2(10),
visas_taken visa_nos;
insert into emppassportDetails values(7934, '12345A123',
visa_nos('v1','v2','v3','v4','v5'));

```

The array type can be created at the Run SQL Command Line prompt, and then can be used by the user to insert data into the emppassportDetails table.

Figure 3.44 shows the output of the array type at the Run SQL Command Line prompt:

The screenshot shows a terminal window titled 'Run SQL Command Line'. The SQL command entered is:

```

SQL> create type visa_nos as varray(5) of varchar2(10);
Type created.

SQL> create table emppassportDetails (
  empno number,
  passportno varchar2(10),
  visas_taken visa_nos
);
Table created.

SQL>

```

Figure 3.44: Creating an Array Type in Oracle

Figure 3.44 shows the array type created in the Oracle database. This type is used by the `InsertEmpPassportDetails.java` file to store the data into the database. The table (`emppassportDetails`) contains the array types to store multiple data of the same type in a column. The column values inserted through Listing 3.19 are stored in one of the columns in the table (`emppassportdetails`).

Figure 3.45 shows the output of Listing 3.19 (InsertEmpPassportDetails.java) using the array types:

```
C:\Windows\system32\cmd.exe
D:\code\JavaEE\Chapter3\Arrays>javac -d . InsertEmpPassportDetails.java
D:\code\JavaEE\Chapter3\Arrays>java InsertEmpPassportDetails
Row inserted, count = 1
Row inserted, count = 2
Row inserted, count = 3
Row inserted, count = 4
Row inserted, count = 5
D:\code\JavaEE\Chapter3\Arrays>
```

**Figure 3.45: Showing the Output of the InsertEmpPassportDetails.java File**

In Listing 3.19, we have used the implementation for Array given by Oracle, which works only with the Oracle JDBC driver; consequently making the application a vendor-dependent application. JDBC 4.0 solves this problem by introducing the `createArrayOf()` method in `java.sql.Connection`. The `createArrayOf()` method of `java.sql.Connection` allows you to create vendor-independent `java.sql.Array` type of object with the given element type and value, as shown in the following code snippet:

```
PreparedStatement ps=con.prepareStatement("insert into emppassportDetails values(?, ?, ?)");
ps.setInt(1,7934);
ps.setString(2,"12345A134");
String s1[]={ "v1", "v2", "v3", "v4", "v5"};
Array a=con.createArrayOf("VARCHAR", s1);
ps.setArray(3,a);
```

We can also retrieve the Array type value from a database using JDBC. Listing 3.20 shows how to read the Array type value from the database using JDBC (you can find the `GetEmpPassportDetails.java` file in the `code\JavaEE\Chapter3\Arrays` folder on the CD):

**Listing 3.20:** Showing the Code for the `GetEmpPassportDetails.java` File

```
package com.kogent.jdbc;
import java.sql.*;
import java.util.*;
/**
 * @author Suchita
 */
public class GetEmpPassportDetails
{
    public static void main(String s[])
        throws Exception
    {
        Driver d= (Driver) ( Class.forName(
            "oracle.jdbc.driver.OracleDriver").newInstance());

        Properties p=new Properties();
        p.put("user","scott");
        p.put("password","tiger");

        Connection con=d.connect(
            "jdbc:oracle:thin:@192.168.1.123:1521:XE",p);

        Statement st=con.createStatement();

        ResultSet rs=st.executeQuery("select passportno, visas_taken from
            empPassportDetails where empno='"+s[0]+"'");

        if (rs.next())
        {
            System.out.println(
```

```

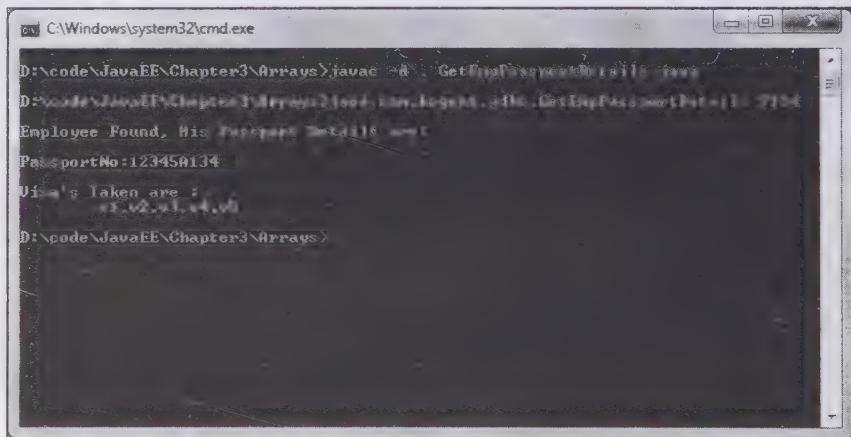
    "\nEmployee Found, His Passport Details are:\n");
    System.out.println("PassportNo:"+rs.getString(1)+"\n");
    System.out.print("Visa's Taken are :\n\t");

    Array a=rs.getArray(2);
    ResultSet rs1=a.getResultSet();
    /*
    The ResultSet produced here to represent Array value has 2 columns where
    1st column represents the element index 2nd column represents the values
    */
    boolean flag=rs1.next();
    while(flag) {
        System.out.print(rs1.getString(2));
        flag=rs1.next();
        if (flag)
            System.out.print(",");
    }
    } //while
} //if
else
    System.out.println("Employee not Found");
System.out.println();
con.close();
} //main
} //class

```

The example shown in Listing 3.20 reads the Array type value from the Oracle database.

Figure 3.46 shows the output of Listing 3.20:



**Figure 3.46: Showing the Output of GetEmpPassportDetails.java**

In the output shown in Figure 3.46, data is selected from the Oracle database. In Listing 3.20, the data is searched based on the specified employee number. In case the specified employee number is not found in the Oracle database, the *Employee not Found* message is displayed.

Note that the Array objects remain valid for at least the duration of the transaction in which they are created. This results in the shortage of resources in case of lengthy transactions. You can use the `free()` method of `java.sql.Array` interface in JDBC 4.0 to release the array resources.

## Describing the Ref Data Type

The `java.sql.Ref` interface represents the `Ref` type values, which are instances of the structured type. Each `Ref` value contains a unique identifier, which points to the `Ref` object. The values are stored either as a column value in a table or as an attribute value in the structured type. Since the `Ref` value is a logical pointer to a SQL structured type, a `Ref` object is also used as a logical pointer to the `Ref` values. `Ref` objects are stored in the database by using the methods of the `PreparedStatement.setRef()` interface.

Table 3.30 describes the methods of the Ref interface:

**Table 3.30: Methods of the Ref Interface**

Method	Description
public String getBaseTypeName()	Returns the name of the SQL structured type referenced by the SQL ref object
public Object getObject()	Retrieves the SQL ref object, which references the SQL structured type
public Object getObject(Map map)	Retrieves the SQL structured type and maps the Java type given by the map specified as an argument
public void setObject(Object value)	Sets the values of the SQL structured type, which is the reference of ref object

After learning how to implement the classes and interfaces of the java.sql package, let's discuss the implementation of the javax.sql package.

## Exploring JDBC Processes with the **javax.sql** Package

The javax.sql package, available in the JDBC API, is also known as the JDBC extension package. The javax.sql package is used to develop the client/server sided applications and provide server sided extension facilities, such as connection pooling and RowSet implementation. In addition, it uses the XA enabled connections for distributed transactions. The javax.sql package provides the following implementations that are used in building server-side applications:

- ❑ **JNDI-based lookup to access databases via logical names**—Allows you to access database resources by using logical names assigned to these resources. In other words, instead of allowing each client to load the driver classes in the respective local virtual machines, you can use the logical names assigned to each resource.
- ❑ **Connection pooling**—Serves as an intermediate layer provided by the javax.sql package to handle multiple connections. In this case, the responsibility for connection pooling is shifted from Application developers to the driver and the application server vendors.
- ❑ **Distributed transaction**—Provides support to handle multiple transactions in the Java EE environment by using the framework provided by the javax.sql package. With this framework, you can enable the support for distributed transactions with minimal configuration.
- ❑ **The RowSet**—Refers to a JavaBeans compliant object that hides ResultSets. The RowSet retrieves and accesses the data stored in a database. A RowSet may be connected when the JDBC connection is established and disconnected when the JDBC connection session ends up.

To understand the JDBC process with the javax.sql packages, let's explore the following broad-level steps in detail:

- ❑ Using DataSource to make a connection
- ❑ Implementing Connection pooling
- ❑ Using RowSet objects
- ❑ Using transactions

### Using *DataSource* to Make a Connection

With the help of the classes and interfaces provided by the javax.sql package, such as DataSource and DriverManager you can establish as well as manage connection with a data source. However, the DataSource mechanism is only preferred because it has many advantages over the DriverManager mechanism. The DataSource interface provides the following advantages, when used to make a connection:

- ❑ The developers need not provide code to implement a driver class.
- ❑ If the properties of a data source or driver changes, instead of modifying the application code, you can simply make the appropriate changes in the configurations of the data source.

- The connections established by using the `DataSource` object have the pooling and distributed transactions capabilities. This object also allows the Web container to communicate with the middle-tier infrastructure. However, the connections established with the help of `DriverManager` do not have the capabilities of connection pooling or distributed transaction.

`DataSource` implementations are provided by the driver vendor. A particular `DataSource` object represents a particular physical data source, and each connection created by `DataSource` is a connection to that physical data source.

The Java Naming and Directory Interface (JNDI) Naming Service is used to provide a logical name for the `DataSource` to make a connection. This naming service uses the Java Naming and Directory Interface™ (JNDI) API. The `DataSource` object can be used to retrieve the logical name associated with the underlying database. The application can then use the `DataSource` object to create the connection to the physical data source it represents.

The `DataSource` object helps in maintaining connection pooling; therefore, it can be used to work with the middle-tier infrastructure. Moreover, a `DataSource` object can also be implemented to work with the middle-tier infrastructure so that the connections it produces can be used for distributed transactions without any special coding.

## Exploring Connection Pooling

Connection pooling means that the connection is reused rather than created each time it is requested. A connection pool facilitates reusability of database connections and maintains a memory cache of connections. The connection pooling module lies at the top layer of the standard JDBC driver product.

This practice of using connection pooling in server-side application is performed in the background. In addition, it does not affect the procedure by which an application is coded. Instead of using the `DriverManager` class, a `DataSource` object (an object implementing `DataSource` interface) is used by an application to obtain a connection from the connection pool. A `DataSource` object is registered with a JNDI Naming service. After the `DataSource` object is registered, it can be automatically retrieved by using the JNDI Naming service. The following code snippet shows the creation of the `DataSource` object in a connection pool:

```
Context ctxt = new InitialContext();
DataSource ds = (DataSource) ctxt.lookup("jdbc/SequeLink");
```

In the preceding code snippet, if the `DataSource` object provides connection pooling, the concerned application automatically benefits from the connection reuse. This can be achieved without any code manipulation. The reused connections from the pool perform tasks similar to the newly created physical connections. When all the required tasks are performed by the application, the connection is explicitly closed. The following code snippet shows the procedure to close the database connection:

```
Connection dbcon = ds.getConnection("scott", "tiger");
// Do some database activities using the connection...
dbcon.close();
```

In the preceding code snippet, the closing event of a pooled connection signals the pooling module to place the connection back in the connection pool for future reuse.

## Traditional Connection Pooling

A general framework has been provided by the JDBC API to provide the support for traditional connection pooling. In traditional connection pooling, third-party vendors provide classes that support the connection pooling mechanism. In this way, the implementation of the specific caching or pooling algorithms can be done by third-party vendors or users. The JDBC4.0 API uses the `ConnectionEvent` class and provides various interfaces to create connection pool. To provide connection pooling in a server-sided application, the `DataSource` must implement following interfaces:

- `ConnectionPoolDataSource` – Specifies the data source that is being used in a connection pool. The `ConnectionPoolDataSource` interface also acts as a factory for the pooled connection objects.
- `PooledConnection` – Refers to an object that manages the hierarchy for connection pool.
- `ConnectionEventListener` – Refers to an object that handles the events generated by a `PooledConnection` object.

- ❑ **JDBCDriverVendorDataSource** – Refers to a class that implements the standard ConnectionPoolDataSource interface. This interface provides hooks, which can be used by the third-party vendors to implement pooling as a layer on top of their JDBC drivers. Moreover, in this case, the ConnectionPoolDataSource interface acts as a factory that creates PooledConnection objects.
- ❑ **JDBCDriverVendorPooledConnection** – Requires a JDBC driver vendor with a class that implements the standard PooledConnection interface to implement the connection pooling mechanism. The third-party vendors implement pooling on JDBC drivers with the help of this interface. In such cases, a PooledConnection object acts as a factory of the Connection objects. A PooledConnection object is the physical connection to the database, while the Connection object created by the PooledConnection object is simply a handle to the PooledConnection object.
- ❑ **PoolingVendorDataSource** – Requires a third-party vendor to provide a class which implements the DataSource interface to implement the connection pooling mechanism in a server-sided application. This interface is the entry point that allows interaction with their pooling module. The ConnectionPoolDataSource object creates PooledConnection objects as per the need.
- ❑ **PoolingVendorConnectionCache** – Specifies that to define the PoolingVendorConnectionCache class, the JDBC 4.0 API does not provide the interfaces, which are to be used between the DataSource object and the connection cache. Usually, a connection cache module contains one or multiple classes. Figure 3.47 shows the PoolingVendorConnectionCache class, which is used as a simple way to convey this concept. The connection cache module must contain a class that implements the ConnectionEventListener interface. Whenever the connection is closed or a connection error occurs, the PoolingVendorConnectionCache interface receives ConnectionEvent objects from PooledConnection objects. Moreover, when a connection closes on a PooledConnection object, the connection cache module returns the PooledConnection object to the cache, as shown in Figure 3.47:

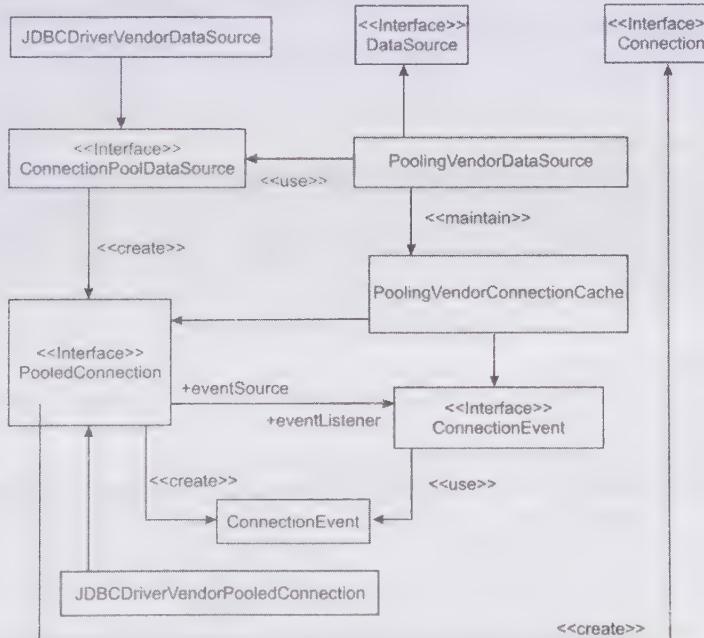


Figure 3.47: Showing the JDBC Connection Pooling Architecture

## Connection Pooling with the **javax.sql** Package

You can also implement the connection pooling mechanism in an application by using the `javax.sql` package. The `javax.sql` package provides a transparent meaning of connection pooling. This approach enables the Application server and the database driver to handle connection pooling internally. It is also important to

remember that as long as you use `DataSource` objects to get connections, connection pooling will automatically be enabled after you configure the Java EE application server.

You should note that the change in the additional connection pool is maintained by the Application server with the coordination of the JDBC driver. In other words, there is no additional programming requirement for JDBC client applications. Instead, the administrator of the Java EE server is required to configure a connection pool on the Application server. The syntax and the names of classes used to configure the connection pool are implementation dependent. However, with a JDBC 4.0 compliant Application server and database driver, the server administrator typically specifies the following:

- A class implementing the `javax.sql.ConnectionPoolDataSource` interface
- A class implementing the `java.sql.Driver` interface
- The size of the pool (minimum and maximum sizes)
- Connection time out
- The authentication parameters, such as loginid and password

The `javax.sql` package provides interfaces and classes to configure the Java EE server to enable connection pooling; therefore, the client application does not implement or access these interfaces directly. The `javax.sql` package specifies three interfaces and one class to implement connection pooling. The interfaces and class for connection pooling provided by the `javax.sql` package are:

- The `javax.sql.ConnectionPoolDataSource` interface
- The `javax.sql.PooledConnection` interface
- The `javax.sql.ConnectionEventListener` interface
- The `javax.sql.ConnectionEvent` class

Let's discuss these interface and classes used for connection pooling in the `javax.sql` package.

#### *The `javax.sql.ConnectionPoolDataSource` Interface*

The `javax.sql.ConnectionPoolDataSource` interface is similar to the `javax.sql.DataSource` interface. However, instead of returning `java.sql.Connection` objects, the `javax.sql.ConnectionPoolDataSource` interface returns the `javax.sql.PoolConnection` objects. The following code snippet lists the methods that return `javax.sql.PooledConnection` objects:

```
public javax.sql.PooledConnection getPooledConnection()
    throws java.sql.SQLException
public javax.sql.PooledConnection
getPooledConnection (String user, String password)
    throws java.sql.SQLException
```

As shown in the preceding code snippet, both the `getPooledConnection()` and `getPooledConnection(String user, String password)` methods return the `javax.sql.PooledConnection` objects.

#### *The `javax.sql.PooledConnection` Interface*

When connection pooling is enabled, objects implementing the `javax.sql.PooledConnection` interface hold a physical database connection. This interface is a factory of `javax.sql.Connection` objects.

The following are the methods provided by the `PooledConnection` interface:

```
public javax.sql.Connection getConnection() throws java.sql.SQLException
```

The `getConnection()` method returns a `java.sql.Connection` object. The returned `Connection` object, in turn, is a proxy for the physical connection held by the `javax.sql.PooledConnection` object. You need to invoke the `close()` method to close the connection with the database. The following code snippet shows the implementation of the `close()` method on the `PooledConnection` object:

```
public void close() throws java.sql.SQLException
```

As shown in the preceding code snippet, the `close()` method throws the `SQLException` exception, if any exception occurs during the closing of the connection with the database.

### The javax.sql.ConnectionEventListener Interface

The connection pooling components implement the ConnectionEventListener interface. The connection pooling components are mainly provided by the driver vendor or other software vendors. The JDBC driver notifies the ConnectionEventListener object, which registers a pooled connection when an application finishes execution. The notification of the event occurs after the application calls the close method on the PooledConnection object. The ConnectionEventListener interface is also notified when the connection is established. The JDBC driver also notifies the listener, before the driver throws the SQLException exception, but the PooledConnection object is already in use. There are two different methods, connectionClosed() and connectionErroroccurred(), containing the ConnectionEventListener interface. The following code snippet represents the connectionClosed() method in the ConnectionEventListener interface:

```
public void connectionClosed(ConnectionEvent event)
```

When the application calls the close() method, the connectionClosed() method is invoked. In this case, the connection pool marks the connection for reuse, as given in the following code snippet:

```
public void connectionErrorOccured(ConnectionEvent event)
```

When fatal connection errors occur, only the connectionErrorOccured (ConnectionEvent event) method is invoked. In this case, the connection pool may close the Connection on this event and remove it from the pool.

### The javax.sql.ConnectionEvent Class

The javax.sql.ConnectionEvent class represents connection-related events and provides information about them. The ConnectionEvent objects are generated when the application closes the pooled connection and the listeners are notified. This event handling is similar to the event handling in Abstract Window Toolkit (AWT) events. It is decided by the connection pool whether or not to add the connection event listeners to the pooled connection and when connection events occur, the connection listeners are notified.

### Implementation of Connection Pooling

The application server implements the mechanism of connection pooling by implementing the ConnectionPoolDataSource class. First, you need to instantiate the ConnectionPoolDataSource class, set its properties, and then bind the class to a name in JNDI context.

The following code snippet shows how to implement the ConnectionPoolDataSource class:

```
com.application.server.ConnPoolDataSource cds = new
    com.application.server.ConnPoolDataSource();
cds.setDatabaseName("myDB");
cds.setServerName("myServer");
Context contxt = new InitialContext();
contxt.bind("jdbc/pooled", cds);
```

The preceding code snippet shows a data source that is created in JNDI. The user can access this data source name to establish a connection. The data source returns a connection.

The data source, which is to be set with a connection, must provide the following properties:

- ❑ **InitialPoolSize**—Specifies the number of connections that the connection pool can maintain during a session.
- ❑ **minPoolSize**—Indicates the minimum number of connections to be maintained in the pool. The 0 value indicates that connections will be created when required.
- ❑ **maxPoolSize**—Indicates the maximum number of connections the pool should entertain. The 0 value indicates that there is no limit.
- ❑ **maxIdleTime**—Indicates the idle time of connections in a pool. It is represented in seconds.

### Using RowSet Objects

The javax.sql.RowSet object is a set of rows from the ResultSet object, or some other data source, such as a file or spreadsheet, represented in tabular form. All RowSet objects inherit the ResultSet interface and can be used as JavaBeans components in a visual Bean development environment. A RowSet is created and configured at design time and executed at run-time. The inbuilt JavaBeans properties enable the RowSet object to be configured and connected to the JDBC DataSource. A group of setter methods is used to pass input

parameters to the command property of the RowSet object. The value assigned to the command property is generally the SQL query, which is used to retrieve the data from the database. All RowSet objects have properties that are defined as getter and setter methods in the implementation classes. The BaseRowSet abstract class helps to set and get the required properties in JDBC RowSet implementations. All the RowSet reference implementations inherit this class; and therefore, have access to the methods of the BaseRowSet class.

As you know that the connection can be obtained in two different ways, either by using the DriverManager mechanism or by using DataSource object. In both these ways, you need to set the username and password properties. In case of DriverManager, you need to set the url and in case of the DataSource object, you need to set the data source name property. You should note that the default value for the type property is ResultSet.TYPE\_SCROLL\_INSENSITIVE, and for the concurrency property is ResultSet.CONCUR\_UPDATABLE. If you are working with a driver or database that does not offer scrollable and updatable ResultSet objects, you can use a RowSet object populated with the same data as a ResultSet object; thereby, making the ResultSet object scrollable and updatable.

A listener for a RowSet object is a component that is to be notified whenever a change or called event occurs in the RowSet object. Due to any of the following changes, the RowSet interface generates an event that is handled by the listeners:

- A cursor movement
- The update, insertion, or deletion of a row
- A change in the entire RowSet content

The listeners must be registered with the RowSet class to receive notifications from a particular RowSet. Therefore, all listeners must implement the RowSetListener interface. A listener for a RowSet object implements the following methods defined in the RowSetListener interface corresponding to the three events discussed in the preceding list:

- cursorMoved**—Includes the actions that a listener should perform when the cursor in the RowSet object moves
- rowChanged**—Specifies the actions that a listener should perform when one or more column values in a row are updated, a new row is inserted, or an existing row is deleted
- rowSetChanged**—Specifies the actions that the listener should perform when the entire RowSet object is populated with new data

Depending on the implementation of an application, the JDBC RowSet objects are categorized as:

- Connected RowSet objects
- Disconnected RowSet objects
- JdbcRowSet objects
- CachedRowSet objects
- WebRowSet objects
- FilteredRowSet object
- JoinRowSet objects

Let's explore these in detail next.

## **Connected RowSet Objects**

A Connected RowSet object creates a connection to a database, by using JDBC driver, and maintains that connection throughout its lifetime. JdbcRowSet is one of the standard Connected RowSet implementations. The JdbcRowSet object is connected to a database, which makes it similar to the ResultSet object. In addition, the JdbcRowSet object is often used as a wrapper to make a nonscrollable and read-only ResultSet object scrollable and updatable.

## **Disconnected RowSet Objects**

A disconnected RowSet object makes a connection to a data source only to read data from the ResultSet object or write the data back to the data source. After reading or writing data to its data source, the RowSet object

disconnects from the data source. As a disconnected RowSet object does not connect to its data source; thereby, the object performs the task of reading and writing data independently. The disconnected RowSet objects are serializable as well as lightweight compared to a JdbcRowSet or ResultSet object. Due to this reason, the disconnected RowSet objects are efficient for thin clients.

Figure 3.48 shows the CachedRowSet interface, which defines the capabilities available to the disconnected RowSet object:

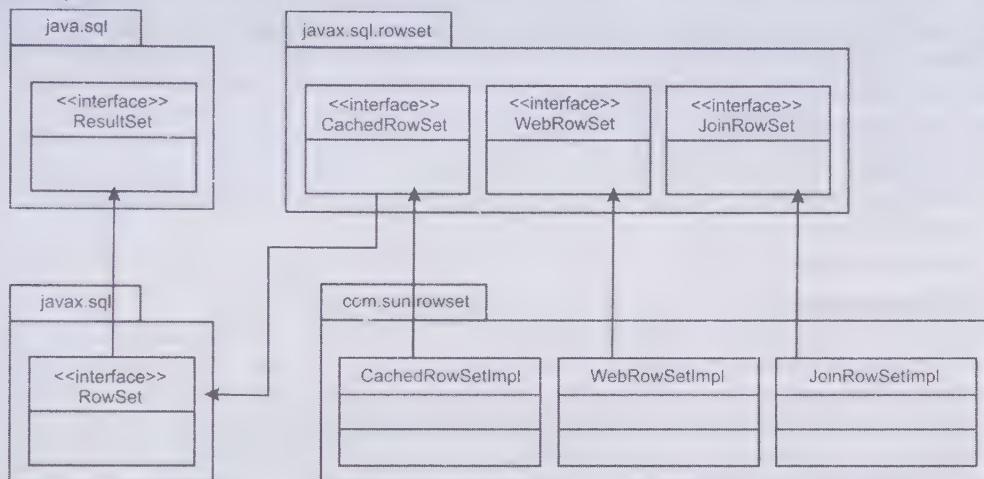


Figure 3.48: Displaying the RowSet Inheritance Hierarchy

## JdbcRowSet Objects

The JdbcRowSet object is simply a wrapper around the ResultSet object and always maintains a connection to its data source, similar to a ResultSet object. The main difference between a JdbcRowSet and ResultSet object is that a JdbcRowSet object has a set of properties and also participates in the JavaBeans event model. The use of the JdbcRowSet object makes it a JavaBeans component. A JdbcRowSet object can be used this way because it is effectively a wrapper for the driver that has obtained its connection to the database. Another benefit of using the JdbcRowSet object is that it makes a ResultSet object as scrollable and updatable. All RowSet objects are scrollable and updatable by default. For example, a JdbcRowSet object populated with the ResultSet data is also scrollable and updatable.

The JdbcRowSetImpl is used as a default constructor to create new instances of the JdbcRowSet objects. A new instance is initialized with default values in the BaseRowSet class, which can be set with new values when required. The commands and properties needed to establish a connection are set, and after which the execute() method is invoked. The new instance does not work until the execute() method is called.

The following code snippet creates a new JdbcRowSetImpl object, sets the command and connection properties, sets the placeholder parameter, and then invokes the execute() method:

```

JdbcRowSetImpl jrs = new JdbcRowSetImpl();
jrs.setCommand("SELECT * FROM TITLES WHERE TYPE = ?");
jrs.setURL("jdbc:myDriver:myAttribute");
jrs.setUsername("cervantes");
jrs.setPassword("sancho");
jrs.setString(1, "BIOGRAPHY");
jrs.execute();
  
```

The preceding code snippet performs the following tasks:

- ❑ Establishes a connection between the RowSet and the database
- ❑ Creates a PreparedStatement object to make a program more interactive and sets its placeholder parameters
- ❑ Executes the statement provided in the setCommand() method to create a ResultSet object

## CachedRowSet Objects

The CachedRowSet object inherits the JdbcRowSet class, in addition to its own capabilities and additional features. This object caches its rows in memory; therefore, it does not need to always connect to its data source. Usually, the CachedRowSet object retrieves rows from a ResultSet object but it can also contain rows from files in tabular formats, such as spreadsheets. The CachedRowSet object is a disconnected RowSet and connects with the data source only when it is reading the data to populate the rows or when it is updating changes in the underlying data source. You can perform the following functions with a CachedRowSet object:

- Create a CachedRowSet object
- Set the properties of the CachedRowSet object
- Fill a CachedRowSet object
- Read data from the CachedRowSet object
- Retrieve the RowSetMetaData object
- Update a CachedRowSet object

Let's discuss each of these in detail.

### *Creating a CachedRowSet Object*

The default implementation for the CachedRowSet object creates a CachedRowSet object. The default constructor is used to create the new instance. The following code snippet shows how to create a new instance of the CachedRowSet object:

```
CachedRowSetImpl crs=new CachedRowSetImpl();
```

In the preceding code snippet, the properties of the CachedRowSet object are set to the default properties of the BaseRowSet object. In addition, the CachedRowSet object has one synchronization provider object RIOptimisticProvider of the SyncProvider type. The classes and interfaces for synchronization provider implementation are provided by the javax.sql.rowset.spi package. The RowSetReader is used by the RIOptimisticProvider objects to read data into the CachedRowSet object, as this RowSet object does not contain any established connection to the database. This RowSetReader object obtains a connection by using the values set either for username, password, and JDBC URL; or for the data source name. The RIOptimisticProvider provider also provides the RowSetWriter object to synchronize any changes made to the rows of the CachedRowSet object while it was disconnected from the underlying data source. If you are not using the RowSetWriter object, the SyncProvider objects are retrieved from the SyncFactory class. The following code snippet is used to get the list of synchronization providers in a CachedRowSet object:

```
java.util.Enumeration Providers=SyncFactory.getRegisteredProviders();
```

The method mentioned in the preceding code snippet returns the list of providers to specify a particular SyncProvider object that the CachedRowset object can use. The following code snippet shows how to create the instance of the CachedRowSet object by providing a specific SyncProvider object:

```
CachedRowSetImpl crs2=new  
CachedRowSetImpl("com.fred.providers.HighAvailabilityProvider");
```

The value for the synchronization parameter can be set using the setSynchProvider() method of CachedRowSet:

```
crs.setSyncProvider("com.fred.providers.HighAvailabilityProvider");
```

### *Setting the Properties of the CachedRowSet Object*

All RowSet objects have common properties, therefore, the properties for the CachedRowSet objects are to be set by using the setter methods available in the RowSet interface. The following code snippet shows how to set the values for the CachedRowSet objects:

```
//basic parameters required to set for establishing a connection with  
Database  
crs.setUsername("user");  
crs.setPassword("password");  
crs.setUrl("jdbc:mySubprotocol:mysubname");  
crs.setCommand("select * from survey");
```

In the preceding code snippet, the `setCommand` method is used to set the command property, which is a query that produces the `ResultSet` object. You can read data into a `RowSet` object from a `ResultSet` object.

#### *Filling a CachedRowSet Object*

To populate data from `ResultSet` object to `RowSet` object, you only have to call the `execute()` method on the `CachedRowSet` object, as shown in the following code snippet:

```
//populate data into rowset object from ResultSet object
crs.execute();
```

In the preceding code snippet, when the `execute()` method is called, the reader of the disconnected `RowSet` object works behind the scene. The `execute()` method is provided by the default `SyncProvider` object, `RIOptimisticProvider`. Then, the `RowSetReader` object gets a connection to the database either by using the JDBC URL or the data source. Next, the reader object executes the query that is to be set for the `command` property. The result of the query is saved in the `ResultSet` object, which is in turn provided to the `CachedRowSet` object.

#### *Reading Data from CachedRowSet Object*

Data is read from a `CachedRowSet` object by using getter methods inherited from the `ResultSet` interface. The following code snippet illustrates how the rows of the `crs` `CachedRowSet` object are iterated and the column values of each row are read:

```
while(crs.next())
{
    String name=crs.getString("NAME");
    int id=crs.getInt("ID");
    Clob comment=crs.getBlob("COM");
    short dept = crs.getShort("DEPT");
    System.out.println(name+" "+id+" "+comment+" "+dept);
}
```

#### *Retrieving RowSetMetaData Object*

The user can retrieve the information about columns in the `CachedRowSet` object by using the `RowSetMetaData` object. The `getMetaData()` method of the `ResultSet` interface returns a `ResultSetMetaData` object, which is further casted to the `RowSetMetaData` object. Finally, the object is assigned to the `rsmd` variable. The following code snippet shows how to retrieve information in the `CachedRowSet` object:

```
RowSetMetaData rsmd=(RowSetMetaData)crs.getMetaData();
int count=rsmd.getColumnCount();
int type=rsmd.getColumnType(2);
```

#### *Updating a CachedRowSet Object*

Updating a `CachedRowSet` object is similar to updating a `ResultSet` object. When the `CachedRowSet` object is disconnected from data source, the updates in the `CachedRowSet` are performed; however, the results of updates are not finally written to data source. To write the results of updates, a connection with the data source has to be established. Therefore, after invoking the `updateRow()` or `insertRow()` method, another method, `acceptChanges()`, is called on the `CachedRowSet` object to write the update results on the database. During the invocation of the `acceptChanges()` method, the `RowSetWriterImpl` object is called on the `CachedRowSet` object internally, which establishes the connection with the data source and also updates the changes in the data source.

The following code snippet shows the steps to update the `CachedRowSet` object:

```
//update 3rd and 4th column of current row
crs.updateShort(3, 58);
crs.updateInt(4, 150000);
crs.updateRow();
crs.acceptChanges();

//Build a new row , inserts into crs and finally inserts into datasource
crs.moveToInsertRow();
crs.updateString("Name", "Shakespeare");
```

```

crs.updateInt("ID", 10098347);
crs.updateShort("Age", 58);
crs.updateInt("Sal", 150000);
crs.insertRow();
crs.moveToCurrentRow();
crs.acceptChanges();

```

In the preceding code snippet, a connection is established corresponding to each call of the `acceptChanges()` method, which is called after calling the `updateRow()` and `insertRow()` methods to change or insert multiple rows. The advantages of using the `CachedRowSet` objects are as follows:

- Obtains a connection to a data source and execute a query
- Reads the data from the resulting `ResultSet` object and populates itself with that data
- Manipulates data and make changes to data while it is disconnected
- Reconnects to the data source to write the changes back to it
- Checks and resolves the conflicts with the data source

The JDBC API does not need to be implemented for using the `CachedRowSet` objects. The `CachedRowSet` object is serializable, which is the main reason to use a `CachedRowSet` object to pass data between different components of an application. Working on a network environment, a `CachedRowSet` object can be used to send the result of query that is executed by Enterprise JavaBeans.

## WebRowSet Objects

A `WebRowSet` object has all the capabilities of a `CachedRowSet` object and is used to read and write the database query results into an XML file. Enterprises on different locations and platforms can communicate through XML; therefore, the XML language has become the standard for Web services communication. As a consequence, a `WebRowSet` object solves a real problem by making it easy for Web services developers to write the Web service programs to send and receive data from a database in the form of an XML document.

### *Creating and Populating a WebRowSet Object*

The new instance of the `WebRowSet` object can be created by using the reference of the `WebRowSetImpl` class. The following code snippet shows the code to create an instance of the `WebRowSet` object:

```

WebRowSet wrs = new WebRowSetImpl();
wrs.populate(rs);

```

In the preceding code snippet, `wrs` has no data; however, it has the default properties of a `BaseRowSet` object. Its `SyncProvider` object is first set to the `RIOptimisticProvider` implementation, which is the default configuration for all disconnected `RowSet` objects. You can set various properties, such as URL, username, password for the `WebRowSet` object, as shown in the following code snippet:

```

wrs.setCommand("SELECT col1,col2 from emp");
wrs.setURL("jdbc:mySubprotocol:myDatabase");
wrs.setUsername("myUsername");
wrs.setPassword("myPassword");
wrs.execute();

```

The preceding code snippet sets the properties for the `WebRowSet` object.

### *Writing and Reading the WebRowSet Object to XML Document*

The `WebRowSet` object can be used to read and write the data into an XML document. The `readXML()` method is used to read the data from the XML document; whereas, the `writeXML()` method allows you to write data in the XML document.

The uses of the `writeXML()` and `readXML()` methods are described as follows:

- Using the `writeXML()` method**—Writes the invoked `WebRowSet` object as an XML document that represents the current state of object. The method writes the XML document to the stream that is passed to it. The stream can be an `OutputStream` object, such as a `FileOutputStream` object, if the user wants to write in binary format; or a `Writer` object, such as a `FileWriter` object, if the user wants to write in characters.

The following code snippet writes the `wrs` `WebRowSet` object as an XML document to the `FileOutputStream` object `fileOutputStream`:

```
java.io.FileOutputStream fileOutputStream = new java.io.FileOutputStream("emp.xml");
wrs.writeXML(fileOutputStream);
```

The `FileWriter` object is used to write the character data to an XML file, as shown in the following code snippet:

```
java.io.FileWriter fileWriter = new java.io.FileWriter("emp.xml");
wrs.writeXML(fileWriter);
```

Two variations of the `writeXML()` method, `fileOutputStream()` and `fileWriter()`, are used for the `WebRowSet` object with the content of a `ResultSet` object before writing it to a stream, as shown in the following code snippet:

```
priceList.writeXML(rs, fileOutputStream);
priceList.writeXML(rs, fileWriter);
```

- **Using the `readXml()` method**—Parses an XML document to construct the `WebRowSet` object. Similar to writing, an XML document, which is to be read, is represented by the `FileInputStream` or `FileWriter` object and is passed to the `readXML()` method.

The following code snippet explains how to read from XML document into a `WebRowSet` object:

```
java.io.FileInputStream fileInputStream = new java.io.FileInputStream("emp.xml");
wrs.readXML(fileInputStream);
```

The `fileReader` object is used to read the XML character data to a `WebRowSet` object, as shown in the following code snippet:

```
java.io.FileReader fileReader = new java.io.FileReader("emp.xml");
wrs.readXML(fileReader);
```

### *Using the `WebRowSet` Object in XMLFormat*

The `WebRowSet` object contains data; and the properties and metadata about the columns. The `WebRowSet` XML schema is an XML document that defines the content of an XML document. It also defines the format in which the document must be presented. This schema is used by both the sender and recipient because it tells the sender how to write the XML document and the receiver how to parse the XML document. The XML document representing a `WebRowSet` object includes the following three types of information:

- **Properties of `WebRowSet` object**—Refer to standard synchronization provider properties, including general `RowSet` properties. A `WebRowSet` object is created and populated from a table having two rows and five columns from a data source. The standard `writeXML()` method describes the internal properties of the `WebRowSet` object.

The following code snippet shows the use of the `writeXML()` method to describe the internal properties:

```
<properties>
<command>select col1, col2 from test_table</command>
<concurrency>1</concurrency>
<datasource/>

<escape-processing>true</escape-processing>
<fetch-direction>0</fetch-direction>
<fetch-size>0</fetch-size>
<isolation-level>1</isolation-level>
<key-columns/>
<map/>
<max-field-size>0</max-field-size>
<max-rows>0</max-rows>
<query-timeout>0</query-timeout>
<read-only>false</read-only>
<rowset-type>TRANSACTION_READ_UNCOMMITTED</rowset-type>
<show-deleted>false</show-deleted>
<table-name/>
<url>jdbc:thin:oracle</url>
<sync-provider>

<sync-provider-name>.com.rowset.provider.RIOptimisticProvider</sync-provider-name>
<sync-provider-vendor>Sun Microsystems</sync-provider-vendor>
<sync-provider-version>1.0</sync-provider-name>
<sync-provider-grade>LOW</sync-provider-grade>
<data-source-lock>NONE</data-source-lock>
</sync-provider>
</properties>
```

- ❑ **Metadata**—Describes the metadata associated with the tabular structure used by a WebRowSet object. Metadata is similar to the java.sql.ResultSet interface. The WebRowSet object is also used to retrieve the metadata information about the ResultSet interface.

The following code snippet shows the columns that are described between the column definition tags:

```
<metadata>
  <column-count>2</column-count>
  <column-definition>
    <column-index>1</column-index>
    <auto-increment>false</auto-increment>
    <case-sensitive>true</case-sensitive>
    <currency>false</currency>
    <nullable>1</nullable>
    <signed>false</signed>
    <searchable>true</searchable>
    <column-display-size>10</column-display-size>
    <column-label>COL1</column-label>
    <column-name>COL1</column-name>
    <schema-name/>
    <column-precision>10</column-precision>
    <column-scale>0</column-scale>
    <table-name/>
    <catalog-name/>
    <column-type>1</column-type>
    <column-type-name>CHAR</column-type-name>
  </column-definition>
  <column-definition>
    <column-index>2</column-index>
    <auto-increment>false</auto-increment>
    <case-sensitive>false</case-sensitive>
    <currency>false</currency>
    <nullable>1</nullable>
    <signed>true</signed>
    <searchable>true</searchable>
    <column-display-size>39</column-display-size>
    <column-label>COL2</column-label>
    <column-name>COL2</column-name>
    <schema-name/>
    <column-precision>38</column-precision>
    <column-scale>0</column-scale>
    <table-name/>
    <catalog-name/>
    <column-type>3</column-type>
    <column-type-name>NUMBER</column-type-name>
  </column-definition>
</metadata>
```

- ❑ **Data**—Describes the data available in a database before the changes are made due to the synchronization of the WebRowSet object. This helps to evaluate the changes between the original and current data. A WebRowSet object contains the ability to synchronize the changes in its data back to the data source. The WebRowSet object provides a table structure and the CurrentRow tag is used to map each row of table. A columnValue tag can contain either the StringData or binaryData tag, depending on its SQL type. You should note that the BLOB and CLOB data types use binaryData tag. They describe a WebRowSet object that has not undergone any changes since its instantiation.

The following code snippet shows the content of the WebRowSet object:

```
<data>
  <currentRow>
    <columnValue>
      firstrow
    </columnValue>
    <columnValue>
      1
    </columnValue>
  </currentRow>
```

```

<currentRow>
    <columnValue>
        secondrow
    </columnValue>
    <columnValue>
        2
    </columnValue>
</currentRow>
<currentRow>
    <columnValue>
        thirddrow
    </columnValue>
    <columnValue>
        3
    </columnValue>
</currentRow>
<currentRow>
    <columnValue>
        fourthrow
    </columnValue>
    <columnValue>
        4
    </columnValue>
</currentRow>
</data>

```

### *Implementing Changes in a Database by Using WebRowSet Objects*

Different operations can be performed on the WebRowSet object to update it. You can update the WebRowSet object by deleting, inserting, and updating an existing row, which are explained as follows:

- ❑ **Deleting a row**—Removes the row from a WebRowSet object. To delete a row, move the cursor to the desired row and invoke the deleteRow method.

The following code snippet shows the deletion of a row, in which the wrs WebRowSet object is used to delete the third row:

```

<data>
    <currentRow>
        <columnValue>
            firstrow
        </columnValue>
        <columnValue>
            1
        </columnValue>
    </currentRow>
    <currentRow>
        <columnValue>
            secondrow
        </columnValue>
        <columnValue>
            2
        </columnValue>
    </currentRow>
    <deleteRow>
        <columnValue>
            thirddrow
        </columnValue>
        <columnValue>
            3
        </columnValue>
    </deleteRow>
    <currentRow>
        <columnValue>
            fourthrow
        </columnValue>
        <columnValue>

```

```

        4
    </columnValue>
</currentRow>
</data>

```

In the preceding code snippet, the XML description marks third row as the `deleteRow` and deletes the row from the `WebRowSet` object.

- **Inserting a row**—Refers to the addition of a new row into the `WebRowSet` object. To insert a new row, move the cursor to the row where the row insertion is to be performed, then call the update methods to insert values into the row, and finally insert that row into `ResultSet` and database. The following code snippet is used to insert a new row into the `WebRowSet` object:

```

wrs.moveToInsertRow();
wrs.updateString(1, "fifththrow");
wrs.updateString(2, "5");
wrs.insertRow();
wrs.acceptChanges();

```

The insertion to the `WebRowSet` object can be performed in the XML file.

The following code snippet shows the XML format insertion to the `WebRowSet` object:

```

<data>
    <currentRow>
        <columnValue>
            firstrow
        </columnValue>
        <columnValue>
            1
        </columnValue>
    </currentRow>
    <currentRow>
        <columnValue>
            secondrow
        </columnValue>
        <columnValue>
            2
        </columnValue>
    </currentRow>

    <currentRow>
        <columnValue>
            newthirdrow
        </columnValue>
        <columnValue>
            III
        </columnValue>
    </currentRow>
    <insertRow>
        <columnValue>
            fifthrow
        </columnValue>
        <columnValue>
            5
        </columnValue>
        <updateValue>
            V
        </updateValue>
    </insertRow>
    <currentRow>
        <columnValue>
            fourthrow
        </columnValue>
        <columnValue>
            4
        </columnValue>
    </currentRow>
</data>

```

- **Updating an existing row** – Creates a specific XML file that holds both the updated value and the value that is replaced. The value that is replaced becomes the original value, and the new value becomes the current value. The following code snippet shows how to move the cursor to a specific row, perform some modifications, and also update the row when the execution of the wrs object is completed:

```
wrs.absolute(5);
wrs.updateString(1, "new4thRow");
wrs.updateString(2, "IV");
wrs.updateRow();
```

The modifyRow tag is used to update the WebRowSet object in an XML document. Both the original as well as updated values are associated within the tags for original row values tracking.

The following code snippet shows the process to update the WebRowSet object in a XML document:

```
<data>
  <currentRow>
    <columnValue>
      firstrow
    </columnValue>
    <columnValue>
      1
    </columnValue>
  </currentRow>
  <currentRow>
    <columnValue>
      secondrow
    </columnValue>
    <columnValue>
      2
    </columnValue>
  </currentRow>
  <currentRow>
    <columnValue>
      newthirdrow
    </columnValue>
    <columnValue>
      III
    </columnValue>
  </currentRow>
  <currentRow>
    <columnValue>
      fifthrow
    </columnValue>
    <columnValue>
      5
    </columnValue>
  </currentRow>
  <modifyRow>
    <columnValue>
      fourthrow
    </columnValue>
    <updateValue>
      new4thRow
    </updateValue>
    <columnValue>
      4
    </columnValue>
    <updateValue>
      IV
    </updateValue>
  </modifyRow>
</data>
```

## FilteredRowSet Objects

A FilteredRowSet object allows the user to limit the number of rows that are visible in a RowSet object so that the user can work only with the relevant data. The user can also apply more than one filter to

FilteredRowSet in one application to work with different sets of rows and columns each time. The filters inherit a WebRowSet object, which inherits the CachedRowSet object. Therefore, a WebRowSet object has the capabilities of both the FilteredRowSet and CachedRowSet objects. In case of JdbcRowSet, filtering is done by using query language, because it is always connected to a data source. The FilteredRowSet object provides a method to filter data without executing a query on the data source, which in turn avoids having connection with the data source and sending queries to it.

#### *Creating a Filter*

A filter is created by using the javax.sql.RowSet.Predicate interface. Each application that wishes to apply a filter must implement the Predicate interface. The FilteredRowSet object enforces filter constraints in two directions, i.e., either column number or column name.

The following code snippet shows the simple implementation of the Predicate interface:

```
import javax.sql.rowset.*;
public class Filter1 implements Predicate
{
    private int lo;
    private int hi;
    private String colName;
    private int colNumber;
    public Filter1(int lo, int hi, int colNumber)
    {
        this.lo = lo;
        this.hi = hi;
        this.colNumber = colNumber;
    }
    public Filter1(int lo, int hi, String colName){
        this.lo = lo;
        this.hi = hi;
        this.colName = colName;
    }
    public boolean evaluate(RowSet rowset) {
        CachedRowSet crs = (CachedRowSet)rowset;
        if (rowset.getInt(colNumber) >= lo &&
            (rowset.getInt(colNumber) <= hi)) {
            return true;
        }else { return false; }
    }
}
```

#### *Using FilteredRowSet Object*

The FilteredRowSet object can be used with the ResultSet object to populate the RowSet object. The following code snippet shows the use of the ResultSet object to populate the RowSet object:

```
FilteredRowSet frs = new FilteredRowSetImpl();
frs.populate(rs);
Range name = new Range("50", "100", "EMP_ID");
frs.setFilter(name);
frs.next() // only IDs from 50 to 100 will be returned.
```

In general, the Predicate object is initialized with the following features:

- ❑ The lower limit of the range within which the values of a column number or column name must lie.
- ❑ The upper limit of the range within which the values of a column number or column name must lie.
- ❑ The column name or number of the value, which must lie within the range of values set by the upper and lower limits. Note that the range of values is inclusive, meaning that a value at the boundary is included in the range.

#### *Updating a FilteredRowSet Object*

The Predicate interface can be applied on the FilteredRowSet object in a bi-directional manner. Any effort to update the FilteredRowSet object that violates the set criteria throws the SQLException exception. The range criteria for the FilteredRowSet object can be changed by applying a new Predicate object to the

instance of the FilteredRowSet object. After changing the criteria of FilteredRowSet, all the updates should be done according to the new criteria set. Updating the FilteredRowSet object is same as updating the CachedRowSet object.

## JoinRowSet Objects

The JoinRowSet interface encapsulates the related data from RowSet objects that form a SQL JOIN relationship. The Joinable interface provides the methods to set, read, and unset a match column. In addition, the Joinable interface should be implemented by all the RowSet objects. The column matching process is the basis of the SQL JOIN operation. The match column may be set by using the appropriate version of the JoinRowSet interface's addRowSet() method. The main purpose of the JoinRowSet interface is to establish a SQL JOIN between disconnected RowSet objects, because they do not connect to data source to make SQL JOIN. A RowSet object can become a part of SQL JOIN relation by adding the RowSet object with JoinRowSet object, because the connected JdbcRowSet object extends the Joinable interface. The Joinable interface is not added in the JoinRowSet object because it is always connected with the data source and can perform SQL JOIN by using SQL query.

### Exploring the Methods Used in the Joinable Interface

The Joinable interface has methods to specify a common column, based on which SQL JOIN is made. However, it does not have the facility to add two RowSet objects into one, which is provided by the JoinRowSet interface. You can set the JoinRowSet constants in the setJoinType method to define the type of the join. The following SQL JOIN constant types can be set on the setJoinType method:

- CROSS\_JOIN
- FULL\_JOIN
- INNER\_JOIN
- LEFT\_OUTER\_JOIN
- RIGHT\_OUTER\_JOIN

#### NOTE

If no join type is provided, the INNER\_JOIN join is set on the setJoinType method, as the default value.

### Using a JoinRowSet Object to form a JOIN

To form the basis of the JOIN relation, you first need to add the RowSet object to the JoinRowSet object. You should note that when the JoinRowSet object is created, it is empty. Therefore, you should define the column in which each RowSet object is to be added to the JoinRowSet object. The RowSet object contains a match column, and the value in each match column should be comparable to the values in the other match column. A match column can be set by using the following methods:

- Matching a column by using the setMatchColumn() method of the Joinable interface before a RowSet object is added to a JoinRowSet object. The RowSet object must implement the Joinable interface to use this method. After setting the match column value, the value can be reset by using the setMatchColumn method at any time.
- Adding a column name or number, or an array of column names or numbers by invoking the addRowSet() method. A match column parameter is passed as an argument in four of the five addRowSet() methods.

The following code snippet adds two CachedRowSet objects to a JoinRowSet object. For simplicity, no SQL JOIN type is set, so the default JOIN type, which is INNER\_JOIN, is established.

The following code snippet shows the implementation of the JoinRowSet object:

```
JoinRowSet jrs = new JoinRowSetImpl();
ResultSet rs1 = stmt.executeQuery("SELECT * FROM EMPLOYEES");
CachedRowSet empl = new CachedRowSetImpl();
empl.populate(rs1);
empl.setMatchColumn(1);
jrs.addRowSet(empl);
ResultSet rs2 = stmt.executeQuery("SELECT * FROM ESSP_BONUS_PLAN");
CachedRowSet bonus = new CachedRowSetImpl();
```

```

bonus.populate(rs2);
bonus.setMatchColumn(1); // EMP_ID is the first column
jrs.addRowSet(bonus);

```

In the preceding code snippet, the EMPLOYEES table, whose match column is set to the first column EMP\_ID is first added to the JoinRowSet object jrs. Then, the ESSP\_BONUS\_PLAN table with the same match column EMP\_ID is added. The rows in the ESSP\_BONUS\_PLAN table are added to jrs, only if the EMP\_ID value ESSP\_BONUS\_PLAN matches with an EMP\_ID value in EMPLOYEE table. In broad terms, everyone in the bonus plan is an employee so all the rows in the ESSP\_BONUS\_PLAN table are added to the JoinRowSet object. The jrs is an inner JOIN of the two RowSet objects based on the EMP\_ID columns. A program can traverse or modify a RowSet object by using RowSet methods, as shown in the following code snippet:

```

jrs.first();
int employeeID = jrs.getInt(1);
String employeeName = jrs.getString(2);

```

The following code snippet adds an additional CachedRowSet object. In this case, the match column (EMP\_ID) is set when the CachedRowSet object is added to the JoinRowSet object, as shown in the following code snippet:

```

ResultSet rs3 = stmt.executeQuery("SELECT * FROM SITE");
CachedRowSet site = new CachedRowSetImpl();
site.populate(rs3);
jrs.addRowSet(site, 1);

```

The JoinRowSet object jrs now contains values from all three tables.

## Working with Transactions

The DBMSs manage the databases over multiple environments where numerous users are working. There may be chances of data loss over multiple environments and the users. Therefore, to overcome such problems, the DBMS provides a mechanism to maintain data integrity within the DBMS. Transactions are used to ensure data integrity when multiple users access and modify data in a DBMS. A database transaction includes the interaction between the databases and users. Transactions are required to ensure data integrity, correct application semantics, and a consistent view of data during concurrent access. In general, DBMS provides the feature of Atomicity, Consistency, Isolation, and Durability for each transaction in a database. These properties are collectively called the ACID (Atomicity, Consistency, Isolation, and Durability) properties.

Let's know about the ACID properties.

### ACID Properties

The ACID properties are maintained by the transaction manager of DBMS to retain the integrity of the data over the database. Let's describe the ACID properties for the transaction mechanism.

#### Atomicity

The guarantee of either all or none of the tasks of a transaction to be performed is defined as atomicity. This property provides an ability to save (commit) or cancel (rollback) the transaction at any point, and controls all the statements of a transaction.

#### Consistency

The Consistency property guarantees that the data remains in a legal state when the transaction begins and ends, implying that if the data used in the transaction is consistent before starting the transaction, it remains consistent even after the end of the transaction. If the data satisfies the integrity constraints of that type, it is known as consistent data or data in legal state.

For example, if an integrity constraint specifies that the age should not be a character and should be a positive value, a transaction is aborted during its execution if this rule is violated.

#### Isolation

The isolation is the ability of the transaction to isolate or hide the data used by it from other transactions until the transaction ends. The isolation is done by preparing locks on the data. The following set of problems may occur when the user performs concurrent operations on the data:

- ❑ **Dirty Read**—Specifies that a transaction tries to read data from a row that has been modified but yet to be committed by other transactions.
- ❑ **Non-repeatable Read**—Occurs when the read lock is not acquired while performing the SELECT operation. For example, if you have selected data under the T1 transaction, and meanwhile if the same is being updated by some other transaction, say T2, then the T1 transaction reads two versions of data. This type of data read is considered as non-repeatable read. It can be avoided by preparing a read lock by transaction T1 on the data that is selected.
- ❑ **Phantom Read**—Specifies the situation when the collection of rows, returned by the execution of two identical queries, are different. This can happen when range locks are not acquired while executing the SELECT query. Consider an example, where in a transaction T1, you have executed query Q1 and got some results (say 10 rows). It is possible that during transaction T1, another transaction T2 has made some changes due to which the execution of the query Q1 within T1 now results in different number of rows (say 11 rows). This problem is referred as phantom read problem, which happens if some other transaction inserts a new record that is being used by an already running transaction.

## Durability

The durability property guarantees that the user has been notified of the successful transaction, which can persist all the statements in the transaction or leave the complete transaction unsaved. This property specifies that after successful execution of the transaction, the system guarantees the updation of data in the database even if the computer crashes after the execution of the transaction.

## Types of Transactions

A database transaction is used to provide data integrity and security to the database. All the JDBC specific drivers are required to provide transaction support for all the database operations. The database operations can include concurrent access of data from a data source. These transaction mechanisms are used to provide a secure way to access the data over multiple environments. The transaction mechanism is categorized into three different types, which are as follows:

- ❑ **Local transaction**—Specifies a transaction whose statements are executed on a single transactional resource through one resource object (that is, through one session). This type of transaction is based on only local networks connected to the data source object. The local transactions are easier to use in a local network. These transactions are not supported for the transactions in multiple networks on a distributed system.
- ❑ **Distributed transaction**—Specifies a transaction whose statements are executed on one or more transactional resources through multiple resource objects. In case of a distributed transaction, the transaction manager is responsible for all the database specific operations. It must support all the ACID properties of the transaction mechanism. A distributed transaction must be synchronized and available at different locations.
- ❑ **Nested transaction**—Specifies a transaction that occurs within the reference of another transaction. It must also satisfy the ACID properties. The changes made by a nested transaction are not visible to the existing or host transaction. The changes occurred in the nested transaction can be notified to the host transaction after they have been committed. This satisfies the Isolation property of the transaction mechanism.

## Transaction Management

Transaction management in the database operation is necessary to maintain the integrity and security of data from unauthorized access. The resource manager in a transaction management system can manage local transactions because all the statements in it are associated with a single session. You need a transaction manager to manage the transactional resource objects required to execute a SQL statement. The JDBC API includes the support for transaction semantics associated with single Connection (Local Transaction) and support to participate in transactions involving multiple resource objects (Distributed Transaction). JDBC API allows you to perform the following operations to execute a transaction containing multiple resource objects:

- ❑ **Setting the Auto Commit attribute**—Allows you to specify when to end a transaction. Executing a transaction is either dependent on a JDBC driver or the underlying data source. JDBC API does not have

any method to start the transaction explicitly. New transaction generally starts when you execute a SQL statement, such as calling the execute, executeUpdate, or executeQuery methods that require a transaction.

The Auto Commit attribute of connection can be set by using the `setAutoCommit(boolean)` method of connection, and calling this method with the true argument enables auto commit. On the other hand, calling the `setAutoCommit(boolean)` method of connection with the false argument disables auto commit. Moreover, JDBC driver provider decides the default argument for the Auto Commit attribute, but in general, it is set to true. If Auto Commit is enabled, JDBC driver commits the transaction as soon as each individual SQL statement is complete. The point at which a statement is considered complete depends on the type of SQL statement as well as what the application does after executing it. For DML (Insert, Update, Delete) and DDL statements, the statement is complete as soon as its execution completes. The following code snippet is used to set the Auto Commit mode before creating a new transaction:

```
// Assume con is a Connection object
con.setAutoCommit(false);
```

If Auto Commit is disabled, the transaction must be explicitly ended by using the `commit()` or `rollback()` method. You can successfully end a transaction and save all the statements present in it by invoking the `commit()` method. However, invoking the `rollback()` method makes the transaction unsuccessful, implying that none of the statements in the transaction are saved. You can disable the auto commit option if you want to group multiple statements into a single transaction and then decide to save or not to save the statements at the end of the transaction.

- **Setting the isolation levels**—Notify the visible data within a transaction. There are four isolation levels used in transaction management, which are as follows:
  - **READ UNCOMMITTED**—Notifies the occurrence of dirty, non-repeatable, and phantom reads.
  - **READ COMMITTED**—Notifies the occurrence of non-repeatable and phantom reads.
  - **REPEATABLE READ**—Notifies the occurrence of only phantom reads.
  - **SERIALIZABLE**—Specifies that all transactions occur in a completely isolated fashion. Dirty, non-repeatable, and phantom reads cannot occur at this isolation level.

The isolation level in a transaction can be specified by using the connection object passed by the connection. The default isolation level is always specified by the underlying data source. Sometimes, the user needs to specify the isolation level explicitly. The JDBC API provides the `setTransactionIsolation(int)` method to set the transaction isolation for a transaction. Similarly, the `getTransactionIsolation()` method is used by the user to retrieve the transaction isolation associated with a connection. If a driver used in a connection does not support the isolation level, the method throws a `SQLException`.

- **Savepoints**—Set points within a transaction. A Savepoint specifies a mark up to which the user can roll back without affecting the rest of the changes of a transaction. The `DatabaseMetaData` interface available in JDBC API provides the methods to support the Savepoint within a transaction. The JDBC API provides the `setSavepoint(String)` method of the `Connection` interface to set a Savepoint in a transaction. The transaction can be rolled back up to the Savepoint by using the `rollback(Savepoint)` method of the `Connection` interface. The following code snippet shows how to set a Savepoint and rollback mechanism in a database:

```
Statement s = conn.createStatement();
int rows = s.executeUpdate("INSERT INTO TABLE1 (COLUMN1) VALUES " + "('FIRST')");
// set Savepoint
Savepoint sp = conn.setSavepoint("SAVEPOINT_1");
rows = s.executeUpdate("INSERT INTO TABLE1 (COLUMN1) " +
"VALUES ('SECOND')");
...
conn.rollback(sp);
...
conn.commit();
```

The preceding code snippet shows how to insert a record into a table in a database, and set a Savepoint, `sp`, in the database. The `INSERT` statement is successfully updated in the database. In the second insertion operation, the transaction is rolled back to the `sp` Savepoint. Therefore, this transaction is cancelled and the changes made

to the database by the second INSERT statement are undone due to the calling of the rollback. You should note that the first INSERT statement is committed even after the rollback of the second INSERT statement.

The Savepoints created during a transaction need to be released after the completion of the database transaction. The `releaseSavepoint()` method of the `Connection` interface can be called to release the Savepoints. In other words, the `releaseSavepoint()` method removes the specified Savepoint from the current transaction. After a Savepoint has been released, the attempts to reference the current transaction in a rollback operation causes a `SQLException` to be thrown.

To understand the concept better, let's create an application called TranMGT. In this application, you need to create a .java (`TransferAmount.java`) file, which is used to perform transaction management. The code snippet for the `TransferAmount.java` file is shown in Listing 3.21 (you can find the `TransferAmount.java` file in the code\JavaEE\Chapter3\TranMGT folder on the CD):

**Listing 3.21:** Showing the Code for the TransferAmount.java File

```
package com.kogent.jdbc;
import java.sql.*;
import java.util.*;
import java.io.*;
/**
 * @author Suchita
 */
public class TransferAmount {
    public static void main(String s[]) throws Exception {
        Driver d= (Driver) (Class.forName(
            "oracle.jdbc.driver.OracleDriver").newInstance());
        Properties p=new Properties();
        p.put("user","scott");
        p.put("password","tiger");
        Connection con=d.connect("jdbc:oracle:thin:@192.168.1.123:1521:XE",p);
        con.setAutoCommit(false);
        String srcaccno=s[0];
        String destaccno=s[1];
        PreparedStatement ps= con.prepareStatement(
            "update bank set bal=bal+? where accno=?");
        ps.setInt(1,500);
        ps.setString(2,destaccno);
        int i=ps.executeUpdate();
        ps.setInt(1,-500);
        ps.setString(2,srcaccno);
        int j=ps.executeUpdate();
        if (i==1&&j==1){
            con.commit();
            System.out.println("Amount transferred");
            con.close();
            return;
        }
        con.rollback();
        System.out.println("Cannot transfer the amount");
        con.close();
    }
}
//main
}//class
```

The application shown in Listing 3.21 is used to transfer money from one account to another in a transaction. A table, `bank`, must be created before executing Listing 3.21, as shown in the following code snippet:

```
Create table bank (accno varchar2(20), bal number (10, 2));
Insert into bank values ('101', 10000);
Insert into bank values ('102', 10000);
```

The `bank` table is created at the Run SQL Command Line prompt to execute the `TransferAmount.java` application. The data in the `bank` table is used by the application to transfer the amount, and the output of this table is shown in Figure 3.49:

```

Run SQL Command Line

SQL> create table bank (accno number(10,2), bal number(10,2))
Table created.

SQL> insert into bank values (102, 1000)
1 row created.

SQL> insert into bank values (101, 2000)
1 row created.

SQL>

```

Figure 3.49: Creating a Table and Inserting Data

Figure 3.49 shows the creation of the required table (bank) for the application shown in Listing 3.21. The application uses the content of the table and performs the transaction. It uses the common SQL queries to update the database and also shows the transaction management by using the Savepoints and rollbacks. Figure 3.50 shows the output of the TransferAmount class:

```

C:\Windows\system32\cmd.exe

D:\codes\JavaEE\Chapter 3\TranMGT>java com.kogent.jdbc.TransferAmount 101 102
Anytwo Trans failed
D:\codes\JavaEE\Chapter 3\TranMGT>

```

Figure 3.50: Showing the Output of TransferAmount.java

Figure 3.50 shows the output of the application created in Listing 3.30 by using the transaction properties. The application initially sets the auto-commit mode as it starts a new transaction in the given data source. Then, the required transactions update the records in the database. The transaction is committed after the updation process is complete. However, if an error occurs, the transaction can be rolled back to undo the changes made to the database.

## Summary

In this chapter, you have learned about JDBC and its basic architecture. The chapter has further explored various JDBC drivers that help an application to establish connection with a database. Next, the chapter has discussed about the new features of JDBC 4.0 and advanced topics, such as Resultset, Updateable, and Scrollable Resultset, batch update, advanced data types, and Rowset. Further, you have learned how to develop the client-server applications by using the `java.sql` and `javax.sql` packages of JDBC API. Finally, you have learned to manage and work with transactions in JDBC applications.

In the next chapter, you learn about Web technologies.

## Quick Revise

**Q1. What is JDBC?**

Ans. JDBC is a specification from Sun Microsystems that provides a standard abstraction (API / Protocol) for Java applications to communicate with different databases.

**Q2. What are the components of JDBC?**

Ans. The components of JDBC are as follows:

- The JDBC API
- The JDBC DriverManager
- The JDBC Test Suite
- The JDBC-ODBC Bridge

**Q3. Explain the different types of JDBC drivers.**

Ans. The different types of JDBC drivers are as follows:

- Type-1 Driver: Refers to the Bridge Driver (JDBC-ODBC bridge)
- Type-2 Driver: Refers to a Partly Java and Partly Native code driver
- Type-3 Driver: Refers to a pure Java driver that uses a middleware driver to connect to a database
- Type-4 Driver: Refers to a Pure Java driver, which is directly connected to a database

**Q4. Name the packages that are used to implement JDBC in an application.**

Ans. The java.sql and javax.sql packages are used to implement JDBC in an application.

**Q5. State the properties of connection pooling.**

Ans. The properties of connection pooling are as follows:

- maxStatements
- initialPoolSize
- minPoolSize
- maxPoolSize
- maxIdleTime
- propertyCycle

**Q6. Name the class that is used to establish a connection to a database.**

Ans. The java.sql.Connection class is used to obtain a connection to a database.

**Q7. Write the code statements used to register the Driver object with the DriverManager class.**

Ans. The code statements used to register the Driver object with the DriverManager class are as follows:

```
Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
where the sun.jdbc.odbc.JdbcOdbcDriver class contains the following code:
public class JdbcOdbcDriver extends ... {
    static { DriverManager.registerDriver (new sun.jdbc.odbc.JdbcOdbcDriver()); }
}
```

**Q8. List the different types of RowSet objects.**

Ans. The different types of RowSet objects are as follows:

- Connected RowSet objects
- Disconnected RowSet objects
- JdbcRowSet objects
- CachedRowSet objects
- WebRowSet objects
- FilteredRowSet object
- JoinRowSet objects

**Q9. Name the interfaces and classes of the javax.sql package that are used for connection pooling.**

Ans. The interfaces and classes of the javax.sql package used for connection pooling are as follows:

- The javax.sql.ConnectionPoolDataSource interface
- The javax.sql.PooledConnection interface
- The javax.sql.ConnectionEventListener interface
- The javax.sql.ConnectionEvent class

**Q10. List the different advanced data types.**

Ans. The different advanced data types are as follows:

- BLOB data type
- Character Large Object (CLOB) data type
- Struct data type
- Array data type
- REF data type

# 4

# Working with Servlets 3.0

<b>If you need an information on:</b>	<b>See page:</b>
Exploring the Features of Java Servlet	152
Exploring New Features in Servlet 3.0	159
Exploring the Servlet API	161
Explaining the Servlet Life Cycle	165
Creating a Sample Servlet	169
Creating a Servlet by using Annotation	173
Working with ServletConfig and ServletContext Objects	174
Working with the HttpServletRequest and HttpServletResponse Interfaces	175
Exploring Request Delegation and Request Scope	190
Implementing Servlet Collaboration	194

The Java Servlet technology provides a simple, vendor-independent mechanism to extend the functionality of a Web server. This technology provides high level, component-based, platform-independent, and server-independent standards to develop Web applications in Java. The Java Servlet technology is similar to other scripting languages, such as Common Gateway Interface (CGI) scripts, JavaScript (on the client-side), and Hypertext Preprocessor (PHP). However, servlets are more acceptable since they overcome the limitations of CGI, such as low performance and scalability.

A servlet is a simple Java class, which is dynamically loaded on a Web server and thereby enhances the functionality of the Web server. Servlets are secure and portable as they run on Java Virtual Machine (JVM) embedded with the Web server and cannot operate outside the domain of the Web server. In other words, servlets are objects that generate dynamic content after processing requests that originate from a Web browser. They are Java components that are used to create dynamic Web applications. Servlets can run on any Java-enabled platform and are usually designed to process HyperText Transfer Protocol (HTTP) requests, such as GET, and POST.

In this chapter, you learn about the Java Servlet API, version 3.0, which can be downloaded from the <http://java.sun.com/products/servlet/index.jsp> Uniform Resource Locator (URL). This API includes two packages, javax.servlet and javax.servlet.http, which provide interfaces and classes to write servlets.

The chapter first explores the general features of Java Servlet, after which you learn about the features of the latest version of Java Servlet, i.e., 3.0. Next, the chapter explains the classes and packages of the Servlet Application Programming Interface (API) used to develop Web applications. You also learn about the life cycle of a servlet and configuring a servlet in the web.xml file. In addition, you learn to create a sample servlet in two ways, by mapping it in the web.xml file and by using annotations. Moreover, the chapter also provides a walkthrough to the noteworthy interfaces of the javax.servlet and javax.servlet.http packages. Toward the end of the chapter, you learn about request delegation, request scope and servlet collaboration.

Just as Servlet 2.2 introduced the concept of self-contained Web applications, Servlet 2.3 introduced filters, Servlet 2.4 provided deprecated classes and methods, and Servlet 2.5 included support for annotations, Servlet 3.0 too includes several new features and enhancements over the previous version. You learn about them later in the chapter.

We begin the chapter by first exploring the features of Java Servlet.

## Exploring the Features of Java Servlet

Java Servlet provides various key features, such as security, performance, as well as the request and response model. Servlets are considered as a request and response model in which requests are sent by users and their appropriate responses are generated by a Web server. In addition, a key feature of a servlet is that you can create multiple instances of a servlet with different data and each servlet can be configured with different names. A Java Servlet also provides support for the security policy used to control accessibility permissions, such as a user accessing a resource. In addition, scripting languages can be used in servlets to dynamically modify or generate Hypertext Markup Language (HTML) pages. Apart from this, servlets also support various HTTP methods, such as GET and POST, which are used to redirect requests and responses.

Now, let's learn about these features in detail.

### Servlet – A Request and Response Model

Servlets are based on the programming model that accepts requests and generates responses accordingly. A developer extends the GenericServlet or HttpServlet class to create a servlet. The service() method in a servlet is defined to handle requests and responses. The following code snippet defines the service() method for the MyServletApplication servlet class:

```
import javax.servlet.*;
public class MyServletApplication extends GenericServlet {
    public void service (ServletRequest request, ServletResponse response)
        throws ServletException, IOException
    {
```

```

}
...
}
```

The `service()` method is provided with request and response parameters. These parameters encapsulate the data sent by a client, which provides access to the parameters and allows servlets to generate responses. Servlets normally use an input stream to retrieve most of their parameters, and an output stream is used to send responses. The following code snippet shows how the `request` parameter is used to invoke the `getInputStream()` method:

```

ServletInputStream input = request.getInputStream();
ServletOutputStream output = response.getOutputStream();
```

In the preceding code snippet, instances of the input and output streams are created, which may be used to read or write data.

## *Servlet and Environment State*

Servlets are similar to any other Java objects and have instance-specific data. This implies that servlets are also independent applications that run within the server environment and do not require any additional classes (which are required by some alternative server extension APIs).

When servlets are initialized, they have access to some servlet-specific configuration data. This enables the initialization of different instances of the same servlet-class with different data, and their management as differently named servlets. The data, provided with each servlet instance at the time of its initialization, also includes some information about the persistent state of an instance. The `ServletContext` object provides the ability to servlets to interact with other servlets in a Web application.

Next, let's discuss the different modes in which a servlet can be used.

## **Usage Modes**

Servlets can be used in various modes. However, these modes are not supported by all server environments. At the core of a request-response protocol, the basic modes in which servlets can be used are as follows:

- ❑ Servlets can be chained together into filter chains by the servers
- ❑ Servlets can support protocols, such as HTTP
- ❑ Servlets serve as a complete, efficient, and portable replacement for CGI-based extensions in HTTP-based applications
- ❑ Servlets can be used with HTML to dynamically generate some parts of a Web document in HTTP-based applications

Now, let's discuss the life cycle of a servlet.

## **Servlet Life Cycle**

When a server loads a servlet, the `init()` method of the servlet is executed. The servlet initialization process is completed before any client request is addressed or before the servlet is destroyed. The server calls the `init()` method once, when the server loads the servlet, and does not call the method again unless the server reloads the servlet. A server cannot reload a servlet after the servlet is destroyed. After initialization, the servlet handles client requests and generates responses. Finally, the `destroy()` method is invoked, to destroy the servlet.

Let's discuss various possible sources from where a servlet is loaded. Moreover, you also explore different situations in which a servlet is loaded.

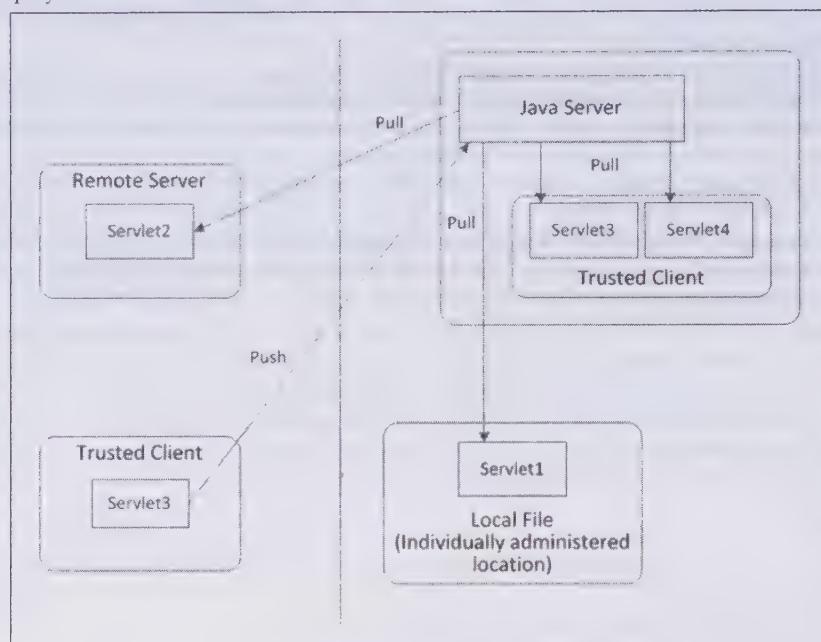
## **Possible Sources of Servlets**

When a client requests for a servlet, the server maps the request of the client and loads the servlet. The server administrator can specify the mapping of client requests to servlets in the following ways:

- ❑ Mapping client requests to a particular servlet, for example, client requests made to a specific database.
- ❑ Mapping client requests to the servlets found in an administered servlets directory. This servlet directory may be shared among different servers that share the processing load for the clients of a website.

- ❑ Configuring some servers to automatically invoke servlets that filter the output of other servlets. For example, a particular type of output generated by a servlet may invoke other servlets to carry out post processing, probably to perform format conversions.
- ❑ Invoking the specific servlets without administrative intervention by properly authorized clients.

Figure 4.1 displays the various sources of servlets:



**Figure 4.1: Displaying the Possible Sources of Loading a Servlet**

Figure 4.1 shows the various possible sources of servlets, which can be as follows:

- ❑ Individually administered locations
- ❑ Directory of servlets that are shared between servers
- ❑ Authorized clients

After having a brief understanding about life cycle of a servlet and exploring its possible sources, let's describe the primary methods of a servlet, which are `init()`, `service()`, and `destroy()`.

### Primary Servlet Methods

The following are the methods used in the life cycle of a loaded servlet:

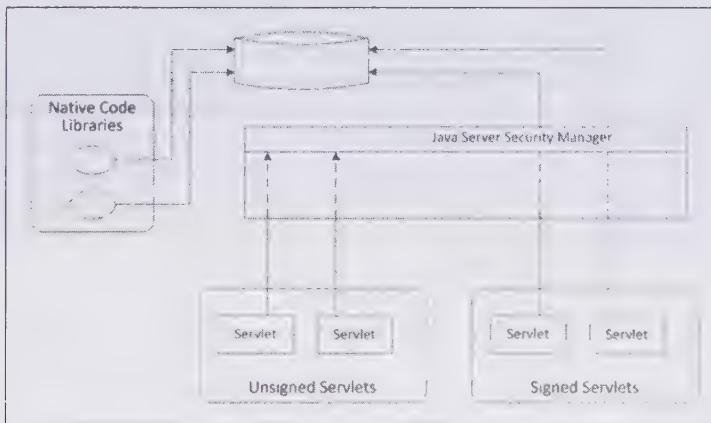
- ❑ `init()`—Refers to the method that an application server uses to load and initialize a servlet. The `init()` method is typically used to create or load objects that will be used by the servlet to handle client requests. The application server calls the `init()` method to provide a new servlet with the information related to its context.
- ❑ `service()`—Helps servlets to handle client requests. Servlets handle many requests after initialization. One `service()` method call is generated by each client request. These requests may be concurrent. This enables servlets to coordinate activities among multiple clients. To share data between requests, the class-static state may be used
- ❑ `destroy()`—Terminates an executing servlet. Servlets process client requests until they are explicitly terminated by the Web server by calling the `destroy()` method. When a servlet is destroyed, it becomes eligible for garbage collection.

## Security Features

Servlets have access to information about their clients. Peer identities can be determined reliably when servlets are used with secure protocols, such as Secure Sockets Layer (SSL). Servlets that rely on HTTP also have access to HTTP-specific authentication data.

Servlets have various advantages of Java. For example, as in Java, memory access violations and strong typing violations are also not possible with servlets. Due to these advantages, faulty servlets do not crash servers, which is common in most C language server extension environments.

Java Servlet provides strong security policy support unlike any other current server extension API. This is because a security manager, provided by all Java environments, can be used to control the permissions for actions such as accessing a network or file. Servlets are not considered trustworthy in general and are not allowed to carry out the actions by default. However, servlets that are either built into the server, or digitally signed servlets that are put into Java ARchive (JAR) files are considered trustworthy and granted permissions by the security manager. A digital signature on any executable code ensures that the organization that created and signed the code takes the guarantee of its trustworthiness. Such digital signatures cannot support answerability for the code by themselves. However, they do provide assurance about the use of that code. For example, all code that accesses network services within a corporate Intranet of the Management Information System (MIS) organization may require having a particular signature, to access those network services. The signature on the code ensures that the code does not violate any security policy. Figure 4.2 displays the approaches to server extensions depicting the Java server security manager layer used by servlets to verify permissions:



**Figure 4.2: Showing the Activities of Signed and Unsigned Servlets**

Figure 4.2 compares two approaches to server extensions:

- ❑ Activities of servlets in the case of signed servlets, which are monitored at fine granularity by the Java security manager
- ❑ Activities of native code extensions in the case of unsigned servlets, which are never monitored

In both cases, the host operating system usually provides very coarse-grained protection.

In languages such as C or scripting languages, extension APIs cannot support fine-grained access controls, even if they do allow digital signatures for their code. This explains that Pure Java™ extensions are fundamentally more secure than current competitive solutions including, in particular ActiveX of Microsoft.

There are some immediate commercial applications for such improved security technologies. At present, it is not possible for most Internet Service Providers (ISPs) to accept server extensions from their clients. The reason is that the ISPs do not have proper methods to protect themselves or their clients from the attacks building on extensions, which use native C code or CGI facilities. However, it has been observed that extensions built with pure Java Servlet can prevent modification of data. Along with the use of digitally signed code, ISPs can ensure that they safely extend their Web servers with the servlets provided by their customers.

## HTML-Aware Servlets

It has been observed that many servlets directly generate HTML formatted text, because it is easy to do so with standard internationalized Java formatted output classes, such as `java.io.PrintWriter`. To dynamically modify HTML pages or generate HTML pages, you do not have to use scripting languages.

You can also use other approaches to generate Java HTML formatted text. For example, some multi-language sites that serve pages in multiple languages, such as English and Japanese, usually maintain language-specific libraries of localized HTML template files. These sites also display the localized HTML templates by using localized message catalogs. Other sites may also have developed HTML generation packages. These packages are particularly well accustomed to other specific needs for dynamic Web page generation, for example, the ones that are closely integrated with other application toolsets.

Servlets may also be invoked by Web servers that provide complete servlet support, to help in preprocessing Web pages by using the server-side include functionality. This kind of preprocessing can be indicated to Web servers by a special HTML syntax. The following code snippet shows the syntax that is used in HTML files to indicate preprocessing of Web pages:

```
<SERVLET NAME=ServletClassName>
  <PARAM NAME=param1 VALUE=val1>
  <PARAM NAME=param2 VALUE=val2>
  If you see this text, it means that the web server providing this page
  does not support the SERVLET tag. Ask your Internet Service Provider to
  upgrade!
</SERVLET>
```

In the preceding code snippet, the invocation style usage of the `SERVLET` tag indicates that a preconfigured servlet should be loaded and initialized in cases where it has not already been done, and then invoked with a specific set of parameters. The output of that servlet is included directly in the HTML-formatted response. Apart from using the `SERVLET` tag, another invocation style can also be used, which allows the passing of the initialization arguments to the servlet and specifies its `CLASS` and `CODEBASE` values directly.

The `SERVLET` tag could be used to insert formatted data. The formatted data can be the output of a Web or database search, user-targeted advertising, or the individual views of an online magazine. HTML-aware servlets can generate arbitrary dynamic Web pages in which typical servlets accept input parameters from different sources. Some of these sources are as follows:

- The input stream of a request, perhaps from an applet
- The Uniform Resource Identifier (URI) of the request
- Any other servlet or the network service
- Parameters passed from an HTML form

The input parameters are used to generate HTML-formatted responses. A servlet often checks with one or more databases, or other data with which the servlet is configured, to decide the exact data that is to be returned with the response.

## HTTP-Specific Servlets

HTTP-specific servlets are those servlets that are used with the HTTP protocol. These servlets can support any HTTP method, such as `GET`, `POST`, and `HEAD`; redirect requests to other locations; and send HTTP-specific error messages. They can also have access to the parameters passed through standard HTML forms. HTTP-specific servlets include the HTTP method to be executed and the Uniform Resource Identifier (URI), which describes the destination of the request. The following code snippet shows some of the methods used in HTTP-specific servlets:

```
String method = request.getMethod(); // e.g. POST
String uri = request.getRequestURI();
String name = request.getParameter("name");

String phone = request.getParameter ("phone");
String card = request.getParameter ("creditcard");
```

In HTTP-specific servlets, request and response data is always provided in the Multipurpose Internet Mail Extensions (MIME) format. This implies that the servlet first specifies the data type and then writes the encoded data. This allows servlets to refer the data format regarding arbitrary sources of input data, and then return the data in the appropriate form for the particular request. Examples of request and response data formats are HTML, graphics formats, such as Joint Photographic Experts Group (JPEG) or Moving Picture Experts Group (MPEG), and data formats that are used by some applications.

In most applications, HTTP servlets are considered better than CGI programs in terms of performance, flexibility, portability, and security. Therefore, rather than using CGI or a C language plug-in, write your next server extension by using the Java Servlet API.

## Performance Features

Let's discuss the performance feature of servlets. One of the most prominent performance features of Java Servlets is that a new process need not be created for every new request received. In most environments, several servlets run in parallel to the server within the same process. This is a big advantage. When you use servlets in such environments with HTTP, performance is assumed to be much better than what it would be if the CGI or Fast-CGI approach was used.

Figure 4.3 displays the different approaches to depict the functionality and performance of a servlet:

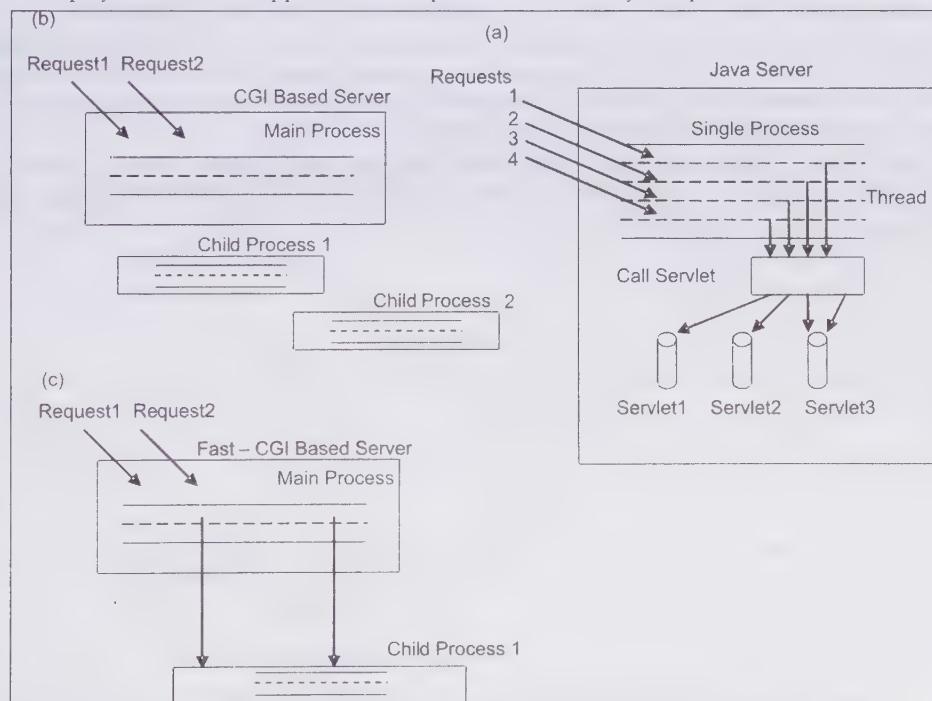


Figure 4.3: Showing Different Servlet Functional Approaches

Figure 4.3 compares the following three server extension approaches:

- ❑ The servlet approach, which allows embedding to be supported inside a server, as shown by section (a) of Figure 4.3
- ❑ The CGI approach, which involves creating a new child process with every new request, as shown by section (b) of Figure 4.3
- ❑ The Fast-CGI approach, which involves creating one child process for multiple requests, as shown by section (c) of Figure 4.3

The difference between these approaches is that the servlet approach require only lightweight thread context switches, whereas Fast-CGI involves heavyweight process context switching on each request, and regular CGI requires even heavier weight process start-up and initialization code on each request. After a servlet is initialized, it can address requests from multiple clients in most environments. This spreads the cost of initialization over many methods. It also enables all client requests made to a service to share data and communication resources as well as use system caches effectively.

Java Servlet can take the advantage of additional processors, with multiple implementations of JVM. The Java Virtual Machine (JVM) enables you to scale applications from entry-level servers to the mainframe class multiprocessors. This also helps to provide better throughput and timely response to clients. Pure Java programs are platform-independent; therefore, they can run on any operating system. In other words, you can select any operating system that best addresses your requirements for any given application. The implementation of Java Servlet is beneficial for many large Web-based applications that use Java and other Internet technologies.

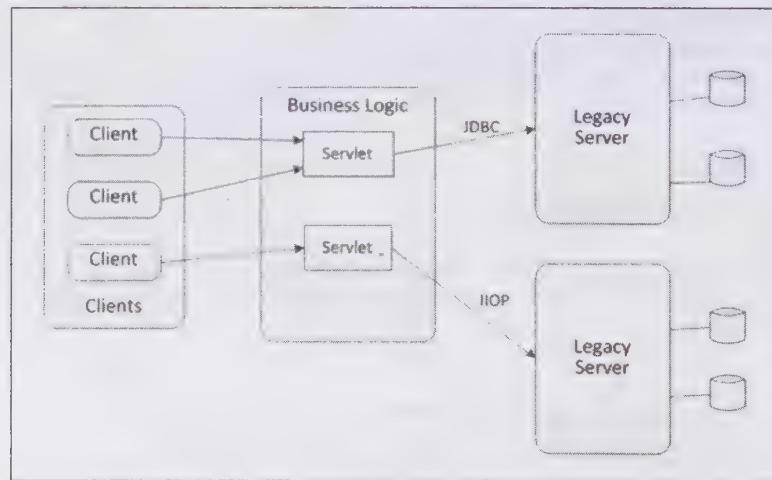
### 3-Tier Applications

Using Java Servlet helps a user to opt for 3-tier applications. Many organizations require you to use multi-tier applications. Many clients and single server models are giving way to a single application, which includes many servers that exchange data between each other.

The first tier of an application may use any number of Java enabled Web browsers. The browsers can include those running on Network Computers (NCs) as well as on personal computers or workstations. Complex tasks related to the user interface are handled in the first tier by Java applets downloaded from two-tier servers. Simpler tasks can be handled by using standard HTML forms.

The second tier of an application involves servlets, which implement the specific business rules and business logic of the application. Such rules can include application-specific access controls for sensitive corporate data.

Figure 4.4 displays the 3-tier structure of servlets:



**Figure 4.4: Showing 3-Tier Structure of Servlets**

Figure 4.4 shows how servlets can be used to connect the second tier of an application to the first tier. A variety of client technologies may be used to connect to other tiers.

The third tier of an application involves data repositories. This tier can be accessed by using relational database interfaces such as Java Database Connectivity (JDBC), or other interfaces supported by legacy data, for example, Remote Procedure Call (RPC)-like protocols such as Open Network Computing (ONC) RPC, Distributed Computing Environment (DCE) RPC, and Common Object Request Broker Architecture (CORBA)/Internet Inter-Orb Protocol (IIOP).

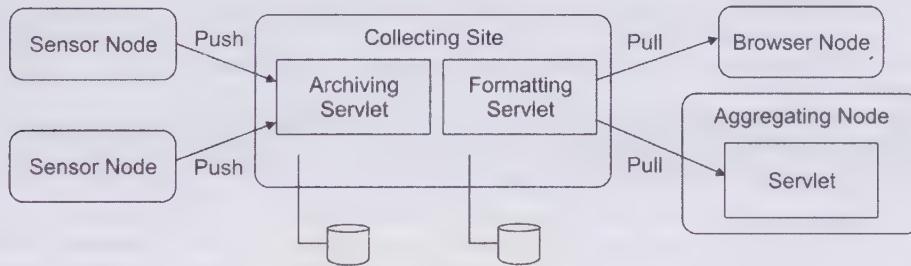
## Web Publishing System

Many organizations have large collections of data. They have to manage and publish the data, which is usually flexible and tends to change. For example, an organization may maintain a collection of both historical and real-time weather data that needs to be presented in easily understood formats in the form of a response to the current application.

In this case, you can use a Web publishing system that provides sites to access historical and real-time weather data from a database. In other words, the required data can be accessed from the database by using JDBC. The weather data (including temperature, wind, rainfall, a frame of image data, or a stream of MPEG data) is sent by a Java equipped remote recording station to a site receiving the data. The servlet processing the data at the collecting site may selectively store the data. For example, it may store data of a specific time period, such as the last two weeks, or it may discard some data immediately.

The collecting sites may receive queries from other sites, such as individual browsers or other collecting sites, to return data in a specific format. In that case, servlets process the saved data and return the response of the query in the appropriate format, such as a Web page with current data and historical tables and graphs, to a user. Some servlets can also perform administrative tasks, such as archiving and deleting data, or pulling data from staging areas, as part of an automated data distribution system.

Figure 4.5 shows communication between two servlets:



**Figure 4.5: Showing Servlet Communication**

Figure 4.5 shows two kinds of servlets used by a collecting site. One is used by the remote sensor nodes to push data to the collecting site, and the other is used by clients to pull data from the collecting site in a specific format. Such clients can include tertiary nodes, which assemble data from multiple collecting sites.

After exploring the features of Java Servlet, let's discuss the new enhancements that have been introduced in Servlet 3.0.

## Exploring New Features in Servlet 3.0

Various new features have been introduced in Servlet 3.0, the new version of Java Servlet. However, before discussing these features, let's briefly review Servlet 2.5, which was the previous version of Java Servlet. The features of Servlet 2.5 have been retained in Servlet 3.0; therefore, let's first discuss the features of Servlet 2.5.

Servlet 2.5 introduced several new advancements, such as changes in the web.xml file and in filter mapping, and support for annotations. We assume that you are familiar with the classes and methods of the previous versions of Java Servlet. The following features and enhancements have been included in Servlet 2.5:

- ❑ Provides support for the new language features of J2SE 5.0, such as generics, the new enum type, and autoboxing. Note that the minimum platform requirement for the Servlet 2.5 specification is JDK 1.5 (J2SE 5.0). This implies that Servlet 2.5 cannot be used in versions below JDK 1.5.
- ❑ Provides support for annotations. An annotation is a special form of syntactic metadata that is added to the Java source code in such a way that processors may alter their behavior based on the metadata information.
- ❑ Introduces changes in the web.xml file that have allowed Java developers to easily configure the resources of an application. In addition, in Servlet 2.5, you can bind all servlets to a filter simultaneously, which was not possible in the earlier versions. Now, it is possible to use an asterisk (\*) within the <filter-mapping>

element as the value of the <servlet-name> element to represent all servlets. Moreover, you can provide multiple matching criteria in the same entry while writing the <servlet-mapping> or <filter-mapping> elements in Servlet 2.5. Earlier, the <servlet-mapping> element supported a single <url-pattern> element; however, now in Servlet 2.5, the <servlet-mapping> element supports more than one url pattern.

- Removes two major restrictions in error-handling and session tracking. Earlier, there was a rule that the resource configured in the <error-page> element could not call the `setStatus()` method to alter the code provided to display an error message. However, the Servlet 2.5 specification no longer prevents the error-handling page to produce a non-error response. Therefore, the error-handling page can do far more than just show an error. Moreover, in the case of session tracking, Servlet 2.5 does not allow you to set response headers for a servlet called by the `RequestDispatcher's include()` method.
- Clarifies certain options available in the Servlet 2.4 specification. These clarifications are as follows:
  - According to the Servlet 2.4 specification, before calling the `request.getReader()` method, you need to call the `request.setCharacterEncoding()` method. However, the specification does not clarify why this needs to be done. The Servlet 2.5 specification describes this properly and states that if you ignore this specification option, the `request.getReader()` method is not executed.
  - The Servlet 2.4 specification does not define what happens if a session id is not specified. However, the Servlet 2.5 specification states that the `HttpServletRequest` interface will return false if the session id is not specified.
  - The Servlet 2.4 specification states that a response should be committed in most situations. The following code snippet shows a situation where the amount of content specified in the `setContentLength()` method of the response is not greater than zero and has been returned to the response:

```
res.setHeader("Host", "localhost");
res.setHeader("Pragma", "no-cache");
res.setHeader("Content-Length", "0");
res.setHeader("Location", "www.kogentindia.com")
```

In the preceding code snippet, a servlet technically ignores the `Location` header because the response must be committed immediately, as the zero byte content length is satisfied. However, the Servlet 2.5 specification states that the response should be committed when the amount of content specified in the `setContentLength()` method of the response is greater than zero, and is returned to the response.

- The Servlet 2.5 specification changes the rules of cross context session management. This feature is used when Servlets dispatch requests from one context to another. The Servlet 2.5 specification states that resources present in a context can refer to the session of that context, irrespective of the origin of a request.

All the preceding features of Servlet 2.5 are also retained in Servlet 3.0, along with the introduction of some new features. The following new features have been included in Servlet 3.0:

- Introduction of annotations as an enhanced feature. When you use annotations to create servlets, using Deployment Descriptor in the form of the `web.xml` file to map the servlet is optional. Developers can directly mark a servlet by using annotations. However, if Deployment Descriptor is also used, it overrides the configuration information provided by using annotations. Some of the annotations introduced in Servlet 3.0 are as follows:
  - **@WebServlet annotation** – Marks the annotated class as a servlet
  - **@WebInitParam annotation** – Specifies the init parameters to be passed to the servlet.
  - **@WebListener annotation** – Marks the annotated class as a listener
- Introduction of Web fragments to implement the concept of modularization of the `web.xml` file. A Web fragment can be assumed as a segment of the `web.xml` file. This implies that one or more Web fragments constitute a `web.xml` file. All configuration information specified in the `web.xml` file can also be specified in the `web-fragment.xml` file. When multiple `web-fragment.xml` files exist for a Web application, the order of their execution can be decided by specifying absolute or relative ordering.

- Introduction of the concept of asynchronous processing, which is implemented by setting the `asyncSupported` attribute of the `@WebServlet` or `@WebFilter` annotation. The `asyncSupported` attribute takes a Boolean value, which should be set to true to obtain asynchronous processing. In asynchronous processing, a servlet does not have to wait for a response from the request made for a resource, such as database.

After discussing the new features and enhancements of Servlet 3.0, let's now explore the Servlet API.

## Exploring the Servlet API

The Servlet API is a part of the Java Servlet specification designed by the Java Community Process (JCP). This API is supported by all servlet containers, such as Tomcat and WebLogic. The Servlet API contains classes and interfaces that define a standard contract between a servlet class and Servlet container. These classes and interfaces are available in the following two packages of the Servlet API:

- `javax.servlet`
- `javax.servlet.http`

Let's learn about these packages in detail in the following sections.

### *Describing the javax.servlet Package*

The `javax.servlet` package contains some interfaces and classes that allow a servlet to access the basic services provided by a Servlet container. The Servlet container provides the implementation of the classes and interfaces packaged under the `javax.servlet` package.

The central abstraction of the Servlet API is the `Servlet` interface. The two classes in the Servlet API that implement the `Servlet` interface are `GenericServlet` and `HttpServlet`. Generally, developers implement the `HttpServlet` interface to create their servlets for Web applications that employ the HTTP protocol.

The basic `Servlet` interface defines a `service()` method to handle client requests. This method is called for each request that is routed to an instance of a servlet by the Servlet container.

The contract between a Web application and a Web container is provided by the `javax.servlet` package. This allows Servlet container vendors to focus on developing the container in the manner most suited to their requirements (or those of their customers), as long as the package provides the specified implementations of the interfaces used in a servlet. The package provides a standard library to process client requests and develop servlet-based applications, from a developer's perspective.

The `Servlet` interface that defines the core structure of a servlet is provided in the `javax.servlet` package, which is the basis for all servlet implementations. However, for most servlet implementations, the subclass from a defined implementation of this interface provides the basis for a Web application.

The various interfaces and classes, such as `ServletConfig` and `ServletContext`, provide the additional services to a developer. An example of such a service is the Servlet container that provides a servlet with access to a client request through a standard interface. The `javax.servlet` package, therefore, provides the basis to develop a cross-platform, cross-servlet container Web application, and allows programmers to focus on developing a Web application.

Developers sometimes also use the `javax.servlet.http` package. Additionally, you need the `javax.servlet` package to build servlet implementations that use a non-HTTP protocol. For example, you can extend classes from the `javax.servlet` package to implement a Simple Mail Transfer Protocol (SMTP) servlet that provides an e-mail service to clients.

Let's now discuss the interfaces, classes, and exception classes of the `javax.servlet` package.

### *Explaining the Interfaces and Classes of the javax.servlet Package*

The `javax.servlet` package comprises fourteen interfaces. While building an application, a programmer can implement seven interfaces, such as `Servlet`, and `ServletRequestListener`. A Servlet container provides the implementation for the following seven interfaces:

- `ServletConfig`

- ServletContext
- ServletRequest
- ServletResponse
- RequestDispatcher
- FilterChain
- FilterConfig

The Servlet container must provide an object for the preceding interfaces to a servlet. The `getServletContext()` method is probably the most important method of the `ServletConfig` Interface. This method returns the `ServletContext` object, which communicates with the Servlet container when you want to perform some action, such as writing to a log file or dispatching requests. There is only one `ServletContext` object for a Web application per JVM. This object is initialized when the Web application starts and is destroyed only when the Web application shuts down. One useful application of the `ServletContext` object is as a persistence mechanism. A programmer may store attributes in the `ServletContext` interface so that they are available throughout the execution of an application and not just for the duration of a request for a resource.

The Servlet container provides the classes that implement the `ServletRequest` and `ServletResponse` interfaces. These classes provide client request information to a servlet and the object used to send a response to the client.

An object defined by the `RequestDispatcher` interface manages client requests by directing them to an appropriate resource on the server.

The `FilterChain`, `FilterConfig`, and `Filter` interfaces are used to implement the filtering functionality in an application. You can also combine the interfaces into chains, implying that you can chain them such that before being processed by a container, the request is filtered through each filter defined in the application. The response goes down the chain in reverse.

A programmer building a Web application implements the following seven interfaces:

- Servlet
- ServletRequestListener
- ServletRequestAttributeListener
- ServletContextListener
- ServletContextAttributeListener
- SingleThreadModel
- Filter

The preceding interfaces are defined so that a Servlet container can invoke the methods defined in the interfaces. Therefore, the Servlet container needs to know only the methods defined in the interfaces. The details of the implementation of the methods are provided by the developers.

The event classes used to notify the changes made to the `ServletContext` object and its attributes are `ServletContextEvent` and `ServletContextAttributeEvent`, respectively.

Initially, the system creates a single instance of a servlet. If a new request is received while the previous one is being processed, a new thread is created for each new user request, with multiple threads running concurrently. This implies that the `doGet()` and `doPost()` methods need to carefully synchronize the access to fields and other shared data because, multiple executing threads may access the data simultaneously. You can implement the `SingleThreadModel` interface in a servlet, if you want to prevent this multithreaded access, as shown by the following code snippet:

```
public class YourServlet extends HttpServlet implements SingleThreadModel
{
    ...
}
```

If you implement the `SingleThreadModel` interface, the system ensures that a single instance of a servlet is never accessed by more than one request thread. This is implemented by queuing all the requests and passing them

one by one to a single servlet instance. However, the server can create a pool of multiple instances, with each addressing one request at a time. This implies that there is no need to worry about simultaneous access to regular fields (instance variables) of a servlet. However, access to class variables (static fields) or shared data stored outside the servlet still needs to be synchronized.

Synchronous and frequent access to servlets can significantly hurt performance (latency). The server remains idle instead of handling pending requests, when a servlet waits for Input/Output (I/O). Therefore, think twice before using the `SingleThreadModel` approach.

#### **Note**

*The `SingleThreadModel` interface has been deprecated as of Java Servlet API 2.4, with no direct replacement*

Two classes, `ServletRequestEvent` and `ServletRequestAttributeEvent`, are used to indicate the life cycle events and attribute events for a `ServletRequest` object. The `ServletContext` object of a Web application is the source of the event.

To read or send binary data to or from a client, the `ServletInputStream` and `ServletOutputStream` classes provide input and output streams, respectively.

Useful implementations of the `ServletRequest` and `ServletResponse` interfaces are provided by the wrapper classes `ServletRequestWrapper` and `ServletResponseWrapper`, respectively. These implementations can be subclassed to allow programmers to adapt or enhance the functionality of the wrapped object for their own Web application. This can be done to implement a basic protocol agreed between a client and a server or to transparently adapt the requests or responses to a particular format that the Web application requires.

## Explaining the Exception Classes of the `javax.servlet` Package

The following two exceptions are present in the `javax.servlet` package:

- ❑ `ServletException`
- ❑ `UnavailableException`

Generally, the `ServletException` exception is thrown by a servlet to indicate a problem with a user request. The problem may be in processing a request or sending of a response.

Whenever the `ServletException` exception is thrown to a Servlet container, an application loses control of the request being processed. It is the responsibility of the Servlet container to clean up the request and return a response to a client. Instead of sending a response, the Servlet container may also return an error page to the client indicating a server problem, depending on implementation and configuration of the container.

A `ServletException` exception should be thrown only as a last resort. The preferred approach to deal with an insuperable problem is to handle the problem and then return the type of the problem to the client.

An application throws the `UnavailableException` exception when a requested resource is not available. The resource can be a servlet, a filter, or any other configuration details required by the servlet to process requests, such as a database, a domain name server, or another servlet.

## Exploring the `javax.servlet.http` Package

The `javax.servlet.http` package contains some interfaces and classes that enhance the basic functionality of a servlet to support HTTP-specific features, such as request and response headers, different request methods, and cookies.

As discussed earlier, there are two classes (`GenericServlet` and `HttpServlet`) in the Servlet API, which implement the `Servlet` interface. `HttpServlet` is an abstract class that extends the `GenericServlet` base class and provides a framework to handle the HTTP protocol. The following section discusses the classes and interfaces of the `javax.servlet.http` package.

## Explaining the Interfaces of the `javax.servlet.http` Package

The `javax.servlet.http` package comprises the following eight interfaces:

- ❑ `HttpServletRequest`

- ❑ HttpServletResponse
- ❑ HttpSession
- ❑ HttpSessionBindingListener
- ❑ HttpSessionContext
- ❑ HttpSessionActivationListener
- ❑ HttpSessionAttributeListener
- ❑ HttpSessionListener

The `HttpServletRequest` interface retrieves data from a client to a servlet for use in the `HttpServlet.service()` method. This interface allows the `service()` method to access the HTTP protocol specified header information. The `HttpServletResponse` interface allows the `service()` method of a servlet to manipulate the HTTP protocol specified header information. This interface also returns the data to its client. The `HttpSession` interface is used to maintain a session between an HTTP client and the HTTP server. This session is used to maintain the state and user identity across multiple connections or requests during a given period.

The `HttpSessionContext` interface provides a group of the `HttpSessions` objects associated with a single session. The `getSessionContext()` method is used by a servlet to get the `HttpSessionContext` object. The `HttpSessionContext` interface also provides various methods to servlets to list the IDs or retrieve a session based on the ID. The `HttpSessionActivationListener` interface notifies a Web container about the activation or passivation of a session object. The `HttpSessionAttribute` interface is implemented to receive the notifications of changes in the attribute lists of sessions within a Web application. The `HttpSessionListener` interface notifies the changes made in the active sessions in a Web application.

#### **Note**

*The `HttpSessionContext` interface has been deprecated as of Java™ Servlet API 2.1 for security reasons, with no replacement.*

The `HttpSessionBindingListener` interface has the `valueBound()` and `valueUnbound()` methods to notify a listener that it is being bound to a session or unbound from a session.

Next, we discuss the classes of the `javax.servlet.http` package.

### Explaining the Classes of the `javax.servlet.http` Package

Apart from the interfaces, the `javax.servlet.http` package also has various classes, which are as follows:

- ❑ Cookie
- ❑ HttpServlet
- ❑ HttpServletRequestWrapper
- ❑ HttpServletResponseWrapper
- ❑ HttpSessionBindingEvent
- ❑ HttpSessionEvent
- ❑ HttpUtils

`HttpServlet` is an abstract class that simplifies the writing of HTTP servlets. As `HttpServlet` is an abstract class, servlet programmers must override at least one of the following methods: `doGet()`, `doPost()`, `doPut()`, `doDelete()` and `getServletInfo()`. The `HttpServletRequestWrapper` class provides a convenient implementation of the `HttpServletRequest` interface to adapt a request to a servlet. Similarly, the `HttpServletResponseWrapper` class provides a convenient implementation of the `HttpServletResponse` interface to adapt the response from a servlet. The `HttpSessionBindingEvent` class communicates to the `HttpSessionBindingListener` object regarding bounding to or unbinding from the `HttpSession` value. The `HttpSessionEvent` class represents event notifications for changes in a session within a Web application. As already discussed, the `HttpSession` interface maintains a session and manages the session with the help of the `Cookie` class. The `HttpUtils` class is a collection of static utility methods useful to HTTP servlets.

After learning about the Servlet API, let's discuss the servlet life cycle next.

## Explaining the Servlet Life Cycle

Servlets follow a life cycle that governs the multithreaded environment in which the servlets run. It also provides a clear perception about some of the mechanisms available to a developer to share server-side resources.

The primary reason why servlets and JavaServer Pages (JSP) outperform traditional CGI is the servlet life cycle. Servlets follow a three-phase life cycle, namely initialization, service, and destruction. This three-phase life cycle is opposed to the single-phase life cycle. Of the three phases, the initialization and destruction phases are performed only once while the service phase is carried out many times.

The first phase of the servlet life cycle is initialization. It represents the creation and initialization of the resources the servlet may need in response to service requests. All servlets must implement the `javax.servlet.Servlet` interface, which defines the `init()` method that corresponds to the initialization phase of a servlet life cycle. As soon as a servlet is loaded in a container, the `init()` method is invoked before servicing any requests.

The second phase of a servlet life cycle is the service phase. This phase of the servlet life cycle represents all the interactions carried out along with the requests that are handled by the servlet until it is destroyed. The service phase of the servlet life cycle corresponds to the `service()` method of the `Servlet` interface. The `service()` method of a servlet is invoked once for every request. Then, its sole responsibility is to generate the response to that request.

The `service()` method takes two object parameters, `javax.servlet.ServletRequest` and `javax.servlet.ServletResponse`. These two objects represent a request for dynamic resource from a client and a response sent by a servlet to the client, respectively. A servlet is usually multithreaded. This implies that a single instance of a servlet is loaded by a servlet container at a given instance, by default. The initialization of the servlet is done only once, and after that, each request is handled concurrently by threads executing the `service()` method of the servlet.

The destruction phase is the third and final phase of the servlet life cycle. This phase represents the termination of the servlet execution and its removal from the container. The destruction phase corresponds to the `destroy()` method of the `Servlet` interface. The container calls the `destroy()` method when a servlet is to be removed from the container.

The invocation of the `destroy()` method enables the servlet to terminate gracefully and clean up any resources held or created by it during execution. To efficiently manage application resources, a servlet should properly use all the three phases of its life cycle. A servlet loads all the required resources during the initialization phase, which may be needed to service client requests. The resources are used during the service phase and then can be given up in the destruction phase.

We have discussed the three events or phases that form the life cycle of a servlet. However, there are many more methods that need to be considered by a Web developer. HTTP is primarily used to access content on the Internet. Though a basic servlet does not know anything about HTTP, a special implementation of the servlet, namely `javax.servlet.http.HttpServlet` has been specifically designed for this purpose.

When the Servlet container creates a servlet for the first time, the container invokes the `init()` method of the servlet. After this, each user request results in the creation of a thread, which calls the `service()` method of the respective instance. Though the servlet in question can implement a special interface, (`SingleThreadModel`), which stipulates that not only a single thread is permitted to run at a time, but also multiple concurrent requests can be made. The `service()` method then calls the `doGet()`, `doPost()`, or any other `doXXX()` method. However, the calling of the `doXXX()` method depends on the type of HTTP request received. Finally, when the server decides to unload a servlet, it first calls the servlet's `destroy()` method.

Let's now discuss the various methods used in the life cycle of the servlet.

## The `init()` Method

As mentioned earlier, the `init()` method is called when a servlet is created for the first time. It is not called again for other user requests. Therefore, the `init()` method is used only for one-time initializations. A servlet is normally created when a user invokes a URL corresponding to the servlet, for the first time; however, the servlet is loaded on the server when a Servlet container maps the user request to the servlet. The following code snippet shows the `init()` method definition:

```
public void init() throws ServletException
{
    // Initialization code...
}
```

Reading server-specific initialization parameters is one of the most common tasks that the `init()` method performs. For example, a servlet might need to know various information details, such as database settings, password files, server-specific performance parameters, hit count files, or serialized cookie data from previous requests.

When you need to read the initialization parameters, you have to first obtain a `ServletConfig` object by using the `getServletConfig()` method, and then call the `getInitParameter()` method on the result. The following code snippet shows how to obtain a `ServletConfig` object:

```
public void init() throws ServletException
{
    ServletConfig config = getServletConfig();
    String param1 = config.getInitParameter("parameter1");
}
```

In the preceding code snippet, notice that the `init()` method uses the `getServletConfig()` method to obtain a reference to the `ServletConfig` object. The object has a `getInitParameter()` method, which can be used to look up the initialization parameters associated with the servlet. Similar to the `getParameter()` method used in the `init()` method of applets, both the input (i.e., the name of the parameter) and the output (i.e., the parameter value) are nothing but Strings.

You can read the initialization parameters by calling the `getInitParameter()` method of the `ServletConfig` object. However, setting up these initialization parameters is the job of the `web.xml` file, which is called Deployment Descriptor, which we discuss in the *Understanding Servlet Configuration* section of this chapter.

## The `service()` Method

Each time a server receives a request for a servlet, the server spawns a new thread and calls for the `service()` method. It is possible that the server spawns a new thread by reusing an idle thread from a thread pool. The `service()` method verifies the HTTP request type (GET, POST, PUT, DELETE) and accordingly calls the `doGet()`, `doPost()`, `doPut()`, `doDelete()` methods. A normal request for a URL or a request from an HTML form that has no METHOD specified results in a GET request. Apart from the GET request, an HTML form can also specify POST as the request method type. The following code snippet explains the implementation of the POST method:

```
<html>
<form name="greetForm" method="post">
```

Now, if you have a servlet that needs to handle both POST and GET requests identically, you may be tempted to override the `service()` method directly rather than implementing both the `doGet()` and `doPost()` methods. However, remember, this is not a good idea. Instead, just you can use the `doPost()` method to call the `doGet()` method (or vice versa), as shown in the following code snippet:

```
@Override
public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
{
    // Servlet code
}
@Override
public void doPost(HttpServletRequest request, HttpServletResponse response)
```

```

throws ServletException, IOException
{
    doGet(request, response);
}

```

In the preceding code snippet, the @Override annotation is used. Though this approach takes a couple of extra lines of code, it has several advantages over the approach of directly overriding of the service() method. One advantage is that you can add support for other HTTP request methods later by adding the doPut(), doTrace() methods in a subclass. Another advantage of using this approach is that you can add support to retrieve the date on which modifications on data have been made by adding the getLastModified() method. However, overriding the service() method eliminates this option because the getLastModified() method is invoked by the default service() method. Finally, as an added advantage, you can get automatic support for the HEAD, OPTION, and TRACE requests.

**NOTE**

If a servlet needs to handle both GET and POST identically, the doPost() method should call the doGet() method or vice versa. Remember, you should not override the service() method directly.

During the entire request and response process, most of the time, you only care about the GET or POST requests. Therefore, you override either the doGet() method or the doPost() method or both. However, if required, you can also override the following methods depending upon the request types:

- The doDelete() method for DELETE requests
- The doPut() method for PUT requests
- The doOptions() method for OPTIONS requests
- The doTrace() method for TRACE requests

Remember, however, that you have automatic support for OPTIONS and TRACE.

The doHead() method is not provided in versions 2.1 and 2.2 of the Servlet API, because in those versions the system answers HEAD requests automatically by using the status line and header settings of the doGet() method. However, the doHead() method is included in version 2.3 to enable the generation of responses to HEAD requests.

## The destroy() Method

The destroy() method runs only once during the lifetime of a servlet, and signals the end of the servlet instance. A Servlet container holds a servlet instance till the servlet is active or its destroy() method is called. The following code snippet shows the method signature of the destroy() method:

```
public void destroy()
```

As soon as the destroy() method is activated, the Servlet container releases the servlet instance.

**NOTE**

It is not recommended to implement the finalize() method in the servlet object; instead, provide the code for the finalization tasks of an application in the destroy() method.

After learning about the life cycle of a servlet, let's understand servlet configuration.

## Understanding Servlet Configuration

You may sometimes need to provide initial configuration information for a servlet. The configuration information for the servlet may include a String or a set of String values, listed in the web.xml file as initialization parameters required during the initialization of the servlet. Due to this functionality, servlets are allowed to have initial parameters specified outside of the compiled code and changed without requiring recompiling of the servlet. Each servlet has an object associated with it, called ServletConfig. This object is created by the container and implements the javax.servlet.ServletConfig interface. The ServletConfig object contains the initialization parameter and you can retrieve the reference of the ServletConfig object by invoking the getServletConfig() method. The following code snippet shows the method provided by the ServletConfig object to access an initialization parameter:

```
getInitParameter(String name)
```

The `getInitParameter()` method returns a `String` object that contains the value of the named initialization parameter or `null`, if the parameter does not exist. The following code snippet shows the `getInitParameterNames()` method of the `ServletConfig` object:

```
getInitParameterNames()
```

The `getInitParameterNames()` method returns the names of the initialization parameters of a servlet as an enumeration of `String` objects. An empty enumeration is returned by the method if the servlet has no initialization parameters.

A servlet can define initial parameters by using the `init-param`, `param-name`, and `param-value` elements in the `web.xml` file. Each `init-param` element defines one initial parameter. This initial parameter must contain a parameter name and value specified by the `param-name` and `param-value` child elements, respectively. A servlet may have as many initial parameters as needed. However, note that initial parameter information for a specific servlet should be specified within the `<servlet>` element for that particular servlet.

A servlet can be configured with the help of the `web.xml` file, which lies in the `WEB-INF` directory of a Web application. This file controls many behavioral aspects of the servlet and JSP. Many servers provide graphical interfaces that allow you to specify initialization parameters and control various behavioral aspects of servlets and JSP pages.

These graphical interfaces are server-specific. However, these interfaces also use the `web.xml` file, which is completely portable. The `web.xml` file contains an XML header, a DOCTYPE declaration, and a `web-app` element. To specify initialization parameters, the `web-app` element must contain a `<servlet>` element with three subelements, which are as follows:

- servlet-name
- servlet-class
- init-param

The `<servlet-name>` element specifies the name that helps you to access the servlet. The `<servlet-class>` element specifies the fully qualified (that is, the package name is included with the servlet class name) class name of the servlet, and the `init-param` element specifies names and values for parameter initialization.

Several changes have been introduced since the Servlet 2.5 specification to the `web.xml` file format to make its use more convenient. For example, in the previous versions of Java Servlet, only one servlet could be bound to a filter at a time, as shown in the following code snippet:

```
<filter-mapping>
  <filter-name>MyFilter</filter-name>
  <servlet-name>MyServlet</servlet-name>
</filter-mapping>
```

In the preceding code snippet, the `MyServlet` servlet is mapped to the `MyFilter` filter; however, in Servlet 3.0, you can bind all servlets at once by using an asterisk. The asterisk is used in the `<servlet-name>` element to represent all servlets. The following code snippet shows you how to bind all servlets to the `MyFilter` filter at once:

```
<filter-mapping>
  <filter-name>MyFilter</filter-name>
  <servlet-name>*</servlet-name>
</filter-mapping>
```

Apart from this, Servlet 3.0 provides support for configuring multiple patterns for a filter. In the earlier versions of Java Servlet, you could use the `<filter-mapping>` element with just one `<url-pattern>` element, whereas Servlet 3.0 supports multiple `<url-pattern>` elements. Consider the following example in which multiple `<url-pattern>` elements have been provided for the `MyServlet` servlet:

```
<servlet-mapping>
  <servlet-name>MyServlet</servlet-name>
  <url-pattern>/servlet/*<url-pattern>
    <url-pattern>/servlet/*<url-pattern>
</servlet-mapping>
```

Listing 4.1 provides a sample `web.xml` file, which maps to a single servlet class named `FirstServlet`:

**Listing 4.1:** Displaying the Code for the web.xml File

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">

  <servlet>
    <servlet-name>FirstServlet</servlet-name>
    <servlet-class>FirstServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>FirstServlet</servlet-name>
    <url-pattern>/FirstServlet</url-pattern>
  </servlet-mapping>
  <session-config>
    <session-timeout>
      30
    </session-timeout>
  </session-config>
</web-app>
```

In Listing 4.1, the `web.xml` file contains an XML header, a DOCTYPE declaration, and a `web-app` element. The `<servlet-name>` element contains the name of the servlet and `<servlet-class>` contains the fully qualified class name of the servlet. The `<servlet-mapping>` element contains two subelements, `<servlet-name>` and `<url-pattern>`. The `<servlet-name>` element contains the name of the servlet as provided in the `<servlet>` element.

Now, let's learn how to create a servlet.

## Creating a Sample Servlet

Let's create a simple servlet in the `FirstApp` application, which handles HTTP request and sets the value of the `name` attribute at the initialization of the servlet, which is displayed on the browser. Moreover, the value of the `init-param, greeting`, is set in the `web.xml` file. Listing 4.2 provides the code for `FirstServlet` (you can find this file on the CD in the `code\JavaEE\Chapter4\FirstApp\src\com\kogent` folder):

**Listing 4.2:** Showing the Code for the FirstServlet.java File

```
package com.kogent;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class FirstServlet extends HttpServlet
{
    @Override
    public void init(ServletConfig config) throws ServletException
    {
        super.init(config);
        config.getServletContext().setAttribute("name", "Pallavi Sharma");
    }
    @Override
    public void doGet(HttpServletRequest request, HttpServletResponse response) throws
    IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter printwrite = response.getWriter();
        printwrite.println("<html>");
        printwrite.println("<head>");
        String greet;
        String name;
        greet = getServletConfig().getInitParameter("greeting");
        name=getServletContext().getAttribute("name").toString();
        printwrite.println("<title>" + greet + "</title>");
        printwrite.println("</head>");
```

```

        printwrite.println("<body>");
        printwrite.println("<h1>" + greet + "</h1>");
        printwrite.println("<h2>" + name + "</h2>");
        printwrite.println("</body>");
        printwrite.println("</html>");
    }
}

```

The FirstServlet servlet class handles the HttpServletRequest object and so the object is extended with the HttpServlet class. The FirstServlet servlet class overrides the init() and doGet() methods. The value for the name attribute is set in the init() method. The doGet() method retrieves the value of the greeting initializing parameter, which will be set in the web.xml file. The doGet() method also displays the value of the name attribute.

In Listing 4.2, the getServletContext() method of the ServletConfig object calls the setAttribute() method, to set the value and the getAttribute() method to retrieve the value of the name attribute. The getInitParameter() method is used to retrieve the value of init-param, which will be set in the web.xml file (Listing 4.3).

Create a JavaEE folder in the C: drive for the applications that you create in all the chapters of this book. This folder can be found on the CD as well.

Now, let's define directory structure for the FirstApp application to store the FirstServlet servlet class, configure the servlet in the web.xml file, and then package, deploy, and run the FirstApp application.

## Exploring Directory Structure

The root directory for all the applications in this book is JavaEE and you will find a folder for each chapter in the CD under this root directory. To run an application on your system, you can either copy the JavaEE folder from CD to the C: drive or create a new JavaEE folder containing the folders for each chapter. For example, for this chapter, the chapter4 folder is created under the root directory, JavaEE.

Figure 4.6 shows the directory structure of the FirstApp Web application:

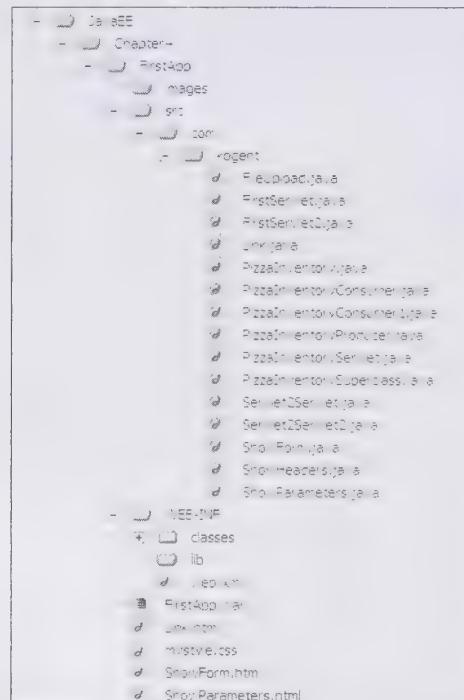


Figure 4.6: Displaying the Directory Structure of the FirstApp Application

In Figure 4.6, FirstApp is the name of the created application. Run the following command from the C:\JavaEE\chapter4\FirstApp\src\com\kogent location to compile the FirstServlet servlet:

```
javac -d C:\JavaEE\chapter4\FirstApp\WEB-INF\classes FirstServlet.java
```

The preceding command compiles the FirstServlet.java file and creates the com.kogent package that contains the class file of FirstServlet. The com.kogent package is created under the classes folder (Figure 4.6).

## Configuring the Servlet

Configuration implies mapping a servlet and providing the initialization parameter values for it. Servlet configuration for any Web application is done in the web.xml file. Listing 4.3 shows how to configure the FirstServlet servlet (you can find this file on CD in the code\JavaEE\Chapter4\FirstApp\WEB-INF folder):

**Listing 4.3:** Showing the Code for the web.xml File

```
<?xml version="1.0" encoding="UTF-8"?>

<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">

    <servlet>
        <servlet-name>firstServlet</servlet-name>
        <servlet-class>com.kogent.FirstServlet</servlet-class>
        <init-param>
            <param-name>greeting</param-name>
            <param-value>Welcome</param-value>
        </init-param>
    </servlet>
    <servlet-mapping>
        <servlet-name>FirstServlet</servlet-name>
        <url-pattern>/FirstServlet</url-pattern>
        <url-pattern>/FServlet</url-pattern>
    </servlet-mapping>
    <session-config>
        <session-timeout>30</session-timeout>
    </session-config>
</web-app>
```

The web.xml file is saved under the WEB-INF folder of the FirstApp application (Figure 4.6). In Listing 4.3, the FirstServlet servlet is mapped to two url-patterns (/FirstServlet and /FServlet). In Servlet 3.0, you can provide multiple url patterns for a servlet in the web.xml file. In Listing 4.3, the greeting initialization parameter is provided Welcome as its value. The FirstServlet servlet class is configured to the com.kogent.FirstServlet class.

## Packaging, Deploying and Running the Web Application

Before deploying the FirstApp Web application, a Web ARchive (WAR) file is created to package the entire Web application. Further, you need to deploy the application on the Glassfish application server and finally you can run the application. Perform the following steps to package, deploy, and run the FirstApp application.

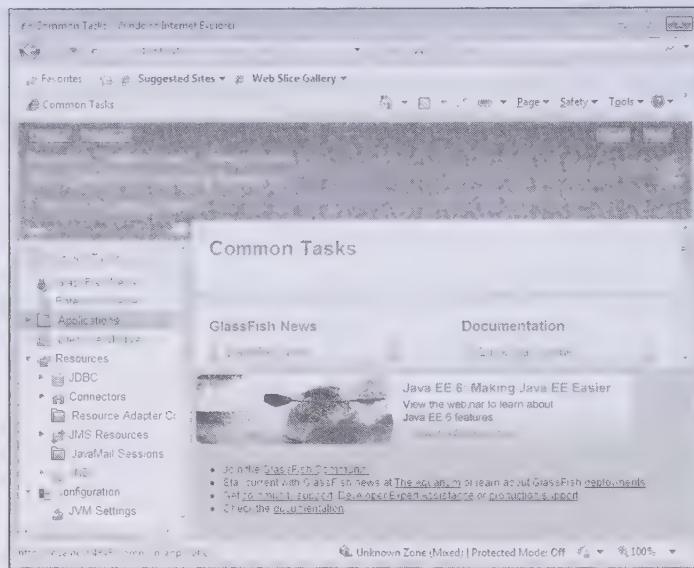
1. Run the following command from the code\JavaEE\chapter4\FirstApp location to create the FirstApp.war file:

```
jar -cvf FirstApp.war .
```

The preceding command creates the FirstApp.war file in the FirstApp folder (Figure 4.6). The WAR file contains the entire directory structure shown in Figure 4.6.

2. Start the Glassfish application server and open the <http://localhost:4848> URL. The Login window appears.

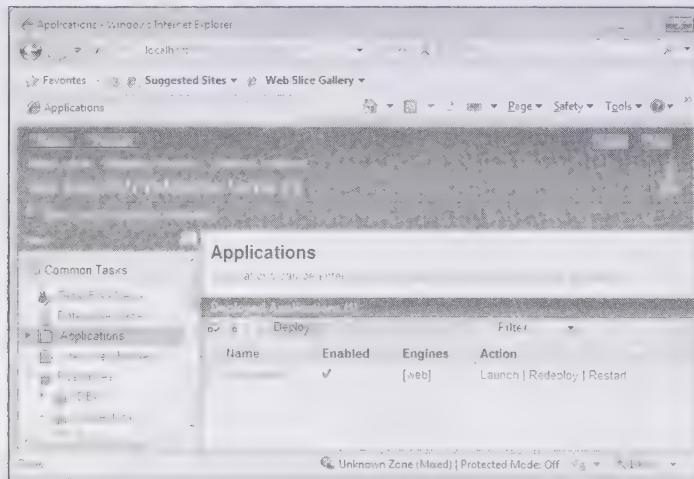
3. Enter admin as the user name and adminadmin as the password. After logging to the Web application, the Index page of the application is displayed.
4. Select the Applications option in the directory tree on the left side of the index page, to deploy the FirstApp WAR file, as shown in Figure 4.7:



**Figure 4.7: Displaying the Admin Console of the Glassfish Server**

After selecting the Applications option, the Applications pane is displayed (Figure 4.8).

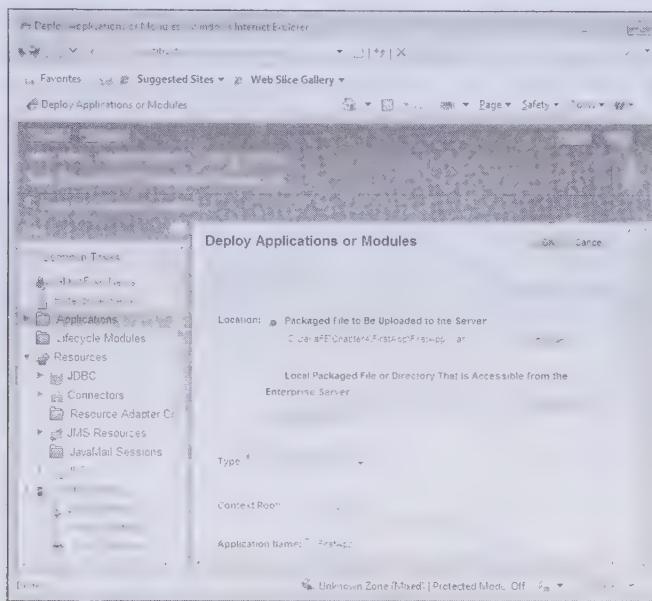
5. Click the Deploy button, as shown in Figure 4.8:



**Figure 4.8: Displaying the Applications Window**

The Deploy Applications or Modules pane is displayed (Figure 4.9).

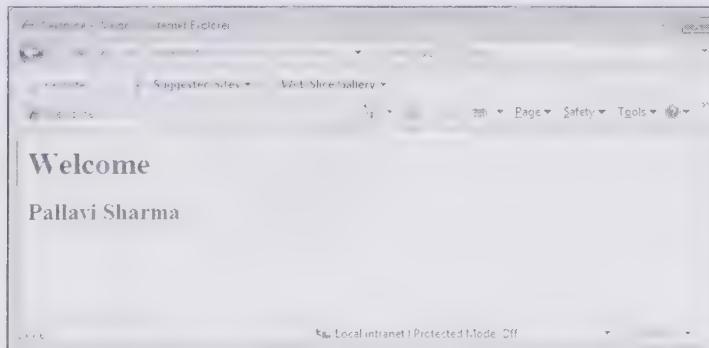
6. Click the Browse button and locate the FirstApp.war file. The location for the FirstApp.war file to be deployed is shown in Figure 4.9:



**Figure 4.9: Displaying the Details of the Web Application to Deploy**

The FirstApp.war file is uploaded and the general information of the FirstApp Web application automatically updated in the required text boxes of the Deploy Applications or Modules pane.

7. Click the Ok button to deploy the FirstApp Web application, packaged into the WAR file. After the application is deployed, you need to run the application to display the output.
8. Open the `http://localhost:8080/FirstApp/FirstServlet` URL. Figure 4.10 displays the output of the FirstApp Web application:



**Figure 4.10: Displaying the Output of the FirstApp Web Application**

Alternatively, you can specify the URL pattern /FServlet in the address bar of the Web browser to access the FirstServlet servlet class.

The `http://localhost:8080/FirstApp/FServlet` URL also displays the same output (Figure 4.10).

After creating and configuring a simple servlet by using Deployment Descriptor (the web.xml file), let's learn how to create a servlet by using annotations.

## Creating a Servlet by using Annotation

Instead of using Deployment Descriptor, a servlet can also be created by using annotations. This is a new feature, introduced in the Servlet 3.0 API, greatly simplifies the creation of a servlet. You need to import two packages,

namely javax.servlet.http.annotation and javax.servlet.http.annotation.jaxrs, to create a servlet by using annotations. Let's discuss how you can re-create the FirstServlet servlet class by using annotations. Listing 4.4 shows the modified code for the FirstServlet servlet, saved as the FirstServlet2.java file (you can find the file on the CD in the code\JavaEE\Chapter4\FirstApp\src\com\kogent folder):

**Listing 4.4:** Displaying the Code for the FirstServlet2.java File

```
package com.kogent;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.annotation.WebServlet;
import javax.servlet.annotation.*;

@WebServlet(name="FirstServlet2", urlPatterns="/FirstServlet2",
    initParams={@WebInitParam(name="greeting",value="welcome") })

public class FirstServlet2 extends HttpServlet
{
    @Override
    public void init(ServletConfig config) throws ServletException
    {
        super.init(config);
        config.getServletContext().setAttribute
            ("name", "Pallavi Sharma");
    }

    @Override
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter printwrite = response.getWriter();
        printwrite.println("<html>");
        printwrite.println("<head>");
        String greet;
        String name;
        greet = getServletConfig().getInitParameter("greeting");
        name = getServletContext().getAttribute("name").toString();
        printwrite.println("<title>" + greet + "</title>");
        printwrite.println("</head>");
        printwrite.println("<body>");
        printwrite.println("<h1>" + greet + "</h1>");
        printwrite.println("<h2>" + name + "</h2>");
        printwrite.println("</body>");
        printwrite.println("</html>");
    }
}
```

Save the FirstServlet2.java file in the src\com\kogent folder (Figure 4.6) and compile the file to generate the .class file. This time, you are not required to provide a mapping for the FirstServlet2 servlet in the web.xml file. Just re-create the FirstApp.war file, deploy the new WAR file, and open the URL <http://localhost:8080/FirstApp/FirstServlet2>. The same output is displayed that was generated for the FirstServlet servlet class (Figure 4.10).

After learning to create a servlet by using annotations, let's briefly discuss the ServletConfig and ServletContext objects in the next section.

## Working with ServletConfig and ServletContext Objects

ServletContext objects help to provide context information in a Servlet container. A ServletContext object is used to communicate with the Servlet container while ServletConfig, which is a servlet configuration object, is passed to the servlet by the container when the servlet is initialized. A ServletConfig object contains a

`ServletContext` object, which specifies the parameters for a particular servlet while the `ServletContext` object specifies the parameters for an entire Web application. The `ServletContext` object parameters are available to all the other servlets in that application.

The following code snippet sets an attribute named `name` with the value `Pallavi Sharma`:

```
public void init(ServletConfig config) throws ServletException
{
    super.init(config);
    config.getServletContext().setAttribute("name", "Pallavi Sharma");
}
```

In the preceding code snippet, the call to the `super.init(config)` method ensures that the `GenericServlet` class receives a reference to the `ServletConfig` object. The implementation of the `GenericServlet` class maintains a reference to the `ServletConfig` object and requires the invocation of the `super.init(config)` method in subclasses.

In the preceding code snippet, `config` is the instance of `ServletConfig` passed as an argument to the `init()` method. The `setAttribute()` method is used to set the value of the `name` attribute. The value of this attribute is accessed with the help of the `getAttribute()` method. This attribute is available for all the servlets in the Web application and can be accessed in any servlet.

Now let's learn to work with the `HttpServletRequest` and `HttpServletResponse` interfaces.

## Working with the `HttpServletRequest` and `HttpServletResponse` Interfaces

We discussed earlier in this chapter that the most common HTTP requests are the `GET` and the `POST` methods. You must implement these methods to handle different types of requests. The servlet container recognizes the type of HTTP request made and passes the request to the correct servlet method. Accordingly, you do not override the `service()` methods as you do for Servlets that extend the `GenericServlet` class; rather, you override the appropriate request methods.

Let's now learn about the `HttpServletRequest` and `HttpServletResponse` interfaces in detail in the following sections.

### Using the `HttpServletRequest` Interface

An `HttpServletRequest` object always represents a client's HTTP request. `HttpServletRequest` is an interface and a subtype of the `ServletRequest` interface. The Web container provider implements this interface to encapsulate all HTTP-based request information, including headers and request methods.

All properties, such as request parameters and attributes of the `ServletRequest` interface are also accessible through the `HttpServletRequest` interface.

Let's learn about the implementation of the `HttpServletRequest` interface under the following subheads:

- ❑ The role of form data
- ❑ Form data and parameters
- ❑ Headers
- ❑ File uploads

### The Role of Form Data

You can understand the role of Form Data better by considering a real-life scenario. When you use a search engine, visit an online bookstore, track stocks on the Internet, or ask a Web-based site for quotes on plane tickets, you may have seen funny-looking URLs such as `http://host/path?user=John+Smith&origin=lon&dest=par`. The part of the URL after the question mark (i.e., `user=John+Smith&origin=lon&dest=par`) is known as Form Data (or query data). This is the most common way by which a server-side program gets information from a Web page. For GET requests, Form Data can be attached to the end of the URL after a question mark (as in the proceeding example). For POST requests, Form Data can be sent to the server separately.

CGI programming involves a tedious traditional approach of extracting the needed information from Form Data. First, you have to read the data one way for GET requests (in traditional CGI, this is usually through the QUERY\_STRING environment variable) and in a different way for POST requests (by reading the standard input in traditional CGI). Second, you have to separate the pairs at the ampersands and then separate the parameter names (left of the equal signs) from the parameter values (right of the equal signs). Third, you have the URL-decode values. All the alphanumeric characters are sent unchanged, but spaces are replaced with plus signs and other characters are replaced with %xx, where xx implies the American Standard Code for Information Interchange (ASCII) or International Organization for Standardization (ISO) Latin-1 value of the character. The process is reversed for the server-side program. For example, if a user enters the values ~jim, ~robert, and ~hall into text fields with the name users in an HTML form, the data is sent as users=%7Ejim%2C+%7Erobert%2C+and+%7Ehall, and the original string is reconstituted by the server-side program. Finally, the fourth reason that makes parsing Form Data in CGI programs a tedious process is that values can be omitted (for example param1=val1&param2=&param3=val3) or a parameter can have more than one value (for example param1=val1&param2=val12&param1=val3). Therefore, special cases need to be applied in your parsing code for these situations.

## Form Data and Parameters

One of the important features of servlets is that parsing of a form is handled automatically. You only need to call the `getParameter()` method of the `HttpServletRequest` object with the case-sensitive parameter name as an argument. The `getParameter()` method is used in the same way when data is sent by the GET request as when it is sent by the POST request. The servlet can identify which request method is used and automatically returns the String value according to the URL-decoded value of the first occurrence of that parameter name. If the parameter exists but has no value, an empty String is returned. In addition, if no such parameter exists, null is returned. You should call the `getParameterValues()` method (which returns an array of Strings) instead of the `getParameters()` method (which returns a single String), if the parameter can potentially have more than one value. The return value of the `getParameterValues()` method is null for parameter names that do not exist. A single element array is returned when only a single value exists for the parameter.

Parameter names are case-sensitive; therefore, for example, `request.getParameter("Param1")` and `request.getParameter("param1")` are not interchangeable.

Finally, for debugging, it is sometimes useful to get a full list of parameters, although most real servlets look for a specific set of parameter names. You can use the `getParameterNames()` method to get the list of parameter names in the form of an enumeration, each entry of which can be cast to a String and used in the `getParameter()` or `getParameterValues()` method. You should note that the order in which the parameter names appear within the enumeration is not specified by the `HttpServletRequest` interface.

Let's now discuss the role of request parameters. Perhaps the most commonly used methods of the `HttpServletRequest` object are `getParameter()` and `getParameters()` that involve getting request parameters. Whenever an HTML form is filled and sent to a server, the fields of the form are passed as parameters. This includes any information sent through input fields, selection lists, combo boxes, check boxes, and hidden form fields. However, the form submission excludes file uploads. Any information passed as a query string is also available on the server-side as a request parameter. The `HttpServletRequest` object includes the following methods to access request parameters:

- ❑ `getParameter(java.lang.String parameterName)`—Takes a parameter name as a parameter and returns a String object representing the corresponding value. This method returns null when it does not find a parameter of the given name.
- ❑ `getParameters(java.lang.String parameterName)`—Allows you to get all the parameter values for the same parameter name returned as an array of Strings. The `getParameters()` method is similar to the `getParameter()` method. However, note that the `getParameters()` method should be used when there are multiple parameters with the same name. Often, an HTML form check box or combo box sends multiple values for the same parameter name.

- ❑ `getParameterNames()`—Returns the parameter names in the form of an enumeration, which are used in a request. This method can be used with the `getParameter()` and `getParameters()` methods, to obtain a list of names and values of all the parameters included with a request.

Let's now create a servlet that reads and displays all the parameters sent with a request. You can use such a servlet to get a little more familiar with parameters, and to debug HTML forms by seeing the information being sent. Listing 4.5 provides the code for such a servlet (you can find the `ShowParameters.java` file on CD in the `code\JavaEE\Chapter4\FirstApp\src\com\kogent` folder):

**Listing 4.5:** Displaying the Code for the `ShowParameters.java` File

```
package com.kogent;

import java.util.*;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ShowParameters extends HttpServlet
{
    public void doGet(HttpServletRequest request, HttpServletResponse
    response) throws IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Request HTTP Parameters Sent</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<p>Parameters sent with request:</p>");
        Enumeration enm = request.getParameterNames();

        while (enm.hasMoreElements())
        {
            String pName = (String) enm.nextElement();
            String[] pvalues = request.getParameterValues(pName);
            out.print("<b>" + pName + "</b>: ");
            for (int i=0;i<pvalues.length;i++)
            {
                out.print(pvalues[i]);
            }
            out.print("<br>");
        }

        out.println("</body>");
        out.println("</html>");
    }

    public void doPost(HttpServletRequest request, HttpServletResponse
    response) throws IOException, ServletException
    {
        doGet(request, response);
    }
}
```

Save the `ShowParameters.java` file in the `src\com\kogent` folder (Figure 4.6) and compile and deploy the `ShowParameters.java` servlet in the `FirstApp` Web application with a mapping to the `/ShowParameters` path. Now, create a few simple HTML forms and use the servlet to see the parameters being sent. In Listing 4.6, `ShowParameters.html` provides a sample HTML form (you can find this file on the CD in the `code\JavaEE\Chapter4\FirstApp` folder):

**Listing 4.6:** Showing the Code for the `ShowParameters.html` File

```
<html>
<head>
<title>Example HTML Form</title>
```

```

<link rel="stylesheet" href="mystyle.css" type="text/css"/>
</head>
<body>
<p>To debug a HTML form set its 'action' attribute to reference the ShowParameters Servlet.</p>
<form action="ShowParameters" method="post">
    Name: <input type="text" name="name"><br>
    Password: <input type="password" name="password"><br>
    Select Box:
        <select name="selectbox">
            <option value="option1">Option 1</option>
            <option value="option2">Option 2</option>
            <option value="option3">Option 3</option>
        </select><br>
    Importance:
        <input type="radio" name="importance" value="very">Very,
        <input type="radio" name="importance" value="normal">Normal,
        <input type="radio" name="importance" value="not">Not<br>
    Comment: <br>
        <textarea name="textarea" cols="40" rows="5"></textarea><br>
        <input value="Submit" type="submit">
</form>
</body>
</html>

```

Save the ShowParameters.html file in the base directory of the FirstApp Web application and deploy the new WAR file. After deploying the file, browse the following URL:

<http://localhost:8080/FirstApp>ShowParameters.html>

The ShowParameters.html page is displayed, as shown in Figure 4.11:

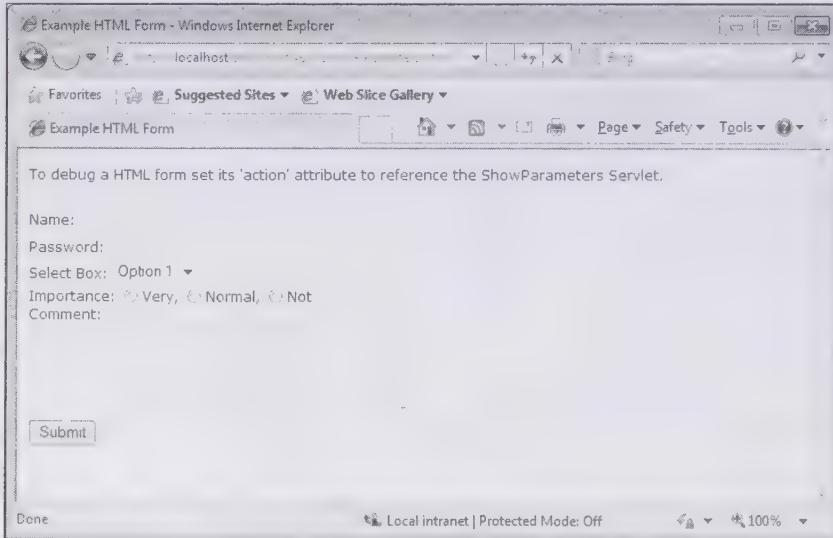
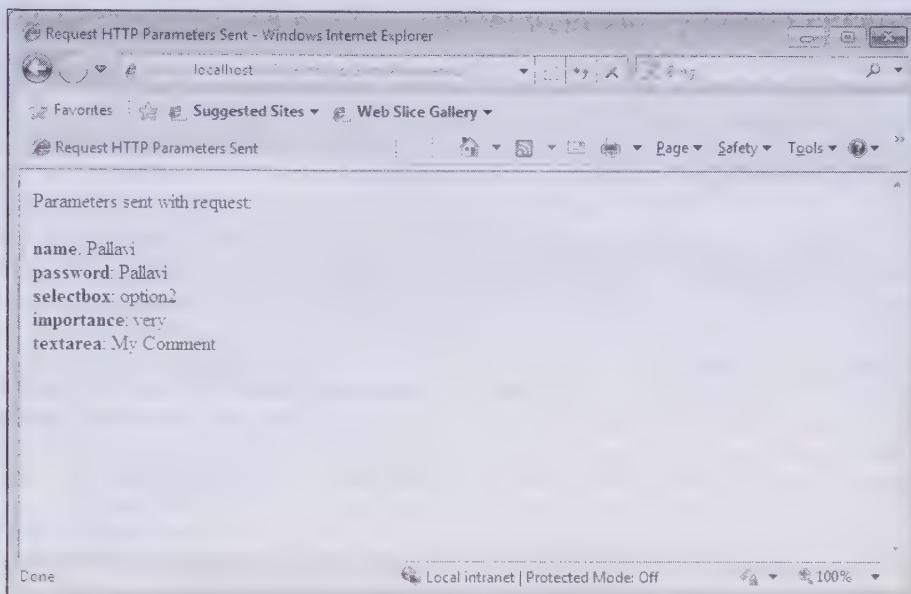


Figure 4.11: Displaying the ShowParameters Page

After entering the relevant details in the HTML form (Figure 4.11), the parameters are sent to the ShowParameters servlet. The parameters sent from the ShowParameters.html page are displayed by the ShowParameters Servlet, as shown in Figure 4.12:



**Figure 4.12: Displaying the Request HTTP Parameters**

On the server-side, each piece of information received from an HTML form is referenced by the same name as defined in the HTML form and is linked to the value that a user has entered for the respective field. The `ShowParameters` servlet calls the `getParameterNames()` method to retrieve a list of all the parameter names and subsequently calls the `getParameters()` method to retrieve the matching value or set of values for each name. The following code snippet shows the implementation of the `getParameterNames()` method:

```
Enumeration enm = request.getParameterNames();
while (enm.hasMoreElements())
{
    String pName = (String) enm.nextElement();
    String[] pvalues = request.getParameterValues(pName);
    out.print("<b>" + pName + "</b>: ");
    for (int i=0;i<pvalues.length;i++) { out.print(pvalues[i]);}
    out.print("<br>");
```

A servlet can fetch information from HTML clients by using parameters. The `ShowParameters` servlet only takes the parameters from the HTML page and displays them back to a client. However, normally, these parameter values are combined and processed with other code to generate responses. Later on in the book, you learn to use this functionality with servlets and JSP to interact with clients, including sending e-mail messages and user authentication.

## HTTP Headers

HTTP headers are set by a client to give information to a server about software that the client is using and how the client wants a server to send back the requested information. HTTP request headers can be accessed from a servlet by calling different methods, which are as follows:

- ❑ `getHeader(java.lang.String name)`—Returns the specified request header value as a `String`. This method returns null if the request does not include a header of the specified name. The header name is case insensitive. A user can use this method with any request header.
- ❑ `getHeaders(java.lang.String name)`—Returns all the specified request header values as an enumeration of `String` objects. Sometimes, a client can send the header values as an enumeration of `String` objects, rather than sending the header as a comma-separated list. Each of these headers can have a different value. If the request includes no header of the specified name, this method returns an empty Enumeration

object. The header name is case insensitive in this method also as well. The user can use this method with any request header.

- `getHeaderNames()`—Returns an enumeration of the names of all the headers sent by a request. In combination with the `getHeader()` and `getHeaders()` methods, the `getHeaderNames()` method can be used to retrieve the names and values of all the headers sent with a request. Some containers do not allow access to HTTP headers. In that case, null is returned.
- `getIntHeader(java.lang.String name)`—Returns the value of the specified request header as an int type. A value of -1 is returned by this method if the request does not contain a header of the specified name. A `NumberFormatException` exception is thrown if the header cannot be converted to an integer.
- `getDateHeader(java.lang.String name)`—Returns the specified request header value as a long value representing a Date object. The returned date is counted as the number of milliseconds since the epoch. The header name is case insensitive. A value of -1 is returned if a request header of the specified name is not found. An `IllegalArgumentException` exception is thrown if the header cannot be converted to a date.

In this way, you can see that HTTP request headers are very helpful to determine diversified information, which can be obtained by calling the preceding listed methods. In the later chapters of the book, HTTP request headers are used as the primary resource to mine data about a client. This includes identifying what language a client would prefer, what type of Web browser is being used, and whether or not the client can support compressed content for efficiency. For now, it is helpful to understand that these headers exist, and to get a general idea about what type of information the headers contain. Listing 4.7 creates a servlet designed to do just that. Save the code of the `ShowHeaders.java` file in the `/src/com/kogent` directory of the `FirstApp` Web application, which is displayed in Figure 4.7. Listing 4.7 shows the code for the `ShowHeaders` class (you can find this file on the CD in the `code\JavaEE\Chapter4\FirstApp\src\com\kogent` folder):

**Listing 4.7:** Displaying the Code for the `ShowHeaders.java` File

```
package com.kogent;
import java.util.*;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ShowHeaders extends HttpServlet
{
    public void doGet(HttpServletRequest request, HttpServletResponse
    response) throws IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Request's HTTP Headers</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<p>HTTP headers sent by your client:</p>");
        Enumeration enm = request.getHeaderNames();
        while (enm.hasMoreElements())
        {
            String headerName = (String) enm.nextElement();
            String headerValue = request.getHeader(headerName);
            out.print("<b>" + headerName + "</b>: ");
            out.println(headerValue + "<br>");
        }
        out.println("</body>");
        out.println("</html>");
    }
}
```

Compile the `ShowHeaders` servlet class and configure the servlet to the `/ShowHeaders` path in the `web.xml` file. The following code snippet provides the configuration of the `ShowHeaders` servlet class in the `web.xml` file:

```
...
```

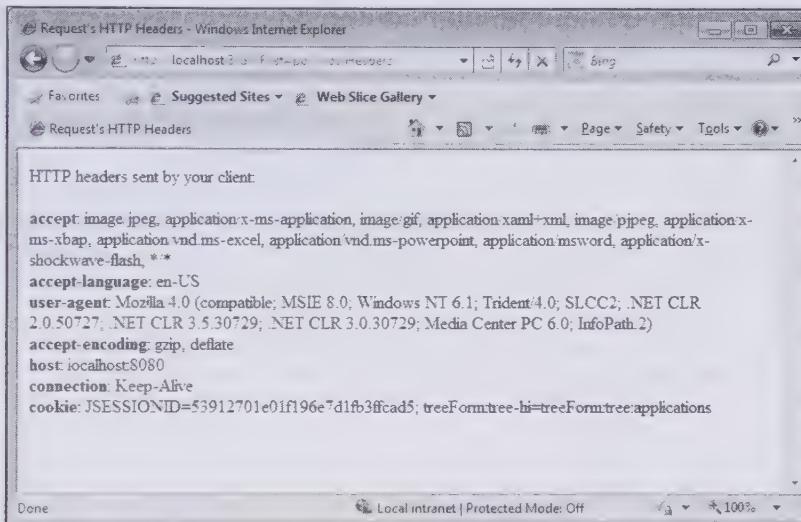
```
<servlet>
```

```

<servlet-name>ShowHeaders</servlet-name>
<servlet-class>com.kogent.ShowHeaders</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>ShowHeaders</servlet-name>
    <url-pattern>/ShowHeaders</url-pattern>
</servlet-mapping>

```

Deploy the FirstApp Web application and after reloading the Web application, browse to <http://localhost:8080/FirstApp>ShowHeaders> to view a listing of all the HTTP headers sent by your browser. Figure 4.13 shows the details of the HTTP headers:



**Figure 4.13: Displaying the Request's HTTP Headers**

Listing 4.7 is a good example to illustrate how headers are normally sent by a Web browser. They are self-descriptive and can therefore be understood easily. You can probably imagine how these headers can be used to infer browser and internationalization information. Table 4.1 lists some of the most relevant request headers:

**Table 4.1: Various Request Headers**

Request header	Description
Accept	Specifies certain media types that can be accepted for a response. Accept headers can be used to specify that the request is limited to a set of desired media types.
Accept-Charset	Indicates the character sets that can be accepted for a response. This header accepts clients that can understand comprehensive or special-purpose character sets. Due to this, the server can represent responses in those character sets. All user agents can accept the ISO-8859-1 character set.
Referer (sic)	Allows a client to state the address (URI) of the resource from which the Request URI was obtained.
Accept-Language	Restricts the set of natural languages that are preferred as a response to a request. Otherwise, this header is similar to the Accept header.
Host	Specifies the Internet host and port number of a requested resource, as obtained from the original URL given by a user or referring resource. This header is mandatory for HTTP 1.1.
User-Agent	Contains information about the user agent (or browser) making a request. This header is used for statistical purposes, to trace violations of protocol, and to automatically recognize user agents.

## File Uploads

File uploads are simple for HTML developers but difficult for server-side developers. Usually, discussions on Servlets and HTML forms conveniently skip the topic of file uploads. However, a servlet developer must have a good understanding of HTML form file uploads. For example, consider a situation where a client needs to upload something besides a simple string of text, such as a picture. In this case, using the `getParameter()` method will not work because it produces unpredictable results. Therefore, to read the picture uploaded the Servlet API provides the MIME type.

There are two primary MIME types for form information, `application/x-www-form-urlencoded` and `multipart/form-data`. In the MIME type `application/x-www-form-urlencoded`, the results in the Servlet API automatically parse out name and value pairs. The information is then available by invoking `HttpServletRequest.getParameter()` or any of the other related methods as described earlier. The second MIME type, `multipart/form-data`, is usually considered difficult, because the Servlet API does not provide any support for it. The information is left as it is and you are responsible for parsing the request body by using either the `getInputStream()` or `getReader()` method.

The Request For Comments (RFC) 1867 memo provided at the <http://www.ietf.org/rfc/rfc1867.txt> URL explains the `multipart/form-data` MIME type and the format of the associated HTTP requests. You can determine how to properly and appropriately handle the information posted to a servlet, by using RFC. This is not a difficult task and is usually not needed, because other developers create complementary APIs to handle file uploads. We discuss this later in this chapter.

A user can actually look at the content of a request when the `multipart/form-data` information is sent. For this, create a file-uploading form and a servlet that reproduces the information obtained from the `ServletInputStream` object. Listing 4.8 provides the code for such a servlet that accepts a `multipart/form-data` request and displays its content as plain text (you can find the `ShowForm.java` file on the CD in the `code\JavaEE\Chapter4\FirstApp\src\com\kogent` folder):

**Listing 4.8:** Displaying the Code for the `ShowForm.java` File

```

package com.kogent;
import java.util.*;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ShowForm extends HttpServlet {
    public void doPost(HttpServletRequest request, HttpServletResponse
        response) throws IOException, ServletException {
        response.setContentType("text/plain");
        PrintWriter out = response.getWriter();
        ServletInputStream sis = request.getInputStream();
        for (int i = sis.read(); i != -1; i = sis.read()) {
            out.print((char)i);
        }
    }
    public void doGet(HttpServletRequest request, HttpServletResponse
        response) throws IOException, ServletException {
        doPost(request, response);
    }
}

```

Save the preceding code as `ShowForm.java` in the `/src/com/kogent` directory of the `FirstApp` Web application. Configure the `ShowForm` servlet to `/ShowForm` in the `web.xml` file by using the following code snippet:

```

...
<servlet>
    <servlet-name>ShowForm</servlet-name>
    <servlet-class>com.kogent.ShowForm</servlet-class>
</servlet>

```

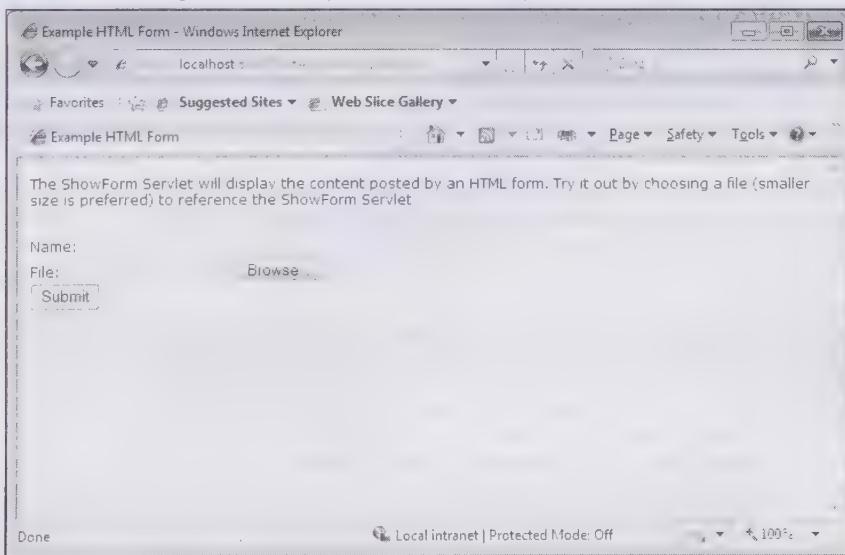
```
<servlet-mapping>
    <servlet-name>ShowForm</servlet-name>
    <url-pattern>/ShowForm</url-pattern>
</servlet-mapping>
```

Now, any information posted by any form can be viewed by directing the request to `http://localhost:8080/FirstApp/ShowForm`. You can run the ShowForm servlet only after creating an HTML form used to upload a file. Listing 4.9 provides the code to create an HTML page (you can find the `ShowForm.html` file on the CD in the `code\JavaEE\Chapter4\FirstApp` folder):

**Listing 4.9:** Displaying the Code for the `ShowForm.html` File

```
<html>
    <head>
        <title>Example HTML Form</title>
        <link rel="stylesheet" href="mystyle.css" type="text/css"/>
    </head>
    <body>
        <p>The ShowForm Servlet will display the content posted by an HTML form. Try it out by choosing a file (smaller size is preferred) to reference the ShowForm Servlet.</p>
        <form action="ShowForm" method="post"
              enctype="multipart/form-data">
            Name: <input type="text" name="name"><br>
            File: <input type="file" name="file"><br>
            <input value="Submit" type="submit">
        </form>
    </body>
</html>
```

Save the code shown in Listing 4.9 as `ShowForm.html` in the base directory of the `FirstApp` Web application and browse the `http://localhost:8080/FirstApp/ShowForm.html` URL. A small HTML form with two inputs, a name and a file to upload is displayed, as shown in Figure 4.14.



**Figure 4.14:** Displaying the `ShowForm` Page

Enter appropriate values for the Name and File fields in the form (Figure 4.14). In this case, a small file is preferred because its content is going to be displayed by the `ShowForm` Servlet. A good candidate for a file to be uploaded is `ShowForm.html`. After entering the values, click the Submit button to show the content of the uploaded file. For example, using `ShowForm.html` as the file displays a form similar to the one shown in Figure 4.15.

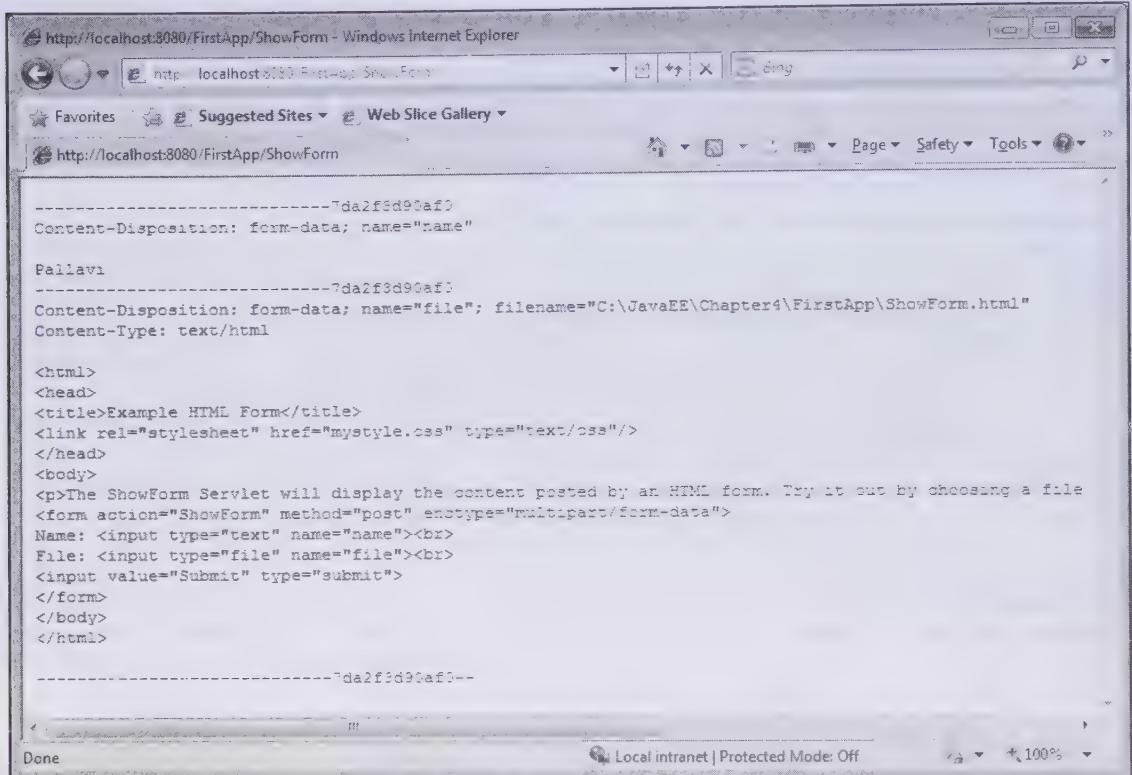


Figure 4.15: Displaying the Output of the ShowForm.html File

The following line is displayed in Figure 4.15 depicting the unique token for the start portion of the multipart of the uploaded file:

-----7da2f3d90af0

The preceding line declares the start of the multipart section and concludes at the ending token, which is identical to the start but has -- appended to it. Between the starting and ending tokens are sections of data (possibly nested multipart) with headers used to describe the content. For example, the Content-Disposition header is displayed as follows:

Content-Disposition: form-data; name="name"

Pallavi

The Content-Disposition header defines the information as part of the form and is identified by the name "name". The value of name is the content that follows it. By default, the MIME type of name is text/plain. The uploaded file is described in the second multipart, shown as follows:

```

Content-Disposition: form-data; name="file";
filename="C:\JavaEE\chapter4\FirstApp>ShowForm.html"
Content-Type: text/html
<html>
<head>
<title>Example HTML Form</title>
<link rel="stylesheet" href="mystyle.css" type="text/css"/>
</head>
<body>
<p>The ShowForm </p>

```

In the preceding code snippet, the Content-Disposition header specifies the form field name to be filled, which corresponds to the field name listed in the HTML form, and describes Content-Type as text/html, as it is not text/plain. The output after the Content-Type header includes code of the ShowForm.html file.

So far, you have learned how the HttpServletRequest object is used to retrieve HTML form parameters from a request by using the getParameter() method. The HttpServletRequest object is also used to retrieve HTTP request header information, upload a file by using the getInputStream() method, and display the content of the uploaded file by using the getWriter() method of the HttpServletResponse object.

After discussing the implementation of the HttpServletRequest interface, let's learn about the HttpServletResponse interface.

## Using the HttpServletResponse Interface

The HttpServletResponse object helps to set an HTTP response header, set the content type of the response, or redirect an HTTP request to another URL. Let's discuss the implementation of the HttpServletResponse interface.

In the previous section, we discussed how to send information back to a client. The HttpServletResponse object is responsible for this functionality, which creates an empty HTTP response. Custom content can be sent back by obtaining an output stream by using either the getWriter() or getOutputStream() method to write the content. A suitable object is returned by these two methods to send either text or binary content to a client. Only one of the two methods can be used with a given HttpServletResponse object. An exception is thrown if you try calling both the methods. The HttpServletResponse object is also used to provide an PrintWriter instance to print the response on a browser. The following code snippet displays the welcome message on the browser:

```
PrintWriter out = response.getWriter();
out.println("<html>");
out.println("<head>");
out.println("<title>Welcome Message</title>");
out.println("</head>");
out.println("<body>");
out.println("<p>Welcome to the Users</p>");
```

In the preceding code snippet, the getWriter() method is used to get an output stream to send the HTML markup. When you use an instance of the PrintWriter object, you need to provide a String object and call the print(), println(), or write() method.

In this section, you learn about the implementation of the HttpServletResponse interface under the following headings:

- Response header
- Response redirection
- Response redirection translation issues
- Auto-refresh/wait pages

### Response Header

The HttpServletResponse object is used to manipulate the HTTP headers of a response and to send the content back to a client. HTTP response headers inform a client the type and amount of content being sent, and the type of server sending the content.

The HttpServletResponse object includes the following methods to manipulate HTTP response headers:

- addHeader(java.lang.String name, java.lang.String value)–Adds a response header having the given name and value. This method can be used to create response headers with multiple values.
- containsHeader(java.lang.String name)–Returns a Boolean value that indicates whether the named response header is already been set or not.
- setHeader(java.lang.String name, java.lang.String value)–Sets the name and value of a response header as specified in the arguments. The previous value is overwritten, if the header is already

set. The `containsHeader()` method can be used to test whether a header is already present or not, before setting its value.

- `setIntHeader(java.lang.String name, int value)`—Sets the name and integer values for a response header, as specified in the arguments. The previous value is overwritten, if the header is already set. The `containsHeader()` method can be used to test whether a header is already present or not, before setting its value.
- `setDateHeader(java.lang.String name, long date)`—Sets the given name and date values for a response header. The date is provided in terms of milliseconds since the epoch. The previous value is overwritten with the new value, if the header is already set.
- `addIntHeader(java.lang.String name, int value)`—Adds a response header having the name and integer values, as specified in the arguments. Response headers can be assigned multiple values when created by using this method.
- `addDateHeader(java.lang.String name, long date)`—Adds a response header having the name and date values, as specified in the arguments. The previous response header values are not overwritten and response headers can have multiple values.

Table 4.2 provides a description of the HTTP response header fields and their values:

Header Field	Header Value
Age	Represents the estimated time since the last response generated from the server. The value of this header field is usually a positive integer.
Content-Length	Indicates the size of the message body, in decimal number of octets (8-bit bytes), sent to a recipient.
Content-Type	Refers to the MIME type corresponding to the content of an HTTP response. A browser can use this value to determine whether the content is rendered internally or launched to be rendered by an external application.
Date	Represents the date and time at which a message originated.
Location	Specifies the location of a new resource in case HTTP response codes redirect a client to such a resource. The location is specified as an absolute address.
Pragma	Specifies the implementation-specific directives that may be applied to any recipient along the request-response chain. <i>No-cache</i> , which indicates that a resource should not be cached, is the most commonly used value.
Retry-After	Indicates the tentative duration for which a service is unavailable to a requesting client. It is used with a 503 (Service Unavailable) response. A date can be returned as a value of this field. The value can also be an integer representing the number of seconds (in decimals) after the time of the response.
Server	Represents information about the server that generated the current response, as a String value.

## Response Redirection

In this section, the response code of HTTP and its different functions have been discussed. The `setStatus()` method of an `HttpServletResponse` object can be used to send any HTTP response code to a client. The servlet sends back a status code 200, OK if everything works smoothly. A status code of 302 is sent by the servlet displaying the Resource Temporarily Moved message, which informs a client that the resource they were looking for is not at the requested URL, but can be found at the URL specified by the Location header in the HTTP response. The 302 response code is very helpful because almost every Web browser automatically follows the new link without informing the user. This allows a servlet to take the request of a user and forward it to any other resource on the Internet.

The 302 response code has excellent uses besides its intended purpose. The reason for this is that the 302 response code has a very common implementation. Most websites often track the users who visit their sites to

get an idea about their interests so that they can send information related to their interests. The technique for tracking website users requires the extracting of the referrer header of an HTTP request. This can be done easily by keeping track of the information sent by a site. The problem originates because the link on a site that directs to an external resource also sends a request back to the originating site from where it was sent. To solve the problem, a clever trick can be used that relies on the HTTP 302 response code. In this trick, rather than providing direct links to external resources, all links can be encoded in such a way that they all lead to the same servlet on your site, but at the same time, the real link can be included as a parameter. After that, link tracking can be implemented by providing the intended link through a servlet. In addition, a 302 status code is sent back to the client along with the real link to visit.

HTTP-aware sites commonly use a servlet to track links. The HTTP 302 response code is used very often because it provides the `sendRedirect()` method in the `HttpServletResponse` object. The `sendRedirect()` method takes one parameter, a `String`, representing the new URL, and automatically sets the HTTP 302 status code with appropriate headers. Using the `sendRedirect()` method and the `java.util.Hashtable` class, it is easy to create a servlet to track the links used. Let's create a servlet, named `Link`, to understand the use of the `sendRedirect()` method. Listing 4.10 shows the code for the `Link.java` file (you can find the `Link.java` file on the CD in the `code\JavaEE\Chapter4\FirstApp\src\com\kogent` folder):

**Listing 4.10:** Displaying the Code for the `Link.java` File

```
package com.kogent;
import java.util.*;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Link extends HttpServlet
{
    static private Hashtable links = new Hashtable();

    String stamp;
    public Link()
    {
        stamp = new Date().toString();
    }

    public void doGet(HttpServletRequest request, HttpServletResponse
response) throws IOException, ServletException
    {
        String lnk = request.getParameter("link");
        if (lnk != null && !lnk.equals(""))
        {
            synchronized (links)
            {
                Integer count = (Integer) links.get(lnk);
                if (count == null)
                {
                    links.put(lnk, new Integer(1));
                }
                else
                {
                    links.put(lnk, new
Integer(1+count.intValue()));
                }
            }
            response.sendRedirect(lnk);
        }
        else
        {
            response.setContentType("text/html");
            PrintWriter out = response.getWriter();
            request.getSession();
            out.println("<html>");
            out.println("<head>");
```

```

        out.println("<title>Links Tracker Servlet</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<p>Links Tracked Since");
        out.println(timestamp);
        if (links.size() != 0) {
            Enumeration enm = links.keys();
            while (enm.hasMoreElements()) {
                String key = (String)enm.nextElement();
                int count =
                    ((Integer)links.get(key)).intValue();
                out.println(key+ " : "+count+" visits<br>");
            }
        } else {
            out.println("No links have been tracked!<br>");
        }
        out.println("</body>");
        out.println("</html>");
    }
}
public void doPost(HttpServletRequest request, HttpServletResponse
response) throws IOException, ServletException {
    doGet(request, response);
}
}

```

Some links used by the Link servlet class are needed to complement the Link servlet. The links can be encoded properly to redirect a user to any resource. Encoding the links only requires passing the real link as a link parameter in a query String. The code given in Listing 4.11 is that of a simple HTML page that includes a few properly encoded links. Save the HTML code provided in Listing 4.11 as Link.html in the base directory of the FirstApp Web application. Listing 4.11 shows the code for the Link.html file (you can find this file on the CD in the code\JavaEE\Chapter4\FirstApp folder):

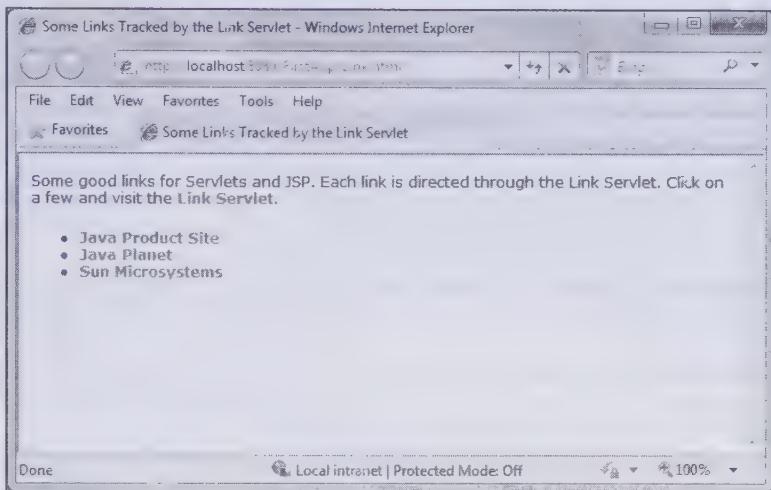
**Listing 4.11:** Displaying the Code for the Link.html File

```

<html>
    <head>
        <title>Some Links Tracked by the Link Servlet</title>
        <link rel="stylesheet" href="mystyle.css" type="text/css"/>
    </head>
    <body>
        Some good links for Servlets and JSP. Each link is directed through
        the Link Servlet. Click on a few and visit the
        <a href="Link">Link Servlet</a>.
        <ul>
            <li><a href="Link?link=http://www.java.sun.com">
                Java Product Site</a></li>
            <li><a href="Link?link=http://www.javaplanet.com">
                Java Planet</a></li>
            <li><a href="Link?link=http://java.sun.com">
                Sun Microsystems</a></li>
        </ul>
    </body>
</html>

```

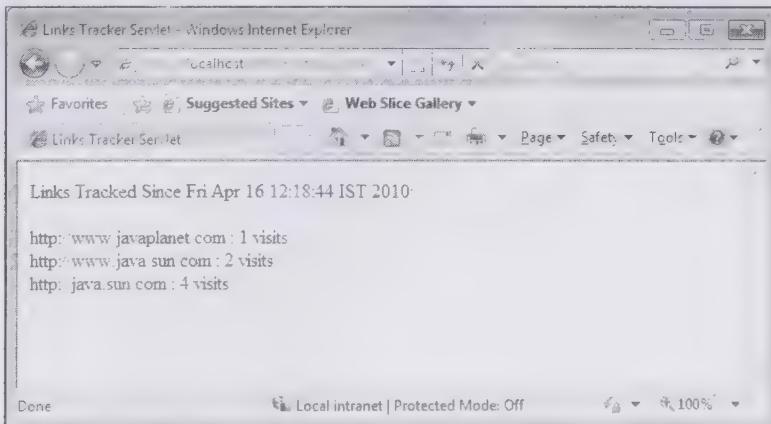
After creating the new FirstApp.war file and deploying the FirstApp Web application, browse the <http://localhost:8080/FirstApp/Link.html> URL. The Link.html page that comprises multiple links is displayed as shown in Figure 4.16:



**Figure 4.16: Displaying the Link.html Page**

Each link on the Link.html page is directed through the Link servlet, which in turn redirects a browser to visit the correct link. Before each redirection, the Link servlet logs the number of times the link has been visited by keying the link URL to an Integer object in a hashtable. You can browse the `http://localhost:8080/FirstApp/Link` URL, to view information about the links visited. The results are same as of the last reloading of the Link servlet. This servlet does not log the information for long-term use.

After clicking each link on the Link.html page multiple times, click the Link Servlet link. You are redirected to the Link servlet, as shown in Figure 4.17:



**Figure 4.17: Displaying the Link Servlet**

Now, let's discuss the concerns related to response redirection translation.

### Response Redirection Translation Issues

Response redirection is a tool that intimates a user about the resource to which a response is to be redirected. It works with any implementation of the Servlet API. However, there is a specific bug that arises while using relative response redirection. Consider the following command, which is used to redirect response:

```
response.sendRedirect("../foo/bar.html");
```

The preceding command works perfectly when used in some servlets but not in others. The problem comes from using the relative back `.. /` to traverse back a directory. A JSP page can use this command correctly; (you have to assume that the browser translates the URL correctly) but the JSP page can use it only if the requested URL is

combined with the correct path and ends on an appropriate resource. For instance, if `http://localhost:8080/foo/bar.html` is a valid URL, then `http://localhost:8080/foo/.../foo/bar.html` should also be valid. However, `http://localhost:8080/foo/foo/.../foo/bar.html` will not reach the same resource.

This might seem an irrelevant problem. However, request dispatching that we introduce in the next section makes it clear why this is an issue. Request dispatching allows requests to be forwarded on the server-side, which means that the requested URL does not change, but the server-side resource that handles the request can be changed. Relative redirections are not always safe; using `.. /` can be bad. The solution is to either use absolute redirections or a complete URL, as shown by the following command:

```
response.sendRedirect("http://localhost/foo/bar.html");
```

Alternatively, you can use an absolute URL from the root, “`/`”, of the `foo` Web application, as shown by the following command:

```
response.sendRedirect("/foo/bar.html");
```

The `HttpServletRequest.getContextPath()` method should also be used, when you can deploy the Web application to a non-root URL, as shown by the following command:

```
response.sendRedirect(request.getContextPath() + "/foo/bar.html");
```

You have already studied about the `HttpServletRequest` object along with the use of the `getContextPath()` method, earlier in this chapter.

## Auto-Refresh/Wait Pages

The other useful response header technique is to send a wait page or a page that auto-refreshes to a new page after a given time period to a user. This tactic is helpful in cases when there is a possibility of getting a response which might take an uncontrollable time to generate or for cases where you want to ensure a brief pause in a response. In this case, the entire mechanism involves setting the refresh response header. The header can be set by using the following command:

```
response.setHeader("Refresh", "time; URL=url" );
```

In the preceding command, `time` is replaced with the amount of seconds the page should wait, and `url` is replaced with the URL that the page should eventually load. For instance, if you want to load the `http://localhost/foo.html` URL after waiting for 10 seconds, the header is set by using the following command:

```
response.setHeader("Refresh", "10; URL=http://localhost:8080/foo.html");
```

The technique of sending a page that auto-refreshes is very helpful because it allows a proper message to be conveyed to clients until their requests are being processed. For example, a simple `your-request-is-being-processed-page`, which automatically refreshes to display the results of the response after a few seconds, can be displayed to the client. Alternatively, the client has to wait until a request is completely processed, before sending back any content. The alternative approach is used in most of the cases. However, this approach requires the client browser to wait for the response. Due to this, sometimes the client may assume that the request may result as timed-out, and may make a time-consuming request twice.

Another practical use for a wait page is to slow down a request. This is done by a developer to get better and more relevant information. For example, a wait page that displays either an advertisement or legal information before redirecting a user to the desired page.

### **NOTE**

*In some situations, the Refresh response header can prove to be helpful. Sometimes it can be considered as the de facto standard; however, it is not a standard HTTP 1.1 header.*

Now, let's learn how to delegate a request to a resource and discuss request scopes.

## Exploring Request Delegation and Request Scope

Request delegation refers to the request of a single client passing through many servlets or other resources in a Web application. The entire process is performed entirely on the server-side, unlike response redirection. Request delegation does not require any action from a client or extra information sent between the client and the

server. Request delegation is available through the `javax.servlet.RequestDispatcher` object, which can be obtained by calling any of the following methods of the `ServletRequest` object:

- ❑ `getRequestDispatcher(java.lang.String path)`—Returns the `RequestDispatcher` object for the path provided as an argument. The path value must start from the base directory / and can direct to any resource in a Web application.
- ❑ `getNamedDispatcher(java.lang.String name)`—Returns the `RequestDispatcher` object for the named servlet. The servlet-name elements of the `web.xml` file define the valid names.

The following two methods are provided by a `RequestDispatcher` object to include different resources and to forward a request to a different resource:

- ❑ `forward(javax.servlet.ServletRequest, javax.servlet.ServletResponse)`—Delegates a request and response to the resource of the `RequestDispatcher` object. A call to the `forward()` method may be used only if no content is previously sent to a client. After the completion of the processing of the `forward()` method, no further data can be sent to the client.
- ❑ `include(javax.servlet.ServletRequest, javax.servlet.ServletResponse)`—Works similar to the `forward()` method, but has some restrictions. Any number of resources can be included by a servlet by using the `include()` method; however, the resource cannot set headers or commit a response.

While a request delegation is often used to break a large servlet into smaller and more relevant parts, a simple case involves separating a common HTML header that is being shared by all the pages of a website. The `include()` method of the `RequestDispatcher` object provides a convenient method to include the header in any other servlet that needs the header. In the case of request delegation, any future changes to the header are automatically reflected on all the servlets. A much more elegant solution to this problem is provided by JSPs. In practice, a servlet request delegation is usually used to break large servlets into smaller and relevant parts.

In addition to this, simple server-side components include request delegations, which are a key part of server-side Java implementations of popular design patterns. As far as Java Servlet and JSP are concerned, design patterns are commonly agreed-upon methods to build Web applications that are robust in functionality and easily maintainable.

You may already know that there are well-defined scopes for variables in Java. Local variables are declared inside methods and are by default only available inside the scope of that method. Instance variables, declared in a class but outside a method or Constructor, are available to all methods in a Java class. There are many other possible scopes as well. These scopes help to keep track of objects and to allow JVM to accurately carry out garbage-collection of the memory. Apart from the various Java variable scopes, such as local and global, the servlets also provide some new scopes. The request scope is introduced by a request delegation.

The request scope and the other scopes are not officially marked by the Servlet specification. A set of methods defined by the servlet specification in the `javax.servlet` package allow you to bind objects to and retrieve objects from various containers (that are themselves objects). As an object bound in this manner is referenced by the container it is bound to, the bound object is not destroyed until the reference is removed. Therefore, bound objects are in the same scope as the container they are bound to. For example, the `HttpServletRequest` object is bound to a container and includes the methods of the `HttpServletRequest` interface. The methods of the `HttpServletRequest` object can be used to bind, access, and remove objects to and from the request scope that is shared by all servlets to which a request is delegated. This is an important concept and can easily be shown in the `Servlet2Servlet` class, created in Listing 4.12.

A request scope can be defined as a method in which an object is passed between two or more servlets with the assurance that the object goes out of scope (that is, it will be garbage-collected) after the last servlet has performed its task. In later chapters, more examples of this concept are provided. Let's first create the `Servlet2Servlet` servlet that passes an object to another servlet, `Servlet2Servlet2`. Listing 4.12 provides the code for the `Servlet2Servlet.java` file (you can find this file on the CD in the `code\JavaEE\Chapter4\FirstApp\src\com\kogent` folder):

#### **Listing 4.12: Displaying the Code for the `Servlet2Servlet.java` File**

```
package com.kogent;
import java.io.*;
```

```

import javax.servlet.*;
import javax.servlet.http.*;

public class Servlet2Servlet extends HttpServlet
{
    public void doGet(HttpServletRequest request, HttpServletResponse
    response) throws IOException, ServletException
    {
        response.setContentType("text/html");
        String param = request.getParameter("value");

        if(param != null && !param.equals(""))
        {
            request.setAttribute("value", param);
            RequestDispatcher rd =
            request.getRequestDispatcher("/Servlet2Servlet2");
            rd.forward(request, response);
            return;
        }

        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Servlet #1</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>A form from servlet #1</h1>");
        out.println("<form>");
        out.println("Enter a value to send to Servlet #2.");
        out.println("<input name=\"value\"><br>");
        out.print("<input type=\"submit\" ");
        out.println("value=\"Send to Servlet #2\"");
        out.println("</form>");
        out.println("</body>");
        out.println("</html>");
    }
}

```

Compile the `Servlet2Servlet` servlet and map it to the `/Servlet2Servlet` URL pattern in the `web.xml` file by using the following code snippet:

```

<servlet>
    <servlet-name>Servlet2Servlet</servlet-name>
    <servlet-class>com.kogent.Servlet2Servlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>Servlet2Servlet</servlet-name>
    <url-pattern>/Servlet2Servlet</url-pattern>
</servlet-mapping>

```

Listing 4.13 shows the code for the `Servlet2Servlet2.java` file (you can find this file on the CD in the `code\JavaEE\Chapter4\FirstApp\src\com\kogent` folder):

#### **Listing 4.13:** Displaying the Code for the `Servlet2Servlet2.java` File

```

package com.kogent;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Servlet2Servlet2 extends HttpServlet
{

    public void doGet(HttpServletRequest request, HttpServletResponse
    response) throws IOException, ServletException
    {

```

```

response.setContentType("text/html");
PrintWriter out = response.getWriter();
out.println("<html>");
out.println("<head>");
out.println("<title>Servlet #2</title>");
out.println("</head>");
out.println("<body>");
out.println("<h1>Servlet #2</h1>");
String value = (String)request.getAttribute("value");
if(value != null && !value.equals(""))
{
    out.print("Servlet #1 passed a String object via ");
    out.print("request scope. The value of the String is: ");
    out.println("<b>" + value + "</b>.");
}
}

else
{
    out.println("No value passed!");
}
out.println("</body>");
out.println("</html>");
}
}

```

Now, save the code given in Listing 4.13, with the name `Servlet2Servlet2.java` in the `/src/com/kogent` directory of the `FirstApp` Web application. In addition, compile the `Servlet2Servlet2` servlet and map it to the `/Servlet2Servlet2` URL pattern in the `web.xml` file, as provided in the following code snippet:

```

<servlet>
    <servlet-name>Servlet2Servlet2</servlet-name>
    <servlet-class>com.kogent.Servlet2Servlet2</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>Servlet2Servlet2</servlet-name>
    <url-pattern>/Servlet2Servlet2</url-pattern>
</servlet-mapping>

```

Redeploy the `FirstApp` Web application and the application is ready for execution. Next, browse the `http://localhost:8080/FirstApp/Servlet2Servlet2` URL. Figure 4.18 shows the servlet response that appears similar to a simple HTML form asking you to enter a value to pass to the second servlet:

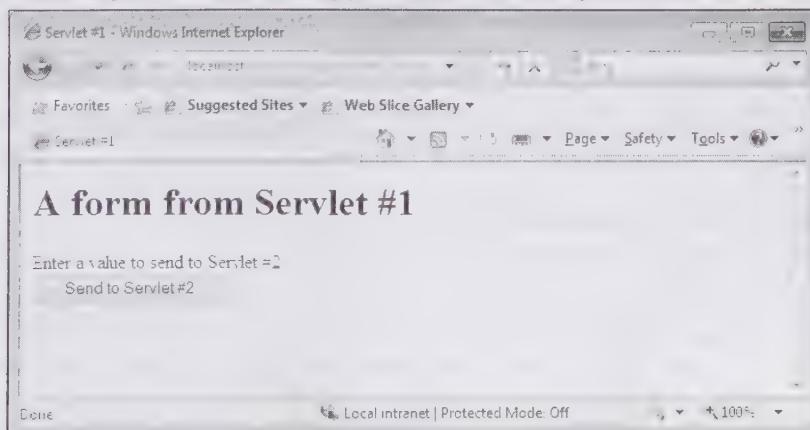
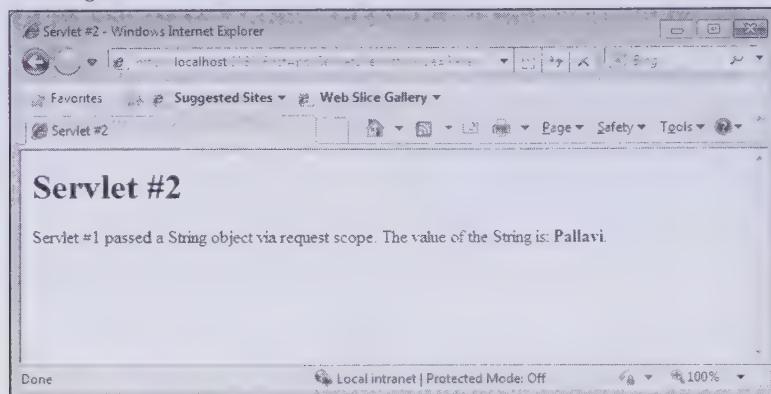


Figure 4.18: Displaying the Output of the `Servlet2Servlet2` Servlet

Type a value in the Enter a value to send to Servlet#2 text box to send to the second servlet. In our case, we have entered the value, Pallavi. Now, click the Send to Servlet #2 button. The second servlet appears, as shown in Figure 4.19:



**Figure 4.19: Displaying Servlet 2 in Progress**

The preceding example demonstrates how the value of one servlet is passed to the other servlet. After discussing the concept of request delegation and request scope, let's now learn how to share information among servlets by using the servlet collaboration technique.

## Implementing Servlet Collaboration

Servlet sometimes cooperate with each other by sharing information. This sort of cooperation is known as servlet collaboration. The collaborating servlets can pass the shared information between each other through method invocations; however, to do this, each servlet is required to know about the servlets with which it is collaborating. This adds unnecessary burden on the server. Several other techniques can be used to carry out for servlet collaboration. Let's discuss these techniques in the following sections.

### Collaboration through the System Properties List

A simple way for servlets in collaboration to share information is by using Java's system-wide Properties list. The Properties list is found in the `java.lang.System` class and holds standard system properties, such as `java.version` and `path.separator` as well as application-specific properties. Servlets can use the Properties list to hold the information they need to share. A servlet can add or change a property by using the `setProperties()` method, as shown in the following code snippet:

```
System.setProperties().put("key", "value");
```

The concerned servlet, or some other servlet running in the same JVM, can later get the value of the property by calling the `getProperties()` method, as shown in the following code snippet:

```
String value = System.getProperty("key");
```

The property can also be removed, by calling the `remove()` method, as shown in the following code snippet:

```
System.getProperties().remove("key");
```

Generally, you should include a prefix while defining the key for a property, which contains the name of the servlet's package and the name of the collaboration group, for example, `com.kogent.Servlet.ShoppingCart`. The `Properties` class is `String`-based, which implies that each key and value is supposed to be a `String`. However, this is not a commonly enforced limitation and can be ignored by servlets if they want to store and retrieve non-`String` objects. Such servlets can use the `Properties` list as a `Hashtable` at the time of storing keys and values because the `Properties` class extends the `Hashtable` class; therefore, the `Properties` list can be treated as a `Hashtable`. For example, a servlet can add or change a property object by calling the `setProperties()` method, as shown in the following code snippet:

```
System.setProperties().put(keyObject, valueObject);
```

The property object is retrieved by calling the `getProperties()` method, as shown in the following code snippet:

```
SomeObject valueObject = (SomeObject)System.getProperties().get(keyObject);
```

The property object is removed by calling the `remove()` method, as shown in the following code snippet:

```
System.getProperties().remove(keyObject);
```

Due to the misuse of the `Properties` list, the `getProperty()`, `list()` and `save()` methods of the `Properties` class throw the `ClassCastException` exception and it is also assumed that each key and value is of the `String` type. Due to this reason, you should use some other technique for Servlet collaboration. JVM should look for the class files for the `keyObject` and `valueObject` arguments in the server's `CLASSPATH`. The class files should not be looked in the default servlet directory where the servlet-class loaders load and reload the servlets.

Servlet collaboration works correctly with the use of property lists if the servlets use the property lists to share insensitive, non-critical, and easily replaceable information. For example, consider a set of servlets that are used to sell pizzas and share a particular pizza on a special day of the week. To implement collaboration between the servlets, the administrative servlet can use a property list to set a special day and the pizza to be served on that day. The following code snippet shows the implementation of the `setProperties()` method to set the value for the pizza and the special day:

```
System.setProperties().put("com.kogent.ServingPizza.special.pizza", "Cheese pizza");
System.setProperties().put("com.kogent.ServingPizza.special.day", new Date());
```

Now, every other servlet on the server is able to access the special properties and display it with the code shown in the following code snippet:

```
String pizza = System.getProperty("com.kogent.ServingPizza.special.pizza");
Date day = (Date)System.getProperties().get("com.kogent.ServingPizza.special.day");
DateFormat df = DateFormat.getDateInstance(DateFormat.SHORT);
String today = df.format(day);
prnwriter.println("Our pizza special today (" + today + ") is: " + pizza);
```

In the preceding code snippet, the `System.getProperty()` method is used to retrieve the value of the `com.kogent.ServingPizza.special.pizza` key.

## *Collaboration through a Shared Object*

Sharing information through a shared object is another way for servlets to collaborate with one another. A shared object can hold a pool of shared information and make it available to each servlet as required. In a sense, the system `Properties` list is a special case example of a shared object. To manipulate an object's data, the shared object often incorporates some business logic or rules. By incorporating a rule that the shared object's data be available only through well-defined methods, the rule protects the shared object's data. The rule helps to protect data integrity and triggers events so that they can handle certain conditions. Moreover, various data manipulations can be abstracted into a single method invocation. This capability is not available in the case of the `Properties` list. The garbage collector is an important aspect of collaborating through a shared object. If at any time a loaded servlet does not reference the object, then the servlet can reclaim it. Therefore, every servlet that uses a shared object must save a reference to the object to keep the garbage collector at bay.

Let's consider the previous example in which we used servlets to sell pizzas. Collaboration between servlets to maintain a shared inventory of ingredients can be implemented through a shared object. For this, you first need to create a shared `PizzaInventory` class. The `PizzaInventory` class is defined to maintain the ingredient count and display the count through public methods. An example of the `PizzaInventory` class is shown in Listing 4.14. Notice that this class is a singleton (a class that has just one instance). This makes it easy for every servlet sharing the class to maintain a reference to the same instance.

Now, let's discuss the shared inventory class. Listing 4.14 shows the code for `PizzaInventory.java` (you can find this file on the CD in the `code\JavaEE\Chapter4\FirstApp\src\com\kogent` folder):

**Listing 4.14:** Displaying the Code for the `PizzaInventory.java` File

```
package com.kogent;
public class PizzaInventory {
    // Protect the constructor, so no other class can call it
```

```

private PizzaInventory() { }

// Create the only instance, save it to a private static variable
private static PizzaInventory instance = new PizzaInventory();

// Make the static instance publicly available
public static PizzaInventory getInstance() { return instance; }

// How many servings of each item do we have?
private int cheese = 0;
private int wheatflour = 0;
private int beans = 0;
private int capsicum = 0;

// Add to the inventory
public void addcheese(int added) { cheese += added; }
public void addwheatflour(int added) { wheatflour += added; }
public void addBeans(int added) { beans += added; }
public void addCapsicum(int added) { capsicum += added; }

// Called when it is time to make a pizza.
// Returns true if there are enough ingredients to make the pizza,
// false if not. Decrements the ingredient count when there are enough.
Synchronized public boolean makePizza()
{
    // Pizza requires one serving of each item
    if (cheese > 0 && wheatflour > 0 && beans > 0 && capsicum > 0)
    {
        cheese--; wheatflour--; beans--; capsicum--;
        return true;           // can make the pizza
    }
    else
    {
        // could order more ingredients
        return false;         // cannot make the pizza
    }
}
}

```

Save the `PizzaInventory.java` file in the `src\com\kogent` directory of the `FirstApp` Web application. `PizzaInventory` maintains an inventory count for four pizza ingredients: cheese, wheatflour, beans, and capsicum. The `PizzaInventory` class holds the count of the ingredients with the private instance variables. Information of the count should be kept in an external database to maintain a record of the quantity of ingredients used to produce pizzas. Each ingredient's inventory count is increased by using the `addCheese()`, `addWheatflour()`, `addBeans()`, and `addCapsicum()` methods. These methods may be called from a servlet accessed by the ingredient prepared in the day.

In the `makePizza()` method, the value of the inventory counts are decreased together. The role of this method is to check whether or not there are enough ingredients to make a full pizza. If there is, then the method decreases the ingredient count and returns `true`. However, if the ingredients are insufficient, then the `makePizza()` method returns `false` (in an improved version, the method may choose to order more ingredients). The `makePizza()` method may be called by a servlet selling pizzas over the Internet, and perhaps also by a servlet communicating with the check-out cash register. Remember that, similar to all the other non-servlet-class files, the class file for `PizzaInventory` is placed somewhere in the server's CLASSPATH (such as `server_root/classes`). Listing 4.15 shows you how a servlet adds ingredients to the inventory.

Let's now create a servlet to add some ingredients to a shared inventory. Listing 4.15 shows the code for the `PizzaInventoryProducer.java` file (you can find this file on the CD in the `code\JavaEE\Chapter4\FirstApp\src\com\kogent` folder).

**Listing 4.15:** Displaying the Code for the `PizzaInventoryProducer.java` File

```

package com.kogent;
import java.io.*;

```

```

import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
import com.kogent.* ;

public class PizzaInventoryProducer extends HttpServlet
{
    // Get (and keep) a reference to the shared PizzaInventory instance
    PizzaInventory inventory = PizzaInventory.getInstance();

    public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException
    {
        res.setContentType("text/html");
        PrintWriter prnwriter = res.getWriter();

        prnwriter.println("<HTML>");
        prnwriter.println("<HEAD><TITLE>Pizza Inventory Producer
                        </TITLE></HEAD>");

        // Produce random amount of each item
        Random random = new Random();

        int cheese = Math.abs(random.nextInt() % 10);
        int wheatflour = Math.abs(random.nextInt() % 10);
        int bean = Math.abs(random.nextInt() % 10);
        int capsicum = Math.abs(random.nextInt() % 10);

        // Add the item sto the inventory
        inventory.addCheese(cheese);
        inventory.addWheatflour(wheatflour);
        inventory.addBeans(bean);
        inventory.addCapsicum(capsicum);

        // Print the production results
        prnwriter.println("<BODY>");
        prnwriter.println("<H1>Added ingredients:</H1>");
        prnwriter.println("<PRE>");
        prnwriter.println("cheese: " + cheese);
        prnwriter.println("wheatflour: " + wheatflour);
        prnwriter.println("beans: " + bean);
        prnwriter.println("capsicum: " + capsicum);
        prnwriter.println("</PRE>");
        prnwriter.println("</BODY></HTML>");

    }
}

```

Save the `PizzaInventoryProducer.java` file in the `src\com\kogent` directory of the `FirstApp` Web application and compile the `PizzaInventory` and `PizzaInventoryProducer` servlets. The following code snippet is used to map the `PizzaInventoryProducer` servlet to the `/PizzaInventoryProducer` by using the `<url-pattern>` element in the `web.xml` file:

```

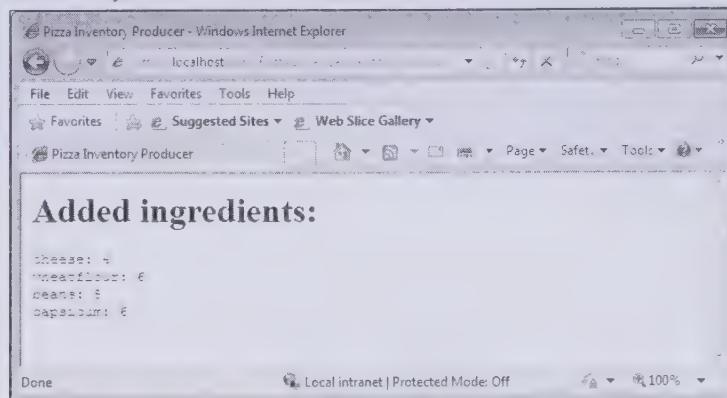
<servlet>
    <servlet-name>PizzaInventoryProducer</servlet-name>
    <servlet-class>com.kogent.PizzaInventoryProducer </servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>PizzaInventoryProducer</servlet-name>
    <url-pattern>/PizzaInventoryProducer</url-pattern>
</servlet-mapping>

```

Create a new `FirstApp.war` file and redeploy the `FirstApp` Web application. Now, browse the `http://localhost:8080/FirstApp/PizzaInventoryProducer` URL to see the output.

A random amount of each ingredient (somewhere between zero to nine servings) is produced and added to the inventory, whenever the `PizzaInventoryProducer` servlet is accessed. Figure 4.20 shows the result of executing the `PizzaInventoryProducer` servlet:



**Figure 4.20: Displaying the Output of the `PizzaInventoryProducer` Servlet**

The `PizzaInventoryProducer` servlet plays the important role of maintaining a reference to the shared `PizzaInventory` instance. This implies that the `PizzaInventory` instance cannot be reclaimed by the garbage collector until the servlet is loaded. The code for the `PizzaInventoryConsumer.java` file is provided on the CD.

Now, let's create a servlet that calls the `makePizza()` method, informing the inventory that it wants to make a pizza. The `PizzaInventoryConsumer.java` file provides the code for the servlet. Save the `PizzaInventoryConsumer.java` file in the `src\com\kogent` directory of the `FirstApp` Web application. Listing 4.16 shows the code for the `PizzaInventoryConsumer.java` file (you can find this file on the CD in the `code\JavaEE\Chapter4\FirstApp\src\com\kogent` folder):

**Listing 4.16: Displaying the Code for the `PizzaInventoryConsumer.java` File**

```

package com.kogent;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import com.kogent.*;

public class PizzaInventoryConsumer extends HttpServlet
{
    // Get (and keep) a reference to the shared PizzaInventory instance
    private PizzaInventory inventory = new PizzaInventory.getInstance();

    public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException
    {
        res.setContentType("text/html");
        PrintWriter prnwriter = res.getWriter();

        prnwriter.println("<HTML>");
        prnwriter.println("<HEAD><TITLE>Pizza Inventory Consumer
                        </TITLE></HEAD>");

        prnwriter.println("<BODY><BIG>");

        if(inventory.makePizza())
        {
            prnwriter.println("Your pizza will be ready
                            in a few minutes.");
        }
    }
}

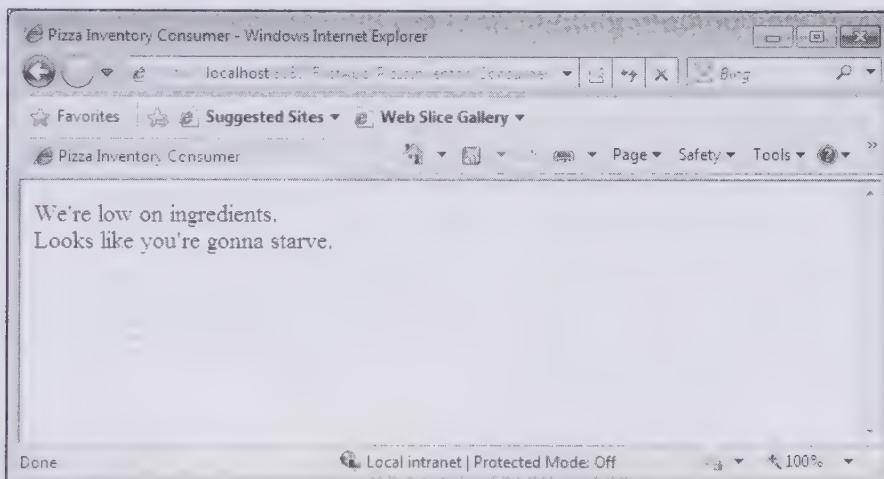
```

```

        else
        {
            prnwriter.println("We're low on ingredients.<BR>");
            prnwriter.println("Looks like you're gonna starve.");
        }
        prnwriter.println("</BIG></BODY></HTML>");
    }
}

```

The PizzaInventoryConsumer servlet does not need to decrease the ingredients count by itself. It maintains a reference to the PizzaInventory instance. This implies that the PizzaInventory instance can be referenced even if the PizzaInventoryProducer servlet is unloaded. Compile the preceding servlet and redeploy the FirstApp Web application. Now, browse the <http://localhost:8080/FirstApp/PizzaInventoryConsumer> URL to see the output. Figure 4.21 shows the output, which is displayed when the PizzaInventoryConsumer servlet is executed:



**Figure 4.21: Displaying the Output of the PizzaInventoryConsumer Servlet**

You can also make a servlet act as a shared object. There is an added advantage of using a shared servlet. Sharing allows a servlet to maintain its state by using its `init()` and `destroy()` methods. In addition, each time a shared servlet is accessed, the servlet can print its current state. Let's re-create the PizzaInventory servlet to implement the concept of sharing a servlet.

Listing 4.17 shows the code for the re-created `PizzaInventoryServlet.java` file (you can find this file on the CD in the `code\JavaEE\Chapter4\FirstApp\src\com\kogent` folder):

**Listing 4.17: Displaying the Code for the PizzaInventoryServlet.java File**

```

package com.kogent;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class PizzaInventoryServlet extends HttpServlet
{
    // How many "servings" of each item do we have?
    private int cheese = 0;
    private int wheatflour = 0;
    private int beans = 0;
    private int capsicum = 0;

    // Add to the inventory as more servings are prepared.
}

```

```

public void addCheese(int added) { cheese += added; }
public void addWheatflour(int added) { wheatflour += added; }
public void addBeans(int added) { beans += added; }
public void addCapsicum(int added) { capsicum += added; }

// Called when it is time to make a pizza.
// Returns true if there are enough ingredients to make the pizza,
// false if not. Decrements the ingredient count when there are enough.
synchronized public boolean makePizza()
{
    // Pizza requires one serving of each item
    if (cheese > 0 && wheatflour > 0 && beans > 0 && capsicum > 0)
    {
        cheese--; wheatflour--; beans--; capsicum--;
        return true; // can make the pizza
    }
    else
    {
        // Could order more ingredients
        return false; // cannot make the pizza
    }
}

// Display the current inventory count.
public void doGet(HttpServletRequest req, HttpServletResponse res)
throws ServletException, IOException
{
    res.setContentType("text/html");
    PrintWriter prnwriter = res.getWriter();

    prnwriter.println("<HTML><HEAD><TITLE>Current
                      Ingredients</TITLE></HEAD>");
    prnwriter.println("<BODY>");
    prnwriter.println("<TABLE BORDER=1>");
    prnwriter.println("<TR><TH COLSPAN=2> Current ingredients:</TH>
                      </TR>");
    prnwriter.println("<TR><TD>Cheese:</TD><TD>" + cheese + "</TD>
                      </TR>");
    prnwriter.println("<TR><TD>Wheatflour:</TD><TD>" + wheatflour +
                     "</TD></TR>");
    prnwriter.println("<TR><TD>Beans:</TD><TD>" + beans + "</TD></TR>");
    prnwriter.println("<TR><TD>Capsicum:</TD><TD>" + capsicum +
                     "</TD></TR>");
    prnwriter.println("</TABLE>");
    prnwriter.println("</BODY></HTML>");
}

// Load the stored inventory count
public void init(ServletConfig config) throws ServletException
{
    super.init(config);
    loadState();
}

public void loadState()
{
    // Try to load the counts
    FileInputStream file = null;
    try
    {
        file = new
              FileInputStream("PizzaInventoryServlet.state");
        DataInputStream in = new DataInputStream(file);
        cheese = in.readInt();
        wheatflour = in.readInt();
    }
}

```

```

        beans = in.readInt();
        capsicum = in.readInt();
        file.close();
        return;
    }
    catch (IOException ignored)
    {
        // Problem during read
    }
    finally
    {
        try
        {
            if (file != null) file.close();
        }
        catch (IOException ignored) { }
    }
}

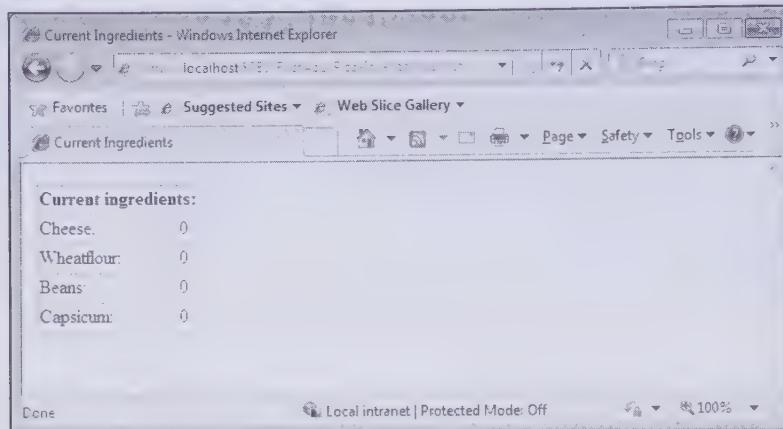
public void destroy()
{
    saveState();
}

public void saveState()
{
    // Try to save the counts
    FileOutputStream file = null;
    try
    {
        file = new FileOutputStream("PizzaInventoryServlet.state");
        DataOutputStream prnwriter = new DataOutputStream(file);

        prnwriter.writeInt(cheese);
        prnwriter.writeInt(wheatflour);
        prnwriter.writeInt(beans);
        prnwriter.writeInt(capsicum);
        return;
    }
    catch (IOException ignored)
    {
        // Problem during write
    }
    finally
    {
        try
        {
            if (file != null) file.close();
        }
        catch (IOException ignored) { }
    }
}
}

```

Now, save the `PizzaInventoryServlet.java` file in the `src/com/kogent` directory of the `FirstApp` Web application. Then, redeploy the `FirstApp` Web application and view the output by accessing the `http://localhost:8080/FirstApp/PizzaInventoryServlet` URL. The `PizzaInventoryServlet` servlet is no longer a singleton but a normal HTTP servlet, which defines an `init()` method that loads its state as well as a `destroy()` method that saves its state. Figure 4.22 shows the output of the `PizzaInventoryServlet` servlet:



**Figure 4.22: Displaying the Output from PizzaInventoryServlet, Showing its State**

Remember that even as a servlet, the `PizzaInventoryServlet.class` file should remain in the server's standard CLASSPATH. This is required to keep the `PizzaInventoryServlet` servlet from being reloaded. Both the `PizzaInventoryProducer` and `PizzaInventoryConsumer` classes can get a reference to the `PizzaInventoryServlet` servlet. The following code snippet shows how to reuse the `PizzaInventoryServlet` servlet:

```
// Get the inventory Servlet instance if we haven't before
if (inventory == null)
{
    inventory = ( PizzaInventoryServlet )
        ServletUtils.getServlet("PizzaInventoryServlet",
            req, getServletContext());
}

// If the load was unsuccessful, throw an exception
if (inventory == null)
{
    throw new ServletException
        ("Could not locate PizzaInventoryServlet");
}
```

In the preceding code snippet, instead of calling `PizzaInventory.getInstance()` method, the producer and consumer classes can ask the `PizzaInventoryServlet` instance from the server.

## Collaboration through Inheritance

Collaborating through inheritance is perhaps the easiest technique of servlet collaboration. In the inheritance technique, each servlet requiring collaboration can extend the same class and opt for inheriting the same shared information. This simplifies the code of the collaborating servlets and allows only the proper subclasses to access the shared information. Moreover, the common superclass can hold a reference to the shared information or hold the shared information itself.

In the following sections, you learn to collaborate with servlets through inheritance in two ways, by inheriting a shared reference and by inheriting the shared information.

### Inheriting a Shared Reference

In case of inheriting a shared reference, a common superclass can hold any number of references to shared business objects, which are easily made available to its subclasses. Such a superclass is shown in Listing 4.18, which we can use for our `PizzaInventory` example.

Listing 4.18 shows the code for `PizzaInventorySuperclass.java` (you can find this file on the CD in the `code\JavaEE\Chapter4\FirstApp\src\com\kogent` folder):

**Listing 4.18:** Displaying the Code for the PizzaInventorySuperclass.java File

```

package com.kogent;

import javax.servlet.*;
import javax.servlet.http.*;
import com.kogent.* ;

public class PizzaInventorySuperclass extends HttpServlet
{
    protected static PizzaInventory inventory = PizzaInventory.getInstance();
}

```

In Listing 4.18, the `PizzaInventorySuperclass` servlet creates a new `PizzaInventory` instance. Now, save the `PizzaInventorySuperclass.java` file in the `src\com\kogent` directory of the `FirstApp` Web application. The `PizzaInventoryProducer` and `PizzaInventoryConsumer` classes can now extend the `PizzaInventorySuperclass` class and inherit a reference to the `PizzaInventory` instance. To understand how the `PizzaInventoryProducer` class can extend and inherit the reference to the `PizzaInventory` instance, the code for the `PizzaInventoryConsumer` servlet is revised in Listing 4.19.

Listing 4.19 shows the revised code for `PizzaInventoryConsumer.java` (you can find this file on the CD in the `code\JavaEE\Chapter4\FirstApp\src\com\kogent` folder):

**Listing 4.19:** Displaying the Code for the `PizzaInventoryConsumer.java` File

```

package com.kogent ;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class PizzaInventoryConsumer extends PizzaInventorySuperclass
{
    public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException
    {
        res.setContentType("text/html");
        PrintWriter prnwriter = res.getWriter();

        prnwriter.println("<HTML>");
        prnwriter.println("<HEAD><TITLE>Pizza Inventory Consumer
        </TITLE></HEAD>");

        prnwriter.println("<BODY><BIG>");
        if (inventory.makePizza())
        {
            prnwriter.println("Your pizza will be ready in a
                few minutes.");
        }
        else
        {
            prnwriter.println("We're low on ingredients.<BR>");
            prnwriter.println("Looks like you're gonna starve.");
        }
        prnwriter.println("</BIG></BODY></HTML>");
    }
}

```

The `PizzaInventory` class does not have to be a singleton anymore. The reason is that subclasses naturally inherit the same instance. The class file for the `PizzaInventorySuperclass` class should be put in the server's CLASSPATH. This is required to keep the `PizzaInventorySuperclass` class from being reloaded.

## Inheriting Shared Information

Apart from allowing servlets to hold shared references, you can also inherit the shared information. For this, you can use a common superclass to hold the shared information by itself and optionally make it available through inherited business logic methods.

The following code snippet shows how the `PizzaInventorySuperclass` class holds its own shared information:

```
public class PizzaInventorySuperclass extends
    HttpServlet {
    // How many "servings" of each item do we have?
    private static int cheese = 0;
    private static int wheatflour = 0;
    private static int beans = 0;
    private static int capsicum = 0;

    // Add to the inventory as more servings are prepared.
    protected static void addCheese(int added) { cheese += added; }
    protected static void addWheatflour(int added) { wheatflour
        += added; }
    protected static void addBeans(int added) { beans += added; }
    protected static void addCapsicum(int added) { capsicum
        += added; }

    // Called when it is time to make a pizza.
    // Returns true if there are enough ingredients to make the pizza,
    // false if not. Decrements the ingredient count when there are enough.
    synchronized static protected boolean makePizza() {
        // ...etc...
    }
    // ...The rest matches PizzaInventoryServlet...
```

Now, let's discuss the difference between the `PizzaInventorySuperclass` and `PizzaInventoryServlet` servlets to analyze what changes are required to inherit the shared information in a servlet. There are only two differences between the two servlets, which are as follows:

- All the variables and methods of the `PizzaInventorySuperclass` servlet are static, which is not the case with the `PizzaInventoryServlet` class. This guarantees that only one inventory is maintained for all the subclasses.
- All the methods of the `PizzaInventorySuperclass` servlet are protected. This implies that the methods will be available only to the subclasses.

With this, we come to the end of the chapter. Let's now recap the main points of the chapter in a short summary.

## Summary

This chapter has discussed the latest version of the Java Servlet API, version 3.0. You have first explored the general features of a Java Servlet, after which the features of Servlet 3.0 have been discussed. The chapter has then explained the classes and packages of the Servlet API that are used to develop Web applications. You have also learned about the life cycle of a servlet and configuring a servlet in the web.xml file. Apart from this, you have learned to create a sample servlet, first by mapping it in the web.xml file and then by using annotations. The chapter has also listed the noteworthy interfaces of the Servlet 3.0 API. At the end of the chapter, you have learned about request delegation, request scope and servlet collaboration.

## Quick Revise

- Q1. The two arguments passed to the `forward()` method of the `RequestDispatcher` class are objects of the ..... classes.
- `HttpServletRequest` and `HttpServletResponse`
  - `HttpServletResponse` and `PrintWriter`

- C. `ServletContext` and `ServletConfig`  
 D. `HttpServletRequest` and `ServletContext`
- Ans. The correct option is A.
- Q2. To get an object of the `PrintWriter` class, we use the `getWriter()` method of the ..... class.
- A. `HttpServletRequest`
  - B. `HttpServletResponse`
  - C. `SessionContext`
  - D. `HttpSession`
- Ans. The correct option is B.
- Q3. A life cycle method of a servlet is .....
- A. `init()`
  - B. `service()`
  - C. `destroy()`
  - D. Above All
- Ans. The correct option is D.
- Q4. We can get an object of the `RequestDispatcher` class by using the `getRequestDispatcher()` method on the ..... class.
- A. `ServletRequest`
  - B. `ServletContext`
  - C. Both A and B
  - D. None
- Ans. The correct option is C.
- Q5. Initialization parameters can be fetched by using the ..... method.
- A. `getAttribute()`
  - B. `getParameter()`
  - C. `getInitParameter()`
  - D. `getServletContext()`
- Ans. The correct option is C.
- Q6. The number of `ServletContext` objects present for an application is .....
- A. 2
  - B. 1
  - C. Not fixed
  - D. Each for a Servlet
- Ans. The correct option is B.
- Q7. `HttpServlet` extends .....
- A. The `javax.servlet.GenericServlet` class
  - B. The `javax.servlet.http.GenericServlet` class
  - C. The `javax.servlet.http.HttpServletRequest` interface
  - D. The `javax.servlet.ServletInputStream` class
- Ans. The correct option is A.
- Q8. The method used to fetch the value of an object in the request scope is .....
- A. `getAttribute()`
  - B. `getParameter()`
  - C. `getInitParameter()`
  - D. None
- Ans. The correct option is A.
- Q9. What is a servlet?
- Ans. A servlet is a simple Java class working on the request-response model. Various interfaces and classes to handle common HTTP-specific services are defined in the Java Servlet API. Each servlet implements the `Servlet` interface by providing implementation of the life cycle methods, `init()`, `service()`, and `destroy()`.
- Q10. Differentiate between the `ServletContext` and `ServletConfig` objects.
- Ans. A `ServletContext` object is used to communicate with a Servlet container while `ServletConfig`, which is a Servlet configuration object, is passed to the servlet by a container when the servlet is initialized. A `ServletContext` object is contained within a `ServletConfig` object.

**Q11.** What is the difference between the `getRequestDispatcher()` methods of the `ServletRequest` and `ServletContext` interfaces?

Ans. Both `getRequestDispatcher()` methods take a String parameter, which is a path to the location where a user's request would be forwarded. The `getRequestDispatcher()` method of the `ServletRequest` interface can accept both types of paths, i.e. the relative path from the requesting servlet and the path relative to the context root. However, the `getRequestDispatcher()` method of the `ServletContext` interface cannot accept the relative path.

**Q12.** Define the `init()` method of a servlet.

Ans. The `init()` method of the `HttpServlet` class is called before a servlet handles the first request and is used to initialize the servlet. This `init()` method also saves the `ServletConfig` object by using the `super.init()` method and stores the initialization details of a servlet. This method is called once only.

**Q13.** How is the `GET` method different from the `Post` method?

Ans. In the case of the `GET` method, all data submitted with an HTML form is attached with a URL. This method is faster and easier to use than the `POST` method but not as secure, and the upper limit of the URL length limits the amount of data transferred. The `Post` method, on the other hand, puts name/value combinations inside the HTTP request body and is therefore more secure. In addition, there is no limit for the amount of data that can be sent in this method.

# 5

# Handling Sessions in Servlets 3.0

## *If you need an information on:*

## **See page:**

Describing a Session	208
Introducing Session Tracking	208
Exploring the Session Tracking Mechanisms	209
Using the Java Servlet API for Session Tracking	217
Creating Login Application using Session Tracking	228

A HyperText Transfer Protocol (HTTP) session stores the information of activities performed by users, such as requesting and accessing resources over a network. An HTTP session helps a servlet to keep track of the activities of a user. The process of keeping a track of a client's state in servlets is termed as session handling, which is also known as session tracking. The process of tracking the details of a session in servlets is similar to other session tracking mechanisms used in previous technologies, such as Common Gateway Interface (CGI).

Servlets provide convenient ways to synchronize multiple sessions between a client and server. As a result, servlets are able to maintain the session state on the server until the client completes the browsing session.

Some common and widely-used techniques to maintain HTTP sessions are persistent cookies, Uniform Resource Locator (URL) rewriting, and hidden form fields. There are few proprietary session tracking strategies, which are vendor dependent that persist state to a disk and/or a database. However, in this regard, there are no clear winners; with pros and cons to each technique.

To implement session tracking in servlets, you need not required to deal with each session tracking technique in your Web applications. Instead, you can use an Application Programming Interface (API), which selects the most appropriate HTTP session technique based on the capabilities of the client and server. The API used for HTTP session technique provides abstract details, such as whether the browser has cookies enabled or the server supports URL rewriting. In addition, to maintain HTTP sessions with servlet session tracking, API provides various possible implementations, such as cookies and URL rewriting.

Firstly, the chapter introduces you to session and how client's state is traced within a session. Further, different session tracking techniques, such as hidden form fields, cookies, and URL rewriting are explored in detail. Later, the chapter explores the session handling API for servlets, which helps overcome the disadvantages of the traditional session tracking techniques. Towards the end, a simple login application is developed to demonstrate the session handling process.

Let's start the chapter with an introduction about a session.

## Describing a Session

A session can be defined as a collection of HTTP requests shared between a client and Web server over a period of time. While creating a session, you require setting its lifetime, which is set to thirty minutes by default. After the set lifetime expires, the session is destroyed and all its resources are returned back to the servlet engine. The succeeding subsections discuss about the life-cycle of a session and focus on how to create and destroy sessions. Prior moving ahead towards the discussion on session life-cycle, let's first discuss about session tracking and its mechanisms.

## Introducing Session Tracking

Session tracking is a process of gathering the user information from Web pages, which can be used in an application. Let's cite an example, a shopping cart application can be taken as the most common example of session tracking. In the shopping cart application, a client accesses the server several times from the same browser and visits several Web pages. After browsing the Web pages, the client decides to purchase some of the items offered by the Web site for sale and clicks the BUY ITEM button. In this case, if a stateless server-side object serves each transaction, and the client's side does not provide any identification on each request; it would not be possible to maintain a filled shopping cart over several HTTP requests from the client. If the user visits a Web page multiple times and selects different items to be added to the shopping cart in each visit, the stateless nature of HTTP might not relate each visit to the same session. Therefore, even writing a stateless transaction data to persistent storage would not be a solution in this regard.

Therefore, session tracking involves identifying the user sessions by related ID numbers and tying the requests to their sessions by using the said ID number. Cookies and URL rewriting are the typical mechanisms for session tracking. Depending upon the Servlet specification, session tracking is implemented through HTTP session objects by the servlet container in the application server. These HTTP session objects are instances of a class and implement the `javax.servlet.http.HttpSession` interface. The `getSession()` method of the `HttpSession` interface is used to create the HTTP session object and the stateful client interaction.

The question may arise about the scope of the HTTP session object. Will it be limited to single request or multiple requests or across users? The scope of the HTTP session object is limited to the single client. It is important to note that you cannot use session objects to share the data between different applications and different clients of the same application. There is only one HTTP session object for each client in each application.

After having a brief overview about session tracking, let's discuss about the various mechanisms that can be used to implement session tracking in servlets.

## Exploring the Session Tracking Mechanisms

To track the session details for a specific user to maintain the session, you need to implement a mechanism in your Web application. You can implement the following session tracking mechanisms to track the session details:

- Cookies
- Hidden form fields
- URL rewriting
- Secure Socket Layer (SSL) sessions

Let's discuss these in detail.

### Using Cookies

While working with session tracking, numerous approaches have been adapted to add a degree of statefulness to the HTTP protocol. Among these approaches, the most widely accepted one is the use of cookies. A cookie is used to transmit an identifier between a server and a client. The transmitting of an identifier is in conjunction with stateful servlets, which can maintain session objects. These session objects are simply the dictionaries that store values (Java objects) together with their associated keys (Java Strings). The following steps describe the usage of cookies:

- After creating a session, the server (container) sends a cookie (as a response from stateful servlet) with a session identifier back to the client. Some other useful information, such as username and password, is also sent with the cookie (all less than 4 KB). The cookie, named JSESSIONID, is sent by the container as a response in the HTTP response header.
- Then, whenever any subsequent request is received from the same Web client session (assuming the client supports cookies), the cookie is sent back by the client to the server as part of the request. In this case, the cookie value is used by the server to look for the session state information to be passed to the servlet.
- Finally, with subsequent responses, the container sends the updated cookie back to the client.

As the container handles the process of sending a cookie, the servlet code is not required while using cookies. A Web browser automatically handles the process of sending cookies back to the server unless the user disables cookies.

The cookie is used by the container to maintain a session. Cookies can be retrieved by a servlet by using the `getCookies()` method of the `HttpServletRequest` object. The cookie attributes can be examined by the servlet using the accessor methods of the `javax.servlet.http.Cookie` objects.

The servlet container sends a cookie to the client. Upon each HTTP request, the cookie is returned back to the server. This way, the session id indicated by the cookie is associated with the request. You should note that you can use HTTP cookies to store information about a session and for each subsequent connection the current session can be looked up and then information about that session is extracted from a location on the server machine. For example, in the following code snippet, the code to retrieve the session information is provided that can be implemented in a servlet:

```
String sesID = makeUniqueString();
Hashtable sesInfo = new Hashtable();
Hashtable hashtab = findTableStoringSessions();
hashtab.put(sesID, sesInfo);
Cookie sesCookie = new Cookie("JSESSIONID", sesID);
sesCookie.setPath("/");
```

```
response.addCookie(sesCookie);
```

The preceding code snippet requests the server, which uses the `hashtab` hash table to associate a session ID of the `JSESSIONID` cookie with the `sesInfo` hash table of data associated with that particular session. A cookie is the most widely used approach for session handling. The servlet's session tracking API handles sessions and performs the following tasks:

- Extracting the cookie that stores other cookie's session identifier
- Setting an appropriate expiration time for the cookie
- Associating hash tables with each request
- Generating a unique session identifier

Now, let's discuss the Cookies API as it would help you to understand about the classes and interfaces used for session tracking.

In Java Servlet API, `Cookie` is a class of the `javax.servlet.http` package, which abstracts the notion of a cookie. You can implement session tracking using cookies with the help of the `addCookie()` and `getCookies()` methods. These methods are provided by the `javax.servlet.http.HttpServletRequest` and `javax.servlet.http.HttpServletResponse` interfaces and are used to add cookies to HTTP responses and to retrieve the cookies from HTTP requests, respectively. A cookie is abstracted by the `Cookie` class. The following code snippet shows a constructor that instantiates a cookie instance with the given name and value:

```
public Cookie(String name, String value)
```

The `Cookie` class provides various methods, such as `getValue()` and `setValue()`, which simplify working with cookies. For all the cookie parameters, the `Cookie` class provides getter and setter methods. The following code snippet shows the implementation of some of the methods of the `Cookie` class:

```
public String getValue()
public void setValue(String newValue)
```

The getter and setter methods can be used to access or set the value of a cookie. Similarly, there exist other methods that can be used to access or change other parameters, such as path and header of a cookie. The following code snippet shows the implementation of the `addCookie()` method provided by the `javax.servlet.http.HttpServletResponse` interface to set cookies:

```
public void addCookie(Cookie cookie)
```

To set multiple cookies, you can call this method as many times as you want. The following code snippet shows the implementation of the `getCookies()` method provided by the `javax.servlet.http.HttpServletRequest` interface to extract all cookies contained in the HTTP request:

```
public Cookie[] getCookies()
```

Now, you can create your servlet to track the activities of a user using cookies by providing the code for the following actions:

- Check if there is a cookie contained in the incoming request
- Create a cookie and send it with the response, if there is no cookie in the incoming request
- Display the value of the cookies, if there is a cookie in the incoming request

Let's create a new Web application, `HandleSession`, to implement session tracking using cookies. This application follows the same directory structure as discussed in *Chapter 2, Web Applications and Java EE 6*. Let's now create the `CookieServlet` servlet class (in the `HandleSession` application) to show how to work with a cookie. Listing 5.1 shows the code of the `CookieServlet.java` file (you can find this file on the CD in the `code\JavaEE\Chapter5\HandleSession\src\com\kogent` folder):

**Listing 5.1:** Showing the Source Code of the `CookieServlet` Servlet Class

```
package com.kogent;
import java.io.*;
import java.util.Random;
import javax.servlet.http.*;
import javax.servlet.ServletException;

public class CookieServlet extends HttpServlet
{
```

```

protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
{
    Cookie[] coki = request.getCookies();
    Cookie tokenCookie = null;
    if(coki !=null)
    {
        for(int i=0; i<coki.length; i++)
        {
            if (coki[i].getName().equals("token"))
            {
                tokenCookie = coki[i];
                break;
            }
        }
    }

    response.setContentType("text/html");
    PrintWriter prnwriter = response.getWriter();
    prnwriter.println("<html><head><title>Extracting the token cookie</title></head><body>");
    prnwriter.println("style=\"font-family:arial;font-size:12pt\"");

    String resetParam = request.getParameter("resetParam");
    if(tokenCookie==null || (resetParam != null &&
        resetParam.equals("yes")))
    {
        Random rnd = new Random();
        long cookieid = rnd.nextLong();
        prnwriter.println("<p>Welcome. A new token " + cookieid + " is now established</p>");
        tokenCookie = new Cookie("token",
            Long.toString(cookieid));
        tokenCookie.setComment("A cookie named token to identify user");
        tokenCookie.setMaxAge(-1);
        tokenCookie.setPath("/HandleSession/CookieServlet");
        response.addCookie(tokenCookie);
    }
    else
    {
        prnwriter.println("Welcome back.. Your token is " +
            tokenCookie.getValue() + ".</p>");
    }

    String requestURLSame = request.getRequestURL().toString();
    String requestURLNew = request.getRequestURL() + "?resetParam=yes";

    prnwriter.println("<p>Click <a href=" + requestURLSame + " > here </a>again to continue browsing with the "+" same identity.</p>");
    prnwriter.println("<p>Otherwise, click <a href = " + requestURLNew +
        "> here</a> to start browsing with a new identity. </p>");
    prnwriter.println("</body></html>");
    prnwriter.close();
}
}
}

```

In Listing 5.1, the `CookieServlet` servlet class first retrieves all the cookies that are contained in the `request` object. In case the cookies are present, the `CookieServlet` servlet class locates the cookie named `token`. If the servlet class does not find a cookie with the name `token`, then the Web container, on the `CookieServlet` servlet class `request`, creates a cookie called `token` and adds it to the response. A random number for the cookie is created by the servlet class. The servlet class creates a cookie with the help of the parameters, as shown in the following code snippet:

```
Name : token
Value : A random number
Comment : A cookie named tokens to identify user
Max-Age : -1 (The value of -1 indicates that the cookie will be discarded when the
Browser exits)
Path : /HandleSession/CookieServlet (This path enables the browser to send the
cookie to only those requests under
http://localhost:8080/HandleSession/CookieServlet)
```

If you are deploying this application on a remote machine (server) and accessing it from another machine (client), you need to set the domain name to be that of the server name; by default, it is localhost. Compile the CookieServlet.java file using the following command:

```
javac -d C:\JavaEE\chapter5\HandleSession\WEB-INF\classes CookieServlet.java
```

The execution of the preceding command would compile the CookieServlet servlet class and store the .class file along with package directory under the WEB-INF\classes directory of the HandleSession application.

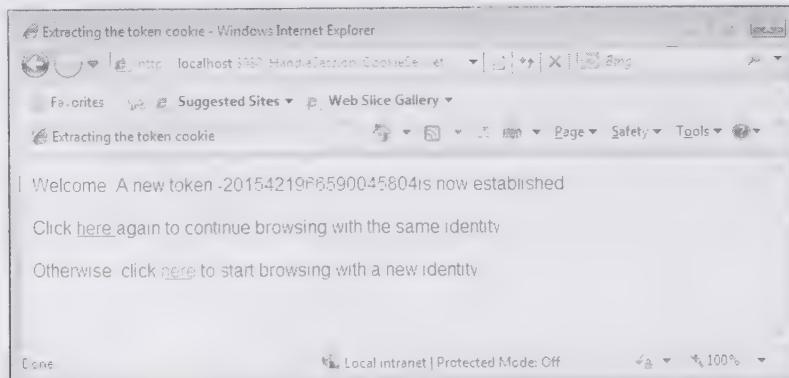
Now, let's create the web.xml file to configure and map the CookieServlet servlet class to the /CookieServlet url pattern. Listing 5.2 shows the code of the web.xml file (you can find this file on the CD in the code\JavaEE\Chapter5\HandleSession\WEB-INF folder):

#### **Listing 5.2: Configuring the CookieServlet Servlet Class**

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">

  <servlet>
    <servlet-name>CookieServlet</servlet-name>
    <servlet-class>com.kogent.CookieServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>CookieServlet</servlet-name>
    <url-pattern>/CookieServlet</url-pattern>
  </servlet-mapping>
</web-app>
```

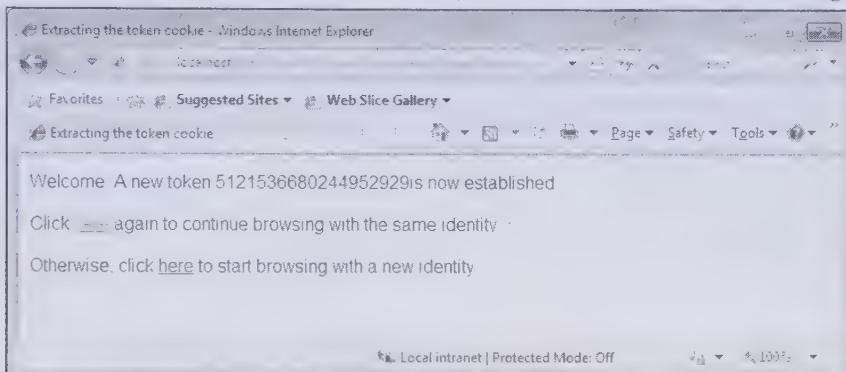
Save the code of Listing 5.2 as the web.xml file under the WEB-INF directory of the HandleSession Web application. Now, package the Web application into HandleSession.war, as discussed in *Chapter 4, Working with Servlets 3.0*, and deploy it on the Glassfish server. Now, run the servlet in a browser by browsing the <http://localhost:8080/HandleSession/CookieServlet> URL. Figure 5.1 displays the output of CookieServlet.java, showing the new token value for the newly created cookie named *token*:



**Figure 5.1: Displaying the Output of CookieServlet Servlet Class**

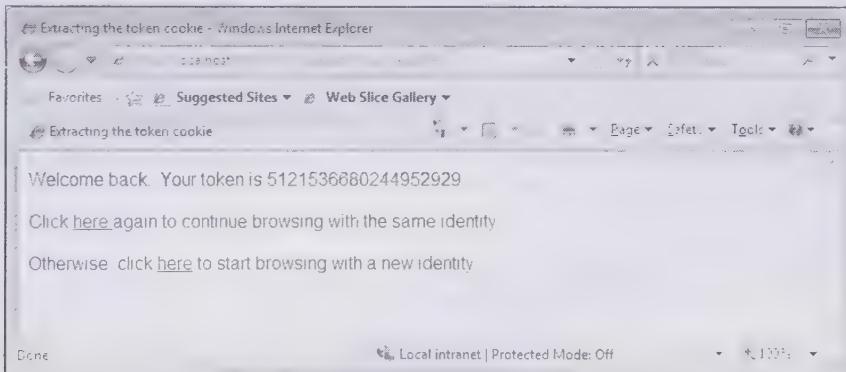
In Figure 5.1, there are two hyperlinks. Click the first hyperlink to browse through the same identity of the cookie. However, to browse with the new identity, click the second hyperlink. When you click the second

hyperlink, a new identity is created, with the new cookie value and it is displayed on the browser. After clicking the second hyperlink, the resetParam parameter is set to yes, as shown in the address bar in Figure 5.2:



**Figure 5.2: Browsing CookieServlet with New Token Value**

Figure 5.2 displays the new token value. Now, to continue with the same token value, as shown in Figure 5.2, click the first hyperlink. After clicking the first hyperlink, the same request URL is retrieved and the browser displays the Welcome back message, as shown in Figure 5.3:



**Figure 5.3: Browsing CookieServlet with the Same Identity**

After discussing how to use cookies, let's now discuss another approach used for session tracking, the hidden form fields.

## Using Hidden Form Fields

The hidden form fields are the fields in a Hypertext Markup Language (HTML) or JavaServer Pages (JSP) form that are not shown to the user and used to store information about a session. You can use the following syntax to use hidden form fields in an HTML page:

```
<INPUT TYPE="HIDDEN" NAME="session" VALUE="...">>
```

In the preceding syntax, the `hidden` value is assigned to the `type` attribute, which implies that the input field is a hidden form field. However, using hidden form fields has a major disadvantage, that is, hidden form fields only work when every page is dynamically generated by a form submission. Therefore, hidden form fields cannot support general session tracking and can only support tracking within a specific series of operations.

## Implementing URL Rewriting

The mechanism of URL rewriting is similar to that of cookies. The URL rewriting mechanism uses the `encodeURL()` method of the response object to encode the session ID into the URL path of a request. The following code snippet shows an example of URL rewriting in which the name of the path parameter is `jsessionid`:

<http://host:port/myapp/index.html?jsessionid=6789>

The server uses the value of the rewritten URL to find the session state information, and to pass the information to the servlet. This is similar to the functionality of cookies. Although, cookies are typically enabled; however, to ensure session tracking using URL rewriting, the `encodeURL()` method is used in the servlets. In addition, the `encodeRedirectURL()` method is used in servlets to redirect to a resource.

According to the Servlet specification, if cookies are enabled, then any call to the `encodeURL()` and `encodeRedirectURL()` methods does not result in any action. In case the cookies are disabled, the servlet can call the `encodeURL()` method of the response object to append a session ID to the URL path for each request. Alternatively, the `encodeRedirectURL()` method is used to redirect a Web page to a resource. As a result, URL rewriting helps in associating the request with the session. URL rewriting is the most commonly used mechanism for session tracking in cases when clients do not accept cookies.

Instead of embedding session information within the forms by using the hidden form fields, URL rewriting stores session details as a part of the URL itself. The following code snippet shows various ways in which the information for a servlet can be requested by using URL rewriting:

```
[1]     http://www.acknowledge.co.uk/Servlet/search
[2]     http://www.acknowledge.co.uk/Servlet/search/23434abc
[3]     http://www.acknowledge.co.uk/Servlet/search?sesID=23434abc
```

In [1], URL rewriting has not been done rather the URL requests for the `search` servlet mapped to `/search` in `web.xml`. In [2], URL has been rewritten at the server to add extra path information. This extra information is embedded in the pages returned to the client. The following code snippet shows how to retrieve the extra path information, provided in [2]:

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
{
    ...
    String sesID = request.getPathInfo(); // return 23434abc from [2]
    ...
}
```

The extra path information works for both the GET and the POST methods in forms and outside the forms with static links. While using technique [3], you can simply rewrite the URL with parameter information, as shown in the following code snippet:

```
request.getParameterValue("sesID");
```

Similar to hidden form fields, URL rewriting provides a means to implement anonymous session tracking. URL rewriting is not limited to forms only. URLs can also be rewritten in static documents to contain the required session information. However, URL rewriting suffers from a disadvantage that the URLs must be dynamically generated and most importantly, the chain of HTML page generation cannot be broken. Therefore, URL rewriting is a tedious and error-prone process.

The mechanism of URL rewriting can be better understood by creating a servlet. Let's create the `TokenServlet` servlet class, which performs the following tasks:

- Checks whether or not the client sends a token with its request
- Creates a new token, if no token has been sent by the client

In addition, the `TokenServlet` servlet class provides two hyperlinks—one that includes the token and other does not.

Listing 5.3 shows the code of the `TokenServlet.java` file (you can find this file on the CD in the `code\JavaEE\Chapter5\HandleSession\src\com\kogent` folder):

#### Listing 5.3: Implementing the URL Rewriting Mechanism

```
package com.kogent;

import java.io.*;
import java.util.Random;
import javax.servlet.http.*;
import javax.servlet.ServletException;

public class TokenServlet extends HttpServlet
{
```

```

protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
{
    String tokensID = request.getParameter("tokens");
    response.setContentType("text/html");
    PrintWriter prnwriter = response.getWriter();
    prnwriter.println("<html><head><title>Tokens</title></head><body>");
    prnwriter.println("style=\"font-family:verdana;font-size:10pt\"");
    if(tokensID==null)
    {
        Random rnd = new Random();
        tokensID = Long.toString(rnd.nextLong());
        prnwriter.println("<p>Welcome. A new token " + tokensID +
        " is now established</p>");
    }
    else
    {
        prnwriter.println("welcome back.. Your token is " + tokensID +
        "</p>");
    }

    String requestURLSame = request.getRequestURL().toString()+"?tokens=" +
    " " + tokensID;
    String requestURLNew = request.getRequestURL().toString();

    prnwriter.println("<p>Click <a href=" + requestURLSame + " > here
    </a> again to continue browsing with the same identity.</p>");
    prnwriter.println("<p>Click <a href=" + requestURLNew + " > here </a>
    to continue browsing with a new identity.</p>");
    prnwriter.println("</body></html>");
    prnwriter.close();
}
}
}

```

Save the code of Listing 5.3 as the TokenServlet.java file in the src\com\kogent directory of the HandleSession Web application. When you click the here hyperlink in the TokenServlet servlet class, the URL path is retrieved using the getRequestURL() method and the URL is rewritten with the parameter information. Compile the TokenServlet servlet class and configure it in the web.xml file, created in Listing 5.2 for the HandleSession Web application. The following code snippet shows how to configure and map the TokenServlet servlet class to the /TokenServlet url pattern:

```

<servlet>
    <servlet-name>TokenServlet</servlet-name>
    <servlet-class>com.kogent.TokenServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>TokenServlet</servlet-name>
    <url-pattern>/TokenServlet</url-pattern>
</servlet-mapping>

```

You can run the TokenServlet servlet class only after packaging the resources of the HandleSession Web application in the HandleSession.war file. Then, you need to redeploy the Web application on the Glassfish server and navigate the http://localhost:8080/HandleSession/TokenServletURL. After navigating this URL, you can notice that the query parameter, tokens, is not included in the initial request. The servlet not only creates a new token but also generates two links. A query string is included in one link, which is passed as a query parameter in the request. Therefore, the servlet can recognize the user from this parameter and display the Welcome back message. This allows the user to continue browsing with the same identity. Another link, that does not include the query parameter, is also generated by the servlet. This link allows the user to continue browsing with a new identity, every time the link is clicked. Figure 5.4 shows the output of the TokenServlet servlet class when it is executed for the first time:

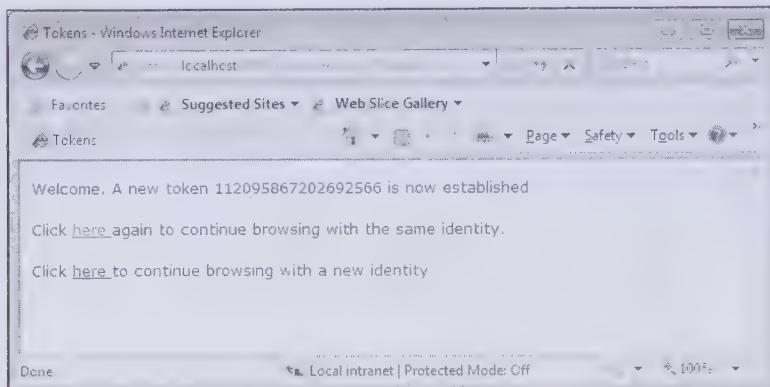


Figure 5.4: Displaying the Output of TokenServlet Servlet Class

After clicking the first link shown in Figure 5.4, the address bar of the browser (Figure 5.5) shows rewritten URL including the tokens parameter in the request. The tokens parameter added in the URL contains the new token identity based on the user's requests. Figure 5.5 shows the output when the first link is clicked:

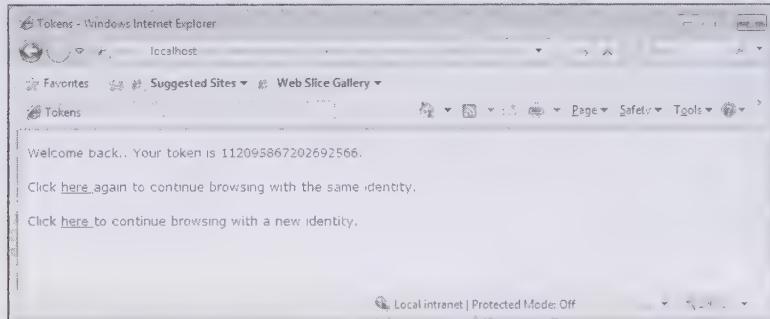


Figure 5.5: Displaying the Output of TokenServlet Servlet Class Using URL Rewriting

Clicking the second link allows the user to continue browsing with a new identity. Figure 5.6 shows the output when the second link is clicked:

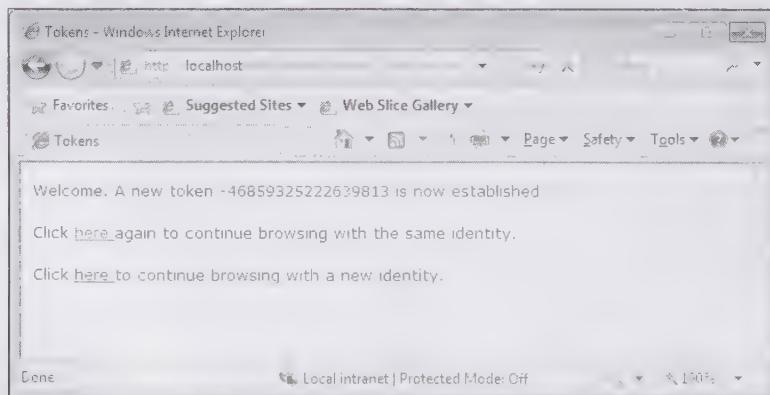


Figure 5.6: Displaying the Output of Browsing the TokenServlet Servlet Class with a New Identity

Although, the URL rewriting mechanism can solve the problem of session tracking, it has two main limitations:

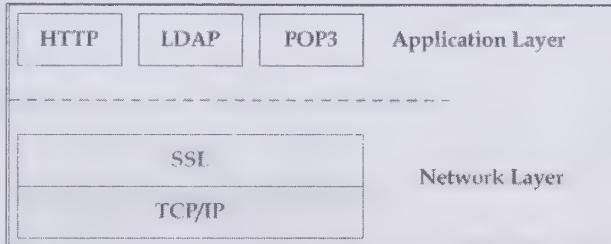
- It is not a secure session tracking mechanism as the token or session id is visible in the URL during a session.

- It can only be used with servlets or other dynamic pages as the links in static pages are hardcoded and cannot change dynamically for every user

You can use cookies as a session tracking mechanism to overcome the preceding problems of the URL rewriting mechanism.

## Using Secure Socket Layer

SSL is used to protect the data during transmission that covers all network services. This layer uses Transmission Control Protocol (TCP)/Internet Protocol (IP) to support typical application tasks that require communication between clients and servers. It is an encryption technology that runs on top of TCP/IP and below application level protocols, such as HTTP. SSL ensures the security of data transported and routed through HTTP. SSL is designed to utilize TCP as a communication layer protocol to provide a dependable, uninterrupted, secure, and authenticated connection between two points over a network. It is used mostly in an HTTP server and client applications. Almost each available HTTP server can support an SSL session. Figure 5.7 shows SSL between application protocols and TCP/IP:



**Figure 5.7: Displaying SSL between Application Protocols and TCP/IP**

Apart from the cookies, hidden form fields, URL rewriting, and SSL mechanisms, the session details of a user can also be tracked by using Java Servlet API. The `HttpSession` interface in Java Servlet API also helps in implementing session tracking. Let's discuss the `HttpSession` interface in the next section and learn how to use it to implement session tracking.

## Using the Java Servlet API for Session Tracking

- Session tracking is a mechanism used to maintain the state of a user within the lifetime of a session. In other words, session tracking is a means to keep track of session data, which represents the data being transferred in a session. As the HTTP protocol is a stateless protocol, the Web container needs to manage the session data in a Web application.

Session tracking is used when session data of a user might be required by a Web server to complete specific operations in the current session. For example, suppose you are shopping on an online book store. You access the online book store Web site and add items to the shopping cart. When you proceed for checkout, then due to session tracking the server would be known that the user who added items to the shopping cart is logging out. The following subsections briefly discuss about history of session tracking and provide a description about how to create and track a session.

## History of Session Tracking

Developers maintain the session state by providing user information into hidden form fields on an HTML page. In addition, they can maintain user's session by embedding the user activities into URLs with a long String of appended characters. You can find good examples of embedding user activities into URLs mostly in search engine sites, which still depend on CGI. These URLs contain the URL parameter name/value pairs that are appended after the reserved HTTP character ?. The search engine sites use the URL parameter name/value pairs to track the user choices. However, appending URL parameters can result in a very long URL that needs to be carefully parsed and managed by the CGI script. The URL parameter name/value pairs cannot be passed through URL from session to session. Once the control over the URL is lost, that is once the user leaves one of the pages, the user information is lost forever.

Later, browser cookies were introduced by Netscape, which can be used by each server to store user related information on the client side. However, one of the drawbacks of using cookies is that cookies are not fully supported by some browsers and most browsers limit the amount of data that can be stored with a cookie.

To overcome the shortcomings faced while using cookies and to maintain the user session, the HTTP Servlet specification was introduced. The HTTP Servlet specification protects the code from the complexities of tracking sessions. Servlets may use the HttpSession object to track the input provided by the user over the span of a single session as well as to share the session details with other servlets.

## Session Creation and Tracking

An instance of a class that implements the javax.servlet.http.HttpSession interface represents each client session in the standard Servlet API. The servlets can use the HttpSession object to set or get the information about the session which must be of the application-level scope. A servlet can retrieve or create the HttpSession object for the user by calling the getSession() method. The getSession() method accepts a boolean argument that specifies whether to create a new session object for the client if no session already exists within the application. Let's now learn how to create a session.

### Creating a Session

A prospective session that has not yet been established is considered new. As HTTP protocol is request-response based; therefore, the HTTP session is considered as new until it is joined by a client. The client is considered to have joined the session when session tracking information is returned to the server indicating that the session has been successfully established. Any next request from the client is not recognized as a part of the session until the client joins the session. A session is considered to be new, in either of the following cases:

- The client does not have knowledge about the session
- The client opts for not joining the session

In both the cases, the servlet container could not correlate a request with the previous request by any means. Therefore, a servlet developer must design the application in such a manner that it could handle a situation where a client has not yet joined a session or will not join a session.

As explained earlier, the servlet container uses the HTTP session objects that implement the javax.servlet.http.HttpSession interface to track and manage the user sessions. The HttpSession interface contains public methods, such as setAttribute() and getAttribute(), to set as well as get the session information, respectively. The following code snippet shows the implementation of the setAttribute() method of the HttpSession interface:

```
void setAttribute(String name, Object value)
```

The setAttribute() method binds the specified object under the specified name, to the session. The following code snippet shows the implementation of the getAttribute() method of the HttpSession interface:

```
Object getAttribute(String name)
```

The getAttribute() method retrieves the object that is attached to the session with the specified name. A null value is returned if there is no match. According to the configuration of a servlet as well as the servlet container, sessions may automatically expire after the specified time or the servlet may invalidate the session explicitly. The following methods can be used by servlets to manage the session life-cycle specified by the HttpSession interface:

- void invalidate() – Invalidates the session instantly and unbinds any bound objects from the session.
- void setMaxInactiveInterval(int interval) – Sets a session timeout interval as an integer value in seconds. Timeout cannot be indicated by a negative value. A value of 0 results in immediate timeout.
- boolean isNew() – Returns true if a new session is created within the request that creates a new session; otherwise, it returns false.
- long getCreationTime() – Returns the time of creation of the session object. The time is measured in milliseconds and is calculated since midnight, January 1, 1970.

- long getLastAccessedTime() – Returns the time associated with the most recent request made by the client during the session. The time is measured in milliseconds and is calculated since midnight, January 1, 1970. Session creation time is returned by the method if the client session has not yet been accessed.

Let's create a servlet to understand the implementation of the HTTP session object. Listing 5.4 provides the code of the MySessionServlet servlet class, which creates the HttpSession object and prints the data held by the request and session objects (you can find the MySessionServlet.java file on the CD in the code\JavaEE\Chapter5\HandleSession\src\com\kogent folder):

**Listing 5.4:** Showing the Code of the MySessionServlet.java File

```
package com.kogent ;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.Date;

public class MySessionServlet extends HttpServlet
{
    public void doGet (HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException
    {
        // Retrieve the session object for the current user session.
        // A new session is created if a session object has not been
        // created previously.
        HttpSession sessionObj = req.getSession(true);

        // Create and update the variable that holds a count indicating
        // the number of times the page has been visited during the current
        // user session.
        Integer count = (Integer) sessionObj.getAttribute("count");

        if (count == null)
        {
            count = new Integer(1);
        }
        else
        {
            count = new Integer(count.intValue() + 1);
        }

        // Save the updated count value to the current user session
        sessionObj.setAttribute("count", count);

        // Displaying the output to the user
        res.setContentType("text/html");
        PrintWriter prnwriter = res.getWriter();

        prnwriter.println("<head><title> " + "Displaying the Details of
        Current User Session" + "</title></head><body>");
        prnwriter.println("<h1>Displaying the Details of Current User
        Session</h1>");
        prnwriter.println("<h2><I>You have visited this page</I><b><font
        color=\"blue\">" + count + "</font></b><I> times.</I></h2><p>");

        prnwriter.println("<BR>");

        // Displaying the request related information from the request
        // object
        prnwriter.println("<Table ALIGN=CENTER Border=\"1\""
        "BorderColor=\"$Red$\">");
        prnwriter.println("<TH COLSPAN=2 ALIGN= CENTER>Summary of Request
        Data</TH>");

        prnwriter.println("<TR><TD WIDTH=\"50%\" ALIGN=LEFT
        VALIGN= CENTER>");
        prnwriter.println("Session ID in Request Object");
    }
}
```

```

prnwriter.println("</TD>");
prnwriter.println("<TD WIDTH=\"50%\">");
prnwriter.println(req.getRequestedSessionId());
prnwriter.println("</TD></TR>");

prnwriter.println("<TR><TD WIDTH=\"50%\" ALIGN=LEFT
VALIGN=CENTER>");
prnwriter.println("Is Session ID in Request from a Cookie");
prnwriter.println("</TD>");
prnwriter.println("<TD WIDTH=\"50%\">");
prnwriter.println(req.isRequestedSessionIdFromCookie());
prnwriter.println("</TD></TR>");

prnwriter.println("<TR><TD WIDTH=\"50%\" ALIGN=LEFT
VALIGN=CENTER>");
prnwriter.println("Is Session ID in Request from the URL");
prnwriter.println("</TD>");
prnwriter.println("<TD WIDTH=\"50%\">");
prnwriter.println(req.isRequestedSessionIdFromURL());
prnwriter.println("</TD></TR>");

prnwriter.println("<TR><TD WIDTH=\"50%\" ALIGN=LEFT
VALIGN=CENTER>");
prnwriter.println("Is Requested Session ID Valid");
prnwriter.println("</TD>");
prnwriter.println("<TD WIDTH=\"50%\">");
prnwriter.println(req.isRequestedSessionIdValid());
prnwriter.println("</TD></TR>");

prnwriter.println("</Table>");

prnwriter.println("<BR><BR>");

// Displaying the session related information from the session
// object
prnwriter.println("<Table ALIGN= CENTER Border=\"1\""
BorderColor="Red\"");
prnwriter.println("<TH COLSPAN=2 ALIGN= CENTER>Summary of Session
Data</TH>");

prnwriter.println("<TR><TD WIDTH=\"50%\" ALIGN=LEFT
VALIGN=CENTER>");
prnwriter.println("Is it a New Session");
prnwriter.println("</TD>");
prnwriter.println("<TD WIDTH=\"50%\">");
prnwriter.println(sessionobj.isNew());
prnwriter.println("</TD></TR>");

prnwriter.println("<TR><TD WIDTH=\"50%\" ALIGN=LEFT
VALIGN=CENTER>");
prnwriter.println("Session ID");
prnwriter.println("</TD>");
prnwriter.println("<TD WIDTH=\"50%\">");
prnwriter.println(sessionobj.getId());
prnwriter.println("</TD></TR>");

prnwriter.println("<TR><TD WIDTH=\"50%\" ALIGN=LEFT
VALIGN=CENTER>");
prnwriter.println("Session Creation Time");
prnwriter.println("</TD>");
prnwriter.println("<TD WIDTH=\"50%\">");
prnwriter.println(new Date(sessionobj.getCreationTime()));
prnwriter.println("</TD></TR>");

prnwriter.println("<TR><TD WIDTH=\"50%\" ALIGN=LEFT
VALIGN=CENTER>");
prnwriter.println("Time at which Session was Last Accessed");
prnwriter.println("</TD>");
prnwriter.println("<TD WIDTH=\"50%\">");
prnwriter.println(new Date(sessionObj.getLastAccessedTime()));
prnwriter.println("</TD></TR>");

prnwriter.println("</Table>");
```

```

prnwriter.println("<BR><BR>");

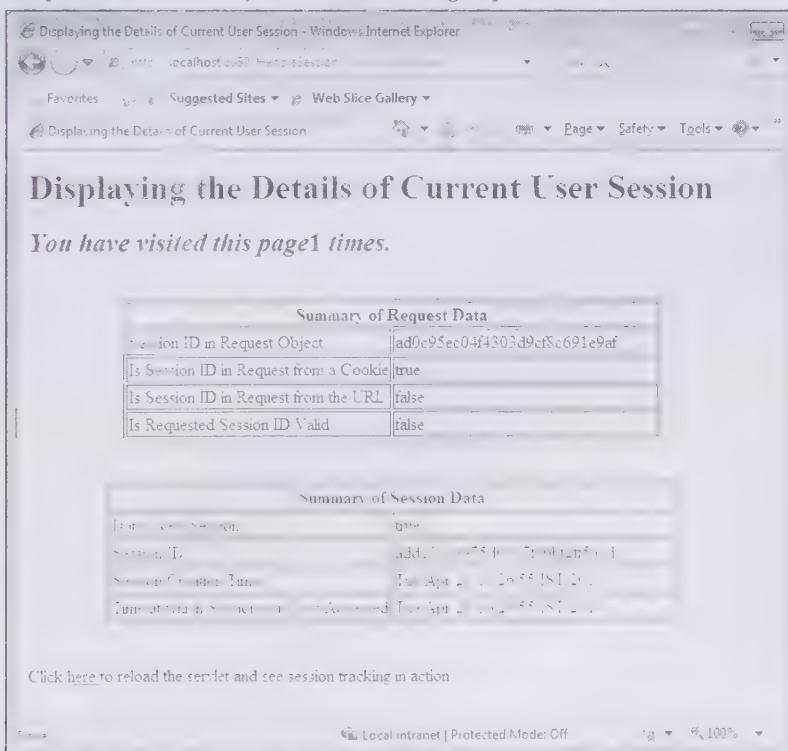
// User can reaccess the servlet using this hyperlink to see
// Session Tracking in action. Clicking the hyperlink refreshes
// the page and increment the session variable count by one every
// time it is clicked.
String url = req.getRequestURL().toString();
prnwriter.println(" Click <a href=" + url + "> here </a> to reload
the servlet and see session tracking in action. <br>");

prnwriter.println("</body>");
prnwriter.close();
}
}
}

```

Save MySessionServlet in the `src\com\kogent` directory of the HandleSession Web application. Now, configure and map the `MySessionServlet` servlet class to the `/MySessionServlet` url pattern in the `web.xml` file and then create `HandleSession.war`. Redeploy the HandleSession Web application on the Glassfish server and browse the URL `http://localhost:8080/HandleSession/MySessionServlet` to see the output of the `MySessionServlet` servlet class.

Figure 5.8 shows the output of the `MySessionServlet` servlet class, displaying the results of the accessor methods on the request and session objects as well as listing request and session data:



**Figure 5.8: Displaying the Output of MySessionServlet Servlet Class**

In Figure 5.8, if a user clicks the `here` link and the cookies are enabled in the browser settings, then the request and session data get changed. In such a case, change the Web browser settings, for example, disable cookies and click the `here` link that causes URL rewriting.

## Tracking a Session Using Servlet API

The Servlet API provides various methods and classes, such as `getSession()` and `HttpSession` that are particularly designed to handle session tracking.

The session tracking API, which is the part of the Servlet API and provides session tracking functionality, can be used in any Web server that supports servlets. However, the level of support depends on the server.

For example, session objects can be written to Java Web Server even if the server disk memory fills up or the server shuts down. However, to take advantage of this option; the items that are placed in the session are required to implement the `Serializable` interface.

### *Exploring Session Tracking Basics*

In session tracking, the user identity is linked with the `javax.servlet.http.HttpSession` object that can be used by the servlets to store or retrieve information about that user. Any set of arbitrary Java objects can be saved in a session object. For example, to store the user's shopping cart content in a database, a user's current `HttpSession` instance is retrieved by using the `getSession()` method of the `HttpServletRequest` interface, as shown in the following code snippet:

```
public HttpSession HttpServletRequest.getSession(boolean create)
```

The current session that is associated with the user who makes the request, is returned by the `getSession()` method. If the `createBoolean` variable has a value `true`, the `getSession()` method creates the session; otherwise, the method returns null. The `getSession()` method must be called at least once before writing any output to the response to ensure that the session is maintained properly.

Let's create another servlet class, `MySessionTrackerServlet`, to have a better understanding of the concept of session tracking. The `MySessionTrackerServlet` servlet class fetches and prints all the attributes associated with the current user session along with a count for the number of times the servlet has been visited by the user during the session. Listing 5.5 shows the code of the `MySessionTrackerServlet.java` file (you can find this file on the CD in the `code\JavaEE\Chapter5\HandleSession\src\com\kogent` folder):

**Listing 5.5:** Showing the Code of the `MySessionTrackerServlet.java` File

```
package com.kogent ;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

public class MySessionTrackerServlet extends HttpServlet
{
    public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException
    {
        // Retrieve the session object for the current user session.
        // A new session is created if a session object has not been
        // created previously.
        HttpSession sessionObj = req.getSession(true);

        // Create and update the variable that holds a count indicating
        // the number of times the page has been visited during the current
        // user session.
        Integer count = (Integer) sessionObj.getAttribute("Count");

        if (count == null)
        {
            count = new Integer(1);
        }
        else
        {
            count = new Integer(count.intValue() + 1);
        }

        // Save the updated count value to the current user session
        sessionObj.setAttribute("Count", count);
        // Add some more attributes to the current user session
        sessionObj.setAttribute("UserName", "Pallavi");
        sessionObj.setAttribute("UserID", sessionObj.getId());
        sessionObj.setAttribute("MyFavouriteColor", "Red");
        // Displaying the output to the user
        res.setContentType("text/html");
        PrintWriter prnwriter = res.getWriter();
    }
}
```

```

prnwriter.println("<HTML><HEAD><TITLE>My Session Tracker
Servlet</TITLE></HEAD>");
prnwriter.println("<BODY><H1>Demonstrating Session Tracking</H1>");

// Display the hit cnt for this page for the current user
prnwriter.println("You have visited this page <b><font
color=\"blue\">" + count + ((count.intValue() == 1) ? "</font><b>" +
time. : "</font></b> times.")); 
prnwriter.println("<P>");

prnwriter.println("<H2><I>Displaying Session Data</I></H2>");
prnwriter.println("<Table Border=\"1\" BorderColor=\"Blue\">");

Enumeration names = sessionObj.getAttributeNames();
while(names.hasMoreElements())
{
    String name = (String) names.nextElement();
    prnwriter.println("<TR><TD WIDTH=\"50%\" ALIGN=LEFT
VALIGN=CENTER><I>"); 
    prnwriter.println(name);
    prnwriter.println("</I></TD><TD>"); 
    prnwriter.println(sessionObj.getAttribute(name));
    prnwriter.println("</TD></TR>"); 
}
prnwriter.println("</TABLE>"); 
prnwriter.println("</BODY></HTML>"); 
}
}

```

The MySessionTrackerServlet servlet class tracks the number of times the client accesses it. When the client accesses the MySessionTrackerServlet servlet class for the first time, the new session is created and each time the client accesses it, the value of count variable is incremented by one and the browser displays the number of times the client has accessed the servlet. The other variables related to the current client session are also displayed by the servlet.

Save the MySessionTrackerServlet servlet class in the src\com\kogent directory of the HandleSession Web application. Configure the MySessionTrackerServlet servlet class in the web.xml file. Then, compile the MySessionTrackerServlet servlet class, package the HandleSession application, and redeploy the Web application on the Glassfish server. Browse the http://localhost:8080/HandleSession/MySessionTrackerServlet URL to view the output, as shown in Figure 5.9:

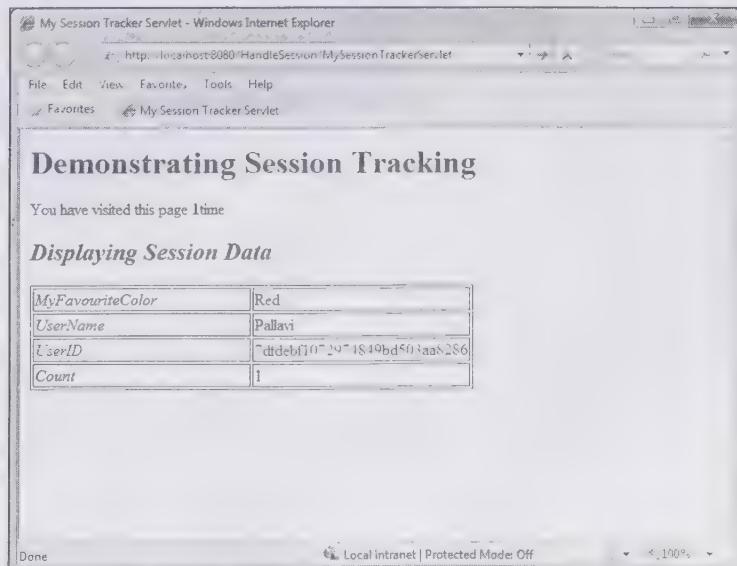


Figure 5.9: Displaying the Output of the MySessionTrackerServlet Servlet Class

The `MySessionTrackerServlet` servlet class first retrieves the `HttpSession` object linked with the current client session. The `getSession()` method, with a Boolean value of `true` as an argument, creates a session if it does not exist for the user. The servlet then fetches and sets the value of the `Integer` object `count`, which is bound to the current user session with a name `Count`. The servlet initializes the `Count` variable, if it is previously null. Otherwise, the servlet increments the value of the `Count` variable by one and resets it for the user session. The servlet also fetches the values of other variables associated with the current user session. Finally, the servlet displays the value for the `count` variable and all the name/value pairs for the variables associated with the current user session.

### Demonstrating Session Life-Cycle with Cookies

Let's now explore the life-cycle of `HttpSession` objects, by creating a servlet class, `TrackSessionLifeCycle`. The servlet examines certain session attributes and provides link to invalidate the existing session. We will also examine the behavior of the servlet in the absence of cookies, and then discuss the ways to generate Web pages that work as desired, irrespective of whether cookies are accepted by the client browser or not.

The `TrackSessionLifeCycle` servlet class generates a page that displays session object data including session ID, creation time of the session object, last accessed time, and max inactive interval for the session. The `TrackSessionLifeCycle` servlet class also provides links to reload the page and invalidate the current user session. Listing 5.6 shows the code of the `TrackSessionLifeCycle.java` file (you can find this file on the CD in the `code\JavaEE\Chapter5\HandleSession\src\com\kogent` folder):

**Listing 5.6:** Displaying the Code of the `TrackSessionLifeCycle` Servlet Class

```
package com.kogent ;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class TrackSessionLifeCycle extends HttpServlet
{
    protected void doGet (HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException
    {
        String url = "/HandleSession/TrackSessionLifeCycle";
        String reqAction = req.getParameter("requestAction");

        HttpSession sessionObj = req.getSession();

        res.setContentType("text/html");
        PrintWriter prnwriter = res.getWriter();
        prnwriter.println("<html>");
        prnwriter.println("<head><title>Demonstrating Session
LifeCycle</title></head>");
        prnwriter.println("<body>");

        if (reqAction != null && reqAction.equals("invalidate"))
        {
            sessionObj.invalidate();
            prnwriter.println("<center><p>Your session has been
invalidated.</p>"); 
            prnwriter.println("Would you like to <a href=\"" + url +
"?requestAction=createNewSession\">"); 
            prnwriter.println("create a new session</a>"); 
        }
        else
        {
            prnwriter.println("<h1><center>Tracking Session Life
Cycle</center></h1>"); 
            prnwriter.println("<br><Table ALIGN= CENTER Border=\"1\">");
```

```

        BorderColor=\"Blue\>");

prnwriter.println("<TR><TD WIDTH=\"50%\" ALIGN=LEFT><I>");
prnwriter.println("Session Status");
prnwriter.println("</I></TD><TD>");
if (sessionObj.isNew())
{
    prnwriter.println("New Session");
}
else
{
    prnwriter.println("Old Session");
}
prnwriter.println("</TD></TR>");

prnwriter.println("<TR><TD WIDTH=\"50%\" ALIGN=LEFT><I>");
prnwriter.println("Session ID:");
prnwriter.println("</I></TD><TD>");
prnwriter.println(sessionObj.getId());
prnwriter.println("</TD></TR>");

prnwriter.println("<TR><TD WIDTH=\"50%\" ALIGN=LEFT><I>");
prnwriter.println("Creation Time:");
prnwriter.println("</I></TD><TD>");
prnwriter.println(new Date(sessionObj.getCreationTime()));
prnwriter.println("</TD></TR>");

prnwriter.println("<TR><TD WIDTH=\"50%\" ALIGN=LEFT><I>");
prnwriter.println("Last Accessed Time:");
prnwriter.println("</I></TD><TD>");
prnwriter.println(new
Date(sessionObj.getLastAccessedTime()));
prnwriter.println("</TD></TR>");

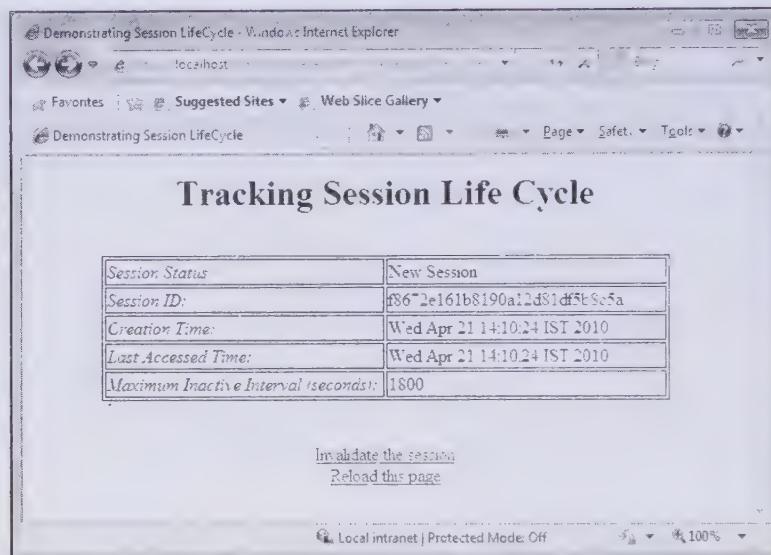
prnwriter.println("<TR><TD WIDTH=\"50%\" ALIGN=LEFT><I>");
prnwriter.println("Maximum Inactive Interval
(seconds):");
prnwriter.println("</I></TD><TD>");
prnwriter.println(sessionObj.getMaxInactiveInterval());
prnwriter.println("</TD></TR>");

prnwriter.println("</TABLE>");
prnwriter.println("<br><br><center><a href =\"" + url +
"?requestAction=invalidate\>\"");
prnwriter.println("Invalidate the session</a>");
prnwriter.println("<br><center><a href =\"" + url + "\>\"");
prnwriter.println("Reload this page</a>\"");
prnwriter.println("</body></html>");
prnwriter.close();
}
}
}

```

Save the code of Listing 5.6 as the `TrackSessionLifecycle.java` file in the `src\com\kogent` directory of the `HandleSession` application. Configure the `TrackSessionLifecycle` servlet class in the `web.xml` file and map it to the `/TrackSessionLifecycle` url pattern. After configuring, compile the `TrackSessionLifecycle` servlet class, package the application into `HandleSession.war`, and redeploy the `HandleSession` application. To run the servlet, browse `http://localhost:8080/HandleSession/TrackSessionLifecycle`.

Figure 5.10 displays the output of `TrackSessionLifecycle` showing the new session since the user is accessing this page for the first time:

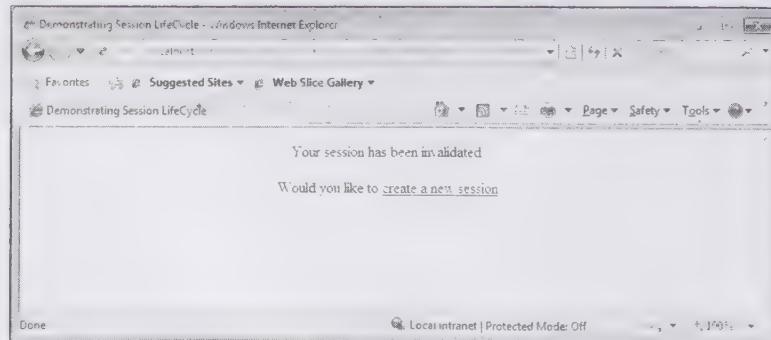


**Figure 5.10: Displaying the Output of TrackSessionLifeCycle Servlet Class**

The else block of the if...else block provided in Listing 5.6 is executed when the servlet is invoked without any parameters and performs the following steps:

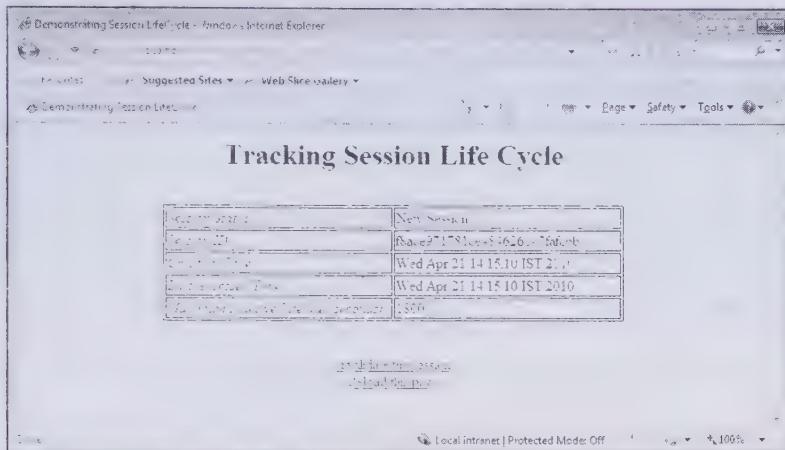
- ❑ Calls the `isNew()` method to check for the status of the session that whether it is new or old.
- ❑ Calls the `getSession()` method on the `HttpSession` object with the true Boolean value passed as an argument.
- ❑ Calls the `getId()` method to get the session ID.
- ❑ Calls the `getCreationTime()` method to get the creation time of the session. When this method returns the creation time as milliseconds since January 1, 1970 00:00:00 GMT, the returned value needs to be converted into a `Date` object by using the new constructor.
- ❑ Calls the `getLastAccessedTime()` method to get the time the session was last accessed.
- ❑ Calls the `getMaxInactiveInterval()` method to get the current max-inactive setting.

After printing the details, such as sessionID, creation time of the session, last accessed time of the session, the servlet generates two links, one to invalidate the session and the other to reload the page. The first link has query string as request Action=`invalidate` that is appended to the URL. When you click the `Invalidate the session` link, the if block of the `doGet()` method is executed. However, the second link simply points to the same page. Ensure that cookies are enabled in your browser configuration so that cookies can be stored on your system. Figure 5.11 displays the browser's output, when the `Invalidate the session` link is clicked:



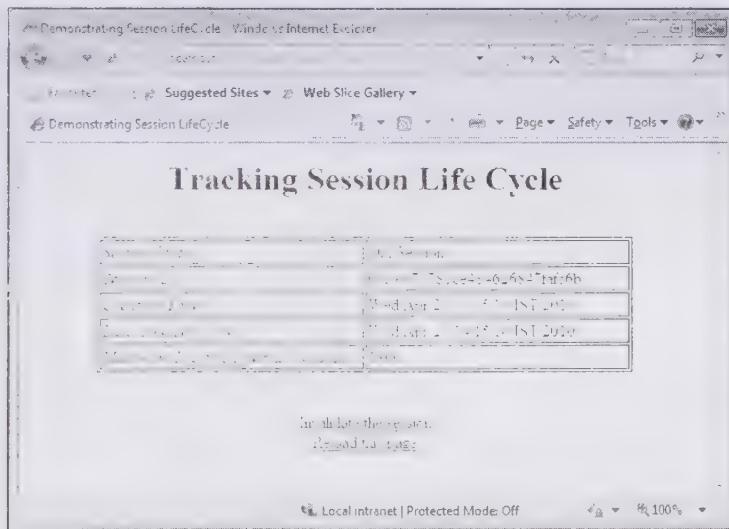
**Figure 5.11: Displaying a Message Depicting the Expiration of a Session**

Now, when you click the `create a new session` link, a new session is created, as shown in Figure 5.12:



**Figure 5.12: Creating a new session**

Now, let's see what happens if the code in the `else` block is executed again. Click the `Reload this page` link. You find that the session status displays that the session is old. Figure 5.13 displays the output when the `Reload this page` link is clicked:



**Figure 5.13: Reloading the Old Session**

You may notice in Figure 5.13 that the session ID and the session creation time are same as they were before reloading the page, as shown in Figure 5.12. Therefore, every time you click the `Reload this page` link, only the last accessed time changes and not the session ID and the session creation time.

This illustrates how simple it is to create and keep track of sessions. Now, let's see what happens if you click the `Invalidate the session` link. As this URL has a query parameter `action=invalidate`, the `if` block of the `doGet()` method will be executed to invalidate the created session.

Now, click the `Create a new session` link. The resulting page should be similar to Figure 5.12. Examine the `if` block of the code of Listing 5.6. This part of the `TrackSessionLifeCycle` servlet class gets the session from the request, calls the `invalidate()` method, and generates a new link back to the previous page.

Now let's create a login application using Session Tracking API. In this application, a user logs on with the username and password and after logging into the application, the user browses the session details. Finally, the user logs out and the session is terminated.

## Creating Login Application using Session Tracking

Let's create the login Web application, in which the user logs on with the username and password, and then creates a session. Till the session is active, the user can browse the session details; however, once the user logs out, the session becomes invalid.

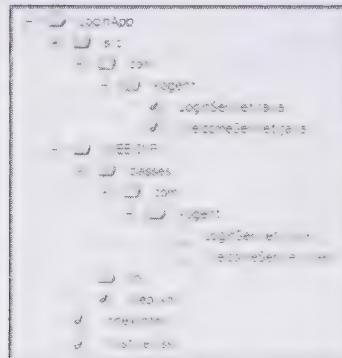
Let's name the Web application as `LoginApp` to demonstrate session tracking. This application has an HTML page and two servlets. These will be configured in the `web.xml` file. You should know the directory structure of the application before creating the application. Therefore, let's first discuss the directory structure of the `LoginApp` application. The directory structure of the `LoginApp` Web application would make it easy to understand where to save the Java, HTML, and configuration files.

### Exploring the Directory Structure of Login Application

The servlets, HTML pages, and the Cascading Style Sheet (CSS) files of the `LoginApp` application are stored under a base directory of the application. Create a folder for your application say, `LoginApp`, in the `C:\JavaEE\Chapter5` folder. Create some more folders, such as `WEB-INF`, `WEB-INF\classes`, `WEB-INF\lib`, and `src`(Figure 5.14). Store different types of files at the proper location in the directory structure, as described in the following statements:

- ❑ All packages containing class files are stored in the `WEB-INF\classes` folder.
- ❑ The configuration file, such as `web.xml`, is stored in the `WEB-INF` folder.
- ❑ All source files (.java files) can be stored in the `src\com\kogent` folder. This folder is optional in your application and you can store your source files at any other location.

Figure 5.14 displays the directory structure of the `LoginApp` application:



**Figure 5.14: Displaying the Root Directory Structure for LoginApp Web application**

As shown in Figure 5.14, `LoginApp` is the root folder containing the `WEB-INF` folder, `src` folder, and `index.html` file. The `WEB-INF` folder has two folders, `classes` and `lib`, and a file `web.xml`. As discussed earlier, the package containing `LoginServlet` and `WelcomeServlet` class files is stored at the `WEB-INF\classes` location under the `LoginApp` directory. The `src\com\kogent` is the optional folder containing source files (`LoginServlet.java` and `WelcomeServlet.java`). The configuration file, `web.xml`, is stored in the `WEB-INF` folder.

### Building the Front-End

The HTML page acts as a front-end of the `LoginApp` Web application. The `index.html` page displays a login page for the users. On the basis of the user details entered in `index.html`, a session is maintained. The user details include the username and password. Listing 5.7 shows the code of the `index.html` file (you can find this file on the CD in the `code\JavaEE\Chapter5\LoginApp` folder):

**Listing 5.7: Showing the Code of the Front-End**

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<HTML>
    <HEAD>
        <TITLE>Login Application</TITLE>
        <link rel="text/css" href="mystyle.css"/>
    </HEAD>
    <BODY>
        <FORM METHOD="POST" ACTION="LoginServlet">
            <H1>Login Application using Session Tracking</H1>
            <TABLE ALIGN=CENTER BORDER="0" >
                <TR>
                    <TD VALIGN=TOP ALIGN=RIGHT>
                        <B>User ID:</B>
                    </TD>
                    <TD VALIGN=TOP>
                        <B><INPUT NAME="username" TYPE="TEXT" MAXLENGTH= "20" SIZE = "20"></B>
                    </TD>
                </TR>
                <TR>
                    <TD VALIGN=TOP ALIGN=RIGHT>
                        <B>Password:</B>
                    </TD>
                    <TD VALIGN=TOP>
                        <B><INPUT NAME="password" TYPE="Password" MAXLENGTH="20" SIZE = "20"></B>
                    </TD>
                </TR>
                <TR><TD VALIGN=CENTER>
                    <B><INPUT VALUE = "Log In" TYPE= "SUBMIT"></B>
                </TD>
                </TR>
            </TABLE>
        </FORM>
    </BODY>
</HTML>
```

Save the code of Listing 5.7 as the `index.html` file in the base `LoginApp` directory and link the `mystyle.css` stylesheet to `index.html`. The style that you want to apply to your HTML page can be stored in the `mystyle.css` file, which is saved in the base `LoginApp` directory. The `codeofmystyle.css` is provided on the CD. After entering the login details, the `LoginServlet` servlet class is requested.

***Creating and Managing a Session***

In the `LoginApp` Web application, two servlet classes are created, `LoginServlet` and `WelcomeServlet`. The `LoginServlet` servlet class creates a new session for each user and retrieves the data from `index.html`. Based on the username and password entered by the user in `index.html`, the `LoginServlet` servlet class sets two new attributes, `username` and `password`, in the session. Listing 5.8 provides the code of the `LoginServlet.java` file (you can find this file on the CD in the `code\JavaEE\Chapter5\LoginApp\src\com\kogent` folder):

**Listing 5.8: Setting the Values of the username and password Attributes**

```
package com.kogent;
import javax.servlet.http.*;
import java.io.*;
public class LoginServlet extends HttpServlet
{
    public void doPost(HttpServletRequest request,HttpServletResponse response)
    {
        try {
            String username = request.getParameter("username");
            String password = request.getParameter("password");
            HttpSession session = request.getSession(true);
            PrintWriter writer = response.getWriter();
            if (session.isNew() != true)
            {
```

```
        writer.println("<h1>Session is Active</h1>");  
        writer.println("<p><a href=\"index.html\">HomePage" + "</a> and  
        return to login page");  
    }  
    else  
    {  
        session.setAttribute("username", username);  
        session.setAttribute("password", password);  
        response.setContentType("text/html");  
        writer.println("<html><body style=\"font-  
family:verdana;font-size:10pt\">");  
        writer.println("<h1>Login Application using Session  
Tracking</h1>");  
        writer.println("<p>Thank you, " + username + ".<p> You are now  
logged in");  
        String newURL = response.encodeURL("WelcomeServlet");  
        writer.println("Click <a href=\"" + newURL + "\">here</a> for  
another servlet");  
        writer.println("</body></html>");  
        writer.close();  
    }  
}  
catch (Exception e)  
{  
    e.printStackTrace();  
}  
}  
}  
}
```

The LoginServlet servlet class checks whether the session is new or not. If the session is not new, the user is prompted to go to the homepage as a session already exists. However, if the session is new, the HTML page is designed, which provides a link to the user to browse and the request is forwarded to WelcomeServlet. The WelcomeServlet servlet class displays the session details of the logged in user. Listing 5.9 provides the code of WelcomeServlet (you can find this file on the CD in the code\JavaEE\Chapter5\LoginApp\src\com\kogent folder):

**Listing 5.9:** Showing the Code of the WelcomeServlet Servlet Class

```
package com.kogent;  
import javax.servlet.http.*;  
import java.io.*;  
import java.util.*;  
public class WelcomeServlet extends HttpServlet  
{  
    public void doGet(HttpServletRequest request, HttpServletResponse response)  
{  
        HttpSession session = request.getSession();  
        try {  
            PrintWriter writer = response.getWriter();  
            if (session == null || session.isNew())  
            {  
                writer.println("You are not logged in");  
            }  
            else  
            {  
                response.setContentType("text/html");  
                writer.println("<html><body style=\"font-  
family:verdana;font-size:10pt\">");  
                writer.println("<h1>Login Application using Session  
Tracking</h1>");  
                writer.println("<p>Thank you, you are already logged in");  
                writer.println("<p>Here is the data in your session");  
                Enumeration names = session.getAttributeNames();  
                while (names.hasMoreElements())  
                {  
                    writer.println(names.nextElement());  
                }  
            }  
        }  
    }  
}
```

```

        String name = (String) names.nextElement();
        Object value = session.getAttribute(name);
        writer.println("<p>name=" + name + " value=" + value);
    }
}
session.invalidate();
writer.println("<p><a href=\"index.html\">Logout" + "</a> and return to login page");
writer.println("</body></html>");
writer.close();
}
catch (Exception e)
{
    e.printStackTrace();
}
}
}

```

When the request is forwarded to the `WelcomeServlet` servlet class, the servlet retrieves the session and checks whether the session is new or not. If the session is new, the logout message is displayed to the user. However, if the same session continues, the session details of the logged in user are displayed on the browser.

The `getAttribute()` method is used to retrieve the values of `username` and `password` attributes. The `getAttributeNames()` method is used to retrieve the values of the attributes that are set during the user session. In the following code snippet, the while loop is used to retrieve the attribute name and its value, which is displayed on the browser:

```

Enumeration names = session.getAttributeNames();
while (names.hasMoreElements())
{
    String name = (String) names.nextElement();
    Object value = session.getAttribute(name);
    writer.println("<p>name=" + name + " value=" + value);
}

```

After displaying the session details, the session is explicitly terminated using the `invalidate()` method. The user can end up the session by clicking the logout link.

The `LoginApp` Web application created a session for the user who had logged in with the `username` and `password`. After logging into the application, the user browses to get the session details. During the session, `username` and `password` have been set as the attributes in the user session, and the values of these attributes are retrieved by the `WelcomeServlet` servlet class and are displayed on the browser.

Now, compile the two servlets from the `src\com\kogent` directory by using the following command:

```
javac -d C:\JavaEE\chapter5\LoginApp\WEB-INF\classes *.java
```

The execution of the preceding command creates the package directory under the `WEB-INF\classes` folder. Prior to packaging, deploying, and running the `LoginApp` Web application, these servlets need to be configured in `web.xml`.

## Configuring the Login Application

To configure the `LoginApp` application, the configuration file used is `web.xml`, which maps the URL path, forwarded by `index.html` to the `LoginServlet` servlet class and also defines the `url pattern` for the `WelcomeServlet` servlet class. Listing 5.10 shows the code of the `web.xml` file (you can find this file on the CD in the `code\Java EE\Chapter 5\LoginApp\WEB-INF` folder):

**Listing 5.10:** Configuring the Servlets and HTML Page of the `LoginApp` Application

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">

<servlet>

```

```

<servlet-name>LoginServlet</servlet-name>
<servlet-class>com.kogent.LoginServlet</servlet-class>
</servlet>
<servlet-mapping>
<servlet-name>LoginServlet</servlet-name>
<url-pattern>/LoginServlet</url-pattern>
</servlet-mapping>
<servlet>
<servlet-name>WelcomeServlet</servlet-name>
<servlet-class>com.kogent.WelcomeServlet</servlet-class>
</servlet>
<servlet-mapping>
<servlet-name>WelcomeServlet</servlet-name>
<url-pattern>/welcomeServlet</url-pattern>
</servlet-mapping>
<session-config>
<session-timeout>30</session-timeout>
</session-config>
<welcome-file-list>
<welcome-file>index.html</welcome-file>
</welcome-file-list>
</web-app>

```

In Listing 5.10, web.xml configures the LoginServlet and WelcomeServlet servlet classes and also provides the url pattern for these servlets. The full package structure has been provided for each servlet class, as shown in Figure 5.14. Moreover, the session timeout has also been set to 30 minutes and the welcome file which will be displayed when the LoginApp Web application runs, is index.html.

Now, before running the LoginApp Web application, it is packaged in the WAR file (LoginApp.war) by using the following command:

```
jar -cvf LoginApp.war .
```

The preceding command creates the LoginApp.war file containing all the files of the LoginApp directory.

## Running the Login Application

After packaging the LoginApp Web application, start the Glassfish server and deploy LoginApp.war. Now, browse <http://localhost:8080/LoginApp> to see the output of the LoginApp application. Figure 5.15 displays the output of index.html, which serve as the Login page:

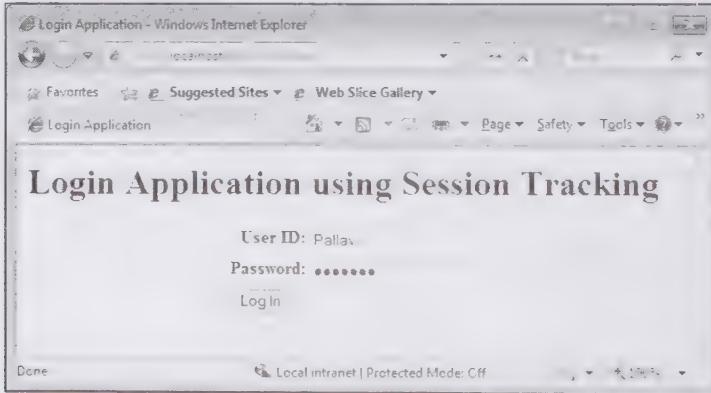
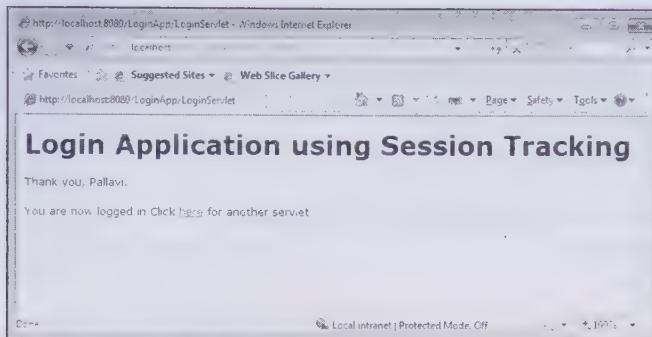


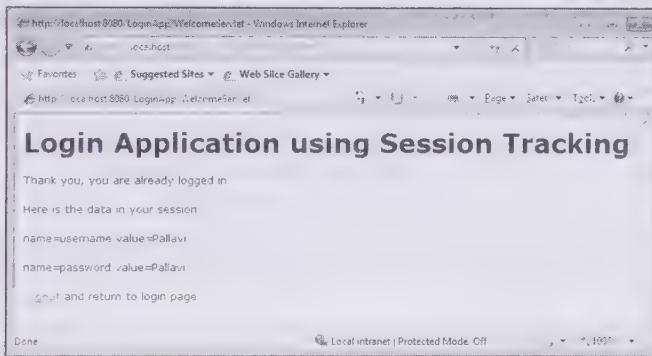
Figure 5.15: Displaying the Output of index.html

In Figure 5.15, enter the User ID and Password and click the Log In button. In our case, we have entered Pallavi in the User ID text box and Pallavi as password in the Password text box. Then, click the Log In button to forward the request to the LoginServlet servlet class. The LoginServlet servlet class creates a new session for the user pallavi and displays a welcome message, as shown in Figure 5.16:



**Figure 5.16: Displaying the Output of the LoginServlet Servlet Class**

The `LoginServlet` servlet class creates a session for the logged in user. To view the session details and value of the attributes set in the session, the user can click the `here` link shown in Figure 5.16. After clicking the link, the request is sent to `WelcomeServlet` and the session details are displayed, as shown in Figure 5.17:



**Figure 5.17: Displaying the Output of the WelcomeServlet Servlet Class**

By clicking the `Logout` link, shown in Figure 5.17, the user can explicitly end the session and return to `index.html`.

Therefore, the `LoginApp` application maintains a session for the user who logs in with the `username` and `password`. Based on the `username` and `password`, the `username` and `password` attributes are set in the session and till the user is in the same session, the session details can be viewed. After logging out, the session is explicitly terminated and the user is no longer in the session and anew session is created for the new user.

This ends up the discussion about session tracking. Let's briefly summarize all the concepts that have been covered so far in this chapter before moving to the next chapter.

## Summary

This chapter has described the implementation of session handling in servlets. Session handling can be done by using many mechanisms, such as Cookies, URL-Rewriting, hidden form fields, and SSL. Apart from this, the session tracking API is also used to handle sessions in servlets. The `javax.servlet.http.HttpSession` interface helps in maintaining the client's session. The various methods, such as `setAttribute()` and `getAttribute()` used to set or retrieve the value of an attribute in a session are discussed in the chapter.

The `LoginApp` Web application has been designed in this chapter, which demonstrates the session tracking of the user logged in by using the `HttpSession` interface. The various other servlets created in the chapter explain URL-Rewriting and working with Cookies.

Apart from session handling, event handling also plays a very important role in servlets. The Servlet 3.0 API defines various classes and interfaces for event handling. The next chapter demonstrates how event handling can be done in servlets.

## Quick Revise

- Q1.** If cookies are disabled on client-side, the alternate session mechanism that can be used is .....  
 A. Either Cookies or URL rewriting      B. URL rewriting  
 C. Cookies and URL rewriting      D. None
- Ans. B
- Q2.** The interface that defines the `getSession()` method is .....  
 A. `HttpServletRequest`      B. `ServletRequest`  
 C. `ServletResponse`      D. `HttpServletResponse`
- Ans. A
- Q3.** The ..... method of `HttpServletRequest` returns null if a session does not exist.  
 A. `getSession()`      B. `getSession(true)`  
 C. `getSession(false)`      D. `getNewSession()`
- Ans. C
- Q4.** The ..... method on the session object is used to remove a set attribute.  
 A. `removeAllValues()`      B. `removeAttribute("attributeName")`  
 C. `removeAttributes()`      D. `removeAllAttributes()`
- Ans. B
- Q5.** Which statement is false:  
 A. URL rewriting can be used to track a session  
 B. SSL has a built in mechanism to obtain the data to define a session  
 C. The name of session tracking cookies must be `JSESSIONID`  
 D. There is no restriction for name of the cookie tracking the session
- Ans. D
- Q6.** What is a session?  
 Ans. A session can be defined as a collection of HTTP requests, over a period of time, between a client and a Web server. When a session is created the lifetime of the session is also set. The session object is destroyed on the expiration of session and all the resources are returned back to the servlet engine.
- Q7.** List the session tracking techniques.  
 Ans. There are basically the following four session tracking techniques:  
 Cookies  
 Hidden Form Fields  
 URL Rewriting  
 Secure Socket Layer (SSL) sessions
- Q8.** How cookies are used to track a session?  
 Ans. Using cookies is the simplest and easiest way to track a session. A unique session id (stored in the form of a cookie) is sent by the server to the client as a part of the response and the same session id saved with the client is sent to the server as a part of the request which helps the server to recognize the unique client session.
- Q9.** List the methods of the `HttpSession` interface which help a servlet to manage session life-cycle.  
 Ans. The `HttpSession` interface provides the following methods to manage session life-cycle:  
 `Invalidate()`  
 `setMaxInactiveInterval(int interval)`  
 `isNew()`  
 `getCreationTime()`  
 `getLastAccessedTime()`
- Q10.** Cookie is a class of the ..... package.  
 Ans. `javax.servlet.http`

# 6

# Implementing Event Handling and Wrappers in Servlets 3.0

**If you need an information on:**

**See page:**

Introducing Events	236
Introducing Event Handling	236
Working with the Types of Servlet Events	237
Developing the onlineshop Web Application	254
Introducing Wrappers	266
Working with Wrappers	268

The concept of event handling allows you to handle the events related to the life cycle of a servlet or session. When an event occurs in a Web application, a relevant listener is notified about the occurrence of the event and the relevant task is performed. To have a better understanding of event handling, let's consider an example. When a client request is received by a Web container, firstly the container maps the request with an appropriate servlet. After the servlet is identified, the Web container loads the servlet class and creates an instance of the servlet. The servlet is then initialized by calling the `init()` method. After calling the `init()` method, the `service()` method passes the request and response objects to the Web container. The Web container invokes the `destroy()` method to remove the servlet, if the servlet is not required. Now, as you can see that in this example, various events, such as initializing a servlet, servicing a request, and destroying the servlet are occurring. These events can be handled by using event handling. To implement the event handling process in this example, you need to create a listener class that is notified by the container about these events.

Apart from handling the events as described in the preceding example, event handling can also be used to handle session level events, such as adding or removing an attribute from a session. In case of life cycle events of a servlet, a context listener is created and notified about the initialization and destruction of a `ServletContext` or the addition and deletion of an attribute from the `ServletContext`. Similarly, a session listener notifies a class when a session is initialized or destroyed, or when an attribute is added or removed from a session.

Java Servlet also allows you to use wrappers to modify the request and response objects. In other words, with the help of wrappers you can add the additional information, such as value of an attribute that is used in a session to the request sent by a user.

This chapter introduces events and explains the process of event handling. The chapter also describes different types of servlet and session level events. Next, you create an onlineshop application to implement the process of handling events. In addition, the chapter also discusses about wrapper classes that are used to modify the request and response data before pre or post operations.

## Introducing Events

An event refers to a set of actions that may occur while an application is running. It could also be defined as a set of actions, such as clicking a button or pressing a key, performed by a user. The following list shows some of the events that may occur during the life cycle of a servlet:

- ❑ Initializing servlets
- ❑ Adding, removing, or replacing attributes in `ServletContext`
- ❑ Creating, activating, passivating, or invalidating a session
- ❑ Adding, removing, or replacing attributes in a servlet session
- ❑ Destroying servlets

Now, let's discuss how these events can be handled in servlets.

## Introducing Event Handling

In Java, events are generated by objects. When an event is fired, a specific method of an object (to be notified) is called. This specific method is then notified about the event and the event object is passed as a parameter to that method. These methods are known as event handlers. However, in servlets, the listener objects are created to listen to events that may occur during the life cycle of a servlet. The event handling can be implemented by using the pre-defined interfaces, which help developers to deal with the `ServletContext` interface and the `HttpSession` interface. A developer can use multiple listeners based on the type of events in a single Web application. These listeners are called event listeners.

The event listeners are Java classes that are notified about the life cycle events of a servlet whenever an event occurs. Apart from the life cycle events, such as creating or destroying the servlets, the event listeners are also notified about the modification of attributes in the `ServletContext` or `HttpSession` object.

A developer creates an event listener class to implement the appropriate listener interface and register it in the `web.xml` file. In Servlet 3.0, it is not mandatory to configure the event listener class; instead, you can simply annotate the listener class. Using annotations, you can declare a listener class as a Web component and can

invoke it in a servlet. The servlet container creates an object of the event listener class while deploying an application. The object is then invoked by the servlet container at runtime.

Earlier, you need to set the shared resources by using the `<context-param>` tag of the web.xml file and then access them by using the `getInitParameter()` method of the `ServletContext` object. However, while setting the shared resources by using the `<context-param>` tag, the developer is not aware of the sequence in which the resources of an application are accessed. Due to this, the developer was not able to provide the code to perform the appropriate task on the occurrence of an event. The solution to this problem is to give major control to only one servlet, which can listen to the application life cycle events. Event listeners are introduced to notify the listener (servlet) about the occurrence of an event.

Let's now explore the different types of servlet events and event listeners.

## Working with the Types of Servlet Events

The events that can occur during the life cycle of a servlet can be grouped into various categories on the basis of their level of occurrence. The following is the list of various levels of servlet events:

- ❑ **Request level events**—Refer to the events related to the client's information to a servlet. There are two event listeners, `ServletRequestListener` and `ServletRequestAttributeListener`, to handle request level events. The `ServletRequestListener` interface is implemented to get notifications about the incoming and outgoing requests' scope in a Web component. The `ServletRequestAttributeListener` interface is implemented to get notifications about the changes made in the request attribute.
- ❑ **Servlet context level events**—Refer to the application level events. The `ServletContextListener` interface is implemented to notify the initialization or destruction of the servlet in the `ServletContext` interface. The class implementing the `ServletContextAttributeListener` interface is notified about the changes made to the attributes of the `ServletContext` object.
- ❑ **Servlet session level events**—Refer to the events that are used to maintain the session of a client. The `HttpSessionListener` and `HttpSessionActivationListener` interfaces are implemented to notify the creation, invalidation, activation, passivation, and timeout events of the `HttpSession` interface. The `HttpSessionAttributeListener` interface can be implemented to notify the changes in the attributes of a session and the class implementing the `HttpSessionBindingListener` interface is notified when a servlet is bound to or unbound from a session.

Let's now discuss the preceding levels of servlet events in detail in the following subsections.

### Implementing the Servlet Context Level Events

A `ServletContext` object is created and associated with a Web application at the time of its deployment. This object stores all the information about the servlets associated with the Web application. The developer can create a database connection when the `ServletContext` object is created, and may close the connection when the `ServletContext` object is destroyed. To handle the events associated with the creation and destruction of the `ServletContext` object, you need to create a listener that implements the `ServletContextListener` interface. This interface helps to notify the listener about the occurrence of the following events:

- ❑ Creation of the `ServletContext` and its availability to service the first request
- ❑ Termination of the `ServletContext`

The `ServletContextAttributeListener` interface helps to notify a listener about any changes or modifications made to the attributes of the `ServletContext` object.

The `ServletContextAttributeListener` event listener notifies about the following three events:

- ❑ Adding a new attribute to the context
- ❑ Replacing an existing attribute in the context
- ❑ Removing an attribute from the context

The `ServletContextListener` and `ServletContextAttributeListener` interfaces have various methods that are invoked by the object of the event listener that implements either of these interfaces to notify about the `ServletContext` events.

## Explaining Servlet Context Life Cycle Event Handling

The events in a servlet life cycle can be monitored and handled by defining an object of a listener. Then, the listener's object invokes an appropriate method based upon the occurrence of the servlet life cycle events. This process is used to handle the servlet context life cycle events.

Let's now discuss about the various methods of the listener interfaces, `ServletContextListener` and `ServletContextAttributeListener` and then define a listener class.

### Describing the `ServletContextListener` Interface

The `ServletContextListener` interface receives notifications about the changes in the `ServletContext` object of the Web application. To receive notification events, the listener class implements the `ServletContextListener` interface, which must be configured in the Deployment Descriptor of the Web application. The `ServletContextListener` interface extends the `EventListener` class.

Table 6.1 describes the methods provided by the `ServletContextListener` interface:

<b>Table 6.1: Methods of the <code>ServletContextListener</code> Interface</b>		
<b>Method</b>	<b>Syntax</b>	<b>Description</b>
contextInitialized	public void contextInitialized (ServletContextEvent sce)	Notifies that the initialization process for the Web application is going to start. During the notification process, a notification about context initialization is sent to all <code>ServletContextListeners</code> prior to initializing any filter or servlet in the Web application.
contextDestroyed	public void contextDestroyed (ServletContextEvent sce)	Notifies that the <code>ServletContext</code> object is going to shut down. All servlets and filters should be destroyed prior to the notification about the <code>ServletContext</code> object destruction.

### Describing the `ServletContextAttributeListener` Interface

The `ServletContextAttributeListener` interface allows us to monitor the changes made to the attribute events of a `ServletContext`. The `ServletContext` attribute events are as follows:

- Adding an attribute
- Replacing an attribute
- Removing an attribute

Now, let's discuss about the `ServletContextAttributeListener` interface.

Implementing the `ServletContextAttributeListener` interface enables a user to receive notifications whenever the attribute list on the `ServletContext` object of a Web application changes. Note that to receive notification events, the implementation class must be configured in the Deployment Descriptor of a Web application.

Table 6.2 describes the methods provided by the `ServletContextAttributeListener` interface:

<b>Table 6.2: Methods of the <code>ServletContextAttributeListener</code> Interface</b>		
<b>Method</b>	<b>Syntax</b>	<b>Description</b>
attributeAdded	public void attributeAdded (ServletContextAttributeEvent scabe)	Notifies about the addition of a new attribute in the <code>ServletContext</code> object. It is called just after the addition of the attribute.

**Table 6.2: Methods of the ServletContextAttributeListener Interface**

Method	Syntax	Description
attributeRemoved	public void attributeRemoved (ServletContextAttributeEvent scab)	Notifies about the removal of an existing attribute from the ServletContext object. It is called just after the removal of the attribute.
attributeReplaced	public void attributeReplaced ServletContextAttributeEvent scab)	Notifies about the replacement of an attribute in the ServletContext object. It is called just after the replacement of an attribute.

Let's learn how to handle events for servlet context by creating a Web application named events.

## Handling Events for Servlet Context

In this section, let's implement the event handling mechanism in ServletContext by creating a Web application called events. In this application, the ServletContextListener object notifies about the creation and destruction of the ServletContext object. In addition, the ServletContextAttributeListener object notifies about adding, replacing, or removing an attribute from the servlet context.

### Implementing the ServletContextListener Interface

You need to create a listener class that implements the ServletContextListener interface and a simple servlet to display a message. The listener class is notified about the initialization and destruction of ServletContext in the events Web application. This listener class implements the ServletContextListener interface.

Listing 6.1 provides the code for the ContextListener class (you can find the ContextListener.java file on the CD in the code\JavaEE\Chapter6\events\src\com\kogent\listeners folder):

#### Listing 6.1: Showing the Implementation of the ServletContextListener Interface

```
package com.kogent.listeners;
import javax.servlet.ServletContext;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import javax.swing.*;
public class ContextListener implements ServletContextListener
{
    private ServletContext cont=null;
    public void contextInitialized(ServletContextEvent evt)
    {
        this.cont = evt.getServletContext();
        System.out.println("Context initialized.....!!!!!");
        JOptionPane.showMessageDialog(null,"Context initialized.....!!!!!");
    }
    public void contextDestroyed(ServletContextEvent evt)
    {
        System.out.println("Context destroyed.....!!!!!");
        JOptionPane.showMessageDialog(null,"Context destroyed.....!!!!!");
        this.cont=null;
    }
}
```

In Listing 6.1, the ContextListener class is notified about the initialization of ServletContext before a servlet is initialized in the events Web application. A dialog box appears displaying the message that the servlet context of the Web application has been initialized, when the contextInitialized() method receives the notification about initialization. The following code snippet demonstrates how to map the ContextListener class in the web.xml file:

```
<listener>
<listener-class>com.kogent.listeners.ContextListener</listener-class>
</listener>
```

Let's create a simple servlet, DemoServlet, to display a hello message to all users.

Listing 6.2 provides the code for the DemoServlet servlet (you can find the DemoServlet.java file on the CD in the code\JavaEE\Chapter6\events\src\com\kogent\servlets folder):

**Listing 6.2: Implementing Event Handling in the DemoServlet Servlet**

```
package com.kogent.servlets;
import java.io.*;
import java.net.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class DemoServlet extends HttpServlet
{
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException
    {
        PrintWriter pw = response.getWriter();
        response.setContentType("text/html");
        pw.println("<html><head></head><body>");
        pw.println("Hello world: This Servlet has been initialized");
        pw.println("</body></html>");
    }
}
```

Now, configure the DemoServlet servlet in the web.xml file. The listener class and the servlet class are mapped in the web.xml file.

Listing 6.3 shows the code for the web.xml file (you can find this file on the CD in the code\JavaEE\Chapter6\events\WEB-INF folder):

**Listing 6.3: Displaying the Code for the web.xml File**

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">

    <listener>
        <listener-class>com.kogent.listeners.ContextListener</listener-class>
    </listener>

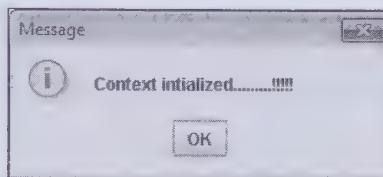
    <servlet>
        <servlet-name>DemoServlet</servlet-name>
        <servlet-class>com.kogent.servlets.DemoServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>DemoServlet</servlet-name>
        <url-pattern>/DemoServlet</url-pattern>
    </servlet-mapping>

    <session-config>
        <session-timeout>30</session-timeout>
    </session-config>

    <welcome-file-list>
        <welcome-file>DemoServlet</welcome-file>
    </welcome-file-list>
</web-app>
```

In Listing 6.3, the DemoServlet servlet is displayed as a welcome page to the users. According to Servlet 3.0 specification, servlets can also be specified as a welcome file. Now, compile the ContextListener listener class and DemoServlet servlet class, and generate the events.war file, which is deployed on the Glassfish application server. When the events.war file is deployed on the Glassfish application server, the Message dialog box appears, as shown in Figure 6.1:



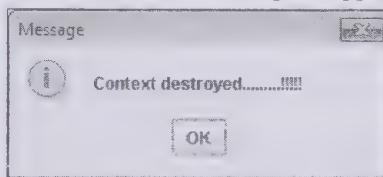
**Figure 6.1: Displaying a Message after the Web Application is Deployed**

After initializing the `ServletContext` of the events Web application, the welcome message can be displayed by browsing the `http://localhost:8080/events` URL.

#### NOTE

*URL stands for Uniform Resource Locator.*

When the events Web application is undeployed, the `contextDestroyed()` method of the `ContextListener` class is invoked and the Message dialog box appears, as shown in Figure 6.2:



**Figure 6.2: Displaying a Message after the Web Application is Undeployed**

Similar to the notification of initialization and destruction of a `ServletContext`, the notifications with respect to the modifications made in attributes of `ServletContext` are provided with the help of the `attributeAdded`, `attributeRemoved`, and `attributeReplaced` methods of the `ServletContextAttributeListener` interface.

#### Implementing the `ServletContextAttributeListener` Interface

The `ServletContextAttributeListener` class implements the `ServletContextAttributeListener` interface and its methods are notified depending upon the event generated.

Listing 6.4 provides the code for the `ServletContextAttributeListener` class (you can find the `ServletContextAttributeListener.java` file on the CD in the `code\JavaEE\Chapter6\events\src\com\kogent\listeners` folder):

**Listing 6.4: Implementing the `ServletContextAttributeListener` Interface in the `ServletContextAttributeListener` Class**

```
package com.kogent.listeners;

import javax.servlet.ServletContext;
import javax.servlet.ServletContextAttributeEvent;
import javax.servlet.ServletContextAttributeListener;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import javax.swing.*;
```

```
public class ServletContextAttributeListener
implements ServletContextAttributeListener {

    //This method is invoked when an attribute
    //is added to the ServletContext object
    public void attributeAdded (ServletContextAttributeEvent scae)
    {
        JOptionPane.showMessageDialog(null,"Attribute Added!!!!!");
    }

    //This method is invoked when an attribute
```

```

//is removed from the ServletContext object
public void attributeRemoved (ServletContextAttributeEvent scae)
{
    JOptionPane.showMessageDialog(null,"Attribute Removed!!!!!");
}

//This method is invoked when an attribute
//is replaced in the ServletContext object
public void attributeReplaced (ServletContextAttributeEvent scae)
{
    JOptionPane.showMessageDialog(null,"Attribute Replaced!!!!!");
}
}
}

```

In Listing 6.4, the `ServletContextAttribListener` class is a simple Java class, which is notified when the attributes from the `ServletContext` object are added, replaced, or removed. The relevant method of this class is invoked depending upon the event fired. Save the `ServletContextAttribListener.java` file at `C:\JavaEE\Chapter 6\events\src\com\kogent\listeners` location.

The following code snippet shows the mapping to be added to the `web.xml` file (Listing 6.3) for the `ServletContextAttribListener` class:

```

<listener>
<listener-class>com.kogent.listeners.ServletContextAttribListener</listener-class>
</listener>

```

Now, let's create a HyperText Markup Language (HTML) page that forwards the request to servlets when a form is submitted. The servlets then detect the type of action, such as adding, removing, or replacing the attributes, and notify to the `ServletContextAttribListener` class.

Listing 6.5 provides the code for the `servletcontextattrib.html` file (you can find this file on the CD in the `code\JavaEE\Chapter6\events` folder):

#### **Listing 6.5: Showing the Code for the `servletcontextattrib.html` File**

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
    <title>Working With Attributes Of the ServletContext Object</title>
    <link rel="stylesheet" href="mystyle.css" type="text/css"/>
</head>
<body>
    <h1>Demonstrating Servlet Context Attribute Listener </h1>
    <form name="theForm" action=".//servletcontextattrib" method="POST">
        <table width="75%">
            <tr><td rowspan="3">Select an Action</td>
                <td><input type="radio" name="action" value="add">
                    Add Attribute To ServletContext</td>
            </tr>
            <tr>
                <td><input type="radio" name="action" value="remove">
                    Remove Attribute From ServletContext</td>
            </tr>
            <tr>
                <td><input type="radio" name="action" value="replace">
                    Replace Attribute In ServletContext</td>
            </tr>
            <tr><td>Enter Servlet Context Attribute Name</td>
                <td><input type="text" name="name" value=""></td>
            </tr>
            <tr><td>Enter Servlet Context Attribute Value</td>
                <td><input type="text" name="value" value=""></td>
            </tr>
            <tr><td colspan="2" align="center">
                <input type="submit" name="btnSubmit" value="Submit">
            </td>
        </tr>
    </table>
</body>

```

```

        </form>
    </body>
</html>

```

In Listing 6.5, an HTML page is designed. This HTML page has three radio buttons for three different actions. There are two textboxes as well, which hold the name of the attribute and its value. Depending upon the action selected in the HTML page, the respective method of the `ServletContextAttribListener` class is called. When the `servletcontextattrib.html` form is submitted, the request is forwarded to the `ServletContextAttrib` class, which is mapped to the `/servletcontextattrib` URL pattern in the `web.xml` file.

Listing 6.6 provides the code for the `ServletContextAttrib` servlet (you can find the `ServletContextAttrib.java` file on the CD in the `code\JavaEE\Chapter6\events\src\com\kogent\servlets` folder):

**Listing 6.6:** Showing the Code for the `ServletContextAttrib.java` File

```

package com.kogent.servlets;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ServletContextAttrib extends HttpServlet
{
    public void service(HttpServletRequest request,
                        HttpServletResponse response)
        throws IOException, ServletException
    {
        response.setContentType("text/html");
        ServletOutputStream out = response.getOutputStream();
        out.print("<html>");
        out.print("<head>");
        out.print("<title>ServletContext Attributes</title>");
        out.print("</head>");
        out.print("<body>");
        ServletContext context = getServletContext();
        String action = request.getParameter("action");
        String name = request.getParameter("name");
        String value = request.getParameter("value");

        if (action == null) {}
        else {
            if (action.equals("add"))
            {
                String test = (String) context.getAttribute(name);
                if (test == null)
                {
                    context.setAttribute(name, value);
                    out.print("Added Attribute To ServletContext object");
                } else {
                    context.setAttribute(name, value);
                    out.print("Replaced Attribute in ServletContext");
                }
            }
            else if (action.equals("remove"))
            {
                String test = (String) context.getAttribute(name);
                if (test == null) {
                    out.print("Attribute does not exist");
                } else {
                    context.removeAttribute(name);
                    out.print("Removed Attribute From ServletContext");
                }
            }
        }
    }
}

```

```

    {
        String test = (String) context.getAttribute(name);
        if (test == null)
        {
            context.setAttribute(name, value);
            out.print("Added Attribute To ServletContext object");
        } else {
            context.setAttribute(name, value);
            out.print("Replaced Attribute in ServletContext");
        }
    }
}

out.print("<center> <br /> <br />");
out.print("<a href='./servletcontextattrib.html'></a>");
out.print("Back To Home Page");
out.print("</a>");
out.print("</center>");
out.print("</body>");
out.print("</html>");
}
}

```

In Listing 6.6, `ServletContextAttrib` is a servlet class that responds to the request sent by the `servletcontextattrib.html` form. The `ServletContextAttrib` class retrieves the data, such as the name of the attribute, the value of the attribute, and the action to be performed. Depending upon the following actions selected by a user, the relevant method is invoked to perform a task:

- ❑ **The add action**—Allows the servlet container to invoke the `attributeAdded()` method to add a new attribute (provided by the user) to the `ServletContext`
- ❑ **The remove action**—Allows the servlet container to invoke the `attributeRemoved()` method to remove the attribute selected by the user from the `ServletContext`
- ❑ **The replace action**—Allows the servlet container to invoke the `attributeReplaced()` method to replace the old value of an attribute with a new value

Save the `ServletContextAttrib.java` file at the `C:\JavaEE\Chapter6\events\src\com\kogent\servlets` location and then compile both the `ServletContextAttribListener` and `ServletContextAttrib` classes.

After compiling these classes, map the `ServletContextAttrib` servlet class in the `web.xml` file (Listing 6.3). The following code snippet shows the mapping for the `ServletContextAttrib` servlet:

```

<servlet>
    <servlet-name>ServletContextAttrib</servlet-name>
    <servlet-class>com.kogent.servlets.ServletContextAttrib</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>ServletContextAttrib</servlet-name>
    <url-pattern>/servletcontextattrib</url-pattern>
</servlet-mapping>

```

Add the mapping for the `ServletContextAttrib` class in the `web.xml` file, which is created in Listing 6.3. Now, create a new `events.war` file to be deployed on the Glassfish application server. After deploying the Web ARchive (WAR) file, browse the `http://localhost:8080/events/servletcontextattrib.html` URL.

Figure 6.3 displays the output of the `servletcontextattrib` HTML page when the attribute name and value are entered in the relevant text boxes, and the `Add Attribute To ServletContext` radio button is selected:

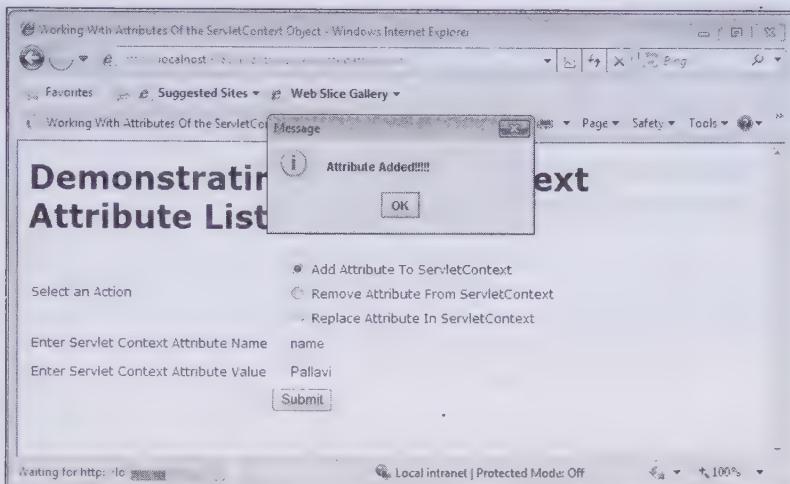


Figure 6.3: Displaying a Message when a New Attribute is Added

In Figure 6.3, when the user clicks the Submit button, a message box appears with the Attribute Added!!!! message.

You can also replace the value of an attribute with a new value by selecting the Replace Attribute In Servlet Context radio button in the servletcontextattrib HTML page and clicking the Submit button. The attribute value is replaced.

Figure 6.4 displays a message box after the ServletContextAttribListener class is notified about the replacement of the attribute value:

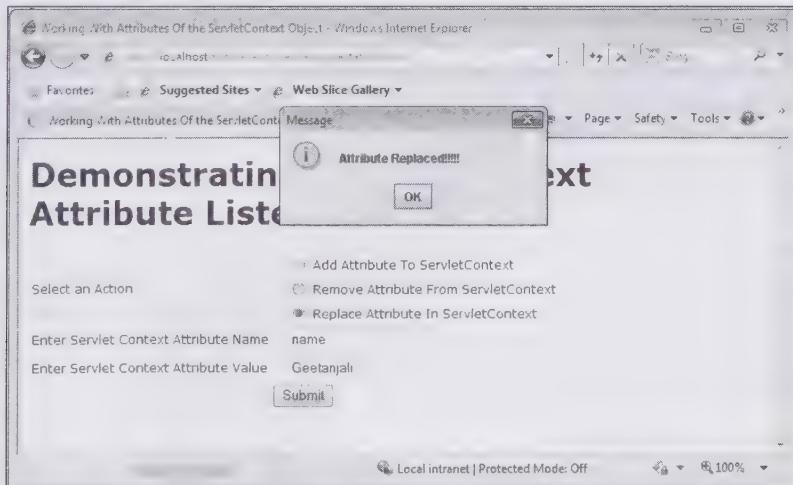
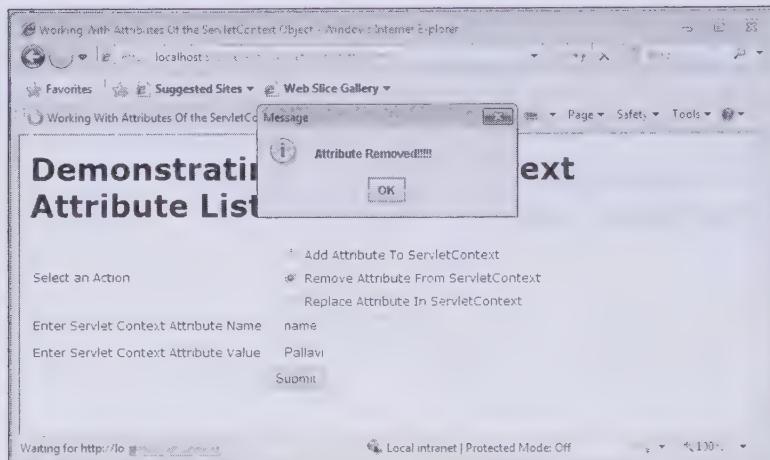


Figure 6.4: Displaying a Message when an Attribute is Replaced

In Figure 6.4, the value of the name attribute is replaced with Geetanjali and the attributeReplaced() method of the ServletContextAttribListener class is invoked.

You can also remove an attribute, when no longer required. In the servletcontextattrib HTML page, you can remove the name attribute by specifying its name and value, selecting the Remove Attribute From ServletContext radio button, and clicking the Submit button.

Figure 6.5 displays the message that appears when the name attribute is removed from the servlet context:



**Figure 6.5: Displaying a Message when an Attribute is Removed**

After learning to notify changes of the attributes to `ServletContextAttributeListener` interface, let's now learn to notify about the changes made in a session.

## Implementing the Servlet Session Level Events

The session level events refer to maintaining the state of a user or a client session. In a session, a class is notified about the servlet session level events during the occurrence of any of the following events:

- Creating a session
- Adding, removing, or replacing the attributes of a session
- Destroying a session

When any of these events occur, the appropriate interface is notified about the event. The `HttpSessionListener` interface is notified about the creation and destruction of a session. The activation or passivation of a session is notified to the `HttpSessionActivationListener` interface. In a session, when new attributes are created or the existing ones are replaced, the changes are notified in the `HttpSessionAttributeListener` interface.

Now, let's discuss how to handle life cycle events of a session.

## Explaining Session Life Cycle Event Handling

The application event listeners can generate notification during creation and destruction of an `HttpSession` object or during the modification of its attributes. A Java class can be defined to handle notifications for the changes made to the `HttpSession` objects. To receive life cycle notification of the `HttpSession` object, let's first understand the `HttpSessionListener` and `HttpSessionActivationListener` interfaces that are implemented to create a listener that handles life cycle events of a session.

### Describing the `HttpSessionListener` Interface

The `HttpSessionListener` interface receives a notification about the creation and invalidation of a session. This interface defines the `sessionCreated` and `sessionDestroyed` methods (Table 6.3). The Java class can implement the `HttpSessionListener` interface to receive life cycle notifications for the `HttpSession` interface.

Implementations of the `HttpSessionListener` interface provide the notification of changes made in the active sessions in a Web application. You need to provide the configuration details about the implementation class in Deployment Descriptor for the Web application so that the application can be notified about the occurrence of events.

Table 6.3 describes the methods provided by the `HttpSessionListener` interface:

**Table 6.3: Methods of the HttpSessionListener Interface**

Method	Syntax	Description
sessionCreated	public void sessionCreated (HttpSessionEvent se)	Notifies that a session was created, where se implies the notification event
sessionDestroyed	public void sessionDestroyed (HttpSessionEvent se)	Notifies that a session is going to be invalidated, where se implies the notification event

The `sessionCreated()` method notifies that a session was created and the `sessionDestroyed()` method notifies that a session is about to be destroyed. The `HttpSessionEvent` class represents event notifications for changing the sessions within a Web application. The constructor of this class constructs a session event from the given source. The `getSession()` method of this interface returns the instance of the session that has been changed.

#### Describing the HttpSessionActivationListener Interface

Objects that are bound to a session are notified about the activation and passivation of a session by the servlet container. While transferring the session details from one Virtual Machine (VM) to another, the container should notify about the event to the class that implements the `HttpSessionActivationListener` interface.

Table 6.4 describes the methods provided by the `HttpSessionActivationListener` interface:

**Table 6.4: Methods of the HttpSessionActivationListener Interface**

Method	Syntax	Description
sessionDidActivate	public void sessionDidActivate (HttpSessionEvent se)	Notifies about the activation of a session
sessionWillPassivate	public void sessionWillPassivate (HttpSessionEvent se)	Notifies about the passivation of a session

Similar to context attributes, the changes in session attributes are notified to the `HttpSessionAttributeListener` interface.

#### Handling Events for Session Attribute

The changes in the attributes of a session can be notified by implementing the `HttpSessionAttributeListener` interface. The classes whose instances are added or removed by a session also implement the `HttpSessionBindingListener` interface, apart from the `HttpSessionAttributeListener` interface. In addition, you also learn about the `HttpSessionBindingEvent` class used to bind or unbind an object from a session.

Let's now discuss the `HttpSessionAttributeListener` interface in detail.

#### Describing the HttpSessionAttributeListener Interface

The `HttpSessionAttributeListener` interface can be implemented to get notifications about the changes made to the attribute lists of sessions within a Web application.

Table 6.5 describes the methods provided by the `HttpSessionAttributeListener` interface:

**Table 6.5: Methods of the HttpSessionAttributeListener Interface**

Method	Syntax	Description
attributeAdded	public void attributeAdded (HttpSessionBindingEvent se)	Notifies about the addition of an attribute to a session. This method is invoked after the addition of an attribute.
attributeRemoved	public void attributeRemoved (HttpSessionBindingEvent se)	Notifies about the removal of an attribute from a session. This method is invoked after the removal of an attribute.

**Table 6.5: Methods of the HttpSessionAttributeListener Interface**

Method	Syntax	Description
attributeReplaced	public void attributeReplaced (HttpSessionBindingEvent se)	Notifies about the replacement of an attribute in a session. This method is invoked after an attribute is replaced.

The attributeAdded, attributeRemoved, and attributeReplaced methods notify about addition, removal, and replacement of an attribute, respectively from the session. These methods are also present in the ServletContextAttributeListener interface, as discussed previously. However, in the HttpSessionAttributeListener interface, these methods are notified about the changes made in the attributes of a session.

In addition to these interfaces, the HttpSessionBindingEvent class is an important class in listening session events. Now, let's discuss the HttpSessionBindingEvent class.

#### *Describing the HttpSessionBindingEvent Class*

Whenever an object is bound to or unbound from a session, the object that implements the HttpSessionBindingListener interface is notified about the event by the HttpSessionBindingEvent class. The HttpSessionBindingEvent class also notifies the HttpSessionAttributeListener interface that is configured in the Deployment Descriptor about bounding, unbounding, or replacing an attribute in a session. A session binds an object by calling the HttpSession.setAttribute() method, and unbinds it by calling the HttpSession.removeAttribute() method. The following constructors are present in the HttpSessionBindingEvent class:

- **Two Arguments Constructor**—Creates an event that notifies an object about its bounding or unbounding from a session. An object needs to implement the HttpSessionBindingListener interface to receive an event. The following code snippet shows the two argument constructor of the HttpSessionBindingEvent class:

```
public HttpSessionBindingEvent(HttpSession session, java.lang.String name)
```

The arguments passed in the preceding code snippet are as follows:

- **session**—Represents the session to which an object bounds or unbounds
- **name**—Represents the name with which an object bounds or unbounds

- **Three Arguments Constructor**—Creates an event that notifies an object about its bounding or unbounding from a session. The object requires to implement the HttpSessionBindingListener interface to receive an event.

The following code snippet shows the three argument constructors of the HttpSessionBindingEvent class:

```
public HttpSessionBindingEvent(HttpSession session, java.lang.String name,  
                               java.lang.Object value)
```

The arguments passed in the preceding code snippet are as follows:

- **session**—Represents the session to which an object bounds or unbounds
- **name**—Represents the name with which an object bounds or unbounds
- **value**—Represents the value with which an object bounds or unbounds

Table 6.6 describes the methods provided by the HttpSessionBindingEvent class to retrieve the value of the bound attribute:

**Table 6.6: Methods of the HttpSessionBindingEvent Class**

Method	Syntax	Description
getSession	public HttpSession getSession()	Retrieves the session object. It overrides the getSession() method in the HttpSessionEvent class.

**Table 6.6: Methods of the HttpSessionBindingEvent Class**

Method	Syntax	Description
getName	public java.lang.String getName()	Retrieves the name with which an attribute bounds or unbounds from a session. A String that specifies the name with which the attribute bounds or unbounds from the session is returned by the method.
getValue	public java.lang.Object getValue()	Retrieves the value of an added, removed, or replaced attribute. The returned value represents the value of an added attribute if the attribute was added (or bound). The returned value represents the value of the attribute that is either removed or unbound from a session. If the attribute was replaced, the returned value represents the old value of the attribute.

After learning about the interfaces and classes that help in session life cycle event handling, let's create the `SessionListener` class in the events Web application. The `SessionListener` class implements the `HttpSessionListener` and `HttpSessionAttributeListener` interfaces so that it is notified about the session level events.

#### *Implementing the HttpSessionListener and HttpSessionAttributeListener Interfaces*

In this subsection, let's create a simple Java class that implements the `HttpSessionListener` interface so that when the attributes of the session are changed, the `SessionListener` class is notified.

Listing 6.7 provides the code for the `SessionListener` class (you can find the `SessionListener.java` file on the CD in the `code\JavaEE\Chapter6\events\src\com\kogent\listeners` folder):

#### **Listing 6.7: Implementing the HttpSessionAttributeListener Interface in the SessionListener Class**

```
package com.kogent.listeners;

import javax.servlet.http.*;
import javax.servlet.*;
import javax.swing.*;
import javax.servlet.http.HttpSessionAttributeListener;
import javax.servlet.http.HttpSessionBindingEvent;

public class SessionListener implements HttpSessionListener,
HttpSessionAttributeListener
{
    public void sessionCreated(HttpSessionEvent hse)
    {
        JOptionPane.showMessageDialog(null, "Session Created!!!!");
    }

    public void sessionDestroyed(HttpSessionEvent hse)
    {
        JOptionPane.showMessageDialog(null, "Session Destroyed!!!!");
    }

    public void attributeAdded(HttpSessionBindingEvent event)
    {
        JOptionPane.showMessageDialog(null,"Attribute Added:" +
event.getName() + "", "" + event.getValue());
    }

    public void attributeRemoved(HttpSessionBindingEvent event) {
        JOptionPane.showMessageDialog(null,"Attribute Removed" + event.getName() + "", "" +
event.getValue());
```

```
    public void attributeReplaced(HttpSessionBindingEvent event) {
```

```

        JOptionPane.showMessageDialog(null,"Attribute Replaced" + event.getName() + ", !" +
        event.getValue());
    }
}

```

Save the SessionListener.java file at the C:\JavaEE\Chapter6\events\src\com\kogent\listeners location. SessionListener is a simple Java class that implements the HttpSessionListener and HttpSessionAttributeListener interfaces. When the HttpSession object is created, the SessionListener class is notified about the creation of the session and a Message box is displayed. When the attributes of this session change, then the SessionListener class is notified about the change and the respective message will be displayed. Similar to other listeners, the SessionListener class is also mapped in the web.xml file (Listing 6.3).

The following code snippet shows the mapping for the SessionListener class:

```

<web-app>
.....
<listener>
    <listener-class>com.kogent.listeners.SessionListener
    </listener-class>
</listener>
.....
</web-app>

```

Now, let's create two servlet classes, SessionCreateServlet and SessionDestroyServlet to create and destroy a session, respectively. The SessionCreateServlet class creates a session and when the HttpSession object is created, the SessionListener class is notified that a session has been created.

Listing 6.8 provides the code for the SessionCreateServlet servlet (you can find the SessionCreateServlet.java file on the CD in the code\JavaEE\Chapter6\events\src\com\kogent\servlets folder):

**Listing 6.8:** Showing the Code for the SessionCreateServlet.java File

```

package com.kogent.servlets;
import java.io.*;
import java.util.Date;
import javax.servlet.*;
import javax.servlet.http.*;
public class SessionCreateServlet extends HttpServlet
{
    @Override
    public void doGet (HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException
    {
        // Get the session object.
        HttpSession session = req.getSession(true);

        // Set content type for the response.
        res.setContentType("text/html");

        // Then write the data of the response.
        PrintWriter out = res.getWriter();
        out.println("<HTML><BODY>");
        out.println("<A HREF=\"/events/mysessiondestroy\">Destroy Session</A>");
        out.println("<h2>Session Created</h2>");
        out.println("<h3>Session Data:</h3>");
        out.println("New Session: " + session.isNew());
        out.println("<br>Session ID: " + session.getId());
        out.println("<br>Creation Time: " + new Date(session.getCreationTime()));
        session.setAttribute("User", "kogent");
        out.println("</BODY></HTML>");
    }
}

```

In Listing 6.8, a new attribute is set by using the `setAttribute()` method of the `HttpSession` object. The SessionCreateServlet class also displays the ID and the time of the session. However, when the user clicks the Destroy Session link, the request is forwarded to the servlet mapped to the /mysessiondestroy url-

pattern. Save the SessionCreateServlet.java file at the C:\JavaEE\Chapter6\events\src\com\kogent\servlets location and then compile it. Now, create the SessionDestroyServlet servlet class which invalidates the session and removes the User attribute from the session.

Listing 6.9 provides the code for the SessionDestroyServlet servlet class (you can find the SessionDestroyServlet.java file on the CD in the code\JavaEE\Chapter6\events\src\com\kogent\servlets folder):

**Listing 6.9:** Showing the Code for the SessionDestroyServlet.java File

```
package com.kogent.servlets;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class SessionDestroyServlet extends HttpServlet
{
    @Override
    public void doGet (HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
        // Get the session object and remove the attribute from the session.
        HttpSession session = req.getSession(true);
        session.removeAttribute("User");

        // Invalidate the session.
        session.invalidate();

        // Set content type for response.
        res.setContentType("text/html");

        // Then write the data of the response.
        PrintWriter out = res.getWriter();
        out.println("<HTML><BODY>");
        out.println("<A HREF=\"/events/index.html\">Reload Welcome Page</A>");
        out.println("<h2>Session Destroyed</h2>");
        out.println("</BODY></HTML>");
        out.close();
    }
}
```

In Listing 6.9, the SessionDestroyServlet servlet class retrieves the session by using the getSession() method and destroys the session by invoking the invalidate() method on the HttpSession object. Save the SessionDestroyServlet.java file at the C:\JavaEE\Chapter6\events\src\com\kogent\servlets location and then compile it.

Let's create the index.html file, which is a simple HTML form used to pass a request to the SessionCreateServlet servlet to create a new session.

Listing 6.10 provides the code for the index.html file (you can find this file on the CD in the code\JavaEE\Chapter6\events\ folder):

**Listing 6.10:** Showing the Code for the index.html File

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<HTML>
    <Head>
        <title>Example Demonstrating Session Event Handling</title>
        <link rel="stylesheet" href="mystyle.css" type="text/css"/>
    </Head>
    <BODY>
        <H2>Session Event Listener</H2>
        <P>
            This example demonstrates the use of a session event listener.
        </P>
        <P>
            <a href="/events/mysessioncreate">Create New Session</a><br><br>
        </P>
        <P>
```

```

Click the <b>Create</b> link above to start a new session.<br>
</P>
</BODY>
</HTML>

```

After creating the listener and the relevant servlets, configure these servlets in the web.xml file (Listing 6.3). The following code snippet shows how to map the SessionCreateServlet and SessionDestroyServlet classes:

```

<servlet>
    .....
    <servlet-name>sessioncreate</servlet-name>
    <servlet-class>com.kogent.servlets.SessionCreateServlet
    </servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>sessioncreate</servlet-name>
    <url-pattern>/mysessioncreate</url-pattern>
</servlet-mapping>

<servlet>
    <servlet-name>sessiondestroy</servlet-name>
    <servlet-class>SessionDestroyServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>sessiondestroy</servlet-name>
    <url-pattern>/mysessiondestroy</url-pattern>
</servlet-mapping>
.....
</servlet>

```

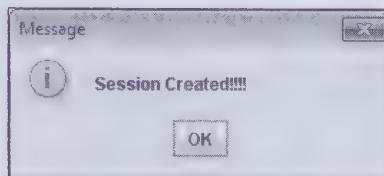
The preceding code snippet maps the SessionCreateServlet to /mysessioncreate and SessionDestroyServlet to /mysessiondestroy url-pattern. Add this code snippet in the existing web.xml file provided in Listing 6.3. Now, compile all the Java source files, create a new WAR file for the events Web application, and deploy the events.war file on Glassfish application server. After deploying the events.war file, browse the URL <http://localhost:8080/events/index.html>.

Figure 6.6 displays the index page of the events Web application:



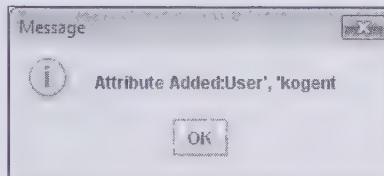
**Figure 6.6: Displaying the index Page of the events Web Application**

If the user clicks the Create New Session link, as shown in Figure 6.6, a new HttpSession object is created. The SessionListener class is notified about the creation of the new session and a Message box appears, as shown in Figure 6.7:



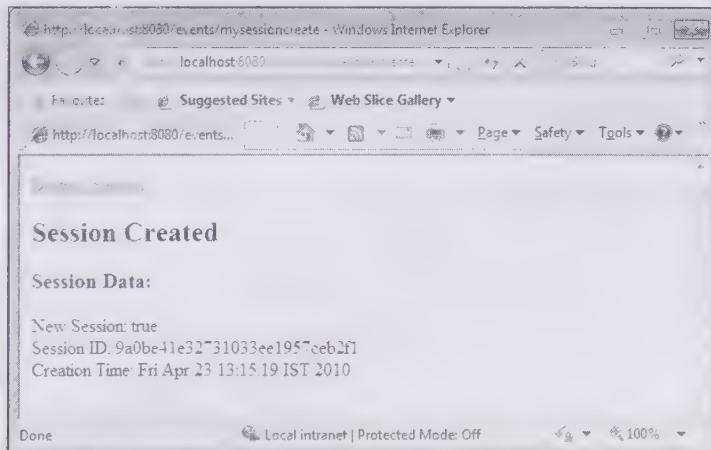
**Figure 6.7: Displaying the Session Created Message**

After receiving the notification that a session has been created, a new attribute called `User` is created. The `SessionListener` class is also notified about the creation of the `User` attribute in the current session. A message box appears with the information about the newly created attribute, as shown in Figure 6.8:



**Figure 6.8: Displaying the Attribute Created Message**

After the `attributeAdded()` method of the `SessionListener` class is notified about adding the `User` attribute, the `SessionCreateServlet` class writes the session ID and the creation time on the browser, as shown in Figure 6.9:



**Figure 6.9: Displaying the Details of the Newly Created Session**

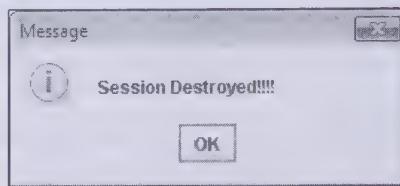
Figure 6.9 displays the session details, such as the session creation time and session ID. Now, if the user clicks the `DestroySession` link, the `SessionListener` class is notified about destroying the servlet. The `User` attribute of this session is removed and the `attributeRemoved()` method is notified about the removal of the attribute.

When the attribute is removed from the session, a message box appears, as shown in Figure 6.10:



**Figure 6.10: Displaying the Attribute Removed Message Box**

After removing the attribute, a message about the invalidation of the session is displayed, as shown in Figure 6.11:



**Figure 6.11: Displaying the Session Destroyed Message Box**

After learning about listeners and how to implement them, let's create an online shopping application.

## Developing the onlineshop Web Application

You should have noticed that when you shop products online, you can add and remove the products from your shopping cart. In this section, you learn to create a sample online shop application named onlineshop using listeners. In this application, you create an onlineshop directory according to the directory structure discussed in Chapter 2, Web Applications and Java EE of this book. Then, you create the products.txt file that contains the details of the available products separated by the | symbol and save the file at C:\JavaEE\Chapter6\onlineshop\WEB-INF location.

Listing 6.11 provides the details of the available products (you can find the products.txt file on the CD in the code\JavaEE\Chapter6\onlineshop\WEB-INF folder):

**Listing 6.11:** Showing the Data for the products.txt File

```
P001|EJB In Simple Steps|14.95
P002|Java EE 5|12.95
P003|Struts Black Book|14.95
P004|C# Project Book|13.95
```

Let's now learn how to create different JavaBeans for the onlineshop Web application.

## Creating the JavaBeans for the onlineshop Web Application

Now, create a JavaBean named Product to set and get the data of various products. Listing 6.12 provides the code for Product JavaBean (you can find the Product.java file on the CD in the code\JavaEE\Chapter6\onlineshop\src\com\kogent\business folder):

**Listing 6.12:** Showing the Code for the Product.java File

```
package com.kogent.business;

import java.io.Serializable;
import java.text.NumberFormat;

public class Product implements Serializable
{
    private String code;
    private String description;
    private double price;

    public Product()
    {
        code = "";
        description = "";
        price = 0;
    }

    public void setCode(String code)
    {
        this.code = code;
    }

    public String getCode()
```

```

{
    return code;
}

public void setDescription(String description)
{
    this.description = description;
}

public String getDescription()
{
    return description;
}

public void setPrice(double price)
{
    this.price = price;
}

public double getPrice()
{
    return price;
}

public String getPriceCurrencyFormat()
{
    NumberFormat currency = NumberFormat.getCurrencyInstance();
    return currency.format(price);
}
}

```

**Listing 6.12:** uses set and get methods to set and retrieve the description, code, and price of each product. The `Product` JavaBean helps in displaying all the available products. When the user picks a single product to add in his cart then the `LineItem` JavaBean sets and gets the details of that product.

**Listing 6.13:** provides the code for the `LineItem` JavaBean (you can find the `LineItem.java` file on the CD in the `code\JavaEE\Chapter6\onlineshop\src\com\kogent\business` folder):

#### **Listing 6.13:** Showing the Code for the `LineItem.java` File

```

package com.kogent.business;

import java.io.Serializable;
import java.text.NumberFormat;

public class LineItem implements Serializable
{
    private Product product;
    private int quantity;

    public LineItem() {}

    public void setProduct(Product p)
    {
        product = p;
    }

    public Product getProduct()
    {
        return product;
    }

    public void setQuantity(int quantity)
    {
        this.quantity = quantity;
    }

    public int getQuantity()
}

```

```

    {
        return quantity;
    }

    public double getTotal()
    {
        double total = product.getPrice() * quantity;
        return total;
    }

    public String getTotalCurrencyFormat()
    {
        NumberFormat currency = NumberFormat.getCurrencyInstance();
        return currency.format(this.getTotal());
    }
}

```

In Listing 6.13, the product details of the selected item are set or get through the instance of the Product JavaBean (Listing 6.12). The LineItem class also calculates the total amount based on the price and quantity of that product. The total amount is then converted into a specified currency format.

Listing 6.14 provides the code for the Cart class to add items into or remove items from the cart (you can find the Cart.java file on the CD in the code\JavaEE\Chapter6\onlineshop\src\com\kogent\business folder):

**Listing 6.14:** Showing the Code for the Cart.java File

```

package com.kogent.business;

import java.io.Serializable;
import java.util.ArrayList;

public class Cart implements Serializable
{
    private ArrayList<LineItem> items;

    public Cart()
    {
        items = new ArrayList<LineItem>();
    }

    public ArrayList<LineItem> getItems()
    {
        return items;
    }

    public int getCount()
    {
        return items.size();
    }

    public void addItem(LineItem item)
    {
        String code = item.getProduct().getCode();
        int quantity = item.getQuantity();
        for (int i = 0; i < items.size(); i++)
        {
            LineItem lineItem = items.get(i);
            if (lineItem.getProduct().getCode().equals(code))
            {
                lineItem.setQuantity(quantity);
                return;
            }
        }
        items.add(item);
    }

    public void removeItem(LineItem item)
}

```

```

    {
        String code = item.getProduct().getCode();
        for (int i = 0; i < items.size(); i++)
        {
            LineItem lineItem = items.get(i);
            if (lineItem.getProduct().getCode().equals(code))
            {
                items.remove(i);
                return;
            }
        }
    }
}

```

As you can see in Listing 6.14, the addItem() method is used to add the items selected by the user to the shopping cart and the removeItem() method is used to remove the selected item from the shopping cart. Now, compile the Java files created in Listing 6.12, 6.13, and 6.14 to create the com.kogent.business package. You also need a Java class to retrieve the product details from a file. Let's create a simple Java class, ProductIO that reads the data from the products.txt file.

Listing 6.15 shows the code for the ProductIO class (you can find the ProductIO.java file on the CD in the code\JavaEE\Chapter6\onlineshop\src\com\kogent\data folder):

**Listing 6.15:** Showing the Code for the ProductIO.java File

```

package com.kogent.data;

import java.io.*;
import java.util.*;
import com.kogent.business.*;

public class ProductIO
{
    public static Product getProduct(String code, String filepath)
    {
        try
        {
            File file = new File(filepath);
            BufferedReader in =
            new BufferedReader(
            new FileReader(file));

            String line = in.readLine();
            while (line != null)
            {
                StringTokenizer t = new StringTokenizer(line, "|");
                String productCode = t.nextToken();
                if (code.equalsIgnoreCase(productCode))
                {
                    String description = t.nextToken();
                    double price =
                    Double.parseDouble(t.nextToken());
                    Product p = new Product();
                    p.setCode(code);
                    p.setDescription(description);
                    p.setPrice(price);
                    in.close();
                    return p;
                }
                line = in.readLine();
            }
            in.close();
        }
    }
}

```

```

        return null;
    }
    catch(IOException e)
    {
        e.printStackTrace();
        return null;
    }
}
public static ArrayList<Product> getProducts(String filepath)
{
    ArrayList<Product> products = new ArrayList<Product>();
    File file = new File(filepath);
    try
    {
        BufferedReader in =
        new BufferedReader(
        new FileReader(file));
        String line = in.readLine();
        while (line != null)
        {
            StringTokenizer t = new StringTokenizer(line, "|");
            String code = t.nextToken();
            String description = t.nextToken();
            String priceAsString = t.nextToken();
            double price = Double.parseDouble(priceAsString);
            Product p = new Product();
            p.setCode(code);
            p.setDescription(description);
            p.setPrice(price);
            String codes = p.getCode();
            products.add(p);
            line = in.readLine();
        }
        in.close();
        return products;
    }
    catch(IOException e)
    {
        e.printStackTrace();
        return null;
    }
}
}

```

In Listing 6.15, the `getProducts()` method loads a file from the specified file path and reads the `products.txt` file. All details of the products are then set in the `ProductIO` class by using the `set` property of the `Product` JavaBean. You should note that when the `ServletContext` is initialized, the `getProducts()` method is invoked through the `CartContextListener` servlet.

After the initialization of the `ServletContext`, the `CartContextListener` servlet sets the attributes required for the application.

Let's now learn to create the `CartContextListener` listener class.

### *Creating the CartContextListener Class*

The `CartContextListener` listener class is notified about the initialization of the context and then the `contextInitialized()` method is invoked. The `CartContextListener` listener class implements the `ServletContextListener` interface.

Listing 6.16 provides the code for the CartContextListener listener class (you can find the CartContextListener.java file on the CD in the code\JavaEE\Chapter6\onlineshop\src\com\kogent\listeners folder):

**Listing 6.16:** Showing the Code for the CartContextListener.java File

```
package com.kogent.listeners;

import javax.servlet.*;
import java.util.*;
import com.kogent.business.*;
import com.kogent.data.*;

public class CartContextListener implements ServletContextListener
{
    public void contextInitialized(ServletContextEvent event)
    {
        ServletContext sc = event.getServletContext();

        // initialize the customer service email address that's used
        // throughout the web site
        String custServEmail = sc.getInitParameter("custServEmail");
        sc.setAttribute("custServEmail", custServEmail);

        // initialize the current year that's used in the copyright notice
        GregorianCalendar currentDate = new GregorianCalendar();
        int currentYear = currentDate.get(Calendar.YEAR);
        sc.setAttribute("currentYear", currentYear);

        // initialize the path for the products text file
        String productsPath = sc.getRealPath("WEB-INF/products.txt");
        sc.setAttribute("productsPath", productsPath);

        // initialize the list of products
        ArrayList<Product> products = new ArrayList<Product>();
        products = ProductIO.getProducts(productsPath);
        sc.setAttribute("products", products);
    }

    public void contextDestroyed(ServletContextEvent event)
    {
    }
}
```

In Listing 6.16, when the context is initialized, the com.kogent.listeners.CartContextListener listener class is notified about the initialization of the context. The CartContextListener listener class sets the custServEmail attribute to the context-param value specified in the web.xml file and the currentYear attribute to the current date. Moreover, the CartContextListener class sets the productsPath attribute to WEB-INF\products.txt. The getProducts() method of the ProductIO class is invoked to return an ArrayList, which is set to the products attribute.

At the time of initializing a ServletContext, four attributes are created, custServEmail, currentYear, productsPath, and products. These attributes can be used as and when required. Now, map the CartContextListener listener class and provide the context-param value in the web.xml file.

Listing 6.17 provides the code to configure the listener and servlet classes in the web.xml file (you can find this file on the CD in the code\JavaEE\Chapter6\onlineshop\WEB-INF folder):

**Listing 6.17:** Showing the Code for the Configuration of Listener and Servlet Classes

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">

<context-param>
```

```

<param-name>custServEmail</param-name>
<param-value>pallavi.sharma@kogentindia.com</param-value>
</context-param>

<listener>
    <listener-class>com.kogent.listeners.CartContextListener</listener-class>
</listener>

<servlet>
    <servlet-name>welcome</servlet-name>
    <servlet-class>com.kogent.servlets.Welcome</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>welcome</servlet-name>
    <url-pattern>/Welcome</url-pattern>
</servlet-mapping>...

<servlet>
    <servlet-name>CartServlet</servlet-name>
    <servlet-class>com.kogent.servlets.CartServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>CartServlet</servlet-name>
    <url-pattern>/cart</url-pattern>
</servlet-mapping>

<servlet>
    <servlet-name>CartNxtServlet</servlet-name>
    <servlet-class>com.kogent.servlets.CartNxtServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>CartNxtServlet</servlet-name>
    <url-pattern>/CartNxtServlet</url-pattern>
</servlet-mapping>

<servlet>
    <servlet-name>Checkout</servlet-name>
    <servlet-class>com.kogent.servlets.Checkout</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>Checkout</servlet-name>
    <url-pattern>/Checkout</url-pattern>
</servlet-mapping>

<welcome-file-list>
    <welcome-file>
        welcome
    </welcome-file>
</welcome-file-list>
</web-app>

```

Listing 6.17 defines `custServEmail` as a context parameter that is assigned a value. The `com.kogent.listeners.CartContextListener` class is also specified as the listener class in Deployment Descriptor. Now, let's design the front end of the onlineshop Web application.

## *Building the Front-end of the onlineshop Web Application*

The front-end of the onlineshop application displays the products available to add in the shopping cart. In the onlineshop Web application, you create servlets, such as `Welcome`, `CartServlet`, `CartNxtServlet`, and `Checkout`. Let's first create the `Welcome` servlet that displays the details of all products by retrieving the value of the `products` attribute. The `products` attribute returns the `ArrayList` in which the description and price of each product is displayed on the browser.

Listing 6.18 shows the code for the `Welcome` servlet class (you can find the `Welcome.java` file on the CD in the `code\JavaEE\Chapter6\onlineshop\src\com\kogent\ servlets` folder):

**Listing 6.18:** Showing the Code for the Welcome.java File

```

package com.kogent.servlets;

import java.io.*;
import java.net.*;
import javax.servlet.*;
import java.util.*;
import javax.servlet.http.*;
import com.kogent.business.Product;
import com.kogent.data.*;

public class welcome extends HttpServlet
{

protected void processRequest(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException
{
    int i = 0;
    String pass =null;
    ServletContext ctx =getServletContext();
    response.setContentType("text/html;charset=UTF-8");

    PrintWriter out = response.getWriter();
    out.println("<html><head><title>Online Shop</title></head><body>");
    out.println("<h1>The onlineshop Application</h1>");
    out.println("<table cellpadding=5 border=1><tr valign=bottom><td align=left><b>Description</b></td><td align=left><b>Price</b></td><td align=left></td></tr>");
    ArrayList<Product> products = new ArrayList<Product>();
    products = (ArrayList<Product>) ctx.getAttribute("products");
    for(i=0; i<products.size(); i++)
    {
        out.println("<tr valign=top>");
        out.println("<td>" + products.get(i).getDescription() + "</td>");
        out.println("<td>" + products.get(i).getPriceCurrencyFormat() + "</td>");
        pass = products.get(i).getCode();
        out.println("<td><a href=./cart?productCode=" + pass + ">Add To Cart</a></td></tr>");
    }
    out.println("</table><br/>");
    out.println("<b> Email ID" + ctx.getAttribute("custServEmail") + "<br/> </b>");
    out.println("</body></html>");
    out.close();
}

@Override
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
{
    processRequest(request, response);
}

@Override
protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
{
    processRequest(request, response);
}

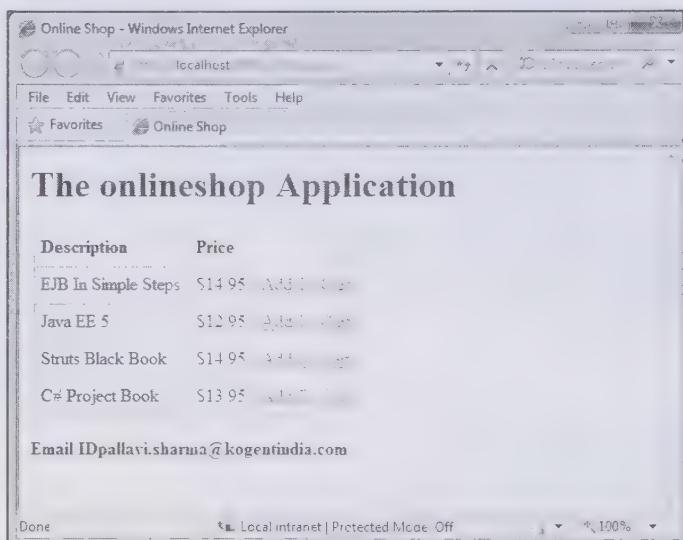
@Override
public String getServletInfo()
{
    return "Short description";
}
// </editor-fold> *
}

```

Listing 6.18 creates a Welcome servlet that retrieves the servlet context and the value of the products attribute. The detail of each product is displayed in the table format and the Add To Cart link is also created for each product, which would add the product in the shopping cart of the user. On clicking the Add To Cart link, the productCode parameter is also passed to the CartServlet servlet class. The productCode parameter contains the code for the product selected by the user to add in the shopping cart. The getAttribute() method also retrieves the value of the custServEmail attribute defined during the initialization of context.

First, you need to compile all the required JavaBeans and servlet classes of the onlineshop application and deploy the application on the Glassfish V3 application server. Next, you can access the <http://localhost:8080/onlineshop> URL to display the first page of the application, i.e., Welcome servlet.

Figure 6.12 displays the Welcome servlet:



**Figure 6.12: Displaying the Products Available in the onlineshop Application**

Figure 6.12 displays the products available in the onlineshop Web application. The description and price of each product is displayed and the user can add the specified product by clicking the link available for each product. When a user clicks the Add To Cart link, the request is forwarded to the Cart Servlet, which creates an HttpSession for the user and retrieves the value of the productCode parameter sent by the Welcome Servlet.

Based on the productCode, the details of that product is retrieved by calling the getProduct() method of the ProductIO class. After that the value for the new attribute named cart is set and the request is dispatched to the CartNxtServlet servlet class.

Listing 6.19 provides the code for the CartServlet servlet (you can find the CartServlet.java file on the CD in the code\JavaEE\Chapter6\onlineshop\src\com\kogent\servlets folder):

**Listing 6.19: Showing the Code for the CartServlet.java File**

```

package com.kogent.servlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import com.kogent.business.*;
import com.kogent.data.*;

public class CartServlet extends HttpServlet
{
protected void processRequest(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException

```

```

    {
        String productCode = request.getParameter("productCode");
        String quantityAsString = request.getParameter("quantity");
        HttpSession session = request.getSession();
        synchronized(session)
        {
            Cart cart = (Cart) session.getAttribute("cart");
            if (cart == null)
            {
                cart = new Cart();
                session.setAttribute("cart", cart);
            }
            //if the user enters a negative or invalid quantity,
            //the quantity is automatically reset to 1.
            int quantity = 1;
            try
            {
                quantity = Integer.parseInt(quantityAsString);
                if (quantity < 0)
                    quantity = 1;
            }
            catch(NumberFormatException nfe)
            {
                quantity = 1;
            }
            ServletContext sc = getServletContext();
            String path = (String) sc.getAttribute("productsPath");
            Product product = ProductIO.getProduct(productCode, path);
            LineItem lineItem = new LineItem();
            LineItem.setProduct(product);
            LineItem.setQuantity(quantity);
            if (quantity > 0)
                cart.addItem(lineItem);
            else if (quantity <= 0)
                cart.removeItem(lineItem);
            session.setAttribute("cart", cart);
            String url = "/CartNxtServlet";
            RequestDispatcher dispatcher =
                getServletContext().getRequestDispatcher(url);
            dispatcher.forward(request, response);
        }
    }
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException
    {
        processRequest(request, response);
    }

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException
    {
        processRequest(request, response);
    }

    @Override
    public String getServletInfo()
    {
        return "Short description";
    }
    // </editor-fold>
}

```

In Listing 6.19, the CartServlet servlet is created. The CartServlet servlet creates a new HttpSession for new users or maintains the session for the existing users. The CartServlet servlet receives the client's request from the Welcome servlet and retrieves the value of the productCode parameter passed with the request. After setting the new attribute cart, the CartServlet servlet forwards the request to the CartNxtServlet servlet class.

However, if the value of the quantity variable is less than or equal to zero, the removeItem() method of the com.kogent.business.Cart class is invoked; however, if the quantity is greater than zero, the addItem() method is called. The RequestDispatcher class is used to forward the request to the CartNxtServlet servlet class.

Listing 6.20 provides the code for CartNxtServlet (you can find the CartNxtServlet.java file on the CD in the code\JavaEE\Chapter6\onlineshop\src\com\kogent\servlets folder):

**Listing 6.20:** Showing the Code for the CartNxtServlet.java File

```
package com.kogent.servlets;
import java.io.*;
import java.net.*;
import javax.servlet.*;
import javax.servlet.http.*;
import com.kogent.business.*;
import java.util.*;
public class CartNxtServlet extends HttpServlet
{
    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
    {
        HttpSession session = request.getSession();
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        out.println("<html><head><title>Online Shop</title></head><body>");
        out.println("<h1>Items in Shopping Cart </h1>");
        Cart carts = (Cart) session.getAttribute("cart");
        ArrayList<LineItem> items = new ArrayList<LineItem>();
        items= carts.getItems();
        out.println("<table border=1 cellpadding=5><tr><th>Quantity</th><th>Description</th><th>Price</th><th>Amount</th></tr>");
        for ( int i=0; i<items.size(); i++)
        {
            out.println("<tr valign=top><td><form action=./cart method=post><input type=hidden name=productCode value=" + items.get(i).getProduct().getCode() + ">");
            out.println("<input type=text size=2 name=quantity value=" + items.get(i).getQuantity() + ">");
            out.println("<input type=submit name=updateButton value=Update></form></td>");
            out.println("<td>" + items.get(i).getProduct().getDescription() + "</td>");
            out.println("<td>" + items.get(i).getProduct().getPrice() + "</td>");
            out.println("<td>" + items.get(i).getTotalCurrencyFormat() + "</td>");
            out.println("<td><form action=./cart method=post><input type=hidden name=productCode value=" + items.get(i).getProduct().getCode() + ">"); 
            out.println("<input type=hidden name=quantity value=0><input type=submit value=RemoveItem></form></td></tr>");
        }
        out.println("</table>");
        out.println("<br><form action=./welcome method=post><input type=submit name=continue value=ContinueShopping></form>");
        out.println("<br><form action=./Checkout method=post><input type=submit name=Logoff value=Checkout></form>");
        out.println("</body></html>");
        out.close();
    }
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
    { }
```

```

    {
        processRequest(request, response);
    }

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException
    {
        processRequest(request, response);
    }

    @Override
    public String getServletInfo()
    {
        return "Short description";
    }
}

```

The code in Listing 6.20 creates the `CartNxtServlet` servlet class, which contains the details of the products added by the user in the shopping cart. The quantity of each product can be updated by the user by clicking the Update button. The amount of the items is also updated based on the price and the quantity of the products selected. The various hidden form fields are used in the `CartNxtServlet` servlet class to send the product ID and quantity (selected by the user) as hidden values with the client's request. If the user wishes to remove any item from his cart, the Remove Item button can be clicked. The `CartServlet` servlet class then invokes the `removeItem()` method as the quantity parameter contains the value, zero.

The various items that you can shop are shown in Figure 6.12, you click the Add to Cart link to add the item in the shopping cart. Once you have clicked the link, the items that are available in your shopping cart are displayed. If you want to further shop for more items, click the ContinueShopping button. In our case, we have added three items in the shopping cart, as shown in Figure 6.13:

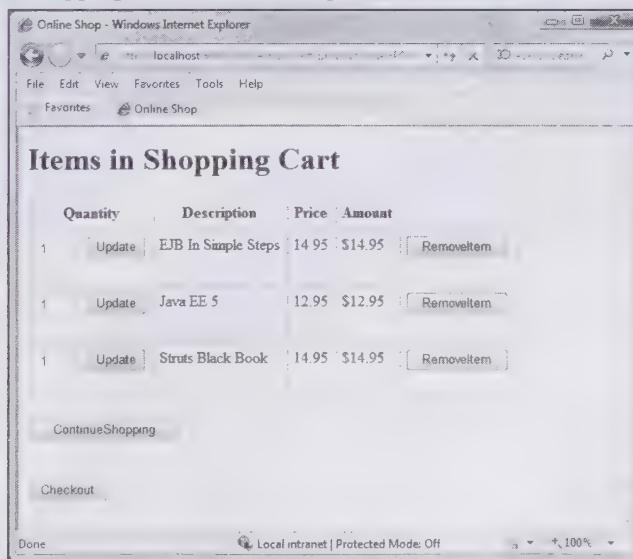


Figure 6.13: Displaying Items in the Online Shopping Cart

If you want to select more items, you can click the ContinueShopping button that again forwards the request to the Welcome servlet. Figure 6.13 displays the shopping cart of the user who has added three products in his cart. Once the user clicks the Checkout button, the request is forwarded to the Checkout servlet and the session is terminated. The Checkout servlet invalidates the session and prompts the user to close the browser window to end the shopping.

Listing 6.21 provides the code for the Checkout servlet (you can find the `Checkout.java` file on the CD in the `code\JavaEE\Chapter6\onlineshop\src\com\kogent\servlets` folder):

**Listing 6.21:** Showing the Code for the Checkout.java File

```

package com.kogent.servlets;
import java.io.*;
import java.net.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class Checkout extends HttpServlet
{
    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
    {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        out.println("<html><head><title>Online Shop</title></head><body>");
        out.println("<h1>The onlineshop Application</h1>");
        HttpSession session = request.getSession();
        session.invalidate();
        out.println("click close to terminate the online shopping");
        out.close();
    }

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException
    {
        processRequest(request, response);
    }

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException
    {
        processRequest(request, response);
    }

    @Override
    public String getServletInfo()
    {
        return "Short description";
    }
    // </editor-fold>
}

```

The servlets created in Listing 6.18, 6.19, 6.20, and 6.21 are provided under the `com.kogent.servlets` package. Save all the servlets and compile them. Prior to the creation of the WAR file, it is essential to map all these servlets in the web.xml file. After mapping all the servlets, create the `onlineshop.war` file and deploy this Web application on the Glassfish application server. Browse the `http://localhost:8080/onlineshop` URL to run the application.]

Let's now learn about wrappers in the next section.

## Introducing Wrappers

One of the features in Java Servlet 3.0 specification is the request and response wrappers. Wrappers are sandwiched between servlet classes and servlet containers and can easily handle the servlet environment by wrapping the servlet's APIs. The core objects of the servlet API are `ServletRequest` as well as `ServletResponse` and the core Hypertext Transfer Protocol (HTTP) specific objects are `HttpServletRequest` and `HttpServletResponse`. These core objects of a servlet cannot be used directly; therefore, wrappers are used to wrap these objects. This section introduces wrapper API and helps you to understand how the interaction with servlets is performed by using wrapper classes.

Let's first discuss about the need of wrappers in detail.

## Exploring the Need for Wrappers

Wrappers allow you to add additional functionalities to the request and response object given by the servlet container. Wrappers allow you to change the data of a user's request as well as add some additional information to that request before sending the data to a servlet or a (JavaServer Pages) JSP page. In addition, wrappers help you to perform some pre/post operations for the calls made on the request or response objects. Moreover, with the help of wrappers, you can change the functionality of the predefined methods on request and response objects with their new APIs. To override a functionality of a predefined method, wrapper objects passed as arguments to calling servlet should be the same as request and response objects of the called servlet.

The following are some situations in which wrappers can be used:

- Making response object capable to provide output in the form of DOM objects on client
- Checking the content type while setting it in response
- Caching the output printed to the client and then flushing it after the entire response is generated

Filters can also use wrappers to wrap a request or response so that it can change the behavior of the request or response to perform some filtering tasks.

## Exploring the Types of Wrapper Classes

The Java Servlet API 3.0 provides the following four wrapper classes:

- javax.servlet.ServletRequestWrapper
- javax.servlet.ServletResponseWrapper
- javax.servlet.http.HttpServletRequestWrapper
- javax.servlet.http.HttpServletResponseWrapper

These classes implement Decorator Pattern, which is a core design pattern that is used to add additional functionality to a particular object. All these four classes are public non-abstract classes provided with one constructor that takes an argument of their respective wrapped type. The `ServletRequestWrapper` class accepts `ServletRequest` as an argument and the `ServletResponseWrapper` class accepts `ServletResponse` as an argument.

### Explaining the `ServletRequestWrapper` Class

The `ServletRequestWrapper` class implements the `ServletRequest` interface. This class is further extended by Web developers to call their overridden methods.

Table 6.7 describes some of the commonly-used methods of the `ServletRequestWrapper` class:

**Table 6.7: Noteworthy Methods of the `ServletRequestWrapper` Class**

Method	Default Behavior
<code>getRequest</code>	Returns the wrapped request object
<code>getAttribute</code>	Calls the <code>getAttribute()</code> request method on the wrapped request object
<code>getParameter</code>	Calls the <code>getParameter()</code> request method on the wrapped request object
<code>getParameterValues</code>	Calls the <code>getParameterValues()</code> request method on the wrapped request object
<code>getRequestDispatcher</code>	Calls the <code>getRequestDispatcher()</code> request method on the wrapped request object
<code>removeAttribute</code>	Calls the <code>getRequestDispatcher()</code> request method on the wrapped request object
<code>setAttribute</code>	Calls the <code>setAttribute()</code> request method on the wrapped request object

### Explaining the `ServletResponseWrapper` Class

The `ServletResponseWrapper` class implements the `ServletResponse` interface. This class is further extended by Web developers to define their overridden methods.

Table 6.8 describes some of the commonly-used methods of the `ServletResponseWrapper` class:

**Table 6.8: Noteworthy Methods of the ServletResponseWrapper Class**

<b>Method</b>	<b>Default Behavior</b>
getBufferSize	Calls the getBufferSize() response method on wrapped response object
getResponse	Returns the wrapped ServletResponse object
getWriter	Calls the getWriter() response method on the wrapped response object
setContentLength	Calls the setContentLength() response method on the wrapped response object
setContentType	Calls the setContentType() response method on the wrapped response object

### Explaining the HttpServletRequestWrapper Class

The HttpServletRequestWrapper class implements the HttpServletRequest interface. This class is further extended by Web developers to define their overridden methods.

Table 6.9 describes some of the commonly-used methods of the HttpServletRequestWrapper class:

**Table 6.9: Noteworthy Methods of the HttpServletRequestWrapper Class**

<b>Method</b>	<b>Default Behavior</b>
getCookies	Calls the getCookies() request method on the wrapped request object
getHeaders	Calls the getHeaders() request method on the wrapped request object
getMethod	Calls the getMethod() request method on the wrapped request object
getPathInfo	Calls the getPathInfo() request method on the wrapped request object
getSession	Calls the getSession() request method on the wrapped request object
getRequestURI	Calls the getRequestURI() request method on the wrapped request object

### Explaining the HttpServletResponseWrapper Class

The HttpServletResponseWrapper class implements the HttpServletResponse interface. This class is further extended by Web developers to define their overridden methods.

Table 6.10 describes some of the commonly-used methods of the HttpServletResponseWrapper class:

**Table 6.10: Noteworthy Methods of the HttpServletResponseWrapper Class**

<b>Method</b>	<b>Default Behavior</b>
addCookie	Calls the addCookie() response method on the wrapped response object
encodeRedirectURL	Calls the encodeRedirectURL() response method on the wrapped response object
encodeURL	Calls the encodeURL() response method on the wrapped response object
sendRedirect	Calls the sendRedirect() response method on the wrapped response object
setHeader	Calls the setHeader() response method on the wrapped response object

## Working with Wrappers

In this section, let's discuss the use of wrappers by creating a Web application, named `wrapper`. When a user accesses the `wrapper` Web application, the `request` parameter is not sent in the request Uniform Resource Identifier (URI); therefore, the `getParameter()` method must return null. Instead of returning null as the value of the parameter, you can customize a servlet to return the `None` string with the help of a wrapper. The `wrapper` Web application also checks whether or not the content type set for the response object is a supported type.

Let's now create the required files for the `wrapper` Web application.

## Creating the Home HTML Page

The Home HTML page consists of a form that has two buttons to send the request with and without wrappers. The user can submit the request by clicking any of these two buttons. One of the buttons sends the request to the WrapperTestServlet servlet, which processes the request using wrappers. The other button sends the request to the TestServlet servlet, which processes the request without using wrappers.

Listing 6.22 shows the code for the Home.html file (you can find the file on the CD in the code\Chapter6\wrapper\ folder):

**Listing 6.22:** Showing the Code for the Home HTML Page

```
<html>
<body>
    <form method=post action="WrapperTestServlet">
        <input type="submit" value="Send Request using
        Wrappers"/><BR><BR>
    </form>
    <form method=post action="TestServlet">
        <input type="submit" value="Send Request without using
        Wrappers"/>
    </form>
</body>
</html>
```

Listing 6.22 shows that the request from the Home HTML page is sent to the WrapperTestServlet class. Let's now create the WrapperTestServlet class to retrieve the value of the parameter passed along with the user's request.

## Creating the WrapperTestServlet.java File

The WrapperTestServlet class retrieves the request parameter that is not sent by the user and then prints the value of the request parameter using the out.println() method. The WrapperTestServlet class uses wrappers and calls the overridden methods, namely getParameter() and setContentType(), of the request and response objects, respectively.

Listing 6.23 shows the code for the WrapperTestServlet.java file (you can find this file on the CD in the code\Chapter6\wrapper\src\com\kogent\servlet folder):

**Listing 6.23:** Showing the Code for the WrapperTestServlet.java File

```
package com.kogent.servlet;

import javax.servlet.*;
import java.io.*;
import com.kogent.mywrappers.*;

public class WrapperTestServlet extends GenericServlet
{
    public void service (ServletRequest req, ServletResponse res)
    throws ServletException, IOException
    {
        MyRequestWrapper mreq = new MyRequestWrapper(req);
        MyResponseWrapper mres = new MyResponseWrapper(res);

        String testparam=mreq.getParameter("myparam");
        mres.setContentType ("text/xml");

        //These methods is invoked just to test
        //whether the wrapper functionality is working or not
        PrintWriter out=res.getWriter ();
        out.println ("<html><body><b>");
        out.println ("Parameter value is "+ testparam);
        out.println ("</b></body></html>");

    } //service
} //class
```

In Listing 6.23, the instances of the MyRequestWrapper and MyResponseWrappper classes are created to modify the current request and response objects.

Let's now learn to create the TestServlet.java file.

### *Creating the TestServlet.java File*

The TestServlet class retrieves the request parameter that is not sent by the user and then prints the value of this parameter using the `out.println()` method. Listing 6.24 shows the code for the `TestServlet.java` file (you can find this file on the CD in the `code\Chapter6\wrapper\src\com\kogent\servlet` folder):

**Listing 6.24:** Showing the Code for the `TestServlet.java` File

```
package com.kogent.servlet;
import javax.servlet.*;
import java.io.*;

public class TestServlet extends GenericServlet
{
    public void service (ServletRequest req, ServletResponse res)
    throws ServletException, IOException
    {
        String testparam=req.getParameter("myparam");
        res.setContentType ("text/xml");

        //These methods is invoked just to test
        //whether the wrapper functionality is working or not
        PrintWriter out=res.getWriter ();
        out.println ("<html><body><b>");
        out.println ("Parameter value is "+ testparam);
        out.println ("</b></body></html>");
    }
}//service
}//class
```

In Listing 6.24, the value of the `myparam` attribute is retrieved and displayed.

Let's now learn to create the `MyRequestWrapper.java` file.

### *Creating the `MyRequestWrapper.java` File*

The `MyRequestWrapper` class extends the `ServletRequestWrapper` class to get the permission to override its methods. The `MyRequestWrapper` class overrides the `getParameter()` method of the `ServletRequestWrapper` class. The `getParameter()` method of the `MyRequestWrapper` class first calls the `getParameter()` method of the `ServletRequestWrapper` super class, which returns the value of the `myparam` attribute and assigns it to string `s1`. If request parameter or `s1` is found to be null, it sets `s1` to `None String` and displays it to the user. In this way, the `getParameter()` method of the `ServletRequestWrapper` class is overridden; otherwise, this method returns the null value.

Listing 6.25 shows the code for the `MyRequestWrapper.java` file (you can find this file on the CD in the `code\Chapter6\wrapper\src\com\kogent\mywrappers` folder):

**Listing 6.25:** Showing the Code for the `MyRequestWrapper.java` File

```
package com.kogent.mywrappers;
import javax.servlet.*;

public class MyRequestWrapper extends ServletRequestWrapper
{
    public MyRequestWrapper (ServletRequest req)
    {
        super(req);
    }

    public String getParameter(String s)
    {
        String s1=super.getParameter(s);
```

```

        if(s1==null)
            s1="None";
        return s1;
    }//getParameter
}//class

```

Let's now create the MyResponseWrapper.java file.

## *Creating the MyResponseWrapper.java File*

The MyResponseWrapper class extends the ServletResponseWrapper class to get the permission to override the methods of the ServletResponseWrapper class. This class overrides the setContentType() method of the ServletRequestWrapper class. The setContentType() method of the MyResponseWrapper class sets the content type of response to text/html using the setContentType() method of the ServletResponseWrapper super class.

Listing 6.26 provides the code for the MyResponseWrapper.java file (you can find the file on the CD in the code\Chapter6\wrapper\src\com\kogent\mywrappers folder):

**Listing 6.26:** Showing the Code for the MyResponseWrapper.java File

```

package com.kogent.mywrappers;

import javax.servlet.*;

public class MyResponseWrapper extends ServletResponseWrapper
{
    public MyResponseWrapper (ServletResponse res)
    {
        super(res);
    }

    public void setContentType(String s)
    {
        if (!s.equals("text/html"))
            s="text/html";
        /*
           Here we consider that our application is designed intelligent
           to provide only html response content
           If we want different types also to be allowed then we need
           to check accordingly
        */
        super.setContentType(s);
    }//setContentType
}//class

```

## *Creating the web.xml File*

The web.xml file provides the mapping for the TestServlet servlet with the /testSer URI.

Listing 6.27 provides the code for the web.xml file (you can find the file on the CD code\Chapter6\wrapper\WEB-INF folder):

**Listing 6.27:** Showing the Code for the web.xml File

```

<?xml version="1.0" encoding="UTF-8"?>

<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">

    <servlet>
        <servlet-name>WrapperTestServlet</servlet-name>
        <servlet-class>com.kogent.servlet.wrapperTestServlet</servlet-class>
    </servlet>

```

```

<servlet-mapping>
    <servlet-name>WrapperTestServlet</servlet-name>
    <url-pattern>/WrapperTestServlet</url-pattern>
</servlet-mapping>

<servlet>
    <servlet-name>TestServlet</servlet-name>
    <servlet-class>com.kogent.servlet.TestServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>TestServlet</servlet-name>
    <url-pattern>/TestServlet</url-pattern>
</servlet-mapping>

</web-app>

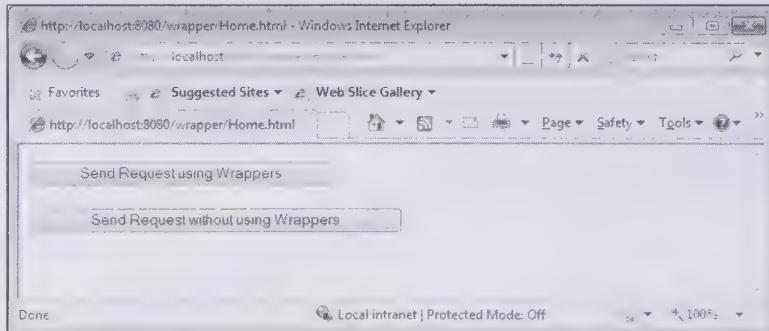
```

After creating the web.xml file, let's now learn to package, deploy, and run the wrapper Web application.

## Packaging, Deploying, and Running the wrapper Web Application

To package the wrapper Web application, you first need to compile all the .java files so that the .class files are generated in the classes directory of the Web application. Next, create the wrapper.war file and deploy the wrapper Web application on the Glassfish application server. Now, browse the <http://localhost:8080/wrapper/Home.html> URL to run the application.

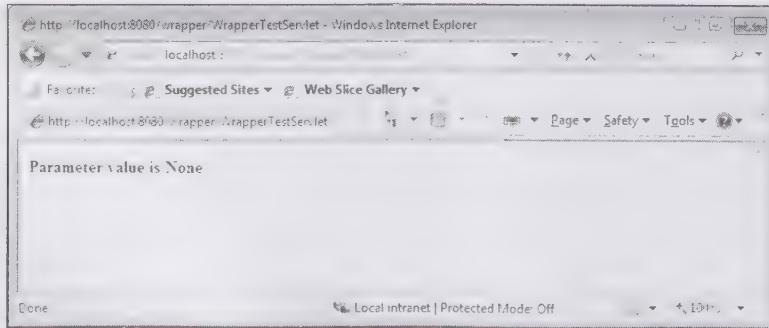
Figure 6.14 shows the output of the wrapper Web application:



**Figure 6.14: Showing the Output of the Home HTML Page**

In Figure 6.14, when you click the Send Request using Wrappers button, the client request is forwarded to the WrapperTestServlet servlet.

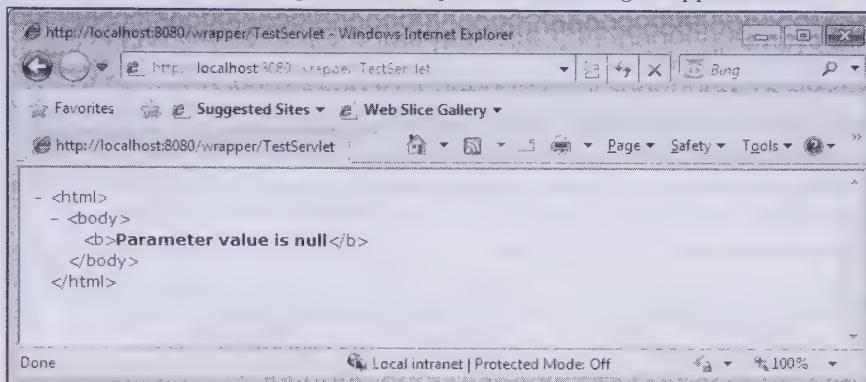
Figure 6.15 shows the output of the WrapperTestServlet servlet:



**Figure 6.15: Displaying the Output when Send Request using Wrappers Button is Clicked**

When you click the Send Request without using Wrappers button, the client request is forwarded to the TestServlet servlet.

Figure 6.16 shows the output on clicking the Send Request without using Wrappers button:



**Figure 6.16: Displaying the Output when Send Request without using Wrappers Button is Clicked**

This completes the discussion about wrappers. Let's now quickly summarize the topics covered in the chapter.

## Summary

This chapter has described about events and event handling process. In addition, you have learned about servlet and session level events that can occur in a Web application and how listeners are notified about an event. Next, you learn to implement the event handling process by creating an onlineshop Web application. The chapter discussed the concept of wrappers and how to implement them in Java Servlet 3.0.

After understanding about event handling and wrappers in servlets, let's learn about JSP and its role in Web applications in the next chapter.

## Quick Revise

**Q1. List the events that can occur during the life cycle of a servlet.**

Ans. The following events can occur during the life cycle of a servlet:

- Initializing servlets
- Adding, removing, or replacing attributes in ServletContext
- Creating, activating, passivating, or invalidating a session
- Adding, removing, or replacing attributes in servlet session
- Destroying servlets

**Q2. What is event handling?**

Ans. Event handling is a concept that is used to handle the life cycle events, such as initializing a servlet, servicing a client request, and destroying an instance of the servlet.

**Q3. What are the various levels of servlet events?**

Ans. The various levels of servlet events are as follows:

- Request level events
- Servlet context level events
- Servlet session events

**Q4. List the listener interfaces used to handle servlet context life cycle events.**

Ans. The servlet context life cycle events can be handled by using the following interfaces:

- ServletContextListener
- ServletContextAttributeListener

**Q5. List the listener interfaces used to handle servlet session level events.**

Ans. The following listener interfaces are used for handling servlet session events:

- HttpSessionListener
- HttpSessionActivationListener
- HttpSessionAttributeListener

**Q6. Explain the methods defined in the ServletContextListener interface.**

Ans. The methods of the ServletContextListener interface are as follows:

- contextInitialized**—Notifies that the initialization process for the Web application is going to start. During the notification process, a notification about context initialization is sent to all the servlet context listeners before initializing any filter or servlet in the Web application.
- contextDestroyed**—Notifies that the ServletContext is going to shut down. All servlets and filters should be destroyed prior to this notification about the ServletContext destruction.

**Q7. Which events are monitored by the ServletContextAttributeListener interface?**

Ans. The ServletContextAttributeListener interface monitors the following events:

- Adding an attribute in a ServletContext
- Replacing an attribute in a ServletContext
- Removing an attribute from a ServletContext

**Q8. Explain the methods of the ServletContextAttributeListener interface.**

Ans. The methods of the ServletContextAttributeListener interface are as follows:

- attributeAdded**—Notifies about the addition of a new attribute to the ServletContext. It is called just after the addition of the attribute.
- attributeRemoved**—Notifies about the removal of an existing attribute from the ServletContext. It is called just after the removal of the attribute.
- attributeReplaced**—Notifies about the replacement of an attribute on the ServletContext. It is called just after the replacement of an attribute.

**Q9. List the various servlet session level events that can occur during a session.**

Ans. The various servlet session level events that can occur during a session are as follows:

- Creating a session
- Destroying a session
- Adding, removing, or replacing the attributes of a session

**Q10. Explain the methods of the HttpSessionBindingEvent class.**

Ans. The methods of the HttpSessionBindingEvent class are as follows:

- getSession**—Retrieves the session object. It overrides the getSession() method of the HttpSessionEvent class.
- getName**—Retrieves the name with which an attribute is bound to or unbound from the session.
- getValue**—Retrieves the value of an added, removed, or replaced attribute.

# 7

# Working with JavaServer Pages (JSP) 2.1

## If you need an information on:

## See page:

Introducing JSP Technology	276
Exploring New Features of JSP 2.1	276
Listing Advantages of JSP over Java Servlet	277
Exploring the Architecture of a JSP Page	277
Describing the Life Cycle of a JSP Page	278
Working with JSP Basic Tags and Implicit Objects	280
Working with Action Tags in JSP	297
Exploring the JSP Unified EL	310
Using Functions with EL	323

JSP stands for Java Server Pages, which is a Java-based technology developed by SUN Microsystems to simplify the development of dynamic Web pages. JSP pages provide a means of separating the presentation logic from the business logic and help to directly embed Java code in static Hyper Text Markup Language (HTML) pages. The presentation logic is provided by HTML tags and the business logic for dynamic content is handled by JSP tags. JSP is a server-side technology and JSP pages are processed on a Web server in the JSP Servlet engine. When a client requests for a JSP page, the JSP Servlet engine first processes the requested page to generate the HTML code for dynamic content, then compiles the page into a servlet, and then executes the resulting servlet to generate an output to the client.

This chapter begins by providing an introduction to the JSP technology. Then, you learn about the new features introduced in JSP 2.1, which is the latest version of the JSP technology. The chapter also discusses the advantages of JSP over Java Servlet and explores the different architectures of JSP pages. Apart from this, the chapter explains the life cycle of a JSP page and also explores the various JSP basic tags, implicit objects, and action tags. Toward the end, the chapter discusses JSP unified Expression Language (EL) and the functions used with EL.

So let's start the chapter with a brief overview about the JSP technology.

## Introducing JSP Technology

JSP is a Java-based Web application development technology and serves as an advancement of the Java Servlet technology. Before the introduction of JSP pages, servlets were used by Java programmers to provide dynamic content to clients. In case of servlets, both the processing of requests and the generation of responses were handled by a single servlet class. Servlets usually contain HTML code embedded in Java code. Therefore, programmers who created servlets needed a thorough knowledge of Java programming and basic knowledge about HTML programming. In addition, servlets do not separate the presentation logic from the business logic in an application.

JSP greatly simplifies the process of creating dynamic Web pages. In JSP, the Java code is embedded within the HTML code to provide dynamic content to a client. There is a clear separation between the presentation logic, which is provided by HTML tags and the business logic, which is handled by embedded Java code. This separation allows you to divide the tasks of providing code for presentation and business logic in an application between designers and programmers having different skills. This implies that, in case of JSP, designers can implement the presentation logic code and the user interface in HTML, and Java programmers can later add the code for dynamic content to implement the business logic. As far as their execution is concerned, JSP pages provide all the benefits of servlets. Apart from this, JSP pages are easier to write than servlets. JSP pages are translated to a servlet during compilation, and the response is sent to a client by generating dynamic content in the form of the servlet.

Various versions of JSP have been released since the development of the JSP specification version 1.2. The latest version of JSP is JSP version 2.1. With the release of every version, several new features have been introduced. Let's now explore the new features that are introduced in JSP 2.1.

## Exploring New Features of JSP 2.1

Various new features have been introduced in JSP 2.1. This version includes Java Standard Tag Library (JSTL) and JavaServer Faces (JSF). Apart from this, new EL syntax is introduced in JSP 2.1, which allows deferred evaluation of the EL expressions. Moreover, EL expressions can now be used to set and retrieve data, and invoke methods. In addition, in JSP 2.1, you can use annotations to configure and include resources and environment data, according to your requirements. The introduction of annotations removes the need for writing entries in Deployment Descriptor files. The enhancement of the EL in JSP 2.1 now provides complete alignment with the JSF technology. Apart from this, JSP 2.1 introduces the concept of qualified functions, which is preferable to using the ternary operator. JSP 2.1 also includes support for literal expressions and enumerations in EL, and the Property Resolver Application Programming Interface (API) is also enhanced. Moreover, JavaBean methods can now be referenced by using EL and invoked by using the Method Binding API.

With this brief overview of the features introduced in JSP 2.1, let's quickly move on and discuss the advantages of JSP over Java Servlet.

## Listing Advantages of JSP over Java Servlet

JSP offers various advantages over Java Servlet. It provides a better server-side scripting support as compared to Java Servlet. Java programmers can create servlets easily, but handling Java code is always a tough job for Web page designers. As stated earlier, a JSP page has HTML code, and some Java code embedded with the HTML code. This makes creating a Web page easy as the basic designing of the page is done by using the HTML code. You can use the Java code to add dynamic content to the page. The Java code is embedded by using different JSP constructs. The following are some reasons to prefer JSP over Java Servlet:

- ❑ Allows you to place static code and components symmetrically in a Web page. These static Web pages are designed by Web designers by using HTML code. Designing static Web pages by using servlets requires a good knowledge of Java, but Web designers may not be comfortable with Java programming constructs. In addition, while using servlets, you have to enclose HTML code in the `out.println()` method, which disables all Integrated Development Environment (IDE) support for generating HTML content. Therefore, using JSP is always easier than using servlets for the same HTML output.
- ❑ Facilitates automatic recompilation of a JSP page by a Web container for any update in the code; whereas, you need to recompile your servlet for every single change in the source code of the servlet.
- ❑ Allows you to access a JSP page directly as a simple HTML page; whereas, servlets cannot be accessed directly and have to be first mapped in the `web.xml` file.
- ❑ Allows you to create a JSP page by using a simple HTML template and the JSP page is automatically handled by the JSP container. However, in servlets you need to provide the Java code to generate dynamic HTML pages.

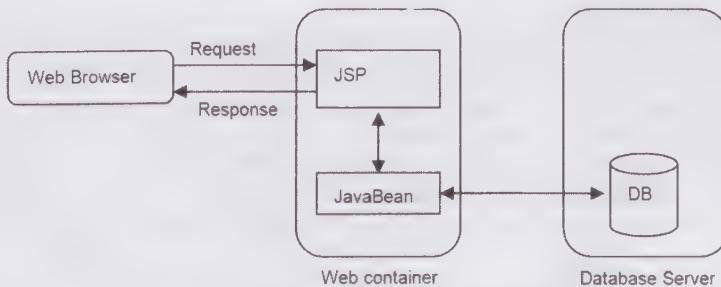
These advantages of JSP make it a popular technology to use in the presentation logic of any Web application. After discussing the advantages of JSP over Java Servlet, let's explore the various architectures of a JSP page in the next section.

## Exploring the Architecture of a JSP Page

According to JSP specifications, there are two approaches to build Web applications. These approaches are known as the JSP Model 1 and JSP Model 2 architectures, respectively. Let's discuss these models in detail in the following sections.

### The JSP Model I Architecture

In the JSP Model I architecture, a Web browser directly accesses JSP pages of a Web container. These JSP pages interact with JavaBeans in an application. When a client sends a request from a JSP page, the response (i.e., a JSP page or servlet) is sent back to the client depending on which hyperlinks are selected or which request parameters are invoked by the client request. If the generated response requires accessing a database, the JSP page uses a JavaBean, which retrieves the required data from the database. Figure 7.1 shows the architecture of JSP Model I:

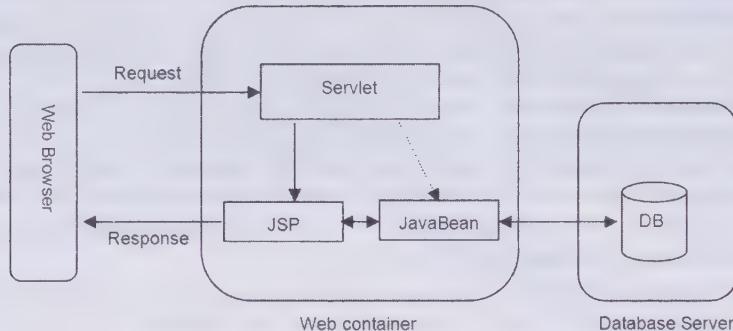


**Figure 7.1: Showing the JSP Model I Architecture**

Let's now discuss the JSP Model II architecture.

## The JSP Model II Architecture

In the JSP Model II architecture, servlets are used by a Web browser to communicate with a JSP page. The servlets are positioned between the Web browser and the JSP pages, and they act as a controller to dispatch requests to the next requested JSP page based on the requested Uniform Resource Locator (URL) and input parameters. The servlet also creates JavaBean instances, if they are required by the JSP page to store data. The Controller servlet then decides which JSP page to forward to as a response, based on the client request. A JavaBean invokes the database server if data is needed from the server. Figure 7.2 shows the architecture of JSP Model II:



**Figure 7.2: Showing the JSP Model II Architecture**

Now that you are familiar with JSP architecture, let's discuss the life cycle of a JSP page in the next section.

## Describing the Life Cycle of a JSP Page

A JSP page is not executed directly. It follows a well defined process that first involves the translation of the JSP page into its corresponding servlet and then the compilation of the source file of the servlet into a class file. The resulting translated servlet is then loaded into memory and initialized, similar to any other servlet. Every time a JSP page is requested, its corresponding servlet is executed. Therefore, most of the stages of the JSP life cycle are similar to that of a servlet. The major stages of the life cycle of a JSP are as follows:

- ❑ The page translation stage
- ❑ The compilation stage
- ❑ The loading & initialization stage
- ❑ The request handling stage
- ❑ The destroying stage (end of service)

Let's now discuss each stage of the JSP life cycle one by one in the following sections.

### The Page Translation Stage

In the page translation stage of the JSP life cycle, a Web container converts the JSP document into an equivalent Java code, that is, a servlet. Usually, a servlet contains Java code with markup language tags, such as HTML or Extensible Markup Language (XML) embedded within the Java code. JSPs, on the other hand, consist primarily of markup language with some Java code embedded in between the markup tags. Therefore, the objective of page translation is to convert a document chiefly consisting of HTML/XML code, to one that has more of Java code, which can be executed by a Java Virtual Machine (JVM).

In the page translation stage, the Web container performs the following operations:

- ❑ Locates the requested JSP page
- ❑ Validates the syntactic correctness of the JSP page
- ❑ Interprets the standard JSP directives, actions, and custom actions used in the JSP page
- ❑ Writes the source code of the equivalent servlet for the JSP page

After a JSP page is translated into a servlet, all the requests for that JSP page are served by its corresponding servlet class. This servlet class works similar to any other simple servlet. The translation of JSP is done automatically by the Web server. JSP page translation occurs only when necessary, that is, at some point before the page has to serve its first request. This process is also performed when the JSP source code is updated and the page is redeployed. A Web container can decide when the translation should occur. The two possible instances where a translation can occur are as follows:

- When a first-time user requests a JSP page.
- When a JSP page is loaded into a Web container. This is also called JSP pre-compilation.

If there is any error in the translation of the page, the container raises the 500 (internal server error) exception and if a change occurs in the JSP page, all the stages of the JSP life cycle are executed again.

## The Compilation Stage

The page translation stage is followed by the compilation stage of the JSP life cycle. In the compilation stage, the Java source code for the corresponding servlet is compiled by the JSP container. The container converts the source code into Java byte (class) code. After the class file is generated, the container decides to either discard the code or retain it for debugging. Generally, most containers discard the generated Java source code by default. After the compilation stage, the servlet is ready to be loaded and initialized.

## The Loading & Initialization Stage

During the loading and initialization stage of the JSP life cycle, the JSP container loads and instantiates the servlet that has been generated and compiled in the translation and compilation stages, respectively. The JSP container, as part of the loading and initialization process, performs the following operations:

- Loading**—Loads the servlet generated during the translation and compilation stages. Normal Java class loading options are used to load the servlet. The loading process is terminated if the Web container fails to load the servlet class.
- Instantiation**—Creates an instance of a servlet class. To create a new instance of the generated servlet, the JSP container uses the no-argument constructor. The JSP translator (part of the JSP container) is responsible for including a no-argument constructor in the servlet generated after the translation of the JSP page.
- Initialization**—Initializes the instantiated object after successful instantiation of the JSP equivalent servlet object. As per the servlet's life cycle, the container initializes the object by invoking the `init(ServletConfig)` method. This method is implemented by the container by calling the `jspInit()` method. This indicates that the `jspInit()` method acts as the `init(ServletConfig)` method of the servlet.

If the loading and initialization stage is performed successfully, the Web container activates the servlet object and makes it available and ready to handle client requests.

### NOTE

A JSP page equivalent servlet instance put into service by a container may not handle any request in its lifetime.

The next stage in the JSP life cycle is the request handling stage.

## The Request Handling Stage

During the request handling stage of the JSP life cycle, only those objects of a JSP equivalent servlet that are initialized successfully are used by the Web container to handle client requests. The container performs the following operations in the request handling stage:

- Creates the `ServletRequest` and `ServletResponse` objects. If a client uses the HTTP protocol to make a request, the container creates the `HttpServletRequest` and `HttpServletResponse` objects, where the request object corresponds to the client request. In other words, the request data and client information can be retrieved by a servlet by using the request object. The response object can be used by the servlet to generate the response to a client.

- ❑ Passes the request and response objects created in the preceding step to the servlet object to invoke the `service()` method. In the case of a JSP equivalent servlet, the container invokes the `_jspService()` method.

Let's now look at the final stage of the JSP life cycle, that is, the destroying stage.

## **The Destroying Stage**

If a servlet container decides to destroy a servlet instance of a JSP page, the container needs to end the services provided by the instance. The servlet container performs the following operations to destroy the servlet instance:

- ❑ Allows all the currently running threads in the `service()` method of the servlet instance to complete their operations. Meanwhile, the servlet container makes the servlet instance unavailable for new requests.
- ❑ Allows the servlet container to invoke the `destroy()` method on the servlet instance, after the current threads have completed their operations on the `service()` method. The `destroy()` method leads to the invocation of the `jspDestroy()` method.
- ❑ Releases all the references of the servlet instance and renders them for garbage collection, after the `destroy()` method is processed. In addition, the servlet's life cycle is complete after the invocation of the `destroy()` method.
- ❑ With this, we complete our discussion on the JSP life cycle. There are various basic tags and implicit objects that constitute the JSP technology. Let's explore them in the next section.

## **Working with JSP Basic Tags and Implicit Objects**

A JSP page consists of various elements or tags, such as scripting elements, implicit objects, and directives. Scripting elements, such as declaration tags, expression tags, and scriptlet tags, help to declare variables and object references in a JSP page. Scripting elements help to generate and display dynamic content. Directives are used in a JSP page to include the content of a static or dynamic file within another file. Directives are also used to import Java files within JSP pages.

The Java Servlet API includes various interfaces that provide useful and suitable abstractions to Java programmers. Some examples of these interfaces are `HttpServletRequest`, `HttpServletResponse`, and `HttpSession`. These abstractions encapsulate an object's implementation. For example, the `HttpServletRequest` interface represents an HTTP request sent from a client along with elements, such as headers and form parameters, as well as provides convenient methods, such as `getParameter()` and `getHeader()`, to extract relevant data from the request. JSP implicit objects provide a convenient way to access the interfaces and objects of the servlet API without writing additional code.

Let's now discuss in detail the basic tags and implicit objects that constitute a JSP page, in the following sections.

### **Exploring Scripting Tags**

JSP scripting tags, also called JSP scripting elements, allow you to add Java coding statements into a JSP page. The Java code incorporated by using scripting elements is translated and generated by the JSP translator while translating the page into a servlet. In most cases, Java is the scripting language used to build a JSP page; however, other supported languages can also be used, depending on the language supported by a Web container.

Let's now learn about the classification of the scripting elements and the use of these elements.

### **Classifying the Scripting Tags**

The JSP scripting tags are categorized into the following three types of tags:

- ❑ Scriptlet
- ❑ Declarative
- ❑ Expression

## The Scriptlet Tag

Scriptlet tags allow you to write the script code (or the Java code) to implement the `_jspService()` method functionality. The JSP translator translates the statements of this tag into the `_jspService()` method of the servlet generated for a JSP page. You can use a scriptlet tag to perform the following tasks:

- ❑ Declaring variables that you can use later in a JSP page. In other words, you can declare variables and write valid expressions within the scriptlet tags.
- ❑ Using any of the JSP implicit objects or any other object declared with a `<jsp:useBean>` element.

Note that if the script code is not valid, an exception is thrown at the compilation stage of the JSP life cycle.

You can include multiple scriptlet tags in a single JSP document. The following code snippet shows the traditional JSP syntax to use a scriptlet tag:

```
<%!
    script code (allows multiple statements)
%>
```

The following code snippet shows the XML-based syntax for a scriptlet tag:

```
<jsp:scriptlet>
    script code (allows multiple statements)
</jsp:scriptlet>
```

Let's now look at an example that illustrates the use the scriptlet tag. The following code snippet shows the syntax to use a scriptlet tag in a JSP page:

```
<html>
<body>
<%
    for (int i=0;i<5;i++)
    {
        out.println(i);
    }
%>
</body>
</html>
```

In the preceding code snippet, the for loop of Java is used in a JSP page between the scriptlet tags to print numbers 1 to 5 in five lines. The following code snippet shows the equivalent servlet code for this JSP page, as generated by the JSP container:

```
public void _jspService(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
{
    ...
    out.write("<html><body>");
    for (int i=0;i<5;i++) {
        out.print(i);
    }
    out.write("</body></html>");
    ...
}
```

## The Declarative Tag

The declarative tag allows you to write the script code you need to provide in the generated servlet class, outside the `_jspService()` method. This tag allows you to declare instance and class variables and implement class methods, such as `jspInit()` and `jspDestroy()`. The following code snippet shows the syntax of the declarative tag:

```
<%!
    script code (allows multiple statements)
%>
```

The following code snippet shows the XML-based syntax for the declarative tag:

```
<jsp:declaration>
    script code (allows multiple statements)
```

```
</jsp:declaration>
```

The following code snippet shows an example illustrating the use of the declarative tag:

```
<%!
int a,b;
int fun(int a)
{
    return a;
}
%>
<%a=1;%>
<%b=fun(a);%>
<%out.println(b);%>
```

In the preceding code snippet, two integer variables, `a` and `b`, and the `fun()` method are defined by using the declarative tag. A value of 1 is then assigned to variable `a`, which is passed as an argument to the `fun()` method. This method processes the argument and stores the result in variable `b`. Finally, the value received by variable `b` is displayed on the browser.

### *The Expression Tag*

An expression tag allows you to write a Java expression, which is then resolved and the resultant value is displayed. The expression tag places the given Java expression in the `out.print()` method during the translation stage of the JSP life cycle. Consequently, the output of the Java expression is displayed with the output of the JSP page.

An expression tag also provides a simple and convenient way to access script (i.e., Java) variables. The following code snippet shows the syntax of the expression tag:

```
<%
    script code (allows only one expression)
%>
```

The following code snippet shows the XML-based syntax for the expression tag:

```
<jsp:expression>
    script code (allows only one expression)
</jsp:expression>
```

Let's now look at an example to use the expression tag. The following code snippet shows the syntax to use an expression tag in a JSP page:

```
<%! int count; %>
<html>
<body>
<% count++; %>
This page is requested by <%=count%> number of users till now.
</body>
</html>
```

The preceding code snippet can be embedded on a JSP page to count the number of requests received for the specific page. Each time a new user accesses the JSP page on the browser, the value of the `count` variable is incremented by 1, and displayed on the browser. For example, when a user accesses the JSP page for the first time, the `count` variable is declared and initialized at 0 (by default). The expression written in the scriptlet tag (`<%count++%>`) increments the value of the `count` variable by one and prints this value on a Web page. Now, when another user accesses this page, the value of the `count` variable is incremented by 1 and the browser displays the result as 2.

Consider the following points when using expression tags:

- ❑ Do not end an expression with ; because the expression given in this tag is placed within the `out.print()` method.
- ❑ You can resolve an expression given in the expression tag to any type, such as `int`, `float`, `double`, `Boolean`, `String`, or `Object`, but not to `void`.

Writing any code violating these rules raises a translation stage error. In addition, nested scripting tags are not allowed in expression tags. This means that a scripting tag cannot have another type of scripting tag. For example, consider the following code snippet:

```
<%
for (int i=0;i<5; i++)
{
    <%=i%> //out.println(+i);
}
%>
```

In the preceding code snippet, the expression tag (`<%= %>`) is nested within the scriptlet tag (`<% %>`), which would raise a translation error.

Let's now learn how to use scripting tags in a Web application.

## Using the JSP Scripting Tags

In this section, we create a Web application to learn the use of JSP scripting tags, such as declarative, expression, and scriptlets. The use of scripting tags varies from simple to complex Web applications. In simple Web applications, servlets and JSP scripting tags are used with JavaBeans to build Web applications. However, in complex applications, scripting tags are used with custom tags and other Model-View-Controller (MVC) frameworks, such as Struts and JSF. In this section, we create a simple Web application by using scripting tags.

Let's create the `ScriptingElements` Web application containing the `ScriptingTags` JSP page, which uses page directives to import the `java.util.*` package, and specify the scripting language as Java. In the declaration tag, three integer variables—`count`, `a`, and `b`—are declared. The `fun()` method accepts a parameter, `a`, and returns an integer value after multiplying the value of `a` with 10. The value returned by the `fun()` method is stored in the `b` variable. The `count` variable is used to keep track of the number of times the Web page is accessed. Listing 7.1 shows the code for the `ScriptingTags` JSP page (you can find the `ScriptingTags.jsp` file on the CD in the `code\Chapter7\ScriptingElements` folder):

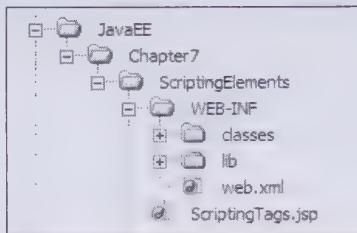
**Listing 7.1:** Showing the Code of the `ScriptingTags.jsp` File

```
<%@ page import ="java.util.*" language="java" %>
<html>
<body>
    <center><h1>Using Scripting Elements</h1></center>
    <%! int count,a,b;
       int fun(int a)
    {
        return 10*a;
    }
    %
    %
    a=1;
    count++;
    for (int i=0;i<5;i++)
    {
        out.println("Value of i in iteration
                    no." +i+ "&nbsp;&nbsp;<b>" +i+ "</b><br/>");
    }
    b=fun(a);
    out.println("Value returned by fun():&nbsp;&nbsp;<b>" +b+ "</b><br/>");
    %
    This page is requested by <b><%=count%></b> number of times on date
    <b><%=new Date()%></b>.
</body>
</html>
```

You do not need to map the JSP page in the web.xml file; however, an empty web.xml file must exist at the appropriate place in the root directory. The following code snippet shows the code to create an empty web.xml configuration file:

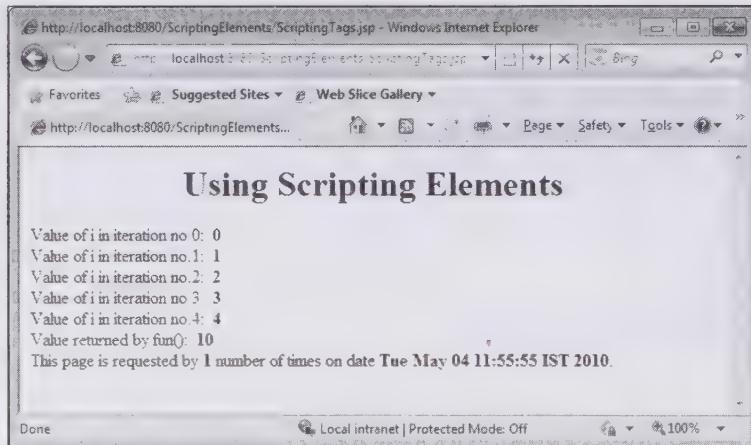
```
<web-app/>
```

Store the ScriptingTags.jsp and web.xml files in the ScriptingElements directory according to the standard directory structure of a Web application. Figure 7.3 shows the directory structure of the ScriptingElements Web application:



**Figure 7.3: Displaying the Directory Structure of the ScriptingElements Web Application**

Now, package the ScriptingElements application into the .war file that can be deployed on the Glassfish V3 application server. Next, deploy the ScriptingElements.war file on the server, after starting the server. Browse the ScriptingElements Web application by using the <http://localhost:8080/ScriptingElements/ScriptingTags.jsp> URL. Figure 7.4 shows the output of the ScriptingTags JSP page as displayed in the Web browser:



**Figure 7.4: Showing the Output of the ScriptingTags JSP Page**

Figure 7.4 shows the use of JSP scripting tags in the ScriptingTags JSP page. The scripting tags are used to iterate the values of the a and b variables. Figure 7.4 also shows the number of times a user has requested the specific Web page.

Apart from scripting elements, you can also use predefined objects, known as implicit objects, in a JSP page. Let's now understand how to implement implicit objects.

## Exploring Implicit Objects

JSP implicit objects are used in a JSP page to make the page dynamic. Java objects within scripting elements can be used to create and access the dynamic content. JSP implicit objects are predefined objects that are accessible to all JSP pages. These objects are called implicit objects because you do not need to instantiate these objects. JSP implicit objects are automatically instantiated by the JSP container while writing the script content in the scriptlet and expression tags. JSP specification standardizes some object reference names, which are available for every

JSP page so that any vendor-implemented translators have to take the responsibility of initializing these objects in the `_jspService()` method.

## **Listing the Features of Implicit Objects**

A JSP implicit object has the following features:

- ❑ Helps Java developers to access the services and resources provided by a JSP container.
- ❑ Helps you to generate the dynamic content of Web pages.
- ❑ Does not allow you to declare a JSP implicit object explicitly because they are automatically instantiated by the JSP container.
- ❑ Allows you to use a JSP implicit object without importing them into your JSP page. This implies that a user only needs to use a reference variable associated with a given object as a JSP implicit object is declared automatically. In other words, a JSP implicit object can be directly used to call the methods associated with them.
- ❑ Helps Java developers in many ways, including handling HTML parameters, sending a request to a Web component, and incorporating the content of a component into a JSP page. A JSP implicit object can also be used to log data through a JSP container, control the output stream, and handle exceptions more efficiently.

Let's now discuss the implicit objects as well as their implementation.

## **Explaining Types of Implicit Objects**

When a JSP page is translated, the nine most commonly used JSP implicit objects are initialized by the JSP Servlet engine in the `_jspService()` method. These JSP implicit objects are listed as follows:

- ❑ request
- ❑ response
- ❑ out
- ❑ page
- ❑ pageContext
- ❑ application
- ❑ session
- ❑ config
- ❑ exception

Let's learn about these objects one by one in detail in the following sections.

### *The request Object*

The `request` object is passed as a parameter to the `_jspService()` method when a client request is made. The `request` object is of the `javax.servlet.http.HttpServletRequest` type. You can use `request` objects in a JSP page in the same way they are used in servlets.

### *The response Object*

The `response` object is used to carry the response of a client request after the `_jspService()` method is executed. This object is of the `javax.servlet.http.HttpServletResponse` type.

### *The out Object*

The `out` implicit object provides access to the output stream of a servlet. This object is a subtype of the `javax.servlet.jsp.JspWriter` class. The `out` object can be used directly in JSP scriptlets to display text data on a browser, as JSP expressions are automatically placed in the output stream. In other words, the `out` object is used to write text data in the output stream.

### *The page Object*

The `page` object refers to a servlet of the JSP page processing a current request. It also works as an instance of the generated servlet in a JSP page. This object is of the `java.lang.Object` type and therefore is rarely used in a JSP page. In addition, you cannot directly use this object to call servlet methods.

### The pageContext Object

The `pageContext` object represents the context of the current JSP page. It provides a single API to manage different scoped attributes. This object is of the `javax.servlet.jsp.PageContext` type.

### The application Object

The `application` object refers to the entire environment of a Web application to which a JSP page belongs. This object is of the `javax.servlet.ServletContext` type.

### The session Object

The `session` object helps to access session data and is of the `HttpSession` type. It is declared if the value of the `session` attribute in a page directive is true. If we explicitly specify the `session` attribute to false, the JSP Servlet engine does not declare this object. In such a case, if you try to access the `session` object, an error is generated.

### The config Object

The `config` object specifies the configuration of the parameters passed to a JSP page. This object is of the `javax.servlet.ServletConfig` type. To show the use of the `config` object within a JSP page, you need to associate a servlet with a JSP file by using the `<jsp-file>` element. After associating a servlet, all the initialization parameters for the named servlet are made available to the JSP page by the `ServletConfig` object.

### The exception Object

The `exception` object is of the `java.lang.Throwable` type and is only available for JSP pages that act as the error handlers for other pages. This object is only available to the JSP error pages that have the `isErrorPage` attribute set to `true` within the page directive.

All the objects discussed so far are available only within scriptlet and expression tags. These objects should not be used in the declaration tag. However, you can use the `getServletConfig()` method if you want to get the `ServletConfig` type of the reference of an object in a declaration tag while implementing the `jspInit()` or `jspDestroy()` method. In addition, you can use the `getServletContext()` method of the `ServletConfig` type to get the `ServletContext` object.

## Using Implicit Objects

Let's now learn to use implicit objects in a JSP page to create a dynamic Web page. In this section, we develop a simple Web application, `ImplicitObjects`, by using implicit objects, such as `request`, `session`, and `pageContext`. This Web application performs three tasks: it overrides the `jspInit()` method to perform initializations, it uses the initialization parameter to implement an object of the `ServletConfig` interface, and it accesses a JSP document placed in a private folder.

The `ImplicitObjects` Web application contains the following pages:

- ❑ The Home HTML page: Represents the index page of the Web application
- ❑ The request JSP page: Displays the welcome message
- ❑ The `pageContext` JSP page: Shows the use of the `pageContext` implicit object
- ❑ The other JSP page: Shows the use of other implicit objects such as `page`, `session`, `out`, `application` and `config`.

The Home HTML page of the `ImplicitObjects` Web application consists of a simple form with a text field, `Name`, and a button, `Invoke JSP`. Listing 7.2 shows the code for the Home HTML page (you can also find the `Home.html` file on the CD in the `code\Chapter7\ImplicitObjects` folder):

**Listing 7.2:** Showing the Code for the `Home.html` File

```
<html>
<body>
<form action="request.jsp">
    Name : <input type="text" name="name"> ...
    <input type="submit" value="Invoke JSP"/>

```

```

</form>
</body>
</html>
```

When a user clicks the `Invoke JSP` button on the `Home` HTML page, the `request` JSP page is displayed. This page displays a greeting message followed by the user name entered on the `Home` HTML page. After you submit the `Home` HTML page, request details, such as type of request, its Uniform Resource Identifier (URI), and request protocol, are displayed in tabular form. These request details are retrieved by using the methods of the `request` JSP implicit object, such as `getMethod()`, `getRequestURI()`, `getProtocol()`, and `getHeader()`. Listing 7.3 shows the code for the `request.jsp` page (you can also find this file on the CD in the `code\Chapter7\ImplicitObjects` folder):

**Listing 7.3:** Showing the Code for the `request.jsp` File

```

<html>
<head>
    <title>
        Using Implicit Objects
    </title>
</head>
<body>

Hello, <b><%=request.getParameter("name")%></b><br/><br/>
Your request details are <br/><br/>
<table border="1">
<tr><th>Name</th><th>Value</th></tr>
<tr><td>request method</td>
<td><%= request.getMethod()%></td></tr>
<tr><td>request URI</td>
<td><%= request.getRequestURI()%></td></tr>
<tr><td>request protocol</td>
<td><%= request.getProtocol()%></td></tr>
<tr><td>browser</td>
<td><%= request.getHeader("user-agent")%></td></tr>
</table>
<%
    if (session.getAttribute("sessionVar")==null)
    {
        session.setAttribute("sessionVar",new Integer(0));
    }
%>

<table>
    <tr><th align=left>Would you like to see use of remaining
        implicit objects?</th></tr>
    <tr>
        <form name=form1 action="pageContext.jsp" method="post">
            <td><input type="radio" name="other" value="Yes">Yes</td>
            <td><input type="radio" name="other" value="No"> No</td></tr>
            <tr><td><input type="submit" value="Submit"></td></tr>
        </form>
    </table>
</body>
</html>
```

After clicking the `Submit` button displayed on the `request` JSP page, the control is transferred to the `pageContext` JSP page. This page uses the `pageContext` implicit object to forward a request to the other JSP page. Listing 7.4 shows the code for the `pageContext.jsp` file (you can also find this file on the CD in the `code\Chapter7\ImplicitObjects` folder):

**Listing 7.4:** Showing the Code for the pageContext.jsp File

```
<HTML>
<HEAD>
<TITLE> Intermediate </TITLE>
</HEAD>

<BODY>
<%
    if("Yes".equals(request.getParameter("other")))
    {
        pageContext.forward("other");
    }
%>
</BODY>
</HTML>
```

The other JSP page shows the use of the page, session, out, config, and application implicit objects. This page overrides the `jspInit()` method, which retrieves the other initialization parameter of the other JSP page from the `web.xml` file. The same initialization parameter value is fetched by calling the `getInitParameter` method of the config implicit object, `config.getInitParameter("count")`. In the other JSP page, we use the page implicit object's `log` method to log the message, `anothermessage (((HttpServlet)page).log("anothermessage"));`. The implicit object, `session`, invokes the `getAttribute()` method to get the value of the `sessionVar` attribute, which shows the number of active sessions in the request JSP page. The implicit object, `application`, is used to retrieve the value of the context parameter, `param1`, set in `web.xml`. We use the `application.getInitParameter("param1")` method to retrieve the value of the `param1` parameter in the other JSP page. Listing 7.5 shows the code for the `other.jsp` file (you can also find this file on the CD in the `code\Chapter7\ImplicitObjects\WEB-INF\jsppages` folder):

**Listing 7.5:** Showing the Code for the other.jsp File

```
<html>
<body>
<%!int count;
public void jspInit()
{
    ServletConfig sc=getServletConfig();
    count = Integer.parseInt(sc.getInitParameter("count"));
    System.out.println("In jspInit");
}

%>

<%
out.println("<H1>Using page, session, out, config, and application
Implicit Objects</H1>");
%>

Count value without using config implicit object: <b> <%=count%></b> <br/>
<
this.log("log message");
((HttpServlet)page).log("anothermessage");
ServletContext ct = config.getServletContext();
out.println("value of sessionVar is:"+"&nbsp;&nbsp;<b>" +
session.getAttribute("sessionVar")+"</b><br/>");
out.println("Server name and version using config implicit
object:"+"&nbsp;&nbsp;<b>" +ct.getServerInfo()+"</b><br/>");
out.println("Value of context parameter param1 obtained using
application implicit object:"+"&nbsp;&nbsp;<b>" +
```

```

application.getInitParameter("param1")+"</b><br/>"); out.println("Count value retrieved using config implicit object:" +
"&nbsp;&nbsp;<b>" +config.getInitParameter("count")+"</b>");

%>
</body>
</html>

```

The web.xml file initializes an application level parameter, param1, with the value param1. In the ImplicitObjects Web application, we configure the other JSP page in web.xml. Although it is not mandatory to declare a JSP page in web.xml, we do it for the following reasons:

- ❑ The initialization parameter count for the other JSP page is configured in web.xml.
- ❑ The other JSP page is placed in a private folder (jsppages) to avoid direct access to the page.

The other JSP page is configured by using the <jsp-file> nested tag of the <servlet> tag. The servlet name specified in the <servlet-name> tag is myjsp, which is mapped to the URI to access this page.

The other JSP page is placed in the WEB-INF/jsppages folder. Listing 7.6 shows the code for the web.xml file (you can also find the file on the CD in the code\Chapter7\ImplicitObjects\WEB-INF folder):

**Listing 7.6:** Showing the Code for the web.xml File

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">

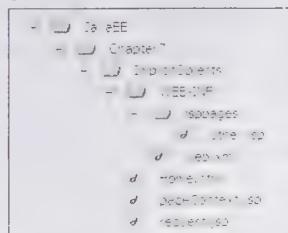
  <context-param>
    <param-name>param1</param-name>
    <param-value>param1</param-value>
  </context-param>

  <servlet>
    <servlet-name>myjsp</servlet-name>
    <jsp-file>/WEB-INF/jsppages/other.jsp</jsp-file>
    <init-param>
      <param-name>count</param-name>
      <param-value>10</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>myjsp</servlet-name>
    <url-pattern>/other</url-pattern>
  </servlet-mapping>

</web-app>

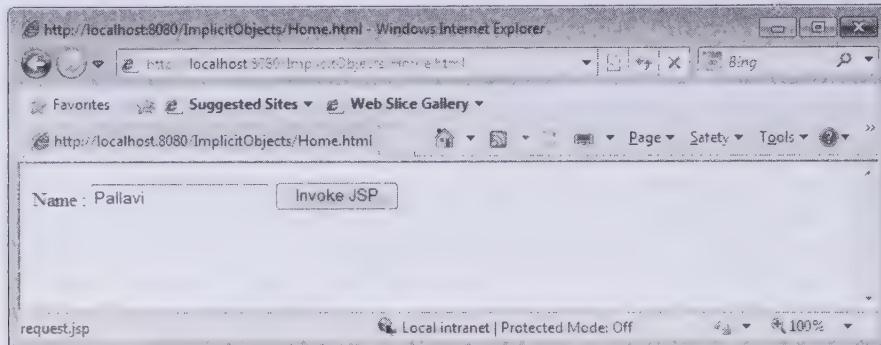
```

To run the ImplicitObjects Web application effectively, you need to make a new folder, say ImplicitObjects, and place the files of the Web application in it, such as Home.html, pageContext.jsp, other.jsp, and web.xml, as shown in Figure 7.5:



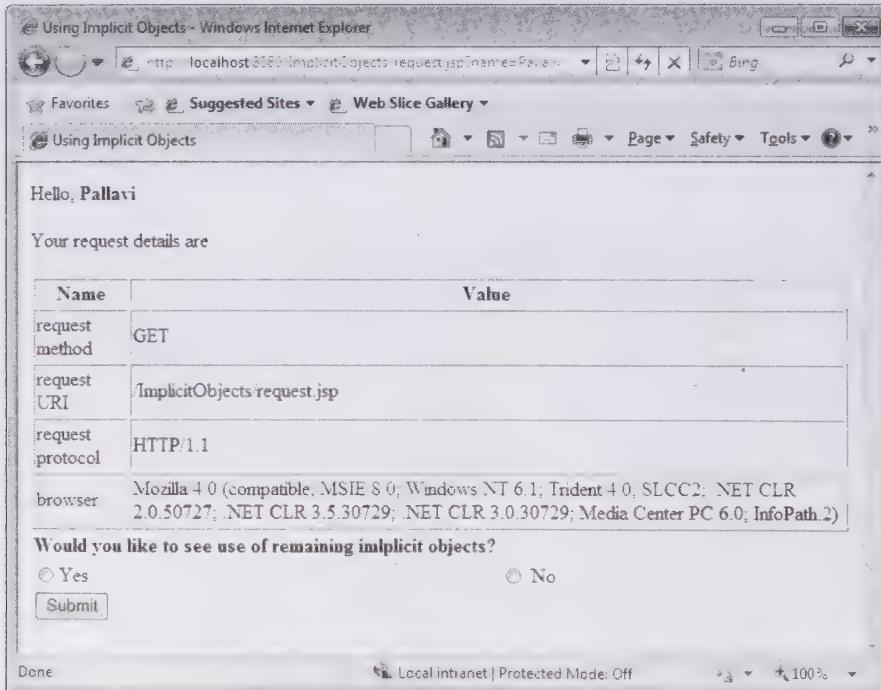
**Figure 7.5:** Showing the Directory Structure for ImplicitObjects Web Application

Now, package the ImplicitObjects application into the .war file and deploy the ImplicitObjects.war file on the Glassfish V3 application server. Next, start the Glassfish server and browse the ImplicitObjects application by using the <http://localhost:8080/ImplicitObjects/Home.html> URL. Figure 7.6 displays the Home page of the ImplicitObjects application:



**Figure 7.6: Displaying the Output of the Home HTML Page**

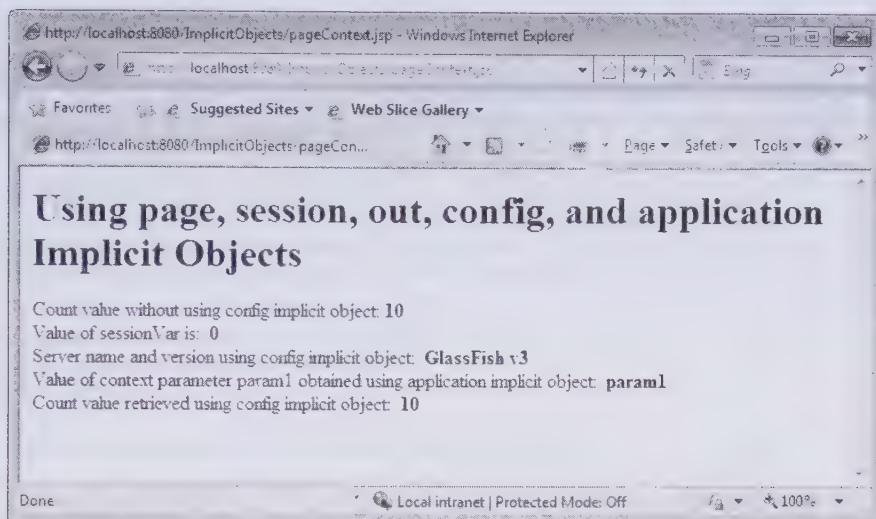
When a user clicks the Invoke JSP button, the request JSP page is displayed, as shown in Figure 7.7:



**Figure 7.7: Displaying the Result of Processing the request JSP Page**

Figure 7.7 shows all request details, such as request type, request URI, browser request details, and request protocols, in the form of a table.

When a user selects the Yes radio button and clicks the Submit button, the pageContext JSP page is displayed, as shown in Figure 7.8:



**Figure 7.8: Displaying the Output of pageContext JSP Page**

Figure 7.8 shows the values of the implicit objects used in the other JSP page in the `ImplicitObjects` application.

Let's now learn about directive tags.

## Exploring Directive Tags

Directive tags provide directions that are used by the JSP translator during the translation stage of the JSP life cycle. These tags are used to set global values, such as class declarations, methods to be implemented, and output content type. The following sections discuss the various types of directive tags and how they are used in a JSP page.

### Explaining the Types of Directive Tags

According to JSP specification, there are three standard directive tags available with all JSP-compliant containers. These directive tags are as follows:

- page
- include
- taglib

Let's discuss each of these directives in detail.

#### The page Directive Tag

The page directive tag holds the instructions that are used by a JSP translator during the translation stage of the JSP life cycle. These instructions affect different properties associated with the whole JSP page. The page directive can be used multiple times in a JSP page, and when used on any part of the JSP page automatically applies to the entire page. However, it is considered a good programming practice to use the page directive at the starting of the JSP page. The syntax of the page directive tag is as follows:

```
<%@page attributes %>
```

The following code snippet shows the XML-based syntax for the page directive tag:

```
<jsp:directive.page attributes/>
```

A total of 11 attributes can be used in the page directive tag, as listed in Table 7.1:

**Table 7.1: Attributes of the page Directive Tag**

Attribute	Value
Language	Takes the type of scripting language to be used in scripting tags, as a value. The default value is Java.
Import	Takes a comma separated list of Java classes as a value.
extends	Takes a complete qualified class name of the class that is extended by the equivalent servlet class written by the translator of the current JSP page.
buffer	Takes the buffer size in kilobytes. The values of this attribute are none, 8 kb, 16 kb, 32 kb, and 64 kb. The default value is 8 kb.
autoFlush	Specifies whether or not to automatically flush the output when the buffer is full. If a <code>true</code> value is given to this attribute, it automatically flushes the buffer as soon as the buffer is full. If this attribute is assigned a <code>false</code> value, an exception is generated when the buffer overflows. The default value of this attribute is <code>true</code> .
isThreadSafe	Takes <code>true</code> or <code>false</code> as its value; the default value is <code>true</code> . This attribute specifies whether a JSP page is thread-safe or not. In other words, the <code>isThreadSafe</code> attribute specifies whether the instance of the equivalent servlet class of the JSP page is capable of handling simultaneous requests or not. If this attribute is assigned a <code>false</code> value, only one thread can use the service provided by one object and if the value is <code>true</code> , simultaneous requests can be handled by the JSP page.
errorCode	Takes the URL path of the page to which a request is to be redirected when an exception is generated in the current page.
isErrorPage	Takes either <code>true</code> or <code>false</code> as its value; the default value is <code>false</code> . This attribute specifies whether the current page is an error page or not. If this attribute is assigned a <code>true</code> value, an additional implicit object, <code>exception</code> , also becomes available for the current JSP page.
contentType	Takes the response content Multipurpose Internet Mail Extensions (MIME) type, and optionally, character encoding. The default value is <code>text/html</code> .
Session	Takes <code>true</code> or <code>false</code> as a value, which indicates whether a session is required or not; the default value is <code>true</code> . If this attribute is assigned a <code>false</code> value, the JSP page cannot use the implicit object, <code>session</code> .
Info	Takes a String, which can be retrieved by using the <code>getServletInfo()</code> method.
pageEncoding	Specifies the encoding type to be used by a Web container to compile a JSP page. Examples of encoding types are ISO-8859-1 and UTF-8.

As stated earlier, a JSP page can contain any number of page directive tags. However, except for the import tag, none of the other tags are allowed to be specified more than once.

### The include Directive Tag

The `include` directive tag is used to combine the content of two or more files during the translation stage of the JSP life cycle. This directive appends the text of the included file to a JSP page, without any processing or modification. The included file can be static or dynamic. If the included file is dynamic, its JSP elements are first translated before being included in the JSP page. In other words, the included file is translated into a page. However, if any changes are made in the included page after the page is translated, the changes are not reflected in the page until the JSP page is translated once again. The syntax of the `include` directive tag is as follows:

```
<%@include file="file path" %>
```

The following code snippet shows the XML-based syntax for the `include` directive tag:

```
<jsp:directive.include file=" file path " />
```

The following code snippet shows an example of using the `include` directive tag to merge the `Test.jsp` and `Test1.html` files:

```
<%@include file="/Test.jsp" %>
<%@include file="/Test1.html" %>
```

Note that if any changes are made to the included page, the behavior of the `<include>` directive tag with respect to the recompilation of the page depends on a Web container. This means that if any changes are made in the included page, some containers recognize and apply the changes to the JSP page, while others do not.

### The `taglib` Directive Tag

The `taglib` directive tag is used to declare a custom tag library in a JSP page so that the tags related to that custom tag library can be used in the same JSP page. The following code snippet shows the syntax of the `taglib` directive:

```
<%@taglib uri="URI" prefix="unique_prefix" %>
```

Any special XML equivalent element for the `taglib` directive tag is not available as the declaration of a custom tag library is done by using the XML namespace syntax.

After discussing the syntax of the various directive tags, let's learn to use these tags in a JSP page.

## Using JSP Directive Tags

JSP directives are used in a JSP page to add functionality to the page. Let's create a Web application called `directiveTags` by using JSP directives. The `directiveTags` application contains the following pages:

- ❑ The `Login` HTML page—Accepts user details and redirects a user to the required JSP page
- ❑ The `LoginProcess` JSP page—Allows you to connect to a database to process a user request
- ❑ The `MyError` JSP page—Represents a JSP page that is called when an error or exception is raised

The Oracle database is used to access the `userdetails` table in the `directiveTags` application. Therefore, you need to create the `userdetails` table to work with the application before creating the required JSP pages. The following code snippet shows the code to create the `userdetails` table and insert data in the table, for the `directiveTags` application:

```
create table.userdetails(
    uname varchar2(15),
    pass varchar2(10));
insert into.userdetails values('Pallavi', 1234567);
```

Let's now create the required pages for the application. Listing 7.7 shows the code for the `Login.html` file, which is the home page of the `directiveTags` application (you can also find the `Login.html` file on the CD in the `code\Chapter7\directiveTags` folder):

**Listing 7.7:** Showing the Code for the `Login.html` File

```
<html>
<body>
    <pre>
        <form action="LoginProcess.jsp">
            <b>User Name</b> : <input type="text" name="uname"/>
            <b>Password</b> : <input type="password" name="pass"/>
            <input type="submit" value="LogIN"/>
        </form>
    </pre>
</body>
</html>
```

The `Login` HTML page allows a user to enter a user name and password, which are stored in a database. After entering the user name and password, the `LoginProcess` JSP page is called to handle the request. Listing 7.8 shows the code for the `LoginProcess.jsp` file (you can also find this file on the CD in the `code\Chapter7\directiveTags` folder):

**Listing 7.8:** Showing the Code for the `LoginProcess.jsp` File

```
<%@page import="java.sql.*" errorPage="/MyError.jsp"%>
<html>
<body>
    <%
        Connection con=null;
        String uname=request.getParameter("uname");
```

```

String pass=request.getParameter("pass");
try
{
    Class.forName("oracle.jdbc.driver.OracleDriver");
    con=DriverManager.getConnection(
        "jdbc:oracle:thin:@192.168.1.123:1521:XE","scott","tiger");
    Statement st=con.createStatement();
    ResultSet rs=st.executeQuery(
        "select * from userdetails where
         uname ='" + uname + "' and
         pass ='" + pass + "'");
    if (!rs.next())
    {
        %>
        User details given for user name : <%=request.getParameter("uname")%> and
        password : <%=request.getParameter("pass")%> are not valid <br/>
        Try again <%@include file="Login.html"%>
    
```

`</body>`
`</html>`

```

    <%
        return;
    } //if
} //try
finally
{
    try
    {
        con.close();
    }
    catch(Exception e){}
} //finally
%>

This is a Home Page <br/>
Welcome, <%=uname%>
</body>
</html>

```

The LoginProcess JSP page shows the use of the JSP page directive, which imports the java.sql package to the LoginProcess JSP page. The java.sql package helps the LoginProcess JSP page to access the userdetails table in the Oracle database that stores user information. After a user submits a request, the LoginProcess JSP page is used to generate the appropriate response for each request made.

The LoginProcess JSP page also includes another JSP page, named MyError, which is used to handle any exception that may occur in the LoginProcess JSP page. In other words, any exception in the LoginProcess JSP page is forwarded to the MyError JSP page. Listing 7.9 shows the code for the MyError.jsp file (you can also find this file on the CD in the code\Chapter7\directiveTags folder):

#### **Listing 7.9:** Showing the Code for the MyError.jsp File

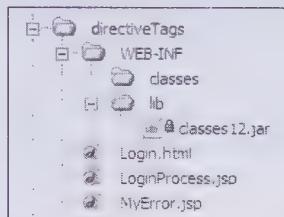
```

<%@page isErrorPage="true"%>
<html>
<body>
    An exception is raised while processing the request: <br/>
    <b>Exception : </b> <br/>
    &nbsp;&nbsp;&nbsp;&nbsp;<%=exception.getMessage()%>
</body>
</html>

```

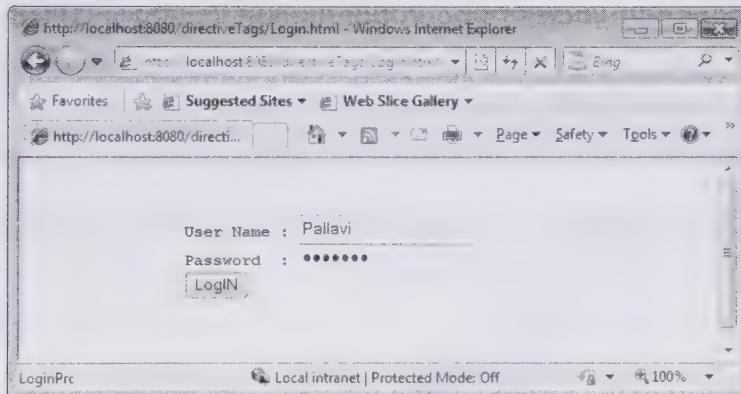
All the files created in the directiveTags application need to be placed properly as shown in Figure 7.9. As the application imports the java.sql package, you need to include a Java ARchive (JAR) file (depending

on the database used in an application) in the lib directory of the application, which is used to access a database within a JSP page. In our case, we have included the classes12.jar file in the lib directory as we are using the Oracle database for the directiveTags application. Figure 7.9 shows the directory structure of the directiveTags application:



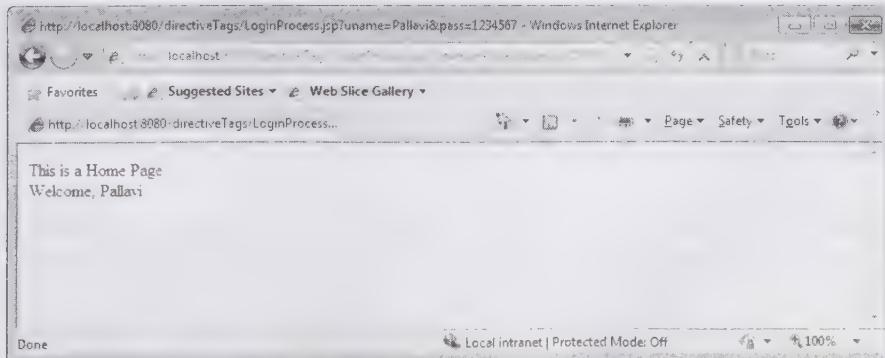
**Figure 7.9: Displaying the Directory Structure for the directiveTags Web Application**

Deploy the directiveTags application on the Glassfish application server. Next, start the Glassfish server and browse the directiveTags application by using the `http://localhost:8080/directiveTags/Login.html` URL. Figure 7.10 shows the output of the Login HTML page:



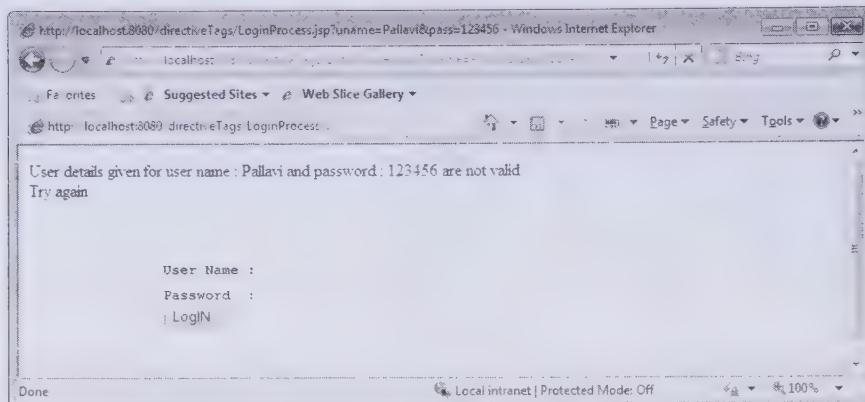
**Figure 7.10: Displaying the Output of the Login HTML Page**

Enter the user name and password on the Login HTML page and click the LogIN button. The JSP Servlet engine checks for the availability of the user name and the password in the Oracle database. If the entries are valid, you are directed to the LoginProcess JSP page, shown in Figure 7.11:



**Figure 7.11: Displaying the Output of the LoginProcess JSP Page for a Valid Login**

Figure 7.11 shows the output for a successful login in the Login HTML page. The database is called each time a user enters values in the user name and password fields. If the entries are not available in the database, the user is redirected to another page, shown in Figure 7.12:

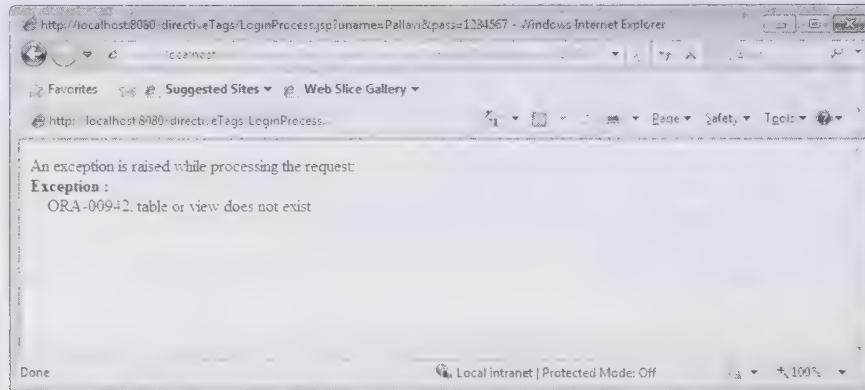


**Figure 7.12: Displaying the Output of the LoginProcess JSP Page for an Invalid Login**

Now, to test the functionality of the `isErrorPage` element used in the `MyError.jsp` file, delete the `userdetails` table from the Oracle database. The following code snippet shows the syntax for deleting a table from the database:

```
drop table.userdetails;
```

When you run the command shown in the preceding code snippet, the `userdetails` table is removed from the database. Now, if you make a request to the `LoginProcess` JSP page by clicking the `LOGIN` button, an `SQLException` exception is raised in the page and the request is forwarded to the `MyError` JSP page, as shown in Figure 7.13:



**Figure 7.13: Displaying an Invalid Access to a Database**

As the table storing user information has been dropped from the database, the Oracle exception `ORA-00942` is raised, stating that the table does not exist in the database.

#### **NOTE**

You may not see the page showing the exception in Internet Explorer. Instead, you get the HTTP 500 Internal Server Error message. To view the exception page in Internet Explorer, uncheck the Show friendly HTTP error messages check box in the Browsing category of the Settings section on the Advanced tab of the Internet Options dialog box. You can open the Internet Options dialog box by selecting the Internet Options option from the Tools menu of the Internet Explorer toolbar.

This completes our discussion on basic tags and implicit objects of JSP. Some action tags are also available in JSP. These tags help Java programmers to perform some basic actions in JSP pages. Let's learn to work with these action tags in the next section.

## Working with Action Tags in JSP

Action tags were first introduced in JSP 1.1, and additional tags were added in the JSP 1.2 and 2.0 specifications. Action tags allow Java programmers to include some basic actions, such as inserting the resources of other pages, forwarding a request to another page, creating or locating JavaBean instances, and setting and retrieving JavaBean properties, in JSP pages. Let's discuss the JSP action tags and learn to declare a JavaBean in a JSP page.

### Exploring Action Tags

Action tags are specific to a JSP page. When a JSP container encounters an action tag while translating a JSP page into a servlet, it generates a Java code corresponding to the task to be performed by the action tag. The following are some important action tags available in a JSP page:

- ❑ <jsp:include>
- ❑ <jsp:forward>
- ❑ <jsp:param>
- ❑ <jsp:useBean>
- ❑ <jsp:setProperty>
- ❑ <jsp:getProperty>
- ❑ <jsp:plugin>
- ❑ <jsp:params>
- ❑ <jsp:fallback>
- ❑ <jsp:attribute>
- ❑ <jsp:body>
- ❑ <jsp:element>
- ❑ <jsp:text>

Now, let's describe these action tags one by one in detail in the following sections.

#### The <jsp:include> Tag

The <jsp:include> action tag facilitates Java programmers in including a static or dynamic resource, such as an HTML or JSP page in the current JSP page, while processing a request. The page to be included is specified by a URL. If the resource to be included is a static HTML page, its content is directly included in the current JSP page. However, if the resource to be included is dynamic, a request is sent to the resource. Then, the resource processes the request and generates a response that is sent back as a result to the JSP page that requested the resource. The result is then included in the JSP page. For example, if the resource included in a JSP page is another JSP page, the requested JSP page is processed and the output content generated by it is included in the requesting JSP page. The following code snippet shows the syntax of the <jsp:include> action tag in a JSP page:

```
<jsp:include attributes>
<!-- Zero or more jsp:param tags -->
</jsp:include>
```

#### NOTE

If the included resource is dynamic, you can use the <jsp:param> tag to supply the name of a parameter with its corresponding value to the requested resource.

The <jsp:include> action tag has certain attributes, which are used to accept values for the specified requested JSP page. The attributes specific to the <jsp:include> tag are as follows:

- ❑ **The page attribute** – Takes a relative URL, which specifies the location of the resource to be included in the JSP page. This attribute allows Java programmers to provide an expression that evaluates to a String representing the relative URL that specifies the resource. The relative URL can be specified directly or as an expression within a page attribute. The following syntax shows how to use the page attribute in a JSP page:

- <jsp:include page="/Header.html"/>

- <jsp:include page="<% =mypath%>" />

The relative URL cannot take a protocol name, port number, or domain name. It starts with the / (forward slash) character, specifying that the URL is taken relative to the context path.

- **The flush attribute**—Takes either the true or false value, to indicate whether the buffer needs to be flushed or not before including a resource. If the true value is assigned to this attribute, the buffer is flushed before including the resource. The attribute takes false as its default value. After the action specified in the <jsp:include> tag is completed, the JSP container continues to process the rest of the JSP page.

You may remember that we talked about the include directive tag under The Include Directive heading of this chapter. Though both the <jsp:include> action tag and the include directive tag are used to include content into a JSP page, they are very different from one another.

The include directive tag incorporates the content of a specified page in a generated servlet page while translating a JSP page during the translation phase. In general, the include directive tag is used to include files, such as HTML, JSP, XML, or a simple .txt file, into a JSP page statically. The file attribute, which is used to refer to the file to be included, is the only mandatory attribute available in the include directive tag. The following code snippet shows the use of the JSP include directive tag in a JSP page:

```
%@include file="test.jsp"
```

In the preceding code snippet, test.jsp is the file to be included in the current JSP page. The content of the test JSP page is incorporated in the current JSP page during the translation stage. You cannot use an expression to specify the URL path to include the resource in the JSP page while using the include directive tag.

The <jsp:include> action tag is used to incorporate the response generated by executing the specified JSP page or servlet. The response is included during the processing of a request, when a request for the page is received from a user. In other words, the specified JSP page is included in the request handling phase of the included JSP page life cycle. The following code snippet shows the use of the <jsp:include> action tag in a JSP page:

```
<jsp:include page= "test.jsp">
```

In the preceding code snippet, the output of the test.jsp file is included in the current JSP page. The respective logic for include is written in the equivalent servlet generated for the JSP page.

Unlike the include directive tag, the <jsp:include> action tag accepts expressions. This implies that, with the <jsp:include> action tag, you can decide the page to include at runtime, which is not possible with the include directive tag.

## The <jsp:forward> Tag

The <jsp:forward> tag forwards a JSP request to another resource, which can be either static or dynamic. If the request is forwarded to a dynamic resource, a <jsp:param> tag can be used to pass a name and value of a parameter to a resource. We use the page directive with the value of the buffer parameter set to none, to specify that the output of the JSP page should not be buffered.

When the output stream is not buffered and some output is written to it, a <jsp:forward> action throws a java.IllegalStateException exception. The <jsp:forward> action in a JSP page works similar to the forward() method of the RequestDispatcher interface. The following code snippet shows the use of the <jsp:forward> action tag in a JSP page:

```
<jsp:forward attributes>
<!-- Zero or more jsp:param tags -->
</jsp:forward>
```

The <jsp:forward> action takes a page attribute to specify the page to which the current request is to be forwarded. The page attribute takes a relative URL to locate a resource. This allows Java programmers to provide an expression, which results in a String value representing the relative URL to locate the resource. You can specify the relative URL directly or by using an expression. The following snippet shows the use of the page attribute in the <jsp:forward> action tag in a JSP page:

```
<jsp:forward page="/Header.html"/>
```

```
<jsp:forward page="<%>=mypath%>"/>
```

The URL specified in the page attribute cannot have a protocol name, port number, or domain name. If the URL starts with the / character, the URL is taken to be relative to the context path; if it does not, the URL is taken to be relative to the JSP page. After the forward action is complete, the JSP container does not continue processing the rest of the JSP page.

## The <jsp:param> Tag

The <jsp:param> tag allows Java programmers to pass a name and value of a parameter to a dynamic resource, while including it in a JSP page or forwarding a request from a JSP page to another JSP page. The <jsp:param> tag is also used to pass parameters to an applet configured in the JSP page by using the <jsp:plugin> tag. The following code snippet shows the syntax to use the <jsp:param> tag in a JSP page:

```
<jsp:param attributes />
```

You can use more than one <jsp:param> tag if you want to pass more than one parameter. The <jsp:param> action tag contains two attributes, name and value, which are discussed as follows:

- ❑ **name**—Specifies the name of the parameter and takes a case-sensitive String literal as its value. The parameter name should be unique. A different name should be used for each parameter.
- ❑ **value**—Specifies the value of the parameter. It takes either a case-sensitive String literal or an expression that is evaluated in the request handling stage of the JSP life cycle.

The following code snippet shows the use of the JSP param and include actions in a JSP page, along with the attributes of the <jsp:param> action tag:

```
<jsp:include page="/MyPage2.jsp">
<jsp:param name="uname" value="User1"/>
<jsp:param name="user_type" value="admin"/>
</jsp:include>
```

## The <jsp:useBean> Tag

To separate the business logic from the presentation logic, it is often a good idea to encapsulate the business logic in a Java object (a JavaBean), and then instantiate and use this Java object within a JSP page. The <jsp:useBean>, <jsp:setProperty>, and <jsp:getProperty> tags help in this task.

The <jsp:useBean> action tag is used to instantiate a JavaBean or locate an existing JavaBean instance, and assign it to a variable name (or id). The object's lifetime can also be specified by providing a value, such as application scope, to the scope attribute. The <jsp:useBean> action tag ensures that the object having the specified id, and which lies in the appropriate specified scope, is available. The object can then be referenced by using its associated id within the JSP page, depending on the scope of the JavaBean. The following code snippet shows the syntax of the <jsp:useBean> action tag:

```
<jsp:useBean attributes>
<!-optional body content-->
</jsp:useBean>
```

The <jsp:useBean> action tag has certain attributes that add extra characteristics to it. For example, the scope attribute of the <jsp:useBean> action tag makes a JavaBean instance available in different scopes. Some attributes specific to the <jsp:useBean> tag are as follows:

- ❑ **id**—Represents the name assigned to a JavaBean, which is later used as a variable to access the JavaBean. The id attribute is used to locate an existing JavaBean instance in the appropriate scope specified in the <jsp:useBean> action tag. This attribute is case sensitive and must follow the naming conventions used to define Java variables.
- ❑ **scope**—Represents the scope in which a JavaBean instance has to be located or created. The value of scope for the JavaBean instance can be page, request, session, or application. The default scope is page. The various scope values can be defined as follows:
- ❑ **page scope**—Indicates that a JavaBean can be used where the <jsp:useBean> action tag is used within a JSP page, until a response is sent back to a client or a request is forwarded to another resource.

- ❑ **request scope**—Indicates that a JavaBean can be used from any JSP page, which processes the same request until a response is sent to a client by the JSP page.
- ❑ **session scope**—Indicates that a JavaBean can be used from any JSP page invoked in the same session as the JSP page that created the JavaBean. The JavaBean instance exists throughout the entire session, and can be accessed by any page sharing the session. The page directive must have a `true` value for the `session` attribute in the JSP page by which the JavaBean is created.
- ❑ **application scope**—Indicates that a JavaBean can be used from any JSP page in the same application as the JSP page that created the JavaBean. This JavaBean exists throughout the session of a Web application, and can be accessed by any page in the application.
- ❑ **class**—Takes the qualified class name to create a JavaBean instance if the JavaBean instance is not found in the given scope. The class specified by the class name assigned to the `class` attribute should not be an abstract class and should have a no-argument constructor. The JSP container calls the no-argument constructor to create a JavaBean instance by using the `new` keyword.

**NOTE**

*The `class` attribute does not allow expressions. This implies that the class name cannot be given dynamically and should be explicitly entered while coding a JSP page.*

- ❑ **beanName**—Takes a qualified class name or an expression of a JavaBean. A JSP container uses the `instantiate` method defined in the `java.beans.Beans` class to instantiate the JavaBean. While instantiating the JavaBean, the `instantiate()` method of the `java.beans.Beans` class first verifies whether the specified name represents the serialized template or not. If the serialized template is found, the `instantiate` method reads it by using a class loader to instantiate the JavaBean. The serialized form is located in the file with the name `packagename.classname.ser`. If the specified name does not represent a serialized template, the `instantiate` method uses the no-argument constructor to create an instance.

**NOTE**

*You can use either the `class` attribute or the `beanName` attribute in a `<jsp:useBean>` tag; both attributes cannot be used together in a `<jsp:useBean>` tag as they refer to the same values.*

- ❑ **type**—Takes a qualified class or interface name, which can be the class name given in the `class` or `beanName` attribute or the super type of a class. This attribute is also used with or without `class` or `beanName`. When the `type` attribute is used with the `class` or `beanName` attribute, the reference of a JavaBean located or created is stored in a reference variable of the type as specified by the `type` attribute. The variable name will be as specified in the `id` attribute. When the `type` attribute is used without the `class` or `beanName` attribute, the JSP container only tries to locate the JavaBean but does not instantiate the JavaBean. If the JavaBean is not found, the JSP container throws the `InstantiationException` exception.

## The `<jsp:setProperty>` Tag

The `<jsp:setProperty>` action tag sets the value of a property by using the setter methods of a JavaBean. Before using the `<jsp:setProperty>` action tag, the JavaBean must be instantiated. In addition, the `name` attribute of the JavaBean must be the same as the reference variable name of the JavaBean instance. The JavaBean can also be instantiated by using the `<jsp:useBean>` action tag as explained earlier. While using the `<jsp:useBean>` action tag, the `name` attribute of the `<jsp:setProperty>` action tag must be the same as the `id` attribute of the `<jsp:useBean>` action tag. These attributes are used to ensure co-ordination between the `<jsp:useBean>` and `<jsp:setProperty>` action tags as both these tags work together. The following code snippet shows the syntax of the `<jsp:setProperty>` tag:

```
<jsp:setProperty attributes/>
```

The `<jsp:setProperty>` tag contains various attributes, which are described as follows:

- ❑ **name**—Takes the name of an existing JavaBean as a reference variable to invoke a setter method. The value of this attribute must match the value of the `id` attribute of the `<jsp:useBean>` tag to set the properties of

a bean. Consequently, you should declare the `<jsp:useBean>` tag before the `<jsp:setProperty>` tag in a JSP page.

- **property**—Takes the name of the property to be set, and specifies the setter method to be invoked. This attribute is used in two different ways. One is to use an asterisk (\*) with the value of the property attribute. Doing this matches all the properties of a JavaBean with the request parameter names. However, if you want to match any specific property of the JavaBean, you need to specify the value of the property attribute with the property name rather than use \*. The following code snippet shows the syntax to use the property attribute with \* and with a specific property name:

```
// Syntax to use property attribute with "*"
<jsp:setProperty name="name of reference variable" property="*" />
// Syntax to use property attribute with specific value
```

```
<jsp:setProperty name="name of reference variable" property="property name" />
```

When you use asterisk (\*) with the property attribute, the value and param attributes are not applicable. However, when you specify a specific property name for the property attribute, the value and param attributes are applicable.

The values sent for a request parameter from a client to a server are always of type `String`. These values must be converted into JavaBean property types. If a property has the `PropertyEditor` class, as indicated in the JavaBeans specification, the `setAsText(String)` method is used to convert the value of the request parameter from `String` to a JavaBean compatible data type. If the `setAsText(String)` method fails to convert the type, an `IllegalArgumentException` exception is thrown by the method.

- **value**—Takes the value that has to be set to the specified JavaBean property. This attribute accepts the value as the `String` type or as an expression that is evaluated at runtime. If the value is not of the `String` type, the value is converted to a JavaBean compatible data type.

#### **Note**

*The value attribute should not be used if the property attribute is set to \*, because \* refers to all properties and we cannot set all the properties by using one value.*

The following code snippet shows the syntax to use the `<jsp:setProperty>` tag with the value attribute:

```
<jsp:setProperty name="name of reference variable"
    property="property name" value=" Property value as string "/>
```

Or

```
<jsp:setProperty name="name of reference variable" property="property name"
    value=" <%= Property value as expression %> "/>
```

- **param**—Specifies the request parameter name whose value is to be assigned to a JavaBean property. When setting JavaBean properties from request parameters, it is not always necessary for the JavaBean to have the same property names as the request parameters. If the param value is not specified, it is assumed that the request parameter and the JavaBean property have the same names.

#### **Note**

*The param attribute must also not be used when the property of a JavaBean is set to \*, because \* refers to all properties and we cannot set requested parameters for all properties by using one value.*

The following code snippet shows the syntax to use the param property:

```
<jsp:setProperty name="reference variable name" property="property name"
    param="request parameter name"/>
```

The following code snippet shows the syntax to set all the properties in a JavaBean while setting the JavaBean properties from the request object:

```
<jsp:setProperty name="abcbean" property="*"/>
```

The following code snippet shows the syntax to set a specific property in a JavaBean while setting the JavaBean properties from the request object:

```
<jsp:setProperty name="abcbean" property="uname" param="uname"/>
```

When setting a JavaBean property to a value, you should specify the value attribute as either a String or an expression that is evaluated at runtime, as shown by the syntax in the following code snippet:

```
<jsp:setProperty name="abcbean" property="uname" value="<% uname %>" />
```

### The <jsp:getProperty> Tag

The <jsp:getProperty> action tag retrieves the value of a property by using the getter methods of a JavaBean and writes the value to the current JspWriter. The following code snippet shows the syntax to use the <jsp:getProperty> action tag:

```
<jsp:getProperty attributes/> [REMOVED]
```

The <jsp:getProperty> action tag takes two attributes, name and property, which are described as follows:

- ❑ **name** – Takes the reference variable name on which you want to invoke the getter method. If a JavaBean is instantiated by using the <jsp:useBean> action tag, the value assigned to the name attribute should match the id attribute value of the <jsp:useBean> action tag. This implies that the <jsp:useBean> action tag must appear before the <jsp:setProperty> tag in a JSP page.
- ❑ **property** – Takes the value of a JavaBean property and invokes the getter method of the property. This attribute takes the name of the property as an argument. For example, if the getUname method needs to be called, the property attribute takes the value uname, as shown in the following code snippet:

```
<jsp:getProperty name="mybean" property="uname" />
```

In the preceding code snippet, the <jsp:getProperty> tag gets the value of the uname property, by using the mybean instance, and includes this value into the output.

#### NOTE

*The <jsp:getProperty> tag is not designed to access Enterprise JavaBeans (EJB) and indexed properties.*

### The <jsp:plugin> Tag

The <jsp:plugin> action tag provides support for including a Java applet or JavaBean in a client Web browser, by using a built-in or downloaded Java plug-in. In the request handling stage of the JSP life cycle, the <jsp:plugin> action tag is substituted by either the <object> or <embed> tag, depending on the browser version. In general, the attributes of the <jsp:plugin> action tag perform the following operations:

- ❑ Specify whether the component added in the <object> tag is a JavaBean or an applet
- ❑ Locate the code that needs to be run
- ❑ Position an object in the browser window
- ❑ Specify a URL from which the plug-in software is to be download
- ❑ Pass parameter names and values to an object

The following code snippet shows the syntax to use the <jsp:plugin> action tag in a JSP page:

```
<jsp:plugin attributes>
  [<!-optionally one jsp:params and/or one jsp:fallback tag can be used -->]
</jsp:plugin>
```

The <jsp:plugin> tag takes some predefined attributes, which are described as follows.

- ❑ **type** – Specifies the type of object that needs to be presented to the browser. The object can be an applet or a JavaBean.
- ❑ **code** – Takes the qualified class name of the object that has to be presented.
- ❑ **codebase** – Takes the base URL where the specified class can be located. This is an optional attribute.
- ❑ **name** – Specifies the name of the instance of a JavaBean or an applet, which helps it to be invoked by the same JSP page to communicate with one another.
- ❑ **archive** – Specifies a comma-separated list representing pathnames. A pathname is used to locate archive files, which are preloaded with a class loader that is present in the directory named codebase. Archive files distinctively improve the performance of an applet and are loaded securely, frequently over a network. The archive attribute of the <jsp:plugin> tag is similar to the archive attribute of the HTML applet tag.
- ❑ **width** – Specifies the initial width, in pixels, for the image shown by an applet or a JavaBean.

- ❑ **height**—Specifies the initial height, in pixels, for the image shown by an applet or a JavaBean.
- ❑ **align**—Specifies the position of an applet. The values the align attribute can take are bottom, top, middle, left, or right. The default value is bottom.
- ❑ **hspace**—Specifies the amount of horizontal space, in pixels, that should be assigned to the left and right side of an applet or a JavaBean displayed by the browser. The value assigned to the hspace attribute must be a nonzero number.
- ❑ **vspace**—Specifies the amount of vertical space, in pixels, that should be assigned to the bottom and top of an applet or a JavaBean. The value assigned to the vspace attribute must be a nonzero number.
- ❑ **jreversion**—Specifies the version of Java Runtime Environment (JRE) as required by the corresponding applet or JavaBean. The default value is 1.2.
- ❑ **nspluginurl**—Specifies the URL from which a client can download the JRE plug-in as required for Netscape Navigator. The value of this attribute is the complete URL specifying a protocol name, port number (this is optional), and domain name.
- ❑ **iepluginurl**—Specifies the URL from which a client can download the JRE plug-in as required for Internet Explorer. The value of this attribute is the complete URL specifying a protocol name, port number (optional), and domain name.

### The <jsp:params> Tag

The <jsp:params> action tag sends the parameters that you want to pass to an applet or a JavaBean. The following code snippet shows the syntax to use the <jsp:params> action tag in a JSP page:

```
<jsp:params>
  <!-- one or more jsp:param tags-->
</jsp:params>
```

To specify more than one parameter value, you can use more than one <jsp:params> action tag within a <jsp:params> tag.

### The <jsp:fallback> Tag

The <jsp:fallback> action tag allows you to specify a text message that is displayed if the required plug-in cannot run. This action tag must be used as a child tag with the <jsp:plugin> action tag. If the plug-in runs and the applet or JavaBean cannot run, the plug-in generally displays a popup window clarifying the error to the user. The following code snippet shows the syntax to use the <jsp:fallback> action tag in a JSP page:

```
<jsp:fallback>
  Text message that has to be displayed if the plugin cannot be started
</jsp:fallback>
```

### The <jsp:attribute> Tag

The <jsp:attribute> action tag is used to specify the value of a standard or custom action attribute. For example, you can use the <jsp:attribute> tag to set the attributes of the <jsp:setProperty> attribute, as shown in the following code snippet:

```
<jsp:setProperty name="mybean">
  <jsp:attribute name="property">uname</jsp:attribute>
  <jsp:attribute name="value">
    <jsp:expression>uname</jsp:expression>
  </jsp:attribute>
</jsp:setProperty>
```

The <jsp:attribute> tag accepts the name and trim attributes, where name specifies the attribute name that you want to set, and trim takes true or false, indicating whether the whitespace coming at the beginning and the end of the <jsp:attribute> tag should be discarded or not. By default, the heading and trailing whitespaces are discarded; consequently, the default value of the trim attribute is true.

**NOTE**

The JSP container discards the heading and trailing whitespaces at transaction time and not at the request handling phase. For example, if you use an expression tag in the body part of an attribute action tag, and the expression tag produces a value with heading and trailing whitespaces, these whitespaces are not discarded by the JSP container. In the preceding code snippet, if the uname variable contains leading and trailing whitespaces, then the container does not eliminate the whitespaces.

If the body of the `<jsp:attribute>` tag is empty, you can specify its value by “”.

**The `<jsp:body>` Tag**

The `<jsp:body>` action tag is used to specify the content (or body) of a standard or custom action tag. Generally, the body content of a standard or custom action invocation is implicitly defined as the body of that tag. However, if one or more `<jsp:attribute>` tags appear in the body of the tag, the body of a standard or custom action can be defined explicitly by using the `<jsp:body>` tag. The `<jsp:body>` tag is required if the standard action or custom tag has multiple `<jsp:attribute>` tags.

You can use the `<jsp:body>` tag to specify the content for the body of JSP actions tags, except for some action tags such as `<jsp:body>`, `<jsp:attribute>`, `<jsp:scriptlet>`, `<jsp:expression>`, and `<jsp:declaration>`.

A tag is said to have an empty body if one or more `<jsp:attribute>` tags appear in the body of a tag invocation with no or empty `<jsp:body>` tags. The following code snippet shows the use of the `<jsp:body>` tag:

```
<jsp:useBean id="mybean">
    <jsp:attribute name="class" trim="true">
        com.kogent.MyBean
    </jsp:attribute>
    <jsp:attribute name="scope">session</jsp:attribute>
    <jsp:body>
        <jsp:setProperty name="mybean" property="*"/>
    </jsp:body>
</jsp:useBean>
```

**The `<jsp:element>` Tag**

The `<jsp:element>` action tag is used to dynamically define the value of the tag of an XML element. This action tag allows you to create an XML tag with a given name. The content of the `<jsp:element>` tag is a template for the attributes and child nodes of the XML tag you want to create. The `<jsp:element>` action tag can be used in JSP pages and tag files. The following code snippet shows the syntax of the element tag:

```
<jsp:element name="name">
    jsp:attribute*
    jsp:body?
</jsp:element>
```

In the preceding syntax, the name attribute of the element tag specifies the name of the XML element it has to create. You can also give an expression as the name of the XML element. The `<jsp:element>` tag can be an empty tag or can contain one `<jsp:body>` or multiple `<jsp:attribute>` attributes, with or without the `<jsp:body>` tag. The following code snippet shows the use of the name attribute with the element action tag:

```
<jsp:element name="mytag"/>
```

The `<jsp:element>` shown in the preceding code snippet creates an empty tag with the name, mytag.

The following code snippet shows another example of using the `<jsp:element>` tag with the `<jsp:attribute>` tag:

```
<jsp:element name="mytag">
    <jsp:attribute name="myatt">Myval</jsp:attribute>
</jsp:element>
```

The `<jsp:element>` tag shown in the preceding code snippet creates an empty element, named mytag, with the myatt=Myval attribute.

The following code snippet shows the use of <jsp:tag> to create a tag with attribute and body content:

```
<jsp:tag name="mytag">
  <jsp:attribute name="myatt">Myval</jsp:attribute>
  <jsp:body>Hello</jsp:body>
</jsp:tag>
```

The preceding code snippet creates the tag named mytag with an attribute, myatt=Myval, and body text content as Hello.

## The <jsp:text> Tag

A <jsp:text> tag is used to enclose template data in an XML tag. The text tag can be used in a JSP page or tag file. The content of a <jsp:text> tag is passed to the implicit object, out. A <jsp:text> tag has no attributes and can appear at any place where template data appears. The following code snippet shows the syntax to use the <jsp:text> tag:

```
<jsp:text> template data </jsp:text>
```

In the preceding code snippet, template data text is used within the <jsp:text> tag. Instead of text, you can also use expressions within the <jsp:text> tag.

After discussing the syntax of various action tags, let's now learn how to declare a JavaBean in a JSP page.

## Declaring a Bean in a JSP Page

Let's create the useBeanEx Web application to learn how to create a JavaBean and declare it in a JSP page. Before declaring a JavaBean in a JSP page, you must create the JavaBean. Therefore, we create a JavaBean called RegForm, which is used to set and get attributes, such as user name and e-mail address, of a user.

The following broad-level steps need to be performed to declare a JavaBean in a JSP page:

1. Create a JavaBean
2. Declare the JavaBean in a JSP page by using the <jsp:useBean> tag
3. Access the properties of the JavaBean
4. Generate dynamic content
5. Deploy and run the useBeanEx application

Let's now learn to implement the steps one by one in the following sections.

## Creating a JavaBean

In this section, we create a JavaBean called RegForm. Listing 7.10 shows the code for the RegForm JavaBean (you can find the code for the RegForm.java file on the CD in the code\Chapter7\useBeanEx\src\com\kogent folder):

**Listing 7.10:** Showing the Code for the RegForm.java File

```
package com.kogent;

public class RegForm implements java.io.Serializable
{
    private String uname, pass, repass, email, fn, ln, address;

    public void setUsername(String s){uname=s;}
    public void setPassword(String s){pass=s;}
    public void setRePassword(String s){repass=s;}
    public void setEmail(String s){email=s;}
    public void setFirstName(String s){fn=s;}
    public void setLastName(String s){ln=s;}
    public void setAddress(String s){address=s;}

    public String getUsername(){return uname;}
    public String getPassword(){return pass;}
    public String getRePassword(){return repass;}
    public String getEmail(){return email;}
```

```

public String getFirstName(){return fn;}
public String getLastName(){return ln;}
public String getAddress(){return address;}
}//class

```

## Declaring a JavaBean in a JSP Page

After creating the JavaBean, you can declare it in a JSP page by using the `<jsp:useBean>` tag. For this, you have to create a JSP page. Listing 7.11 shows the code for a JSP page named RegProcess, in which we declare the JavaBean created in Listing 7.10 (you can find the RegProcess.jsp file on the CD in the code\Chapter7\useBeanEx\ folder):

### Listing 7.11: Showing the Code for the RegProcess.jsp File

```

<%@page errorPage="Registration.html"%>
<html>
<body>

<jsp:useBean id="regform">
  <jsp:attribute name="class" trim="true">com.kogent.RegForm</jsp:attribute>
  <jsp:attribute name="scope">session</jsp:attribute>
  <jsp:body>
    <jsp:setProperty name="regform" property="*"/>
  </jsp:body>
</jsp:useBean>

<form action="RegProcessFinal.jsp"><pre> <b>
First Name : <input type="text" name="first_name"/>
Last Name : <input type="text" name="last_name"/>
Address : <input type="text" name="address"/>

<input type="submit" value="Register"/>
</b></pre></form>
</body>
</html>

```

In Listing 7.11, the `<jsp:useBean>` action tag creates an instance of a JavaBean class according to the attributes specified in the JSP page. The following is a brief description of the attributes of the `<jsp:useBean>` action tag:

**id**—Specifies the name of the JavaBean that you want to declare

**class**—Specifies name of the JavaBean class, which helps the JSP container to search the JavaBean class and create its instance.

**scope**—Specifies the scope of the JavaBean

The preceding attributes help the JSP container to create an instance of the JavaBean class. The `<jsp:useBean>` action tag has some more attributes along with the attributes that are used in the RegProcess JSP page.

## Accessing JavaBean Properties

You can verify the properties of a JavaBean, such as password, e-mail, and firstName, by using the `<jsp:getProperty>` action tag. This action tag uses the name and property tags to read the JavaBean properties. Let's create the ViewRegistrationDetails.jsp file to show the use of the `<jsp:getProperty>` action tag to read JavaBean properties. Listing 7.12 shows the code for the ViewRegistrationDetails JSP page (you can find the ViewRegistrationDetails.jsp file on the CD in the code\Chapter7\useBeanEx\ folder):

### Listing 7.12: Showing the Code for the ViewRegistrationDetails.jsp File

```

<jsp:useBean id="regform" type="com.kogent.RegForm" scope="session"/>
<%@page errorPage="Registration.html"%>
<html>

```

```

<body> <pre>

<b>User Name :</b> <jsp:getProperty name="regform" property="userName"/>
<b>Password :</b> <jsp:getProperty property="password" name="regform"/>
<b>Email ID :</b> <jsp:getProperty name="regform" property="email"/>
<b>First Name :</b> <jsp:getProperty name="regform" property="firstName"/>
<b>last Name :</b> <jsp:getProperty name="regform" property="lastName"/>
<b>Address :</b> <jsp:getProperty name="regform" property="address"/>

</pre>
<form method=post action="javascript:alert('The remaining process is
under development');">
<input type="submit" value="Register"/>
</form>
</body>
</html>

```

Before reading JavaBean properties, the JSP container needs to locate the instance of the JavaBean class you create. After the JSP container locates the instance of the JavaBean class, the container starts reading the JavaBean properties by using the `<jsp:getProperty>` tag. Listing 7.12 shows the attributes used with the `<jsp:getProperty>` tag. The `name` attribute specifies the name of the JavaBean class to the JSP container and the `property` attribute specifies the JavaBean properties that have to be read.

## Generating Dynamic Content within a JSP Page

The next step is to create an HTML page, named Registration, to retrieve the required data from users to set the properties of the RegForm JavaBean. The Registration.html file contains a button called Register. When a user clicks this button after providing the required user details, the user is directed to the RegProcess JSP page to set the properties of the RegForm JavaBean, based on the details entered by the user. In other words, we use the Registration HTML page to generate dynamic content in the RegProcess JSP page. Listing 7.13 shows the code for the Registration HTML page (you can find the Registration.html file on the CD in the code\Chapter7\useBeanEx\ folder):

**Listing 7.13:** Showing the Code for the Registration HTML Page

```

<html>
<body>
<pre>
<form action="RegProcess.jsp">
<pre><b>User Name :</b> <input type="text" name="userName"/>
<b>Password :</b> <input type="password" name="password"/>
<b>RePassword :</b> <input type="password" name="rePassword"/>
<b>Email ID :</b> <input type="text" name="email"/>
<input type="submit" value="Register"/>
</pre><b></b></pre>
</form>
</pre>
</body>
</html>

```

In Listing 7.13, the param names such as `userName`, `password`, and `rePassword` are the same as the property names in the RegForm JavaBean. However, other param names, such as `first_name` and `last_name`, are different from the property names, such as `firstName` and `lastName`. Therefore, the RegProcess JSP page is created to set these properties in the RegForm JavaBean.

Using the RegProcess JSP page, you can declare a JavaBean in JSP. This page also collects more information about the user in the RegForm JavaBean in addition to what is collected by using the Registration.html page. A Register button is provided in the RegProcess JSP page. When you click this button after entering the required user information, you are directed to the RegProcessFinal JSP page. Listing 7.14 provides the code for the RegProcessFinal JSP page (you can find the RegProcessFinal.jsp file on the CD in the code\Chapter7\useBeanEx\ folder):

**Listing 7.14:** Showing the Code for the RegProcessFinal JSP Page

```
<%@page errorPage="Registration.html"%>

<jsp:useBean id="regform" class="com.kogent.RegForm" scope="session"/>

<jsp:setProperty name="regform" property="firstName" param="first_name"/>
<jsp:setProperty name="regform" property="lastName" param="last_name"/>
<jsp:setProperty name="regform" property="address"/>

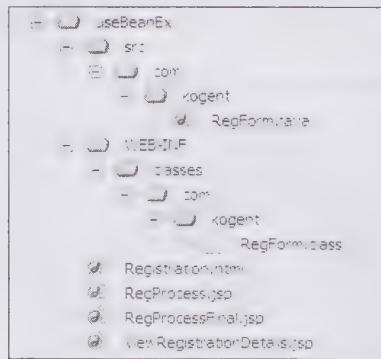
<html>
<body>
    <pre>
        Your registration details are valid,
        <a href="viewRegistrationDetails.jsp">Click</a> to view Registration
        Details and confirm.
    </pre>
</body>
</html>
```

#### NOTE

Before using the `<jsp:setProperty>` or `<jsp:getProperty>` tag in a JSP page, you must declare a JavaBean in that JSP page by using the `<jsp:useBean>` tag.

## Deploying and Running the Application

You must place the pages and other resources of the useBeanEx application as per the correct directory structure, as shown in Figure 7.14:

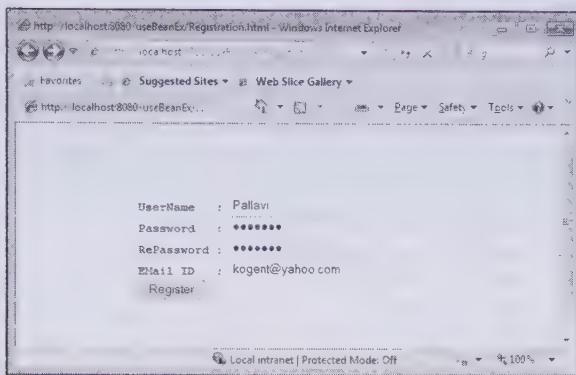


**Figure 7.14:** Showing the Directory Structure for the useBeanEx Application

#### NOTE

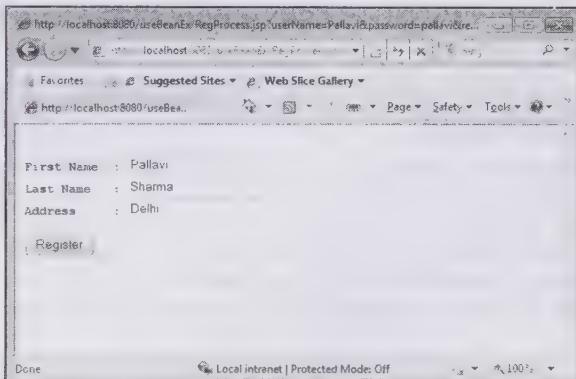
It is optional to include Deployment Descriptor (web.xml), containing an empty `</web-app>` tag in the WEB-INF directory of the useBeanEx application.

Create the Web ARchive (WAR) file for the application and then deploy the useBeanEx application on the Glassfish V3 application server. Next, browse the useBeanEx application by using the `http://localhost:8080/useBeanEx/Registration.html` URL and enter the details shown in Figure 7.15:



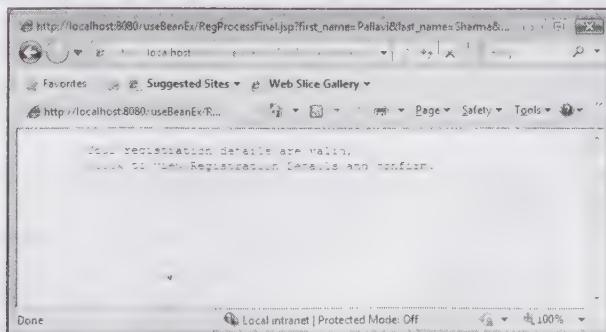
**Figure 7.15: Showing the Output of the Registration HTML Page**

Click the Register button to continue with the registration process. In this application, the user name, password, and e-mail id are stored in the RegBean JavaBean by using the `<jsp:useBean>` and `<jsp:setProperty>` tags. In addition, the RegBean instance is set in the session scope. After entering the details and clicking the Register button (Figure 7.15), the RegProcess JSP page appears, where users are required to provide more information about themselves, as shown in Figure 7.16:



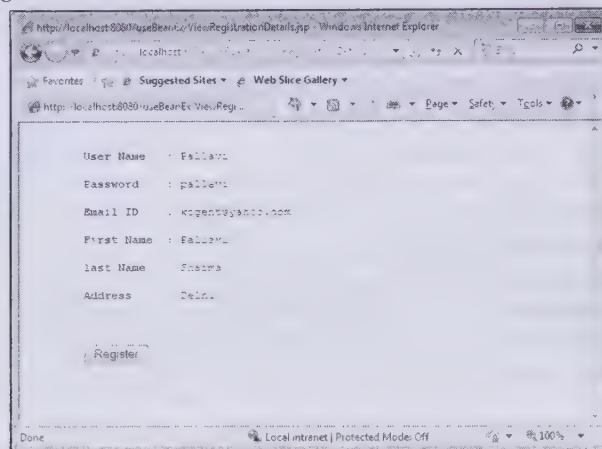
**Figure 7.16: Showing the Registration Process Window**

Clicking the Register button (Figure 7.16) stores the additional information, which includes the first name, last name, and address values, in the RegBean JavaBean. The next screen confirms the validity of the registration details provided by you. After clicking the Register button, you are directed to the RegProcessFinal JSP page, as shown in Figure 7.17:



**Figure 7.17: Showing the RegProcessFinal JSP Page**

Click the hyperlink (Figure 7.17) shown in the RegProcessFinal JSP page to display all the details you have entered, as shown in Figure 7.18:



**Figure 7.18: Displaying the Registration Details of a User**

Figure 7.18 shows the user details confirming that the RegBean JavaBean is successfully created and set. This completes our discussion on JSP action tags. Let's now explore the JSP unified EL, which can also be used to fetch application data.

## Exploring the JSP Unified EL

EL is a language that allows JSP programmers to fetch application data stored in JavaBeans components. It is a new feature introduced in JSP 2.0, and in JSP 2.1, EL of JSP 2.0 and JSF 1.1 are combined to form a single unified EL (EL 2.1). The languages have been combined so that JSP programmers do not have to worry about the different life cycles of JSP and JSF and can take advantage of the new mechanisms that JSF applications use. The EL specification was previously defined in the JSP 2.1 specification. It now has its own independent specification document, which states that EL is generally applicable to a variety of technologies and is not dependent on the JSP specification.

There has been a lot of variation in the methods used to embed the Java code or the business logic in presentation layers. At present, the following methods are used to call the Java code from a JSP page:

- ❑ Placing the entire Java code in a JSP page. This method is useful when small Java code is to be incorporated in a JSP page. This method also requires JSP programmers to be familiar with the Java technology.
- ❑ Defining separate helper classes that encapsulate the entire Java code and calling these helper classes in a JSP page. In this case, the JSP page simply contains few blocks of code to use the helper classes.
- ❑ Using JavaBeans and JSP page action tags, such as `<jsp:useBean>`, `<jsp:getProperty>`, and `<jsp:setProperty>`, to call the Java code.
- ❑ Using JSP EL, which uses short notations to access and display object properties.
- ❑ Creating tag handler classes to embed the Java code. These classes are invoked by using custom tags, which are similar to XML tags.

In the *Declaring a JavaBean in a JSP Page* heading of this chapter, you learned how to access JavaBean properties to show the output data in a JSP page by using standard JSP action tags, such as `<jsp:useBean>` and `<jsp:getProperty>`. However, it becomes complex when we access a JavaBean property, which is a collection or some other JavaBean.

The incorporation of EL to the JSP technology has helped reduce the use of scriptlets in JSP pages. EL expressions cannot be used in JSP scriptlets, expressions, or declaration elements. On the other hand, EL expressions provide short hand notations to retrieve, present, and manipulate Web application data.

Let's now understand the basic syntax of using EL and discuss the various types of EL expressions. You also learn about the different types of tag attributes as well as resolve EL expressions. Apart from this, you learn to work with EL operators and EL objects.

## *Understanding the Basic Syntax of using EL*

EL expressions are enclosed between the \${ and } characters. The most general example of an EL expression is \${object.data}. The object in the preceding expression can be any Java object representing different scopes, such as request, and session. The use of EL expressions greatly reduces the quantity of code written for a task. JSP EL has the following features:

- ❑ Deferred expressions, which can be assessed at various stages of a page's life cycle
- ❑ Method expressions, which call the methods to carry out event handling, validation, and other functions for JSF User Interface (UI) components
- ❑ Value expressions, which can set and get the data of external objects
- ❑ A flexible mechanism, which allows customization of variable and property resolution to evaluate an EL expression
- ❑ Expressions to carry out arithmetic calculations

## *Classifying EL Expressions*

EL expressions can be categorized into the following types:

- ❑ Immediate and deferred expressions
- ❑ Value expressions
- ❑ Method expressions

Let's learn about these in detail in the following sections.

### **Immediate and Deferred Expressions**

Two constructs are used to represent EL expressions, \${expr} and #{expr}. In a JSP page, \${expr} is used for expressions that need to be evaluated immediately and #{expr} is used for expressions that are evaluated at a later time. Therefore, an EL expression that uses the \${expr} syntax is called an immediate expression, and the EL expression that uses the #{expr} syntax is called a deferred expression.

In immediate expressions, the JSP container processes the expression and provides the corresponding response immediately at the time of request handling. Deferred expressions are also introduced in the JSF technology. In deferred expressions, the evaluation of the expressions is deferred due to multiple phases in the JSF life cycle. All operations, such as component data validation and event handling, need to be performed in a specific order. Therefore, the evaluation of a deferred expression is postponed to an appropriate time.

Immediate expressions may be used in template text or as values of the attributes of a JSP tag that takes a runtime value. Such expressions can only read but cannot set the value of a property. The following code snippet is an example of a JSP tag using an immediate expression:

```
<fmt:formatNumber value="${sessionScope.cart.sum}" />
```

In the preceding code snippet, the \${ sessionScope.cart.sum } expression retrieves the value of the sum property from the JavaBean named cart . The sum property specifies the total price of all constituent items in the cart.

In deferred expressions, JSF Controller evaluates the expression in different phases of the JSF life cycle, based on the component in which the expression is used. The following code snippet shows an example of a JSF HTML tag that uses a deferred expression:

```
<h:inputText id="name" value="#{employee.name}" />
```

In the preceding code snippet, when the JSF page containing the <h:inputText> tag is requested, JSF implementation evaluates the #{employee.name} expression in the Render Response phase of the JSF page life cycle. After submitting this page, JSF assesses this expression in more than one phase of its life cycle, during which operations such as retrieving the value of the #{employee.name} expression from the previous page, validating the expression, and setting the value of the expression to the name property of the managed JavaBean

are performed. This means that deferred expressions are lvalue expressions, that is, expressions that can read or write data on a JavaBean. The \${employee.name} expression acts as an rvalue expression, which only reads data during the initial request and acts as the lvalue expression during postback.

## Value Expressions

Value expressions are used to refer to objects such as JavaBeans, collections, enumerations, and implicit objects, and their properties. An object is referred to by using the value expression containing the name of the object. Suppose the \${employee} expression is used in a JSP page, where employee refers to the name of a JavaBean. When the Web container comes across the \${employee} expression, it invokes the PageContext.findAttribute (String) method internally, which searches for the employee JavaBean in the request, session, and application scopes. If the employee JavaBean does not exist, a null value is returned.

To refer to an enum constant by using value expression, a String literal is used. The following code snippet shows the syntax for an enumeration declaration:

```
public enum PlayCardSuite {hearts, spades, diamonds, clubs}
```

The following code snippet shows the syntax to access a constant defined by the PlayCardSuite enumeration by using EL:

```
${PlayCardSuite.spades}
```

The expression shown in the preceding code snippet is an example of a value expression, which is used to refer to the enum constant of the PlayCardSuite enumeration.

Now let's discuss the syntax of value expressions to access the properties and elements of a collection in a JavaBean. For this, we use the (.) and ([ ]) operators. For example, the name property of the employee JavaBean can be referred to by using either the \${employee.name} or \${employee["name"]} expression. Double or single quotes can be used to represent a String literal. To access a specific element of a collection, such as a list or an array inside a JavaBean, the [ ] notation having an int value is used. For example, \${employee.phonenos[1]} accesses the second phone number of the phonenos array.

Value expressions are of two types, rvalue and lvalue. An rvalue expression can only read data, and not write data. On the other hand, the lvalue expression can read as well as write data. Examples of rvalue expressions are \${"hello"}, \${employee.age+20}, and \${false}. EL supports all types of literals in Java, such as Boolean, integer, floating point, and String.

Value expressions may be embedded in static text or as a value of a tag attribute that can take an expression. The following code snippet shows how to embed a value expression in static text:

```
<prefix:tag>
    some text ${expr} some text
</prefix:tag>
```

In the preceding code snippet, the \${expr} expression is embedded within a text String. If the body of the <prefix:tag> tag is dependent on another tag, then the embedded expression is not evaluated.

## Method Expressions

Method expressions are used to call public methods, which return a value or object. Such expressions are usually deferred expressions. JSF HTML tags represent UI components on a JSF page. These tags use method expressions to call functions that perform operations such as validating a UI component or handling the event generated on a UI component. The following code snippet shows the use of method expressions in a JSF page:

```
<h:form>
    <h:inputText
        id="email"
        value="#{employee.email}"
        validator="#{employee.validateEmail}" />
    <h:commandButton
        id="submit"
        action="#{customer.submit}" />
</h:form>
```

The various elements shown in the preceding code snippet can be briefly described as follows:

- The inputText tag – Shows the UI Input component in the form of a text field on a Web page