Enterprise JavaBeans Development Using VisualAge for Java

Joaquin Picon, Patrizia Genchi, Maneesh Sahu, Martin Weiss, Alain Dessureault
Take Note!

Before using this information and the product it supports, be sure to read the general information in Appendix E, “Special Notices” on page 391.

First Edition (June 1999)

This edition applies to VisualAge for Java Enterprise Edition 2.0 with the Enterprise JavaBeans technology update for use with the Windows Operating System and WebSphere Advanced Edition 2.0.

Comments may be addressed to:
IBM Corporation, International Technical Support Organization
Dept. QXXE Building 80-E2
650 Harry Road
San Jose, California 95120-6099

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 1999. All rights reserved
Note to U.S Government Users – Documentation related to restricted rights – Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.
Contents

FIGURES ................................................................................................. xi
Tables ..................................................................................................... xvii
Preface .................................................................................................... xix
The Team That Wrote This Redbook ....................................................... xix
Acknowledgments ................................................................................... xx
Comments Welcome ................................................................................ xx

Part 1. Enterprise JavaBeans Technology: Essentials ..................... 1

Chapter 1. Moving Business Logic To EJB Technology .......... 3
1.1 Stand-Alone Applications ................................................................. 3
1.2 High Volume Transactions ............................................................... 4
1.3 The SanFrancisco Application Frameworks ................................. 4
1.4 Distributed Object-Oriented Technologies ................................. 7
    Microsoft DCOM ........................................................................... 7
    CORBA ......................................................................................... 8
1.5 Summary .......................................................................................... 9

Chapter 2. EJB Beans: Quick Start .................................................... 11
2.1 Roles ................................................................................................. 12
    Provider ......................................................................................... 12
    Deployer ......................................................................................... 12
    Application Assembler ................................................................. 12
    Container Provider ......................................................................... 12
    Server Provider ............................................................................. 13
    System Administrator .................................................................... 13
2.2 Components ...................................................................................... 13
    Container ......................................................................................... 13
    Session and Entity Beans ............................................................. 13
        Session Beans ........................................................................... 14
        Entity Beans ............................................................................ 14
2.3 An Overall Picture ........................................................................... 15

Chapter 3. Developing a Session Bean .............................................. 19
3.1 Session Bean Basics ......................................................................... 19
    Basic Components of a Session Bean .......................................... 19
    Stateless Versus Stateful Session Beans ..................................... 20
Session Bean Life Cycle ............................................... 21
   Creation State ..................................................... 21
   Ready State ........................................................ 21
   Pooled State ........................................................ 22
   Removal State ....................................................... 22
3.2 Developing Session Beans ......................................... 23
   Writing the Enterprise Bean Class ............................... 23
      Implementing the Business Methods ......................... 26
      Implementing the ejbCreate Methods ....................... 28
      Implementing the SessionBean Interface .................... 29
   Writing the Home Interface ..................................... 31
   Writing the Remote Interface .................................. 32
3.3 Developing Session Bean inside VisualAge for Java ......... 34
   Creating EJB Beans .................................................. 34
      First step: Adding EJB Groups ................................ 35
      Second step: Adding EJB Beans to EJB Groups ............. 39
   Generating Deployed Code ....................................... 55
      The deployment descriptor .................................... 56
      The ejb-jar File ................................................... 64
   Deploying an Enterprise Bean .................................... 64
   Generate the Test Client ......................................... 66
   Testing EJB Beans ................................................... 67
      Setting Name Service and EJB Server Properties .......... 68
      Starting the Name Service and EJB Servers ................. 71
      Restarting Servers .............................................. 73
      Running the Generated Test Client ......................... 73
3.4 Writing Your Own Client Application ........................... 79
   Importing Required Java Packages .............................. 80
   Instantiating an EJB Object ...................................... 80
   Stopping the Name Service and EJB Servers ................... 85
3.5 Running the Client outside VisualAge for Java ............... 85
   Exporting EJB Groups and EJB Beans ........................... 85
      Exporting EJB Beans to EJS or EJB Jar Files ............... 85
      Exporting EJB Beans to Client Jar Files .................... 88
   Testing the Client outside VisualAge for Java ............... 90
3.6 Deployment on WebSphere Advanced Edition 2.0 ............. 91
3.7 Deployment for Other Enterprise JavaBeans Servers .......... 97
3.8 Deployment Using WebSphere Tools ............................ 99

Chapter 4. Developing a Container-Managed Persistence Bean 107
4.1 Container-Managed Persistence Basics .......................... 107
4.2 Developing a CMP Bean .......................................... 108
   Creating a CMP Bean ............................................... 109
   Projects, Packages and Reserved Package Organization ...... 118
Understanding the Generated Types .................................................. 120
Adding Fields to the Bean ................................................................. 122
Removing a Field .............................................................................. 127
Regarding the Other Modifiers (Final, Transient, Static) ..................... 129
Working with the Key Class ............................................................... 130
4.3 Understanding the Generated Methods ......................................... 131
Life Cycles of Enterprise Bean Instances ......................................... 132
  Creation State .............................................................................. 132
  Pooled State ............................................................................... 132
  Ready State ................................................................................ 132
  Removal State .......................................................................... 133
The CustomerBean's Generated Methods .......................................... 133
The CustomerHome Interface's Generated Methods ......................... 135
The CustomerKey's Generated Methods ........................................... 136
4.4 Adding Remote Interface Methods .............................................. 137
Methods Inherited from javax.ejb.EJBObject .................................. 139
Using Valid Parameters and Return Values for Java RMI .................. 139
4.5 Modifying or Adding Home Methods .......................................... 140
4.6 Generating Database Schema and mapping ................................... 140
  The Top-Down Approach ............................................................ 143
  The Bottom-Up Approach .......................................................... 146
  The Meet-in-the-Middle Approach .............................................. 146
Creating the CMP Sample Database ................................................. 146
Importing a Database Schema into VisualAge for Java .................... 147
Defining a Map between EJB Beans and the Database Schema .......... 151
4.7 Adjusting the Deployment Descriptor Attributes .......................... 156
4.8 Generating Your Deployed Code .................................................. 158
4.9 Adding Special Finder Methods .................................................. 158
  Insert a Finder Method Declaration in the CustomerHome Interface 158
  Create a Query String in the CustomerFinderHelper Interface .......... 159
4.10 Testing Your Bean ...................................................................... 161

Chapter 5. Developing a Bean-Managed Persistence Bean ............... 171
5.1 Bean-Managed Persistence Entity Bean ....................................... 171
5.2 The TransactionRecord Entity Bean .......................................... 172
  Creating the Entity Bean .............................................................. 172
  Adding and Defining BMP Fields ................................................ 174
  Understanding the Generated Methods ...................................... 176
  Modifying the Key Class .............................................................. 178
  Adding Remote Methods ............................................................. 178
  Modifying and Adding Home Methods ....................................... 179
5.3 XML—The eXtensible Markup Language ...................................... 180
  Introduction to XML ................................................................. 180
Chapter 11. Inheritance .............................................. 271
11.1 Inheritance .................................................. 271
11.2 Requirements for the Inheritance ..................... 272
11.3 Programming Model ...................................... 272
    Remote Interface .......................................... 273
    EJB Homes ................................................ 273
    Primary Key Class ...................................... 273
11.4 Mapping Schemes for Relational Databases .......... 274
11.5 Developing the BankAccount Hierarchy ............. 276
    Developing the Remote Interfaces .................. 276
    The BankAccount Superclass ......................... 277
    Developing Subclass EJB Beans .................... 277
    Proxy Superclass .................................. 278
    Defining the Inheritance Mapping Scheme .......... 280
11.6 Accessing the BankAccount Hierarchy .............. 281
    The TypeConverter Class ................................ 282
11.7 Relationships and Inheritance ....................... 283
    SingleLink ............................................... 284
        Constructor ......................................... 284
        public BankAccount getBankAccount() .......... 284
    ManyLink .............................................. 284
        Constructor ......................................... 284
        public Enumeration getBankAccounts() ....... 284
        public void addBankAccount(BankAccount acc) 285
        public void removeBankAccount(BankAccount acc) 285
        public void removeAllMembers() .................. 285
11.8 The Future of EJB Inheritance ....................... 285

Chapter 12. Bank Implementation Entity Beans .......... 287
12.1 Developing the BankAccount Hierarchy ............. 288
    Developing the BankAccount Bean .................. 288
        Add the CMP Fields ................................ 288
        Modify the Generated Methods .................. 288
        Add the Remote Methods .......................... 289
        Specify the Relationships ....................... 289
    Developing the Children EJB Beans ................. 289
12.2 Developing the Bank Entity Bean .................... 289
    Add Fields ............................................ 290
    Add New Methods ...................................... 292
    Modify the Generated Methods ..................... 293
    Add Methods to the Remote Interface .............. 295
12.3 Developing the Customer Entity Bean ............... 296
Add Fields ....................................................... 296
Add New Methods ............................................ 298
Modify the Generated Methods ......................... 299
Add Methods to the Remote Interface .................. 300
Adding Finder Methods to CustomerHome ............. 301
12.4 Developing the TransactionRecord Entity Bean .... 301
Add Fields .................................................. 302
Add New Methods ............................................ 304
Modify the Generated Methods ......................... 305
Add Methods to the Remote Interface .................. 306
Adding Finder Method to TransactionRecordHome .... 306
12.5 Creating the Bank Schemas ......................... 307
12.6 Creating the Bank Maps .............................. 308

Chapter 13. Bank Implementation UserSB ................. 311
13.1 The UserSB Sequence Diagrams .................... 312
UserSB Customer Authentication Sequence Diagram .... 312
UserSB Customer GetAccounts Sequence Diagram ..... 314
UserSB Customer Deposit Sequence Diagram .......... 315
UserSB Customer Withdraw Sequence Diagram ......... 316
UserSB Customer GetBalance Sequence Diagram ...... 316
UserSB Customer GetHistory Sequence Diagram ...... 317
UserSB Customer Transfer Sequence Diagram .......... 318
UserSB Customer PayBill Sequence Diagram .......... 319
13.2 Creating the UserSB Session Bean ................. 320
Add Fields .................................................. 321
Add New Methods ............................................ 326
Modify the Generated Methods ......................... 332
Add Methods to the Remote Interface .................. 334
Generate Deployed Code .................................. 335

Chapter 14. Bank Implementation AdminSB ............... 337
14.1 Developing the AdminSB Session Bean .............. 338
Add Fields .................................................. 338
Add New Methods ............................................ 339
Modify the Generated Methods ......................... 344
Add Methods to the Remote Interface .................. 345
Generate Deployed Code .................................. 345

Part 3. Appendixes ........................................... 347

Appendix A. Hints and Tips ................................ 349
A.1 Importing EJB Beans from a Jar File ................. 349
Enterprise JavaBeans Development Using VisualAge for Java

A.2 Moving or Copying EJ B. Beans between Repositories ............ 353
A.3 Updating WebSphere Class Path .................................. 354
A.4 Starting and Stopping WebSphere Advanced Edition 2.0 .... 355

Appendix B. Transaction Samples Appendix ......................... 357
B.1 Installing and Running the Transaction Samples ............... 357
   Creating the DB2 Database .................................. 357
   Importing the transaction.dat Repository ...................... 358
   Adding the TransactionSamples Project to the Workspace ..... 358
   Generating the Deployed Code and Running the Samples ...... 359
B.2 Output Samples ................................................. 360

Appendix C. ITSOBANK Database .................................... 369
C.1 Database Definition DDL ....................................... 369
C.2 Sample Data of ITSOBANK Tables ............................. 373

Appendix D. Samples and BANK Application ......................... 377
D.1 Content of the CDROM ......................................... 377
D.2 Running the ITSOBANK application ............................ 378
   Inside VisualAge for Java .................................... 379
   Inside WebSphere Advanced Edition 2.0 ........................ 381
   Testing a bank scenario ..................................... 383

Appendix E. Special Notices ........................................ 391

Appendix F. Related Publications .................................... 395
F.1 International Technical Support Organization Publications .. 395
F.2 Redbooks on CD-ROMs ........................................ 396
F.3 Other Publications ............................................. 396

How To Get Itso Redbooks ........................................... 399
How IBM Employees Can Get ITSO Redbooks ................. 399
How Customers Can Get ITSO Redbooks ......................... 400
IBM Redbook Order Form ........................................ 402

Glossary ................................................................. 403

List of Abbreviations .................................................. 413

Index ................................................................. 415

ITSO Redbook Evaluation ............................................ 423
FIGURES

1. SanFrancisco Basic Architecture .......................... 5
2. SanFrancisco Future Model ................................ 7
3. Global Enterprise JavaBeans View from Client and Server ...... 17
4. The TransferBean Class ..................................... 25
5. The Business Methods of the TransferBean Class ............... 27
6. The ejbCreate Method of the TransferBean Class .............. 29
7. Implementing the SessionBean Interface in the TransferBean Class 31
8. The TransferHome Home Interface ............................ 32
9. The Transfer Remote Interface ............................... 33
10. VisualAge for Java Main Window ............................ 35
11. EJBs Page .................................................. 37
12. Adding a New EJB Group .................................... 38
13. EJB Page after the Creation of the New Group ................ 39
14. Create EJB Bean SmartGuide ............................... 40
15. Create ReadCustomerInfo EJB ............................... 42
16. Adding Import Statements .................................. 43
17. EJB Workspace after the Creation of the New EJB Bean ....... 44
18. Create Field SmartGuide .................................... 47
19. Create fileName Field ..................................... 48
20. Create Method SmartGuide .................................. 50
21. Create Method SmartGuide—First Page ....................... 51
22. Create Method SmartGuide—Second Page ..................... 52
23. ReadCustomerInfoBean in the EJB Workspace ................ 55
24. Properties Window .......................................... 58
25. Environment Properties Page ................................ 59
26. Environment Properties Page after Definition of a Property .... 60
27. Definition of the Properties Field ........................... 61
28. Getting Properties from the Context ........................ 61
29. Retrieve Environment Variable Value from Properties ......... 61
30. Method properties Window .................................. 62
31. Method Selection ........................................... 63
32. Modify Method Control Descriptor Window .................... 63
33. EJB Workspace after the EJB Deployment Code Generation .... 65
34. Generate Test Client ....................................... 67
35. EJB Server Configuration ................................... 68
36. Location Service Properties Dialog .......................... 69
37. Persistent Name Service Property Dialog ...................... 70
38. EJB Server Properties Dialog ............................... 71
39. Console after All Servers Are Started ............................................. 72
40. Connect Window for the ReadCustomerInfo Session Bean ............ 75
41. Home Test Client Page ................................................................. 76
42. Remote Test Client Page ............................................................... 77
43. Testing Remote Methods ............................................................. 78
44. Inspector Window ......................................................................... 79
45. Read Customer Client Window ...................................................... 81
46. Exporting jar Files ......................................................................... 86
47. Export to an EJB jar File ............................................................... 87
48. Export to an EJB Client jar File ..................................................... 89
49. IBM WebSphere Application Server Administration Login ........... 92
50. Container Page inside WebSphere .................................................. 93
51. Deploy Jar File Window ................................................................. 95
52. Redeployment Selection Window .................................................. 95
53. EJB Jar Copy Window .................................................................... 96
54. Exporting to EJB Jar Files ............................................................. 97
55. Undeploy Jar File Window ............................................................. 98
56. EJB Jar Undeploy Result Window ................................................. 99
57. WebSphere Navigation Path ........................................................... 100
58. The jet Tool .................................................................................. 101
59. Message from jet Tool ................................................................. 102
60. The jet Tool after Importing the ReadCustomerInfo Bean ............ 103
61. Using jet Tool to Modify a Property Value .................................... 104
62. The SmartGuide Window .............................................................. 110
63. Project Not Found .......................................................................... 112
64. Create EJB SmartGuide ................................................................. 114
65. EJB Class Attributes and Interfaces Definition .............................. 115
66. Customer CMP Entity Bean Generated Classes ............................. 117
67. Association between EJB Reserved Package and Project ................ 119
68. Association between EJB Class Package and Project ...................... 120
69. CustomerCMP Entity Bean's Fields ............................................... 123
70. Create Field SmartGuide .............................................................. 125
71. New Fields Created ....................................................................... 126
72. CMP Fields Icon .......................................................................... 127
73. Dummy Member Field Declaration ............................................... 128
74. Workbench without Dummy Field ............................................... 129
75. Methods added to the Remote Interface are identified by an Icon ... 138
76. Remote Interface generated Methods ......................................... 139
77. Association Between CMP Fields and Database Columns ............. 141
78. Selection with Schema Browser: Database, Tables, Converters ....... 142
79. With Map Browser Associate Fields and Columns ....................... 143
80. EJB Server Properties ................................................................... 144
81. Characteristics of DB2 Table and Columns Created for CMP Fields 145
125. Creating the EJ BHome Object ........................................... 209
126. Changing the JNDI Name under VisualAge for Java .............. 210
127. Changing the JNDI Name with Jet Tool ............................... 211
128. Creating the EJB Object .................................................. 212
129. Accessing an EJB from a Different Machine......................... 213
130. Opening the EJB Server Configuration .................................. 213
131. EJB Server Configuration .................................................. 214
132. Persistent Name Server Properties ...................................... 214
133. Location for Persistent Naming Information .......................... 215
134. EJB Server Properties ...................................................... 216
135. OTS Overview ............................................................... 223
136. SimpleClient Application .................................................. 226
137. MyTransactionTest.test1() method ...................................... 227
138. MyTransactionTest.test2() Method ...................................... 228
139. MyTransactionTest.test3() Method ...................................... 229
140. Access Control ............................................................... 242
141. System View of the Bank Application .................................... 252
142. Bank Application Class Diagram ......................................... 254
143. Single-table Mapping ........................................................ 274
144. Root/leaf Mapping ........................................................... 275
145. Distinct-table Mapping ...................................................... 275
146. BankAccount Inheritance Hierarchy ...................................... 276
147. Simulating Inheritance by delegation .................................... 278
148. Mapping the BankAccount Hierarchy to the Database ............... 280
149. UserSB Customer Authentication Sequence Diagram ............... 313
150. UserSB GetAccounts Sequence Diagram ............................... 314
151. UserSB Deposit Sequence Diagram ..................................... 315
152. UserSB Withdraw Sequence Diagram ................................... 316
153. UserSB getBalance Sequence Diagram .................................. 317
154. UserSB getHistory Sequence Diagram .................................. 318
155. UserSB transfer Sequence Diagram ..................................... 319
156. UserSB PayBill Sequence Diagram ....................................... 320
157. Properties for the UserSB ................................................... 321
158. UserSB Generated Code ..................................................... 335
159. Import from an EJB Jar File window ..................................... 350
160. Import from CreateCustomerInfo EJB Jar File ....................... 351
161. EJB Workspace after Importing and EJB.jar ......................... 353
162. WebSphere Application Server Java Engine Setup ................... 354
163. ITSOEJBSamples EJB Group .............................................. 359
164. ITSOBANK Database Data Definition Language (Part1) ........... 370
165. ITSOBANK Database Data Definition Language (Part 2) .......... 371
166. ITSOBANK Database Data Definition Language (Part 3) .......... 372
167. ITSOBANK Data (Part 1) .................................................... 373
168. ITSOBANK Data (Part 2) ..................................................... 374
169. ITSOBANK Data (Part 3) ..................................................... 375
170. ITSOBANK Data (Part 4) ..................................................... 376
171. SG24-5429 Directory Structure ............................................. 378
172. Welcome Page ................................................................. 384
173. Login Page ................................................................. 385
174. Account Page ............................................................... 386
175. Account Information Page .................................................. 387
176. Account Balance Page ...................................................... 388
177. Transfer Funds page ......................................................... 389
178. Fund Transferred Page ...................................................... 390
## Tables

1. Container-Managed Fields Mapping .................................................. 145
2. Creating the Different DOM Nodes in the Hierarchy .......................... 185
3. Accessing the Different DOM Nodes .................................................. 186
4. Scenario 1 - TX_REQUIRED ............................................................... 232
5. Scenario 2 - TX_MANDATORY ............................................................. 232
6. Scenario 3 TX_NOT_SUPPORTED .......................................................... 233
7. Scenario 4 TX_REQUIRES_NEW ............................................................ 233
8. Scenario 5 - TX_REQUIRED ............................................................... 234
9. Create Field bankName ................................................................. 290
10. Create Field customerLink ............................................................... 291
11. Create Field accountLink ............................................................... 291
12. Create Fields for the Customer Bean ............................................... 297
13. Create Field bankAccountLink ........................................................ 297
14. Create Field bank ................................................................. 298
15. Create Field accountID and traccID ............................................... 302
16. Create Field transamt ................................................................. 303
17. Create Field transtype ................................................................. 303
18. Create Field accountLink ............................................................... 304
19. Create Bank Schema .................................................................... 307
20. Bank Table Maps ........................................................................ 308
21. Create Bank Datastore Map ............................................................ 308
22. Bank Property Map Relationship ..................................................... 309
23. Create Field bankAccountHome ....................................................... 322
24. Create Field bankHome ............................................................... 322
25. Create Field custKey ................................................................. 323
26. Create Field customerHome ........................................................... 323
27. Create Field customerRef .............................................................. 324
28. Create Field initialCtx ................................................................. 324
29. Create Field loginPassword ............................................................ 325
30. Create Field loginStatus .............................................................. 325
31. Create Field transactionRecordHome .............................................. 326
32. Create Field for AdminSB ............................................................. 339
Preface

With the introduction of the Sun Microsystems Enterprise JavaBeans (EJB) architecture specification, application developers can now focus on writing the business logic necessary to support their application without having to deal with the intricacies of the underlying middleware, which still gives crucial services such as transactions, security, naming and persistency.

IBM is exploiting this specification in a family of compatible Java application servers conforming to IBM’s Enterprise Server for Java (EJS) specification. In addition, IBM has already introduced the support for development of EJB beans into its award-winning Java development tool: VisualAge for Java version 2.0. By including in this new version the EJS runtime, VisualAge for Java provides a unique Rapid Application Development environment to develop, debug and test EJB beans.

In this book we explore the EJB specification in depth and apply this knowledge to develop an application including all different facets of EJB beans: Session bean, Bean-Managed Persistence and Container-Managed Persistence entity beans, transaction (JTS), security, and naming service (JNDI). We also address relationships and inheritance which are issues not yet addressed by the EJB specification.

The Team That Wrote This Redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization San Jose Center.

Joaquin Picon is a consultant at the International Technical Support Organization, San Jose Center. He writes extensively and teaches IBM classes worldwide on application development, object technology, CORBA, and Enterprise JavaBeans technology. Before joining the ITSO, Joaquin worked at the IBM Application Enabling Center Of Competency in France. Joaquin holds a degree in telecommunications from the Institut National de Telecommunications (http://www.int-evry.fr).

Patrizia Genchi is an I/T Specialist in IBM Italy. She has been working for IBM for nine years, and has been involved in networking projects for six years working for the Tivoli lab in Rome. She now works for the Java Technology Center of the IBM EMEA. She has three years of experience in
developing JAVA applications. She has worked as a SanFrancisco Application technical consultant in the last two years. She holds a degree in Mathematics from the University of Bari.

**Martin Weiss** is an Advisory IT Specialist in Switzerland. He has been with IBM since 1978, working in application development (IBM mainframe, AS/400, OS/2, Windows NT). Since 1993 he has focused on object technology, primary as a developer and mentor in VisualAge customer projects (Smalltalk, C++, Java).

**Maneesh Sahu** is a software engineer in IBM, Bangalore. He has two years of experience in developing JavaBeans. He holds a degree in computer engineering from Karnataka Regional Engineering College (http://www.krec.ernet.in). His areas of expertise include developing visually wirable components and application development in VAJava and XML. He has written extensively on the subject of developing Java and XML applications.

**Alain Dessureault** is an IT Architect in IBM Canada. He has 7 years of experience in all phases of the software life cycle (analysis, requirements, architecture, design, construction, implementation and testing). He holds a degree in Computer Science from the Universite Laval. His areas of expertise include object-oriented tools processes and languages, distributed systems and client-server systems architecture, design and programming. He has written extensively on systems architecture and distributed computing.

## Acknowledgments

This book would not have been possible without the help of the following people who contributed to the content of the book:

The editing team and staff at the International Technical Support Organization, San Jose Center

Dave Ings, Howard Borenstein, Daniel Zimmerman
VisualAge for Java Development, IBM Toronto Lab, Toronto

Graeme Dixon, Keith Uher
WebSphere Advanced Edition 2.0 Development, IBM Transarc, Pittsburgh

Elias Bayeh
WebSphere Advanced Edition 2.0 Development, IBM Raleigh Lab, Raleigh
Verlyn Johnson,  
San Francisco Marketing Support, IBM Rochester

David S. Linthicum, Chief Technology Officer, SAGA, http://www.sagus.com

Chris Crenshaw, President, Nova Laboratories, http://www.nova-labs.com

Comments Welcome

Your comments are important to us!

We want our redbooks to be as helpful as possible. Please send us your comments about this or other redbooks in one of the following ways:

❑ Fax the evaluation form found in “ITSO Redbook Evaluation” on page 423 to the fax number shown on the form.

❑ Use the electronic evaluation form found on the Redbooks Web sites:
  For Internet users http://www.redbooks.ibm.com
  For IBM Intranet users http://w3.itso.ibm.com

❑ Send us a note at the following address:
  redbook@us.ibm.com
With the introduction of the Sun Microsystems Enterprise JavaBeans (EJB) architecture specification, we have another new model for creating and packaging business logic. Having a new technology does not mean it has to be used right now in any situation. But you certainly need to understand all the benefit brought by EJB beans and start planning when to introduce this technology in your organization.

1.1 Stand-Alone Applications

If you are going to develop an application running on a single workstation for a single user, you may consider using the Java language, but you may not need to create your business logic as EJB beans.
However, be aware that business logic encapsulated in EJB beans can be directly incorporated into a stand-alone application. This gives you the flexibility to easily transform it into a multi-user, transactional distributed application.

### 1.2 High Volume Transactions

At the other end of the spectrum, you are currently running, or plan to run, high volume transactions applications. In that case, you have probably already been using CICS mainframe software, which manages transactions like selling and shipping products, billing customers and trading stocks, for years. If not, you should start using CICS. As businesses are adopting the Internet, the number of electronic transactions is exploding, and only application servers like CICS can provide the kind of reliability and scalability required by mission-critical applications. While workstation based NT systems expect to be able to support around 1500 concurrent users by end of 1999, CICS can handle as many as 250,000 concurrent users at a time.

In addition to using sophisticated technologies, such as Parallel Sysplex, with resource managers sharing data across the sysplex providing high availability, CICS business transaction services (BTS) allow you to control complex business transactions. Using BTS, a top-level program manages the interrelationship, ordering parallel execution, commit scope, recovery, and restart of the actions that make up the business transaction.

If this environment describes your requirements for running your application, then it may be a little bit too early to start using Enterprise JavaBeans technology, even though EJB support in CICS/390 is coming.

### 1.3 The SanFrancisco Application Frameworks

Assuming your application does not have these high transaction volume constraints, you may be looking to commercial application frameworks. This may be the solution, for a particular domain, to speed up your application development.

A significant group of business management systems capabilities are common within a particular application domain. These capabilities support the core business activities (processes) performed by any business within this domain.
For instance, the amount of overlap among various implementations of General Ledger is significant. Therefore, building an application from scratch means that a large percentage of the total investment goes into building standard components rather than into creating new components that represent added value.

These considerations led to the idea of commercial application frameworks, which are a set of classes representing business entities that cooperate to implement core business processes. The availability of object-oriented frameworks allows developers to rely on a base of existing business processes and business objects.

The SanFrancisco Application Business Components (SanFrancisco) is the answer IBM gives to this requirement. This is a set of components that address the commercial area, such as General Ledger (GL), Account Payable/Account Receivable (AR/AP) Warehouse Management (WM), and Order Management (OM).

SanFrancisco has a strong technological infrastructure and has a layered architecture that shields developers from implementation details of function, such as distributed computing and transaction management.

The SanFrancisco architecture is made of three integrated layers:
- Foundation
- Common Business Objects (CBOs)
- Core Business Processes (CBPs)

The architecture is illustrated in Figure 1.
The Foundation layer is the core technological layer of SanFrancisco and provides the fundamental services such as distributed object creation, synchronization, persistence, and a consistent application development model. It encapsulates the technological aspects of cross-platform distributed object management and provides an easy-to-use API. It also includes support for security, provides distributed transaction processing, and forms a middleware between a client application and server.

The Common Business Objects layer is built on top of the Foundation layer. It provides advanced components like Company, Address, BusinessPartner that are common across business domain. It provides generalized mechanisms to be used to solve business application problems.

Core Business Processes are the top layer of SanFrancisco. This layer provides the fundamental behavior and structure for particular domains. SanFrancisco provides the behaviors and structures needed to allow developers to customize the Core Business Processes for use in their applications. For example, SanFrancisco contains the General Ledger Core Business Process, which includes the architecture, design, and default logic to build a General Ledger application. The developer, instead of building it from scratch, needs only to enhance and extend this layer to build a customized General Ledger application. (For details, refer to the redbook SanFrancisco Concepts and Facilities, SG24-2157-00.)

Building applications with SanFrancisco allows the developer to concentrate only on business needs, without taking care of technological issues like distributing computing.

Today the SanFrancisco Foundation overlaps the functions provided by EJB servers. It also provides a programming model which includes functions not yet addressed by EJB servers. On the other hand, EJB servers include some functions not provided by SanFrancisco. For example, CORBA access is not delivered by SanFrancisco, because SanFrancisco does not support IIOP access. You can integrate SanFrancisco business components with emerging technologies such as WebSphere, but components built for one environment will not automatically run in the other.

If you want to build a GL application which is CORBA compliant, you may choose to build your own EJB components, but it has to be clear that in this case you have to write 100% of your business code. You cannot use the SanFrancisco business components as the starting point for a CORBA compliant application.
IBM has announced that it plans to migrate the SanFrancisco business components to run on the WebSphere family of servers. The SanFrancisco component development started long before the EJB specification was released. But the SanFrancisco programming model will be migrated to the EJB server model. This will provide CORBA access to SanFrancisco components. The Websphere family of EJB servers will be enhanced to provide the functions needed by the new version of the SanFrancisco components, as shown in Figure 2.

![SanFrancisco Future Model](image)

**Figure 2. SanFrancisco Future Model**

### 1.4 Distributed Object-Oriented Technologies

The growth of Object-Oriented technologies has facilitated the creation of standards providing the frameworks necessary to develop within a 3-tier architecture. The Object Management Group (OMG) defined the Common Object Request Broker Architecture (CORBA) standard in 1991 to help vendors develop applications that can cooperate in a networking environment. Microsoft did the same in its Distributed Common Object Model (DCOM).

**Microsoft DCOM**

The Distributed Component Object Model (DCOM) is Microsoft's solution to distributed object architectures. It uses Object Remote Procedure Calls (ORPCs) to carry messages between components. Key features engineered
into the DCOM architecture include language independence, transport neutrality, and static and dynamic object method invocation. Because DCOM supports Windows platforms, DCOM-based applications can take full advantage of existing Microsoft services such as security and transactions. Some vendors are migrating DCOM to other platforms such as UNIX, and DCOM eventually will become truly cross-platform.

One thing is clear, DCOM / MTS applications will need to interoperate with EJB beans and coexist on the same machine.

**CORBA**

In the recent years, several companies have put on the market products supporting the CORBA standard.

There is no doubt that CORBA offers the most mature environment for distributed objects. Even though not a single vendor provides the entire set of object services, they still offer the largest set of services necessary to run such distributed object oriented applications.

In addition, with a product like Component Broker, developers do not need to know how to program persistence, security, or transactional integrity, because the appropriate code is generated for them. Therefore, IT organizations can develop their business logic, then select the qualities of service necessary for the deployment of the business application. This idea of separating deployment concerns from actual business logic is built into Component Broker from the very beginning. However, the contract between Component Broker's business objects and the containers in which they are executed has not been standardized.

Such a contract between business objects and their containers has been standardized in the form of the Enterprise JavaBeans specification. This specification opens a new world of opportunities for tool builders, ranging from those who build high-end enterprise-class tools, to IDE tool vendors who simplify the creation, assembly, and reuse of components during development and deployment.

Today you can already deploy your EJB beans into Component Broker's containers. As the EJB specification evolves, EJB beans running in Component Broker will benefit from advanced features such as caching and queries already available to Component Broker's business objects.
1.5 Summary

The Enterprise JavaBeans specification helps significantly in the separation of business logic from the middleware intricacies of dealing with major services like persistence, transaction, naming and security.

If this is not yet a technology you can start using in any situation, the situation may change very quickly. As we have seen major commercial frameworks, CORBA, and high transaction volume environments have plans to support EJB technology soon. Developing business logic in the form of EJB beans may give a complete platform independence, and for the first time, a unique capability to move from the workstation to more reliable, secure, and scalable mainframe environments by the magic of redeployment.

Before we start looking at how to develop EJB beans with VisualAge for Java, we propose to refresh your mind with a quick description of the major components and roles found in the EJB specification.
2 EJB Beans: Quick Start

Since Sun Microsystems' March 1998 announcement of its Enterprise JavaBeans (EJB) 1.0 specification, most of the companies already involved in CORBA, transaction monitors, or databases have announced that their products will implement the specification.

The EJB specification provides a solution for a clear separation of the business logic and the intricacies of dealing with persistency, transactions, and other middleware-related services.

Now you can create applications and concentrate exclusively on the business logic packaged as EJB beans. The difficult part of dealing with all other services required by applications is provided by a container framework in which EJB beans are deployed and executed.

Clients of EJB beans are of course new EJB clients using Remote Method Invocation (RMI) and Java Naming and Directory Interface (JNDI) services. However, CORBA clients written in C++, as ActiveX, or in Java can be EJB clients as well. By using CORBA services, a CORBA client can have full access to EJB beans.
2.1 Roles

The separation of the business logic from other tasks requires the definition of different roles.

Provider

The EJB beans provider is the developer who understands the business logic and knows how to translate it into Java code. The provider needs to have an understanding of the interfaces and semantics of EJB beans.

The EJB beans provider is responsible for deciding, among other issues, whether the enterprise bean is persistent or not. If the enterprise bean is persistent, its provider must provide a list of fields with persistent state. The provider also must decide whether they are going to manage the persistency, using such support as files, databases, or CICS transactions, or let the container manage the persistency. Another decision the provider must make concerns the behavior of the enterprise bean when it is involved in the scope of a transaction. The provider’s responsibility ends with the packaging of the EJB beans and all associated files as a jar file, which is then given to the deployer.

Deployer

The EJB beans deployer is responsible for installing the EJB classes in the EJB server. The deployer understands EJB technology and the runtime server environment, and can map the developer’s requirements and configure them using EJB server tools. The deployer also must ensure that the EJB beans are accessible through JNDI.

Application Assembler

The application assembler writes applications that use EJB beans. The assembler has a client view of the EJB application and can create new, composed EJB beans and develop applets, servlets, or native CORBA applications that use EJB beans.

Container Provider

The container provider focuses mainly on providing scalability, security, and transactional behavior for EJB beans. The accompanying tools generate the "glue" code that creates the link between the business logic and the
underlying services. The container intercepts client calls to the EJB beans and executes the appropriate services that ensure the transactional, persistent behavior expected for the EJB beans.

**Server Provider**

The server provider brings operating system and middleware services to the container. In most cases, the same vendor provides the server and container, because there are as yet no specifications to define the interface between a container and a server.

**System Administrator**

The system administrator ensures that the system is working properly by using the monitoring and management tools provided by the server and container providers.

## 2.2 Components

In this section we present an overview of the major EJB components.

**Container**

A container is where an EJB bean lives. It can contain one or more EJB classes. These classes are identified by their home interface, which acts like a factory that allows clients to create, find, and remove instances of an EJB bean.

By using the Deployment Descriptor during EJB development, or later on by using deployment tools, it is possible to give a meaningful name to the home interface. The container places this name in the naming space that is accessible from clients using JNDI.

**Session and Entity Beans**

There are two types of EJB beans:

- Session beans
- Entity beans
Session Beans

The life time of a session bean is typically that of its client. A session bean contains conversational states that are not persistent and will not survive a server failure. However, a session bean must indicate to its container a state management mode:

- STATELESS
- STATEFUL

In order for a container to manage efficiently a large number of bean instances, it can take an instance out of memory and store it in permanent storage. This is called passivation. When this bean is invoked again, the container creates a new instance and initializes it with the data saved during passivation. This is called activation.

Therefore, if the session bean contains a conversational state that must be preserved between method invocations, the session bean indicates STATEFUL management mode. Otherwise, the container considers the session bean STATELESS and never uses passivation, but instead can destroy it in case of memory resource shortage. Because all instances of stateless session beans are the same, the container can decide to use any available instance to satisfy a client request.

Entity Beans

Entity beans are used to represent persistent data. In the most common case, their fields are mapped into a relational database. But in more complex situations, the persistent data can result from the invocation of an application or the execution of an existing CICS transaction.

The persistent state of an entity bean is defined in two flavors:

1. **Bean-managed persistence (BMP)**
   With a BMP entity bean, the EJB beans provider manages the persistent state of the bean by coding database calls or any type of access to permanent storage. It is the provider’s responsibility to save and restore the state of the EJB bean when called by the container on such methods as: ejbFind, ejbLoad, and ejbStore.
   A BMP entity bean is inappropriate for large applications. This becomes obvious when you think about a large number of entity beans, each accessing a given database. For portability and scalability, use a Container-managed persistence entity bean.

2. **Container-managed persistence (CMP)**
   With CMP entity beans, you do not have to know which source is used to provide the persistent state of the bean. You just have to specify which fields are persistent. Thus there is complete portability, and the EJB provider can focus on the business logic.
The difficult part is for the container. Each container provider also provides the tools to map EJB fields to databases or any existing application. In addition, it can efficiently manage database access by using a shared pool of connections and caching data. Vendors will differentiate their offerings by the number of containers they provide to access different sources, and the reliability and scalability of their implementation.

When CMP is used, an EJB bean can be transparently moved from one container to another regardless of whether it is provided by the same vendor. The business logic remains untouched!

Every entity bean has a unique identity within its home. Its identity is defined by a primary key. The primary key can be any class type. The only restriction is that the class must implement `serializable`, because it must be possible to pass the primary key between the client and the server.

The primary key is used by the client to create or find an instance of an EJB bean.

### 2.3 An Overall Picture

Without going into extensive details, let us look at how the major components relate to each other from a developer's perspective.

Let us assume you want to develop the EmployeeID EJB bean. You first have to buy an EJB server with appropriate containers supporting the source of the persistency of your data. With the containers, the vendor must also provide tools to deploy your EJB beans packaged in jar files.

Assuming you have all this, you can start developing your EJB beans.

For each EJB bean, you need to decide whether it is a session bean or an entity bean. If it is a session bean, you have to inform its container about its management mode: STATEFUL or STATELESS. For an entity bean with maximum portability, scalability, and reliability, select a CMP bean. Otherwise implement it as a BMP bean. But do not forget that mixing persistency state management with business logic is not good for portability.

For each bean, you have to provide an implementation, additional classes, and interfaces. Assuming that your bean name is EmployeeID, you have to:

1. Develop an EmployeeIDBean class
2. Develop an EmployeeID interface
3. Develop an EmployeeIDHome interface
4. If it is an entity bean, provide an EmployeeIDKey
5. Create a deployment descriptor file
6. Create a manifest file
7. Package all of these files inside a deployment jar that the container deployment tool uses to install and execute the EJB beans.

The information set in the deployment descriptor contains the beanHomeName. This is the JNDI name you give to the EmployeeIDHome. It can be any name that makes sense in your organization, such as “applications/travelExpenses.” At deployment time, the deployer can still change it. When the container instantiates the EJB bean home, it registers in the naming space this name bound to EmployeeIDHome (see Figure 3).

Later on, an EJB client uses the JNDI API to look up the home and then continues by creating, finding, or removing instances of the object managed by its home.
One of the major benefits of EJB technology comes from the separation of the business logic from the middleware supporting it. The specification also defines the contract with the container provider who provides such services as transactions, persistency, and security and the tools to generate the glue code that realizes the link between these services and the business logic.

To make the EJB offering even more attractive, VisualAge for Java 2.1 comes with complete EJB development support, including WebSphere 2.0 runtime. You can now develop your EmployeeIDBean and let the tool generate every thing else including the container deployment classes. You can develop, debug, and test any type of EJB beans: session beans, BMP entity beans, or CMP entity beans. VisualAge for Java also generates the jar file with all information required to deploy it in containers such as WebSphere, Component Broker, or any other vendor container. All these capabilities are detailed in this book.
3 Developing a Session Bean

3.1 Session Bean Basics

Session bean support is mandatory for an EJB 1.0 specification compliant container. Session beans are transient objects that exist for the duration of a single user session. A session represents work in progress on behalf of a single client.

In this chapter, we look at what needs to be implemented by a developer: how he deploys, debugs, and tests a session bean.

Basic Components of a Session Bean

Every session bean must have the following components:

- Bean class
  This class encapsulates the data associated with the enterprise bean and contains the developer-implemented business methods that access this data. It also contains the methods used by the container to manage the life cycle of an enterprise bean instance. Clients (whether they are other...
enterprise beans or user applications) never access objects of this class directly; instead, they use the container-generated classes associated with the home and remote interfaces to manipulate the enterprise bean.

- **Home interface**
  This interface contains the methods used by the client to create and remove instances of the enterprise bean. This interface is implemented by the container during enterprise bean deployment in a class known generically as the EJB home class.

- **Remote interface**
  Once the client has used the home interface to gain access to an enterprise bean, it uses this interface to invoke the business methods defined in the bean class. This interface is implemented by the container during enterprise bean deployment in a class known generically as the EJB object class.

Unlike an entity bean, a session bean does not have a primary key class. A session bean does not require a primary key class because instances of session beans cannot be uniquely identified from each other.

### Stateless Versus Stateful Session Beans

Session beans encapsulate data and methods associated with a user session, task, or ephemeral object. By definition, the data in a session bean instance is ephemeral; if it is lost, no real harm is done. For example, at a bank, a session bean might represent a funds transfer, the creation of a customer profile or new account, and a withdrawal or deposit. If a fund transfer is being typed in when a server crashes, the data associated with the bank accounts involved always remains consistent. Only the transfer data, which can always be retyped, is lost.

The manner in which a session bean is designed determines whether its data is shorter-lived or longer-lived:

- If a session bean needs to maintain specific data across methods, it is referred to as a stateful session bean. When a session bean maintains data across methods, it is said to have a conversational state. A Web-based shopping cart applet is a classic example of a stateful session bean. As the shopping cart user adds or subtracts items to or from the shopping cart, the underlying enterprise bean must maintain a record of the content of the cart. Once a particular client begins using an instance of a stateful session bean, the client must continue to use that instance as long as the specific state of that instance is required. If the bean is lost before the contents of the shopping cart is committed to an order, the shopper must load up a new shopping cart.
If a session bean does not need to maintain specific data across methods, it is referred to as a stateless session bean. The example Transfer enterprise bean developed in “The TransferBean Class” on page 25 provides an example of a stateless session bean. For stateless session beans, a client can use any instance to invoke any of the session bean’s methods.

**Session Bean Life Cycle**

This section describes the life cycle of a session bean instances. Differences between stateful and stateless session beans are noted.

**Creation State**

A session bean’s life cycle begins when a client invokes a create method defined in the bean’s home interface. In response to this method invocation, the container does the following:

1. Creates a new memory object for the session bean instance.
2. Invokes the session bean’s setSessionContext method. (This method passes the session bean instance a reference to a session context interface that can be used by the instance to obtain container services and get information about the caller of a client-invoked method.)
3. Invokes the session bean’s ejbCreate method corresponding to the create method called by the client.

**Ready State**

Once a session bean instance is created, it moves to the ready state of its life cycle. In this state, clients can invoke the bean’s business methods defined in the remote interface. The actions of the container at this state are determined by whether a method is invoked transactionally or nontransactionally:

- **Transactional method invocations**
  When a client invokes a transactional business method, the session bean instance is associated with a transaction. Once a bean instance is associated with a transaction, it remains associated until that transaction completes. (Furthermore, an error results if a client attempts to invoke another method on the same bean instance if invoking that method causes the container to associate the bean instance with another transaction or with no transaction.) The container then invokes the following methods:

  1. The afterBegin method, if that method is implemented by the bean class. (The bean needs to implement the SessionSynchronization interface).
2. The business method in the bean class that corresponds to the business method defined in the bean's remote interface and called by the client.

3. The bean instance's beforeCompletion method, if that method is implemented by the bean class and if a commit is requested prior to the container's attempt to commit the transaction.

The transaction service then attempts to commit the transaction, resulting either in a commit or a rollback. When the transaction completes, the container invokes the bean's afterCompletion method, passing the completion status of the transaction (either commit or rollback).

If a rollback occurs, a stateful session bean can roll back its conversational state to the values contained in the bean instance prior to beginning the transaction. Stateless session beans do not maintain conversational state, so they do not need to be concerned about rollbacks.

Nontransactional method invocations
When a client invokes a nontransactional business method, the container simply invokes the corresponding method in the bean class.

Pooled State
The container has a sophisticated algorithm for managing which enterprise bean instances are retained in memory. When a container determines that a session bean instance is no longer required in memory, it invokes the bean instance's ejbPassivate method and moves the bean instance into a reserve pool. When this method returns, the container saves the instance's state to a data source. A session bean instance cannot be passivated when it is associated with a transaction.

If a client invokes a method on a passivated session bean instance, the container activates the instance by restoring the instance's state from the data source and then invoking the bean instance's ejbActivate method. When this method returns, the bean instance is again in the ready state. Because every stateless session bean instance of a particular type is the same as every other instance of that type, stateless session bean instances are not passivated or activated. These instances exist in a ready state at all times until their removal.

Removal State
A session bean's life cycle ends when a client or the container invokes a remove method defined in the bean's home interface and remote interface. In response to this method invocation, the container calls the bean instance's ejbRemove method.
If you attempt to remove a bean instance while it is associated with a transaction, the javax.ejb.RemoveException is thrown. Once a bean instance is removed, any attempt to invoke a method on that instance causes the java.rmi.NoSuchObjectException to be thrown.

A container can implicitly call a remove method on an instance after the lifetime of the EJB object has expired. The lifetime of a session EJB object is set in the deployment descriptor with the timeout attribute.

### 3.2 Developing Session Beans

In their basic make up, session beans are similar to entity beans. However, the purpose of these two types of enterprise beans is very different.

From a component point of view, one of the biggest differences between the two types of enterprise beans is that session beans do not have a primary key class. Session enterprise beans do not require primary keys because session EJB objects are created, associated with a specific client, and then removed as needed, whereas entity EJB objects represent permanent data in a data storage area that can be uniquely identified with a primary key. Because the data for session beans is never permanently stored, the session bean class does not have methods for storing data to and loading data from a data source.

Every session bean must contain the following basic parts:

- The enterprise bean class. For more information, see “Writing the Enterprise Bean Class” on page 23.

- The enterprise bean's home interface. For more information, see “Writing the Home Interface” on page 31.

- The enterprise bean's remote interface. For more information, see “Writing the Remote Interface” on page 32.

### Writing the Enterprise Bean Class

A session bean class defines and implements the business methods of the enterprise bean, and implements the methods used by the container:

- During the creation of enterprise bean instances

- To inform the enterprise bean instance of significant events in the instance's life cycle.
By convention, the enterprise bean class is named NameBean, where Name is the name you assign to the enterprise bean. The enterprise bean class for the example Transfer enterprise bean is named TransferBean.

Every session bean class must meet the following requirements:

- It must be public, it must not be abstract, and it must implement the javax.ejb.SessionBean interface. For more information, see “Implementing the SessionBean Interface” on page 29.
- It must define and implement the business methods that execute the tasks associated with the enterprise bean. For more information, see “Implementing the Business Methods” on page 26.
- It must define and implement an ejbCreate method for each way in which you want it to be able to instantiate the enterprise bean class. For more information, see “Implementing the ejbCreate Methods” on page 28.

Stateful session beans may need to synchronize their conversational state with the transactional context in which they operate. For example, a stateful session bean may need to reset the value of some of its variables if a transaction is rolled back, or it may need to change these variables if a transaction successfully completes.

If a bean needs to synchronize its conversational state with the transactional context, the bean class must implement the javax.ejb.SessionSynchronization interface. This interface contains methods to notify the session bean when a transaction begins, when it is about to complete, and when it has completed. The enterprise bean developer can use these methods to synchronize the state of the session enterprise bean instance with ongoing transactions.

Note: The enterprise bean class can implement the enterprise bean’s remote interface, but this is not recommended. If the enterprise bean class implements the remote interface, it is possible to inadvertently pass the this variable as a method argument.

Figure 4 shows main parts of the enterprise bean class for the example Transfer bean. The sections that follow discuss these parts in greater detail. A session bean can be either stateful or stateless. In a stateless session bean, none of the methods depend on the values of variables set by any other method, except for the ejbCreate method that sets the initial (identical) state of each bean instance. In a stateful enterprise bean, one or more methods depend on the values of variables set by some other method. The Transfer bean is stateless. If the Transfer bean’s transferFunds method was dependent on the value of the balance variable returned by the getBalance method, the TransferBean would be stateful.
import java.rmi.RemoteException;
import java.util.Properties;
import javax.ejb.*;
import java.lang.*;
import javax.naming. *
import com.ibm.ejs.doc.account. *
public class TransferBean implements SessionBean {
    private SessionContext mySessionCtx = null;
    private InitialContext initialContext = null;
    private AccountHome accountHome = null;
    private Account fromAccount = null;
    private Account toAccount = null;

    private void ejbActivate() throws RemoteException { }

    private void ejbCreate() throws RemoteException {
        ...
    }

    private void ejbPassivate() throws RemoteException { }

    private void ejbRemove() throws RemoteException { }

    public float getBalance(long acctId) throws FinderException,
    RemoteException {
        ...
    }

    public void setSessionContext(javax.ejb.SessionContext ctx)
    throws java.rmi.RemoteException {
        ...
    }

    public void transferFunds(long fromAcctId, long toAcctId, float amount)
    throws java.rmi.RemoteException {
        ...
    }
}

Figure 4. The TransferBean Class
Implementing the Business Methods

The business methods of a session bean class define the ways in which the data encapsulated in instances of the enterprise bean can be manipulated. The business methods implemented in the enterprise bean class cannot be directly invoked by a client. Instead, the client invokes the corresponding methods defined in the enterprise bean's remote interface, by using an EJB object associated with an instance of the enterprise bean, and the container invokes the corresponding methods in the enterprise bean instance.

Therefore, for every business method implemented in the enterprise bean class, a corresponding method must be defined in the enterprise bean's remote interface. The enterprise bean's remote interface is implemented by the container in the EJB object class when the enterprise bean is deployed.

Figure 5 shows the business methods for the TransferBean class. The getBalance method is used to get the balance for an account. It first locates the appropriate Account EJB object and then calls that object's getBalance method.

The transferFunds method is used to transfer a specified amount between two accounts (encapsulated in two Account entity EJB objects). After locating the appropriate Account EJB objects by using the findByPrimaryKey method, the transferFunds method calls the add method on one account and the subtract method on the other.

Like all finder methods, findByPrimaryKey can throw both the FinderException and RemoteException exceptions. The try/catch blocks are set up around invocations of the findByPrimaryKey method to handle the entry of invalid account IDs by users. If the session bean user enters an invalid account ID, the findByPrimaryKey method cannot locate an EJB object, and the finder method throws the FinderException exception. This exception is caught and converted into a new RemoteException exception containing information on the invalid account ID.

Obtaining the EJB home object is discussed in “Implementing the ejbCreate Methods” on page 28.
public class TransferBean implements SessionBean {
    ...
    private Account fromAccount = null;
    private Account toAccount = null;
    ...
    public float getBalance(long acctId) throws FinderException, RemoteException {
        AccountKey key = new AccountKey(acctId);
        try{
            fromAccount = accountHome.findByPrimaryKey(key);
        } catch(FinderException ex) {
            throw new FinderException("Account " + acctId + " does not exist.");
        }
        return fromAccount.getBalance();
    }
    ...
    public void transferFunds(long fromAcctId, long toAcctId, float amount)
    throws RemoteException, InsufficientFundsException, FinderException {
        AccountKey fromKey = new AccountKey(fromAcctId);
        AccountKey toKey = new AccountKey(toAcctId);
        try {
            fromAccount = accountHome.findByPrimaryKey(fromKey);
        } catch(FinderException ex) {
            throw new FinderException("Account " + fromAcctId + " does not exist.");
        }
        try {
            toAccount = accountHome.findByPrimaryKey(toKey);
        } catch(FinderException ex) {
            throw new FinderException("Account " + toAcctId + " does not exist.");
        }
        try {
            toAccount.add(amount);
            fromAccount.subtract(amount);
        } catch(InsufficientFundsException ex) {
            throw new InsufficientFundsException("Insufficient funds in " + fromAcctId);
        }
    }
    ...
}

Figure 5. The Business Methods of the TransferBean Class
Implementing the ejbCreate Methods

You must define and implement an ejbCreate method for each way in which you want an enterprise bean to be instantiated. A stateless session bean must have only one ejbCreate method, which must return void and contain no arguments; a stateful session bean can have multiple ejbCreate methods. Each ejbCreate method must correspond to a create method in the enterprise bean’s home interface. (Note that there is no ejbPostCreate method in a session bean as there is in an entity bean.) Like the business methods of the enterprise bean class, the ejbCreate methods cannot be invoked directly by the client. Instead, the client invokes the create method in the bean instance’s home interface, and the container invokes the ejbCreate method. If an ejbCreate method is executed successfully, an EJB object is created.

Each ejbCreate method must meet the following requirements:

- It must return void.
- It must contain code to set the values of any variables needed by the EJB object.

Figure 6 shows the ejbCreate method required by the TransferBean class. When a container invokes this method, the static method getInitialContext is called to set the enterprise bean’s initialContext variable. Next, a JNDI lookup is performed by calling the initialContext object’s lookup method and passing the name of the Account enterprise bean in string form. The lookup yields a temporary object of type java.lang.Object, which is passed to the static method narrow contained in the Account enterprise bean’s AccountHomeHelper class to get the AccountHome EJB object. (The AccountHomeHelper class is one of the helper classes generated when the Account enterprise bean is deployed.)
Developing a Session Bean

Implementing the SessionBean Interface

Every session bean class must implement the methods inherited from the javax.ejb.SessionBean interface. The container invokes these methods to inform the enterprise bean instance of significant events in the instance's lifecycle. All of these methods must be public, return void, and throw the java.rmi.RemoteException exception.

```
public class TransferBean implements SessionBean {
    private javax.ejb.SessionContext mySessionCtx = null;
    private InitialContext initialContext = null;
    private AccountHome accountHome = null;

    public void ejbCreate() throws RemoteException {
        // Get the initial context
        try {
            Properties properties = new Properties();
            ... properties.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY, 
            "com.ibm.jndi.CosNaming.CNInitialContextFactory");
            initialContext = new InitialContext(properties);
            } catch(Exception ex) {
            ... }
            
            // Lookup the home interface using the JNDI name
            try {
            java.lang.Object o = initialContext.lookup("Account"); // this is the JNDI name
            if (o instanceof org.omg.CORBA.Object) {
            accountHome = AccountHomeHelper.narrow((org.omg.CORBA.Object) o);
            } else { // Unexpected object type
            ... }
            } catch (Exception e) { // Error getting the home interface
            ... }
        }
    }
```

Figure 6. The ejbCreate Method of the TransferBean Class
❑ **ejbActivate**

This method is invoked by the container when the container selects an enterprise bean instance from the instance pool and assigns it a specific existing EJB object. This method must contain any code that you want to execute when the enterprise bean instance is activated.

❑ **ejbPassivate**

This method is invoked by the container when the container disassociates an enterprise bean instance from its EJB object and places the enterprise bean instance in the instance pool. This method must contain any code that you want to execute when the enterprise bean instance is passivated (deactivated).

❑ **ejbRemove**

This method is invoked by the container when a client invokes the remove method inherited by the enterprise bean's home interface (from the javax.ejb.EJBHome interface). This method must contain any code that you want to execute when an enterprise bean instance is removed from the container.

❑ **setSessionContext**

This method is invoked by the container to pass a reference to the javax.ejb.SessionContext interface to a session bean instance. If an enterprise bean instance needs to use this context at any time during its life cycle, the enterprise bean class must contain an instance variable to store this value. This method must contain any code required to store a reference to the context.

As shown in Figure 7, except for the setSessionContext method, all of these methods in the TransferBean class are empty because no additional action is required by the bean for the particular life cycle states associated with these methods. The setSessionContext method is used in a conventional way to set the value of the mySessionCtx.
Writing the Home Interface

A session bean’s home interface defines the methods used by clients to create and remove instances of the enterprise bean and obtain meta data about an instance. The home interface is defined by the enterprise bean developer and implemented in the EJB home class created by the container during enterprise bean deployment. The container makes the home interface accessible to clients through the Java Naming and Directory Interface (JNDI).

By convention, the home interface is named NameHome, where Name is the name you assign to the enterprise bean. For example, the Transfer enterprise bean’s home interface is named TransferHome.

Every session bean’s home interface must meet the following requirements:

- It must extend the javax.ejb.EJBHome interface. The home interface inherits several methods from the javax.ejb.EJBHome interface.
- Each method in the interface must be a create method, that corresponds to a ejbCreate method in the enterprise bean class. Unlike entity beans, the home interface of a session bean contains no finder methods.
- The parameters and return value of each method defined in the interface must be valid for Java RMI. In addition, each method's throws clause must include the java.rmi.RemoteException exception class.
Figure 8 shows the relevant parts of the definition of the home interface (TransferHome) for the example Transfer bean. This interface defines one create method.

```java
import javax.ejb.*;
import java.rmi.*;
public interface TransferHome extends EJBHome{
  Transfer create() throws CreateException, RemoteException;
}
```

Figure 8. The TransferHome Home Interface

A create method is used by a client to create an enterprise bean instance. A stateful session bean can contain multiple create methods; however, a stateless session bean can contain only one create method with no arguments. This restriction on stateless session beans ensures that every stateless session bean instance is the same as every other instance of the same type. (For example, every Transfer bean instance is the same as every other Transfer bean instance.)

Each create method must be named create and it must have the same number and types of arguments as a corresponding ejbCreate method in the EJB object class. The return types of the create method and its corresponding ejbCreate method are always different. Each create method must meet the following requirements:

- It must return the type of the enterprise bean’s remote interface. For example, the return type for the create method in the TransferHome interface is Transfer.

- It must have a throws clause that includes the java.rmi.RemoteException exception, the javax.ejb.CreateException exception class, and all of the exceptions defined in the throws clause of the corresponding ejbCreate method.

**Writing the Remote Interface**

A session bean’s remote interface provides access to the business methods available in the enterprise bean class. It also provides methods to remove an enterprise bean instance and to obtain the enterprise bean’s home interface and handle. The remote interface is defined by the enterprise bean developer and implemented in the EJB object class created by the container during enterprise bean deployment.
By convention, the remote interface is named Name, where Name is the name you assign to the enterprise bean. For example, the Transfer enterprise bean's remote interface is named Transfer.

Every remote interface must meet the following requirements:

- It must extend the javax.ejb.EJBObject interface. The remote interface inherits several methods from the EJBObject interface.
- You must define a corresponding business method for every business method implemented in the enterprise bean class.
- The parameters and return value of each method defined in the interface must be valid for Java RMI.
- Each method's throws clause must include the java.rmi.RemoteException exception class.

Figure 9 shows the relevant parts of the definition of the remote interface (Transfer) for the example Transfer bean. This interface defines the methods for transferring funds between two Account bean instances and for getting the balance of an Account bean instance.

```java
import javax.ejb.*;
import java.rmi.*;
import com.ibm.ejs.doc.account.*;
public interface Transfer extends EJBObject {
    ...
    float getBalance(long acctId) throws FinderException, RemoteException;
    ...
    void transferFunds(long fromAcctId, long toAcctId, float amount)
        throws InsufficientFundsException, RemoteException;
}
```

Figure 9. The Transfer Remote Interface

By stepping through the different tasks required to develop an EJB bean, you have probably realized that a lot of code can be generated automatically. VisualAge for Java does this for you, and a lot more, which will be described in the rest of this book.
3.3 Developing Session Bean inside VisualAge for Java

Before starting to develop a session bean, we need to describe the VisualAge for Java EJB beans development environment. This consists of multiple tools that can be categorized into the following groups:

• Tools for creating EJB beans
• Tools for generating deployed code for a WebSphere™ EJS
• Tools for testing EJB beans before you install them into EJSs

All of these tools are accessible from the EJBs page of the Workbench. This page is the heart of the EJB Beans development environment. This is where your EJB bean groups and individual EJB beans reside, and it is where you accomplish almost all of your EJB beans development activities. This environment provides all of the necessary run-time support for the IBM WebSphere Application Server. It also leverages the built-in team and versioning capabilities of VisualAge for Java to help you maintain both your EJB source code and generated code.

Creating EJB Beans

The EJB beans development environment provides tools to help you create EJB beans that can be deployed in a WebSphere EJS or in EJB containers provided by other vendors. These tools allow you to accomplish core development activities in the EJBs page, such as creating EJB groups and EJB beans. The tools also allow you to accomplish other development activities, such as writing and editing business logic, importing or exporting EJB beans, and managing EJB beans. A tool is also provided to help you map entity beans to back-end data stores, such as relational databases. There are also tools to help you set deployment descriptor and control descriptor properties prior to running the tools used to generate the deployed code. The VisualAge for Java main window is shown in Figure 10.
In this chapter we create a stateless session bean called ReadCustomerInfo. It has the responsibility to read data from a serialized file and give as output a CustomerObject initialized with the data contained in the .ser file. The "customer.ser" file must exist in your environment, and in order to have flexibility to change its location, we set it using the environment property in the deployment descriptor as explained in “Generate Deployment Code inside VisualAge for Java” on page 65. The "customer.ser" file is already provided in the CdRom\Part1Samples\SessionBean directory of the CDROM provided with this book.

**First step: Adding EJB Groups**

In a typical VisualAge for Java environment, you need to create a project in order to organize your work. Once you have decided what project you need for your EJB beans development work, you can add EJB groups to the EJBs page to hold your EJB beans.
An EJB group is a logical group that allows you to organize your EJB beans. You can perform global operations on an EJB group that iterate on all of the EJB beans that reside in the group. For example, if you select an EJB group to export to an EJB jar file, all of the EJB beans contained in the group are exported.

Generally, when you work with EJB beans, you need to perform the following three main actions:

1. In the Workbench Projects page, create a project.
2. In the EJBs page, create an EJB group. When doing this, you are asked to specify the package name. If it does not exist, it is created for you and put in the project specified in step 1.
3. Create EJB beans inside the group specified in step 2.

There are two ways to add an EJB group to the EJBs page:

- Create a new EJB group.
- Retrieve one or more existing EJB groups from the repository.

To add EJB groups:

1. In the Workbench Projects page, from the menu, select **Selected->Add Project**. The Add Project SmartGuide appears.
2. In the Create a new project named field, type **ITSO Bank**. This is the name assigned to the new project that will contain your EJB beans, then click **Finish**. The new project appears in the projects page. When creating projects, we suggest that you create a project to hold the EJB beans and their related classes, and another project to hold the client application code that will access the EJB beans. This helps to keep your code organized. We have not done this for our small session bean sample because there are very few classes to manage.
3. In the Workbench Projects page, click on the **EJBs** tab. The EJBs page appears, as shown in Figure 11.
4. From the EJBs menu, select **Add-EJB Group**. The Add EJB Group SmartGuide appears.

5. Beside the Project field, click the **Browse** button and look for the **ITSO Bank** the project that we want to contain the EJB group, then click **OK**.

6. You can do one of the following:
   - Add an EJB group by creating a new group.
   - Add one or more EJB groups by retrieving existing EJB groups from the repository.

For our sample purpose we now create a new group. To add an EJB group by creating a new group, ensure that the **Create a new EJB group named** radio button is selected, and type **ITSOEJBGroup**, which is the name we want to assign to the new group. Your window should now appear as shown in Figure 12.
Figure 12. Adding a New EJB Group

7. Click **Finish**. The EJB group is added to the EJBs pane, as shown in Figure 13.
Second step: Adding EJB Beans to EJB Groups

Once you have added your new EJ B group to the EJBs page, you can add EJ B beans to the EJ B groups in one of the following ways:

- Creating new EJ B beans
- Retrieving existing EJ B beans from the repository
- Importing EJ B beans from EJ B jar files

This topic discusses how to add EJ B beans by creating new EJ B beans. Information about retrieving existing EJ B beans from the repository and adding EJ B beans by importing them from a jar file is found in the Appendix A.1, “Importing EJ B Beans from a Jar File” on page 349.
To add EJB beans:

1. In the EJBs pane, select an EJB group to contain your EJB bean and click mouse button 2.
2. From the pop-up menu, select **Add-EJB**. The Create EJB Bean SmartGuide appears, as shown in Figure 14.

Using the Create EJB SmartGuide, you can add an EJB bean by either creating a new EJB bean or by retrieving one or more existing EJB beans from the Repository.
Let us see how to add a new EJB bean by creating a new EJB bean:

1. Ensure that the Create a new EJB radio button is selected.

2. In the Bean name field, type ReadCustomerInfo, which is the name that you want to assign to your new EJB bean.

3. Since you are creating a new EJB bean class, you need to specify the type of EJB bean you want to create: session bean, entity bean with bean-managed persistence (BMP), or entity bean with container-managed persistence (CMP) fields. The ReadCustomerInfo bean is a session bean, so from the Bean type drop-down list, choose Session Bean.

4. In the Project field, type the project name, modify the default name, or click the Browse button and select the project that contains (or will contain) the EJB bean class from the list of projects, then click OK. By default, the project to which the EJB group belongs is displayed. In our case the project is ITSO Bank.

5. In the Package field, type the project name, modify the default name, or click the Browse button and select the package that contains (or will contain) the EJB bean class, then click OK. If you enter a package that does not exist, a new package is created. Type itso.samples.ejb.sessionbean in the package field.

6. In the Class field, either accept the default value or type in a name for the new class. We recommend that you use the default value so that all the implementation types of the EJB bean are named properly.

7. Since we do not provide any superclass for our ReadCustomerInfo bean, we do not fill the Superclass field.

Before clicking the Next button, your Create EJB SmartGuide should look as shown Figure 15.
Figure 15. Create ReadCustomerInfo EJB

8. After clicking on **Next**, you are presented with the Define EJB Class Attributes and Interfaces panel (see Figure 16).
9. Default EJB Home Interface and EJB Remote Interface names are automatically generated. You can change them or keep the default values. The EJB Key Class is not accessible because session beans do not need a key.
10. In the Add import statements to the bean class, click on **Add Package** button.

11. This opens a new window where you can select packages and add them by clicking **Add** button.

12. Repeat the previous step for each package. Once all packages have been added, you can close the window by clicking on **Close**. The list of added packages appears now in the Add import statements to the bean class.

13. You can follow the same process to add Interfaces that your bean implements. You don't need to add javax.ejb.Session. The generated code automatically implements this interface because the tool knows we are creating a session bean.

14. Click on **Finish** at this point.

Now your EJB workspace page should look as shown in Figure 17.

![Figure 17. EJB Workspace after the Creation of the New EJB Bean](image-url)
As you can see in Figure 17, if you select the ITSOEJBGroup, the **Types** pane is populated with three classes:

- ReadCustomerInfo (remote interface of the ReadCustomerInfo EJB)
- ReadCustomerInfoBean (bean class of the ReadCustomerInfo EJB)
- ReadCustomerInfoHome (home interface of the ReadCustomerInfo EJB)

If you go back to “Developing Session Beans” on page 23, you can see what is needed in order to create an EJB bean, and how much easier is to create it using VisualAge for Java. In fact VisualAge for Java automatically generates the three classes required for creating an EJB bean by following the specification rules.

You can have a look of what is generated inside each of this class by simply selecting it in the Types pane and looking at the displayed methods in the Methods pane. As you can see, the remote interface is empty. It is up to you to decide which are the methods you need to put in the Remote interface ReadCustomerInfo.

The Home interface instead shows the create() method that our session bean at least must have. Remember, since the ReadCustomer session bean is a stateless session bean, we do not implement any additional create() method.

You can add new methods to your bean class and make them appear in the Home or Remote interface. In order to do this you never manipulate the source code of these two interfaces directly. A detailed explanation on how to do this is found in “Adding Methods to EJB Beans” on page 49.

The ReadCustomerInfo bean is a simple stateless session bean that accesses a serialized file in order to read simple customer information. For this reason, you need to do some additional steps before you can start developing your session bean.

In C:Rom\Part1Samples\SessionBean directory, we provide the following CustomerObject.jar file. It contains a CustomerObject serialized JavaBeans to be used by the CustomerInfo session bean. Import it into your ITSO Bank project. An itso.samples.ejb.sessionbean.model package should now show up in the ITSO Bank project.

Now we can proceed with building our ReadCustomerInfo session bean. Go back to the EJB workspace. We need to create new fields to the ReadCustomerInfo session bean. Since this is a stateless session bean, the fields are private attributes for the bean.
Adding Fields to EJB Beans

When you create an EJB bean in the EJB bean development environment, certain fields are created automatically in the bean class. In the case of a session bean, the javax.ejb.SessionContext mySessionCtx is created. This topic explains how to add three more fields to the EJB bean after it has been created. We need to define the following fields:

```java
private String fileName = null;
private FileInputStream fInput;
private ObjectInputStream input;
```

You can display EJB fields by selecting the EJB bean in the EJBs pane, and then selecting the Fields icon in the upper right corner of the Types pane to toggle to the Fields pane. The Fields pane and the Types pane share the same space on the EJBs page and can be displayed alternately by selecting the icon in the upper right corner of the pane. To display the Types pane, select the Types icon.

The reason why we declared all the fields as private is because our session bean is stateless. If you want to make it a stateful session bean you have to declare all the fields as public so that the container takes care of their state during passivation.

To add a field to the bean class:

1. In the Types pane of the EJBs page, select the EJB bean class to which you want to add a field.
2. Click mouse button 2, then select Add Field from the pop-up menu.
   The Create Field smartguide appears, as shown in Figure 18.
3. In Field Name, enter `fileName`.

4. Select **String** in the Field Type combo box.

5. Type `null` in the Initial Value field.

6. Select **private** as Access Modifier. Our bean is Stateless; this is why we declare private fields. If the bean had been declared Stateful, then we should have declared the fields as public.

7. Check the **Access with getter and setter methods** checkbox so getters and setters methods are generated by the tool for you.

The SmartGuide should now look as shown in Figure 19.
8. Click **Finish**.

You need to repeat the above steps to create the fields:

- Input of type FileInputStream
- Input of type ObjectInputStream

Since we all make mistakes, we may have created a field that we really do not need. The question is: How do we delete fields?
Deleting Fields from EJB Beans

There is no automatic way to delete a field from an EJB bean; all the operations should be done manually. What you need to do is the following:

1. In the Types pane of the EJBs page, select the EJB bean class from which you want to delete a field.
2. From the Methods pane select the corresponding setter and getter method (use Ctrl key for multiple selection). Click mouse button 2, then select Delete.
3. In the Types pane of the EJBs page, select the EJB bean again and delete the field instance manually from the source code.

In this version of VisualAge for Java, selecting a field and delete is not yet provided.

After the creation of all necessary fields, we carry on by adding new methods to the session bean. We need to add three new methods. The openFile() and closeFile() methods are needed only for accessing the serialized file customer.ser. The readCustomer() instead is used to read the information from the serialized file and return them into a CustomerObject.

Modifying methods or adding methods in the EJB Development Environment is similar to modifying or adding methods elsewhere in VisualAge for Java.

Adding Methods to EJB Beans

To add a new method do the following:

1. In the Types pane of the EJBs page, select the EJB bean class to which you want to add a method. Select ReadCustomerInfoBean.
2. Click mouse button 2, then select Add Method from the pop-up menu. The Create Method SmartGuide appears, as shown in Figure 20.
3. Select the **Create a new method** radio button.
4. Type `CustomerObject readCustomer()` as shown in Figure 21.
5. Click **Next** to go to the next page.

6. Select **public** as the Access Modifiers and, since the `readCustomer` method can throw an `IOException`, you need to add this exception in the **What exception may this method throw?** list. In order to do this, click the **Add** button and type the exception name. When you are finished click **Close** (see Figure 22).

   You need to go on the second page only if you want to change the modifiers attribute, or if you want the SmartGuide to handle additional exceptions. If these conditions do not apply, you can skip step 5 and step 6.
Figure 22. Create Method SmartGuide—Second Page

7. Click **Finish** to create the new method.

You now need to create the other two methods by following the same steps. The two methods have the following signature:

```java
public void openFile();
public void closeFile() throws java.io.IOException;
```

Now we need to fill the methods with the right code and understand what we must do with these new business methods.

**Here is the code to be inserted in each method:**
private void closeFile() throws java.io.IOException{
    if(input!=null){
        input.close();
        input=null;
    }
}

private void openFile() {
    try {
        //Need to initialize the fileName field from what is defined in
        //the environment
        fileName=(String)props.get("customerFile");
        fInput= new FileInputStream(fileName);
        input = new ObjectInputStream(fInput);
    }
    catch(Exception e){
        System.out.println("Error in opening: " + e);
    }
}

public CustomerObject readCustomer() throws java.io.IOException{
    try{
        openFile();
        if (fInput.available() > 0){
            System.out.println("INIT --> Reading object");
            CustomerObject cO=(CustomerObject)input.readObject();
            System.out.println("END --> Reading object");
            closeFile();
            return cO;
        }
    }
    catch(Exception e)
    {
        e.printStackTrace(System.out);
    }
    return null;
}

After you create methods, you need to promote to the remote interface those
methods that will be accessed by a client. In our case, only the
readCustomerInfo method needs to appear on the remote interface.
Adding Methods to the Remote Interface

To promote a method to the remote interface, do the following:

1. Select the method and then select **Methods-Add-to-EJB Remote Interface**. So to promote the readCustomer() method to the remote interface select it and from the Methods menu select **Add to - EJB Remote Interface**. Another way to do the same thing is select the method, click the mouse button 2 and from the popup menu select **Add to - EJB Remote Interface**. Once the method has been promoted you can notice a small icon appearing near the method. This icon helps you to identify all the methods that have been declared in the remote interface.

Since the ReadCustomerInfo bean does not have any further create or finder methods, we do not need to add any additional methods to the home interface. We will see this option later in the book.

Removing Methods from the Remote Interface

After you have added a method to the remote interface, you can also remove it. To do this in the EJB bean class select the method and then **Methods - Remove From - EJB Remote Interface**. In this case the method is only removed from the remote interface but still exists in the EJB bean class. If you really want to get rid of the method, delete it from the EJB bean class instead, then the corresponding method in the remote interface is automatically removed.
Your workspace should now look as shown in Figure 23:

![Figure 23. ReadCustomerInfoBean in the EJB Workspace](image)

You have now completed the development steps for the ReadCustomerInfoBean. Let us see how to deploy it and test it inside VisualAge for Java.

**Generating Deployed Code**

Once an enterprise bean is developed, the next step is to deploy that bean into a container. However, before you can deploy an enterprise bean, you must first create the deployment descriptor, which defines the transaction, security, and other environment specifications that determine how the enterprise bean is managed by the container.
The deployment descriptor

The deployment descriptor contains attribute and environment settings that define how the container invokes enterprise bean functionality.

The content of the deployment descriptor is decided during different phases:

- Application analysis
- Bean development
- Bean deployment

For example, during analysis you may decide that a bean is going to use a file name that must not be hardcoded. Then, during the development phase, the developer decides that the file name is going to be passed to the bean by using a property name. At last, during deployment, the deployer can change the value of the property name with a new value pointing to a different fully qualified path and file name (see Figure 61 on page 104).

Every enterprise bean (both session and entity) must have a deployment descriptor that contains settings for the following attributes; these attributes can be set for the entire enterprise bean or for the individual methods in the bean. The container uses the definition of the bean-level attribute unless a method-level attribute is defined, in which case the latter is used.

- Transaction attribute—Defines the transactional manner in which the container invokes a method. The values for this attribute are described in Chapter 7, “Transactions” on page 217.
- Transaction isolation level attribute—Defines the degree to which transactions are isolated from each other by the container.
- Access control attribute—Defines an access control entry that identifies users or roles that are permitted to access the methods in the enterprise bean. The values for this attribute are defined by enterprise bean deployers and the system administrator.
- RunAsMode and RunAsIdentity attributes—The RunAsMode attribute defines the identity used to invoke the method. If a specific identity is required, the RunAsIdentity attribute is used to specify that identity.

The deployment descriptor for a session bean must also contain settings for the following attributes. These attributes can be set on the bean only; they cannot be set on a per-method level.

- The State management attribute defines the conversational state of the sessionBean. This attribute must be set to either STATEFUL or STATELESS.
The Timeout attribute defines the timeout value in seconds associated with this session bean. This is a value needed for the container. Idle instances of a session bean are removed by the container after this specified time period has elapsed.

Deployment descriptors can be created by using the tools within an integrated development environment (IDE) like IBM’s VisualAge for Java, or by using the stand-alone tools contained in WebSphere.

You provide this information to VisualAge for Java by using property dialogs. Later, when you wish to export your EJB bean in an EJB jar file in order to deploy it in other environments, the serialized deployment descriptor is generated for you, as well as all the required files that go into the deployable jar file.

In case you do not specify any information that normally goes into the deployment descriptor, VisualAge for Java creates a deployment descriptor with default values.

To launch this SmartGuide, from the EJB Workspace:
1. Go to the EJBs pane.
2. Select ITSOEJBGroup.
3. Select ReadCustomerInfo bean class.
4. Click mouse button 2, then select Properties.

The Properties window appears, as shown in Figure 24.
In the first page that appears, you will recognize all of the values described in the section “The deployment descriptor” on page 56. We keep the default values on this first page.

If you need to change the JNDI name or control attributes at the bean level, do one or more of the following:

- In the Enter the JNDI name for BeanHome field, type the JNDI name to associate with the EJB bean in the JNDI name space. The container will bind the EJB bean's home interface with a JNDI name that includes the name you specified. A client who wishes to access your EJB bean uses the JNDI in order to find it in a naming context.

- In the Transaction Attribute drop-down box, select a transaction attribute. This attribute tells the container how to manage transaction scopes before and after the execution of the EJB method.

- In the Isolation Level drop-down box, select an isolation level. This level tells the container what isolation level to set on the database connections used by the EJB bean at the start of each transaction.
In the **Run-As Mode** drop-down box, select a run-as mode. This selection tells the container the security identity to associate with the execution of the EJB method.

- For session EJB beans only, in the **State Management Attribute** drop-down box, specify whether or not you want the container to maintain state information for the EJB bean.

- For session EJB beans only, in the **Session Timeout Value** (seconds) field, specify after how many seconds a session should timeout.

During runtime, our bean needs to know where to get a file called `customer.ser`, which contains a serialized instance of a Customer object. We pass this information through the environment properties.

Click the **Environment** tab. The Environment tab page appears, as shown in Figure 25.

![Environment Properties Page](image)

Figure 25. Environment Properties Page

You can do one or more of the following:

- To set an environment property, specify the variable name in the **Variables** field and its associated value in the **Values** field, and click **Set**.
To select an environment property defined for the EJB bean, click on the properties in the **Environment Properties** list, and click OK.

To delete an environment property, select the property you want to delete, and click **Delete**.

In order to add the property we need, to do the following:

1. In the Variables field, type `customerFile`.
2. In the Values field type `c:\customer.ser`. This indicates where the "customer.ser" is located. We assume the file is located under the root directory. If you copy it somewhere else, ensure that the property is updated.

The properties window should now appear as shown in Figure 26.

![Properties window](image)

**Figure 26. Environment Properties Page after Definition of a Property**
If you want to use Environment properties, you need to define a Properties field in the bean class (see Figure 27).

```java
public class ReadCustomerInfoBean implements javax.ejb.SessionBean {
    private javax.ejb.SessionContext mySessionCtx = null;
    // Needed to hold the environment properties
    private Properties props;
    private String fileName = null;
    private FileInputStream fInput;
    private ObjectInputStream input;
}
```

Figure 27. Definition of the Properties Field

This Properties field is initialized in the setSessionContext() method as shown in Figure 28.

```java
public void setSessionContext(javax.ejb.SessionContext ctx) {
    mySessionCtx = ctx;
    // Need to get the environment information
    props = ctx.getEnvironment();
}
```

Figure 28. Getting Properties from the Context

Now you need to initialize the fileName variable reading the information from the properties. We do it in openFile() (see Figure 29).

```java
public void openFile() {
    try {
        fileName = (String)props.get("customerFile");
        fInput = new FileInputStream(fileName);
        input = new ObjectInputStream(fInput);
    } catch (Exception e) {
        System.out.println("Error in opening: " + e);
    }
}
```

Figure 29. Retrieve Environment Variable Value from Properties

We do not need to change the control attributes at the method level, but if you need to do this, click the **Method** tab. The **Method** page appears, as shown in Figure 30.
Figure 30. Method properties Window

To add a method-level control descriptor, click Add. The **Add Control Descriptor at Method Level** dialog appears, as shown in Figure 31.
Figure 31. Method Selection

If you select a method from the list and click **OK**, the following dialog appears, as shown in Figure 32.

The displayed attribute values are those defined at the bean level. Modifications done in this window affect the selected method only.
The ejb-jar File

The ejb-jar file is the file format used to package enterprise beans; this file uses the standard Java programming language Archive File. The ejb-jar file can be used to contain individual enterprise beans, multiple enterprise beans, and entire enterprise bean-based applications.

For an individual enterprise bean, an ejb-jar file must contain the following files:

- The .class files that make up the bean
- The deployment descriptor
- The manifest file that identifies the contents of the .jar file.

Manifest files are organized into sections that are separated by blank lines; each section corresponds to a file stored in the .jar file. Each section contains one or more tag-value pairs with the syntax tag:value. The section corresponding to the deployment descriptor file for each enterprise bean in a .jar file must contain the following headers:

  Name: deploymentDescriptorFile
  Enterprise-Bean: True

The manifest file must be named META-INF/MANIFEST.MF.

An ejb-jar file can be created by using the tools within an integrated development environment (IDE) like IBM's VisualAge for Java, or by using the stand-alone tools contained in WebSphere.

Deploying an Enterprise Bean

When you deploy an enterprise bean, the deployment tool creates the following elements:

- The container-implemented EBJ BHome object and EBJ BObject classes from the enterprise bean's home and remote interfaces (and the persister and finder classes for entity enterprise beans with CMP).
- The Java ORB, stub, and skeleton files required for remote method invocation (RMI).
- Helper and holder classes for the home and remote interfaces.

An enterprise bean can be deployed by using the tools within an integrated development environment (IDE) like IBM's VisualAge for Java, or by using the stand-alone tools contained in WebSphere.

NOTE: If you add a remote method and promote it to the remote interface after generating the deployed code, you must regenerate the deployed code.
**Generate Deployment Code inside VisualAge for Java**

We are now ready to generate the deployment code for the Bean:

1. In the EJBs pane, select the ReadCustomerInfo bean class.
2. Click mouse button 2, then select **Generate->Deployed Code** item from the EJB bean menu.

The Types pane is now populated with the generated classes, as shown in Figure 33.

![Figure 33. EJB WorkSpace after the EJB Deployment Code Generation](image-url)
Generate the Test Client
VisualAge for Java contains a complete EJB server runtime environment. When you request VisualAge for Java to generate deployed code, it does it for the WebSphere Advanced Edition 2.0 containers. Then you can immediately run your EJB beans, debug them, and test them inside your IDE. To make the developer’s life even easier, it also generates a test client.

A test client can be generated for each EJB bean. The purpose of the test client is to test each of the methods in the home and remote interfaces of your bean. The client features a graphical user interface that allows you to select methods from the home and remote interfaces from a list, insert parameters, and run the methods.

To generate the EJB test client, do the following:
1. Go to the EJBs pane.
2. Select ITSOEJBGroup.
3. Select ReadCustomerInfo bean class.
4. Click mouse button 2, then select Generate-Test Client item from the EJB menu. A default EJB test client class appears in the Types pane of the EJBs page under the name ReadCustomerInfoTestClient, as shown in Figure 34.
Testing EJB Beans

To run and test the ReadCustomerInfo session bean, you need to create an EJB server configuration in which you can test your EJB bean.

Configuring a server configuration for your EJB bean consists of adding the EJB group that contains your EJB beans to a server configuration.

To create an EJB server configuration:
1. From the EJBs page, select ITSOEJBGroup.
2. Click mouse button 2, and then select **Add to - Server Configuration** from the pop-up menu.

The EJB Server Configuration browser appears, as shown in Figure 35.

![EJB Server Configuration](image)

**Figure 35. EJB Server Configuration**

An EJB server entry is added in the Servers pane. The EJB server contains the ITSOEJBGroup.

Once you create your EJB server configuration, you may need to set some properties. Even though, in our case, we use default values, let us have a look at the different settings that can be used.

**Setting Name Service and EJB Server Properties**

The name servers that are used by the EJB Server Configuration browser are configurable. The name servers work with the default values that are configured for them. This topic describes how to change them if so desired. Note that once set, any name server or EJB server properties are saved even if you exit VisualAge for Java.
Setting Properties for the Location Service Daemon

To set the properties for the Location Service Daemon:

1. In the EJB Server Configuration browser, in the Servers pane, select the Location Service Daemon.
2. Click mouse button 2 and select Properties from the pop-up menu. The Location Service Daemon properties dialog appears, as shown in Figure 36.

3. Type in the port number for the ORB Listener Port. Note that the value for the ORB Listener Port must match the value for the LSD Port in the Persistent Name Server.

Setting Properties for the Persistent Name Server

To set the properties for the Persistent Name Server:

1. In the EJB Server Configuration browser, in the Servers pane, select the Persistent Name Server.
2. Click mouse button 2 and select **Properties** from the pop-up menu. The Persistent Name Server Properties dialog appears, as shown in Figure 37.

![Persistent Name Service Property Dialog](image)

**Figure 37. Persistent Name Service Property Dialog**

3. Modify the default values if necessary.

The **Initial Root** value is a working directory where certain information is written about the persistent data.

In the **LSD Name** field, the Location Service Daemon can run on another machine other than the supplied default of localhost. Note that the **LSD Port** number must match the **ORB Listener Port** of the Location Service Daemon.

### Setting Properties for the EJB Server

If you just want to test and debug an EJB bean and associated client application that you are running within VisualAge for Java, you should not need to change the default properties that are automatically established for the EJB server. However, only if you are testing an entity EJB bean, you must ensure that the database URL references a database that is configured on your system.

To set the properties for the EJB server:

1. In the EJB Server Configuration browser, in the Servers pane, select the EJB server.

2. Click mouse button 2 and select **Properties** from the pop-up menu. The Properties for EJB Server dialog appears, as shown in Figure 38.
3. If you are testing an entity EJB bean, in the **Database URL** field, ensure that the URL of a database is specified that is configured on your system, for example, jdbc:db2:MyDatabase.

4. If your database is set up so that a user ID and password are required to access the database, then specify a user ID and password in the **Database User ID** and **Database Password** fields.

5. If you changed the default port values, ensure that the **LSD Port** number matches the **ORB Listener Port** of the Location Service Daemon.

Since we are dealing with session beans, we do not need to be concerned about persistence aspects.

**Starting the Name Service and EJB Servers**

Once you have created an EJB server configuration, you need to start the Location Server Daemon, the Persistent Name Server, and your EJB server before you can test your EJB bean.

To start the servers:

1. In the EJBs page, select **EJBs - Open To - Server Configuration**. The EJB Server Configuration browser appears.

2. In the Servers pane, select the **Location Service Daemon** and click mouse button 2, then select **Start Server**.
3. In the Servers pane, select the **Persistent Name Server** and click mouse button 2, then select **Start Server**. You can also select a server, and from the tool bar, click on 🔄. 
In the same way, you can stop a server by clicking on the icon 🔄. The Console window appears in the background and monitors the two servers.

4. In the Console, check that the Location Service Daemon and the Persistent Name Server are listening by selecting each server (one at a time) in the All Programs pane and looking for the following messages in the Standard Out pane:

   Location service daemon listening...
   NameServer is listening...

To start the EJB server:

1. Once the Location Service Daemon and the Persistent Name Server are running, in the Servers pane of the EJB Server Configuration browser, select the **EJB server (server1)** and click the 🔄 icon to start the server. The Console window now also monitors the EJB servers, as shown in Figure 39.

![Figure 39. Console after All Servers Are Started](image)

2. In the Console, check that the EJB server is ready to process a call from the client by selecting the EJB server entry at the bottom of the All
Programs pane and then looking for the following message in the Standard Out pane:

Server is listening...

3. You can change the source code of any EJB bean in the EJB server configuration and debug it without stopping the EJB server. However, if you change the home or remote interfaces, you must do the following:
   1. Stop the EJB server. (Information about stopping the EJB Server is found in the topic “Stopping the Name Service and EJB Servers” on page 85.)
   2. Regenerate the deployed code.
   3. Start the EJB server again.

Note: If you create an EJB server configuration that contains a number of EJB beans, and then you decide that you want to delete one of the EJB beans, you should stop the EJB server before deleting the EJB bean.

Restarting Servers

In some situations, you may want to perform a restart on a server that is already running. Essentially, this resets the running server to the state it was in when it was first started.

To restart a server:

1. In EJBs page, select EJBs - Open To - Server Configuration. The EJB Server Configuration browser appears.
2. In the Servers pane, select the server that you want to restart.
3. Click mouse button 2, and select Restart Server from the pop-up menu.
   In the Servers pane, an icon 🔄 appears beside the server to indicate that it is running.

Running the Generated Test Client

Now we are ready to run the test client to test the EJB bean's home and remote interface methods, and to debug the ReadCustomerInfo EJB implementation.

When running the test client, you can set breakpoints in your EJB implementation and use the VisualAge for Java debugger as you would normally do.

All of the Java projects found in the workspace are automatically added to the test client classpath if you start the test client from either the EJB Server Configuration browser or the EJBs page. The test client will also
automatically locate the EJB bean using the JNDI name specified as part of the EJB bean’s deployment descriptors.

If you start the test client outside of the EJB Server Configuration browser or the EJB page, you need to set the classpath of the test client program.

We recommend that you start the test client in either the EJB Server Configuration browser or in the EJB page.

We now describe the basic steps required to run the test client.

To run the default test client:

1. In the EJB Server Configuration browser, ensure that the Name Service servers and the EJB Server are running. (Information about starting these servers is found in the topic, “Starting the Name Service and EJB Servers” on page 71.

2. In the EJB pane of the EJB Server Configuration browser, select the ReadCustomerInfo EJB bean for which you have created a test client. (If the EJB bean does not appear in the EJB pane of the EJB Server Configuration browser, you must add the EJB bean to a server configuration by following the instructions in the topic, “Testing EJB Beans” on page 67.
3. Click on the Run Test Client icon to run the test client. The Connect page of the Test Client window appears, as shown in Figure 40:

![Figure 40. Connect Window for the ReadCustomerInfo Session Bean](image)

4. The **Provider URL** field contains the hostname of the server you are connecting to. In this case `iiop://` stands for `iiop://localhost`.

   The **JNDI Name** field contains the JNDI name for your EJB. To find out which is the JNDI name of your EJB bean, please refer to “Generate Deployment Code inside VisualAge for Java” on page 65 on how to get to the EJB Properties window (see Figure 24 on page 58).

   The **Initial Context Factory** field contains the class name to be used as the context factory of the naming service.

   The **Home Interface** contains the fully qualified name of the EJB Home interface.

   The **Home Helper** field contains the fully qualified name of the Helper class of the EJB bean being tested.

   To establish a connection to the name server, click the **Connect** button. The client constructs an initial naming context, and ask the name server to look up an instance of the home interface for the ReadCustomerInfo EJB bean you are testing. Once the connection is made and an instance of the home interface is retrieved, the test client automatically switches to the Home Interface page, as shown in Figure 41.
Figure 41. Home Test Client Page

Figure 41 shows the test client for the ReadCustomerInfo bean. ReadCustomerInfo is a stateless session bean, so its home interface (ReadCustomerInfoHome) contains only one create method. The other methods shown are defined by the javax.ejb.EJBHome interface, which ReadCustomerInfoHome extends, and they are marked with a trailing :: EJBHome.

5. In the Methods pane, select the create() method. Click the Send button in order to create the remote interface object. This sends the method call to the EJB server to create the new ReadCustomerInfo bean. The resulting remote interface object will be used by the test client for testing remote methods. When the remote interface object is retrieved, the test client will switch to the Remote Interface page, as shown in Figure 42.
In the Remote Interface page, the Methods pane contains the list of methods defined by the remote interface, plus methods defined by the interface `javax.ejb.EJBObject`.


7. Click the **Send** button to send the method to the EJB bean. The result of the method call is displayed below the Methods pane. The result should look as shown in Figure 43.
8. If you want to examine the result object, click the **Inspect** button. This launches the VisualAge for Java Inspector, as shown in Figure 44, as shown in Figure 44.
After testing the ReadCustomerInfo EJB bean, we can now build our own client application.

### 3.4 Writing Your Own Client Application

We have provided the client application for you. You can find it in the SB client.jar file in the C:\\CdRom\Part1\Samples\SessionBean directory. Import the SB client.jar file into your ITSO BANK project.

An enterprise bean can be accessed by different types of clients:

- A Java servlet
- A Java Server Page (JSP)
- A Java application
- Another enterprise bean.

While this chapter discusses the first two types of Java clients only, all of the programming tasks described in this chapter also apply to enterprise beans acting as clients to other enterprise beans. This chapter assumes that you understand how to write a Java servlet and a Java application. To access and
use an enterprise bean in a Java client, the client must contain code that does the following:

- Imports the Java packages required for naming, remote method invocation, and EJB interaction.
- Instantiates the bean's EJB object. For more information, see "Instantiating an EJB Object" on page 80.
- Refreshes the reference to each EJB object associated with a session bean if the reference becomes invalid.

**Importing Required Java Packages**

While the Java packages required for any particular enterprise bean client vary, the following packages are required by all clients:

- `java.rmi.*` — This package contains the classes and methods required for remote method invocation (RMI).
- `java.util.*` — This package contains various Java utility classes, such as Properties, Hashtable, and Enumeration used in a variety of ways throughout all enterprise beans.
- `javax.ejb.*`
- `javax.naming.*`
- The package for each enterprise bean with which the client interacts.

**Instantiating an EJB Object**

To invoke a bean's business methods, a client must create or find an EJB object for that bean. Once the client has created this object, it can invoke methods on it in the standard way. To create or find an instance or a bean's EJBObject, the client must instantiate the EJBHome object for that bean. The EJBHome object can then be used to create or find (for entity beans only) an instance of the bean's EJBObject.

The ReadCustomerInfo client application is a simple application that shows the data read from the .ser file in a window, like Figure 45.
Developing a Session Bean

Figure 45. Read Customer Client Window

The ReadCustomerInfo client application is made of two EJB beans, the ReadCustomerClient and the ViewController. It follows the MVC paradigm. The ReadCustomerClient is the visual part and the ViewController is the controller on the visual part. This means that all the access to the EJB bean and the action performed on the button really goes to the controller.

The model is represented by the ReadCustomerInfo EJB bean. The ViewController has a `create(String,String,String)` method that is called inside the `getCustomer()` method of the ViewController class. The parameters passed here are, respectively, the hostname, the naming service class to be used, and the port number.

When the `windowOpened` event is detected for the ReadCustomerClient, an instance of ViewController is created with the right parameters.

The code in this section is taken from the example Java application ViewController.java; all of this code is contained in the application's `create(String,String,String)` method.

The Enterprise JavaBeans specification contains instructions for using classes in the `javax.rmi` package to locate a bean's home interface. Unfortunately, this package has not been implemented. Therefore, IBM has implemented an alternative way to locate a home interface by using the IBM CosNaming naming and directory service.

Locating an enterprise bean's home interface is a two-step process. First, you create a `javax.naming.InitialContext` object. Then, you use the `InitialContext` object to get the EJ BHome object.
The code below shows what is required to create the InitialContext object. To create this object, you construct a java.util.Properties object, assign environment values to the Properties object, and then pass the object as the argument to the InitialContext constructor.

```java
public void create(String host, String port) {
    ReadCustomerInfoHome rCHome = null;
    javax.naming.InitialContext initContext = null;

    // Get the initial context
    try {
        System.out.println("Retrieving initial context...");
        java.util.Properties properties = new java.util.Properties();
        // local name server
        System.out.println("-------Set properties to: javax.naming.Context.PROVIDER_URL, iiop://"+host+":"+port);
        properties.put(javax.naming.Context.PROVIDER_URL, "iiop://"+host+":"+port);
        // IBM name services
        System.out.println("-------Set properties to: com.ibm.jndi.CosNaming.CNInitialContextFactory");
        System.out.println("-------Invoke: javax.naming.InitialContext(properties)");
        initContext = new javax.naming.InitialContext(properties);
    } catch (javax.naming.NamingException e) {
        System.out.println("Error retrieving the initial context: " + e.getMessage());
        System.exit(0);
    } catch (Exception e1) {
        System.out.println("Error retrieving the initial context: " + e1.getMessage());
        e1.printStackTrace(System.out);
        System.exit(0);
    }

    // lookup the home interface using the JNDI name
    .......
    // create a new ReadCustomerInfo to set
    .......
    set...
}
```

Once the `InitialContext` object is created, the application uses it to create the EJBHome object. This creation is accomplished by invoking the lookup method which takes the JNDI name of the enterprise bean in String form.
and returns a java.lang.Object object. The returned object is filtered, using the static method narrow, to obtain an EJ BHome object for the specified enterprise bean.

The narrow method is contained in the home helper class generated during bean deployment. For the ReadCustomerInfo bean, this home helper class is named ReadCustomerInfoHomeHelper.

```java
public void create(String host, String port) {
    ReadCustomerInfoHome rCHome = null;
    javax.naming.InitialContext initContext = null;
    // Get the Initial context
    try {
        System.out.println("Retrieving the home interface...");
        java.lang.Object o = initContext.lookup("ReadCustomerInfo"); // this is the JNDI name
        if (o instanceof org.omg.CORBA.Object)
            rCHome = ReadCustomerInfoHomeHelper.narrow((org.omg.CORBA.Object) o);
    } catch (javax.naming.NamingException e) {
        System.out.println("Error retrieving the home interface: " + e.getMessage());
        System.exit(0);
    } // endtry
    // Create a new ReadCustomerInfo to set
    ...
    set..
}
```

The code below shows how to create the EJ B object by using the EJ BHome object. The EJ B object is obtained by calling a create method.

```java
public void create(String host, String port) {
    ReadCustomerInfoHome rCHome = null;
    javax.naming.InitialContext initContext = null;
    // Get the Initial context
    try {
        System.out.println("Creating new ReadCustomerInfo...");
        ReadCustomerInfo rC = null;
    }
}
```
try {
    rC = rCHome.create();
    System.out.println("ReadCustomerInfo created!");
} 
catch (Exception e) {
    System.out.println("Exception creating new ReadCustomerInfo: " +
            e.getMessage());
    e.printStackTrace(System.out);
    System.exit(0);
} // end try
if(rC== null)
    System.out.println("Unable to creating ReadCustomerInfo");
//this method sets the ReadCustomerInfo instance in the ViewController class
setReadCustomers(rC)

You can open the ReadCustomerClient to the visual composition editor in order to understand the application flow.

To start and test the ReadCustomerClient client inside VisualAge:
1. In the EJB Server Configuration browser, ensure that the Name Service servers and the EJB server are running.
2. In the Workbench Projects page or other page where your client resides, select your client class and click mouse button 2, then select Properties. The Properties dialog appears.
3. In the command line argument type hostname 900 as parameters (in the VisualAge for Java IDE, the Persistent Name Server runs on port number 900).
4. Click the Class Path tab.
5. Beside the Project Path field, click the Edit button. The Class Path dialog appears.
6. Select the IBM WebSphere Test Environment project and click OK.
7. At the bottom of the Properties dialog, select the Save in repository (as default) check box and click OK.
8. Select your client class, then click mouse button 2 and select Run - Run main.
9. When the ReadCustomerClient window appears, click the Get Customer Data button. The result should be as shown in Figure 45

Once you have finished testing your EJB bean, you need to stop the server.
Stopping the Name Service and EJB Servers

To stop a server:
1. In the EJB Server Configuration browser, in the Servers pane, select the server and
2. Click on the icon to stop the server.

The Console window indicates that the server has been terminated.

Removing EJB Servers

You cannot remove the Location Service Daemon or the Persistent Name Server. You may, however, remove EJB servers from the EJB Server Configuration browser.

To remove an EJB server:
1. In the EJB Server Configuration browser, in the Servers pane, select the EJB server that you want to remove.
2. Click on the icon to stop the server (if it is not already stopped).
3. Select the EJB server again, click mouse button 2, and select Remove Server from the menu. The EJB server is removed from the list of servers in the Servers pane.

3.5 Running the Client outside VisualAge for Java

Exporting EJB Groups and EJB Beans

Before starting to go into the details of how to deploy a client application outside VisualAge for Java, we need to give you some general information on how to export EJB beans from VisualAge for Java.

Exporting EJB Beans to EJS or EJB Jar Files

Once you have tested your EJB beans, you can export them to an EJS jar file, EJB jar file, or client jar file. (Information about exporting to a client jar file is found in the topic “Exporting EJB Beans to Client Jar Files” on page 88.)

An EJS jar file contains both EJB code and deployed classes. An EJB jar file contains EJB code, but no deployed classes.

NOTE: All the EJS runtime files are excluded automatically from the jar file.
To export an EJB bean to an EJS or EJB jar file:
1. In the EJBs pane of the EJBs page, select one or more EJB groups or EJB beans, as shown in Figure 46.

2. Click mouse button 2, then do one of the following:
   - To export EJB code and deployed classes, select **Export - EJS JAR** from the pop-up menu. After the jar file is created, you can install it on a WebSphere EJS.
   - To export EJB code, select **Export - EJB JAR** from the pop-up menu. After the jar file is created, you can deploy it into an EJB container using the deployment tool provided by the EJB Container provider.

If you choose, the Export to an EJB Jar File SmartGuide appears, as shown in Figure 47.
3. In the **jar file** field, type the name of the jar file you want to create. Alternatively, click the **Browse** button to select an existing jar file to overwrite. The SmartGuide marks all of the class files associated with the selected EJB groups or individual EJB beans for exportation to the jar file.

4. To select individual file types to export, click one or more of the following check boxes:
   - **beans**: export all the beans in the EJB group or the individual EJB bean you selected.
• **class files**: export all the bytecode files for the selected EJB group or EJB bean.

• **java files**: export all the source code files for the selected EJB group or EJB bean.

• **resource**: export all the non-.java and non-.class files for the selected EJB group or EJB bean.

• **Details**: by default, all the files for the selected file type are exported. Click the **Details** button next to a file type to see a list of the files that will be exported, and to specify individual files to export.

5. To ensure that any types or resources referenced by the selected EJB group or EJB bean are also exported, click the **Select referenced types and resources** button.

6. Specify additional options as appropriate using the **Options** check boxes as follows:

   • **Include debug attributes in .class files**: If you want to debug the exported classes using an external debugger, click this check box to include debugging information for the exported classes.

   • **Compress the contents of the jar file**: Compress the exported files.

   • **Overwrite existing files without warning**: If you are exporting to an existing jar file, and you do not want to be warned about overwriting existing files, click this check box.

   • **Automatically open a web browser on created .html files?**: Do not click this check box, since we do not have any HTML files.

7. Click **Finish**.

You will follow essentially the same steps if you decide to export to an EJS .jar file. The only difference will be the content of the file exported.

**Exporting EJB Beans to Client Jar Files**

A client jar file contains all the classes and interfaces used to access EJB beans.

**NOTE**: If you work as recommended with two different projects, one for the client and one for the server, then instead of adding the server project to the classpath of the client project, we recommend that you include the client jar file in the client classpath before executing the client application. This provides better code isolation between the client and server code.

To export an EJB bean to a client jar file:
1. In the EJBs pane of the EJBs page, select one or more EJB groups. In our case, you just select the ReadCustomerInfo session bean.

2. Click mouse button 2, then select **Export - Client J AR** from the pop-up menu.

The Export to an EJB Jar File SmartGuide appears, as shown in Figure 48.

![Figure 48. Export to an EJB Client jar File](image)

As you can see from Figure 48, the content of this dialog is exactly the same as the previous dialog in Figure 47. The meaning of the fields is the same; the only difference is the content of the .jar file.
Testing the Client outside VisualAge for Java

If you want to test the client application outside VisualAge for Java and access the enterprise beans running inside VisualAge for Java, you need to do the following:

- Export an EJB bean to a client.jar file
- Export client code

**Export an EJB bean to a client.jar file**

As described in the topic, "Exporting EJB Beans to Client Jar Files" on page 88, export the client .jar file to a file called CreateCustomerInfo.jar. Export it to the C:\ root directory. If you decide to export it to a different directory, you will need to change the class path accordingly.

When exporting the client code, you need to add the content of the following package itso.samples.ejb.sessionbean.client.

Export into a .jar file called SBClient.jar. We pack all together because we have few classes, but the right thing to do is to have two different .jar files one for the client code, and one for the code that in our case is the CustomerObject EJB bean only.

In the next session, assume you have exported the .jar file in the C root directory:

You will now have to check that you have the following files in the c:\ directory:

- CreateCustomerInfo.jar
- CustomerObject.jar
- SBClient.jar
- customer.ser

Open a DOS prompt, check your class path, and check that you have the following in your classpath:

`.;
c;c:\CreateCustomerInfo.jar;
c:\CustomerObject.jar;
c:\SBClient.jar;
c:\WebSphere\AppServer\lib\ejs.jar;
c:\WebSphere\AppServer\lib\jndi.jar;
c:\IBM\Java\eab\runtime20;
c:\IBM\Java\eab\runtime;`
NOTE:

- You should not have to be concerned about the VisualAge for Java and DB2 paths; they should have been set during the installation procedure of both products; but you will definitely need to manually add the one regarding WebSphere.

- If you have exported the jar files to different directories, make sure that those changes are reflected in the classpath. But remember that the customer.ser file has to be in the C:\ root directory.

Make sure the EJB environment is up and running inside VisualAge for Java, which means that all EJB servers are started.

Run the Java client from the DOS Prompt using the following command:

```java
java itso.samples.ejb.sessionbean.client.ReadCustomersClient iron
com.ibm.jndi.CosNaming.CNInitialContextFactory 900
```

(Be sure you use the right hostname instead of iron.)

### 3.6 Deployment on WebSphere Advanced Edition 2.0

Next we will explain how to deploy the ReadCustomerInfo inside WebSphere. Before proceeding, you must have WebSphere Advanced Edition 2.0 up and running.

We have experienced problems with empty containers inside WebSphere. If you are not going to use a container, you should remove it.

Since we do not intend to use the defaultEntityContainer, therefore we will remove it. To remove a container, start the IBM WebSphere ApplicationServer 2.0 Administration utility. Refer to Figure 49.
Login to the administration server. In the left pane choose **Enterprise Java Services - Containers**. On the right side of the window choose **defaultEntityContainer** and click **Remove**. You should now have only the **defaultSessionContainer** shown in the list. Refer to Figure 50.
In Figure 50 you can see the container settings:

- **Container Name** field: This shows the container name you assigned to the container.
- **Container Class** files: This contains the class name of the container services used by WebSphere. In this case, we have used the com.ibm.ejs.container.EJSSessionContainer for the session beans. The com.ibm.ejs.container.EJSEntityContainer has been used for the entity beans.
- **JDBC URL** field: You need to specify this information only if you are defining a container that is supposed to contain entity beans; in this case you have to specify something like the following:
  
  jdbc:db2:ejs_samp

- **EJB Jar Directory** field: This is the directory where the WebSphere server looks to find the jar file deployed in the container. The directory is called deployedEJBs. It can be found under the following directory: drive\WebSphere\Appserver.
• **Container Database Authentication**: In this field, you see the Container User ID and password (which is the same as your DB2 administrator userid and password).

• **Deployed EJB Jar File list**: Here, you can see the list of the already deployed jar files.

In the left pane select **Enterprise Java Services - EJB Jar Files** to see which EJB beans can be deployed and which have already been deployed.

We now want to deploy the ReadCustomerInfo session bean inside WebSphere.

1. From VisualAge for Java, go to the EJB workspace and select the ReadCustomerInfo bean. Export it to the EJS jar file as described in the topic “Exporting EJB Beans to EJS or EJB Jar Files” on page 85.

2. Export the EJS jar file in the following directory: drive:\WebSphere\AppServer\deployableEJBs. Call it ReadCustomerInfo.jar. Remember that you also need to export the client .jar file and the client application, as described in “Testing the Client outside VisualAge for Java” on page 90.

3. From **Enterprise Java Services** select **EJB Jar Files**. You now see the ReadCustomerInfo.jar file in the **EJB Jar File** list.

4. Select the ReadCustomerInfo.jar file; the content of this file appears in the right pane.

5. Click the **Deploy** button. The window shown in Figure 51 appears:

Before you can deploy the ReadCustomerInfo.jar inside WebSphere, you have to make sure that WebSphere has in its classpath a reference to the CustomerObject.jar file. If it does not, you get an error message saying it could not read the .jar file when attempting to deploy. The reason why you get this is that the ReadCustomerInfo uses the CustomerObject class, which is found in the CustomerObject file. To find out how to update the WebSphere class path, please refer to Appendix A.3, “Updating WebSphere Class Path” on page 354.

---

**Note**

Before you can deploy the ReadCustomerInfo.jar inside WebSphere, you have to make sure that WebSphere has in its classpath a reference to the CustomerObject.jar file. If it does not, you get an error message saying it could not read the .jar file when attempting to deploy. The reason why you get this is that the ReadCustomerInfo uses the CustomerObject class, which is found in the CustomerObject file. To find out how to update the WebSphere class path, please refer to Appendix A.3, “Updating WebSphere Class Path” on page 354.
6. Make sure that the container selected is the right one (defaultSessionContainer in our case). Click the **Deploy** button. The window shown in Figure 52 appears:

![Deploy Jar File Window](image)

**Figure 51. Deploy Jar File Window**

7. The Window in Figure 52 box shows three options: **Regenerate** is used for JAR files that have never been deployed; it creates the stubs and skeletons and database tables necessary in a deployed EJB. **Redeploy Existing** is used for any predeployed samples; it uses the existing stubs and skeletons contained in the JAR file and creates the necessary database tables (only for entity bean). **Cancel** allows you to revert to the previous screen. Since we have exported an EJS jar file from VisualAge for Java, the ReadCustomerInfo bean is already deployed. Click **Redeploy Existing**. If everything is OK with the deployment, you get the message shown in Figure 53.
8. Log off from the administration server. Stop and restart your WebSphere Application Server for this to take effect.

- To stop WebSphere:
  - Shut down the Web server (from Start - Settings - Control Panel - Services and select Lotus Domino Go Webserver, click the Stop button.)
  - Shut down the WebSphere ServletService to stop Application Server. (Select the WebSphere Servlet Service from Start - Settings - Control Panel - Services and push the Stop button.)

- To start WebSphere:
  - The server starts automatically when you start your Web server (from Start - Settings - Control Panel - Services and select Lotus Domino Go Webserver, click the Start button.)

You are now ready to test your environment using the ReadCustomerClient application. Open a DOS Window check your classpath as explained in “Testing the Client outside VisualAge for Java” on page 90.

From the DOS Prompt, run:

java itso.samples.ejb.sessionbean.client.ReadCustomersClient iron
com.ibm.jndi.CosNaming.CNInitialContextFactory 9019

**NOTE**: In the VisualAge for Java IDE, the Persistent Name Server runs on port number 900. However, in WebSphere Advanced Edition 2.0, it runs on port number 9019. The port number value can be configured. This is the reason why when you write your client application you need to pass in the right port number.
3.7 Deployment for Other Enterprise JavaBeans Servers

If you plan to deploy your EJB bean in a non-WebSphere environment, what you have to do is to prepare an EJB Jar file as described in “Generating Deployed Code” on page 55. You can do this with VisualAge for Java by choosing the option Export-EJB Jar as shown in Figure 54.

![Figure 54. Exporting to EJB Jar Files](image-url)
Once you get your new .jar file, you can deploy it in any EJ B container using the tools the container provides to you. You need to follow the container tool instructions.

Just to help you understand the common procedures, we will deploy this new .jar file inside WebSphere. In this case we work as VisualAge for Java do not provide us the possibility to export directly to the EJS jar.

Before we can deploy the jar again, we need to undeploy it.

1. From Enterprise Java Services select EJB Jar Files. You now see the ReadCustomerInfo.jar file in the EJB Jar File list.

2. Select the ReadCustomerInfo.jar file; the content of this file appears in the right pane.

3. Click Undeploy button. The window shown in Figure 55 appears:

```
Undeploy Jar File ReadCustomerInfo.jar from a C... [ ]

Undeploy from Container: defaultSessionContainer

Undeploy Cancel

Figure 55. Undeploy Jar File Window
```

4. Make sure the container selected is the right one (defaultSessionContainer in our case). Click the Undeploy button. The window shown in Figure 56 appears:
Developing a Session Bean

5. From VisualAge for Java, go to the EJB workspace and select the ReadCustomerInfo bean. Export it to an EJB jar file as described in the topic “Exporting EJB Beans to EJS or EJB Jar Files” on page 85.

6. Export the EJB jar file in the following directory.
   :drive:\WebSphere\AppServer\deployableEJBs.

7. Follow the steps for the deployment again. This time it will take longer, since all the EJS files need to be created.

8. Logoff from the administration server. Stop and restart your WebSphere Application Server for this to take effect.

3.8 Deployment Using WebSphere Tools

WebSphere Advanced Edition 2.0 provides a tool to manually generate a deployment descriptor file for an enterprise bean (and place that descriptor file in an existing .jar file). This tool is named jet and can be found under the following WebSphere product path:

\WebSphere\AppServer\Samples\ejs\

We will give you an example of how to use the jet tool. To run jet, navigate the WebSphere path as in the window shown in Figure 57:
Select jet.bat and double click on it. The jet will start and the window shown in Figure 58 will appear.
To generate a deployment descriptor file for an enterprise bean with the jet tool, do the following:

1. Specify the full path of the .jar file containing your enterprise bean in the Input field. You can use the Browse button to obtain the file that you need. If you created your input .jar file correctly, all of the fields in either the EntityBean or SessionBean window are filled in with the default deployment descriptor values for your bean.

Try to import the ReadCustomerInfo.jar we used in "Deployment for Other Enterprise JavaBeans Servers" on page 97. The ReadCustomerInfo.jar is an EJB jar file generated by VisualAge for Java,
and it already contains a deployment descriptor. This is the reason why you get the message shown in Figure 59:

Figure 59. Message from jet Tool

Click the **Yes** button to examine the content of the Deployment descriptor.
Your **jet** tool should now appear as shown in Figure 60:
2. Specify the full path of the ejb-jar file to create in the Output field. You can use the Browse button to obtain the file that you need. (If you want to create an XML file containing the descriptor only, specify the full path to that file in the XML Output field. Note that this file is not required.). You generally specify the same name as the input. Here we just want to see how the .jar file is modified with the deployment descriptor. So just type the file name.

3. You will notice that only the SessionBean tab is available for the selection. This is why the tool recognize the type of bean it is dealing with. Select the SessionBean tab.

4. You can navigate in the jet tool tag and you will discover a match of the information shown with the property dialog in Figure 24 on page 58. To create a deployment descriptor, you must specify appropriate values for the type of bean you are creating as described in “Generate Deployment Code inside VisualAge for Java” on page 65.
5. By selecting the Environment tab, you can modify the property `customerFile` and assign it a value different from what the developer set during development time (see Figure 26 on page 60). For example, we change the path to access the file `customer.ser`:

- from:  `c:\customer.ser`
- to:    `z:\ITSO\SG245429\Redbook\customer.ser`

as shown in Figure 61:

![Figure 61. Using Jet Tool to Modify a Property Value](image)

6. When the attribute and other values are set, press the `Build` button to build the deployment descriptor file and add that file to the output .jar file.

For more details on the `jet` tool please refer to Websphere Enterprise Server for Java Programming Guide.

At this point you can deploy your EJB bean inside WebSphere.

During deployment, an ejb-jar file is generated from an ejb-jar file. The ejb-jar file contains the container-specific stub, skeleton, implementation, helper, and holder classes required by the container.
The WebSphere **EJB Deploy** command can be used to manually deploy an enterprise bean. This command-line tool can also be used to generate the database tables required for entity beans. The syntax of the EJB Deploy command follows:

```java
java com.ibm.ivj.ejb.tools.deployment.EJBDeploy ejb-jarFile workingDirectory ejs-jarFile[options]
```

The following command parameters are required:

- **ejb-jarFile**—Identifies the ejb-jar input file to be deployed.
- **workingDirectory**—Identifies a working directory that is used by the command to store temporary files during deployment.
- **ejs-jarFile**—Identifies the ejs-jar output file to be created during deployment.

Optionally, you can specify one or more of the following command parameters to fine-tune the actions of the EJB Deploy command:

- **-analyze**
  Directs the command to analyze the contents of the ejb-jarFile to ensure that it contains all required files and that the classes and interfaces define a valid enterprise bean.

- **-codegen**
  Directs the command to create EJS implementation classes only; stub and skeleton classes are not generated.

- **-dburl**
  Directs the command to create the specified database table for use by the enterprise bean. The argument to this parameter has the following format: dbAPI:database:tableName. For example, to create a database table named sample in an IBM DB2 database with the Java Database Connectivity (JDBC) API, the argument is jdbc:db2:sample.

- **-dbuid dbUserId**
  Directs the command to associate the specified database user ID with the bean so that the container can access the database on the bean’s behalf. This parameter must be used in conjunction with the -dbpass parameter.

- **-dbpass dbPassword**
  Directs the command to associate the specified database password with the Bean so that the container can access the database on the bean’s behalf. This parameter must be used in conjunction with the -dbuid parameter.

- **-force**
  Directs the command to ignore verification errors (those that can be discovered by using the -analyze parameter) and create the ejs-jar file.
- **noclean**
  Directs the command to refrain from deleting the classes written to the workingDirectory during deployment.

- **-t**
  Directs the command to report on its progress during deployment.

The follow command deploys the example ReadCustomerInfo.jar ejb-jar file into an ejs-jar file named ReadCustomerInfoEJS.jar using the working directory C:\TestApp:

```bash
C:\TestApp> java com.ibm.ivj.ejb.tools.deployment.EJBDeploy ReadCustomerInfo.jar C:\TestApp ReadCustomerInfoEJS.jar
```

Once you have deployed your enterprise beans, you have to copy the ReadCustomerInfoEJS in the deployable directory inside WebSphere and follow the steps in “Deployment on WebSphere Advanced Edition 2.0” on page 91.
4 Developing a Container-Managed Persistence Bean

4.1 Container-Managed Persistence Basics

As you read in the Introduction, a container-managed persistence (CMP) bean is an entity bean for which the container handles the interactions between the enterprise bean and the data source. The container uses the getContainerManagedFields() method from the deployment descriptor to find out which fields it is responsible for. The container is responsible for the instance fields synchronization with the persistent store. When you develop a Container-managed persistence bean you do not have to be concerned about which type of persistent mechanism is used.

This section examines the development of entity beans with CMP. While much of the information in this section also applies to entity beans with BMP, there are some major differences between the two types. For information on
the tasks required to develop an entity bean with BMP, see Chapter 5, “Developing a Bean-Managed Persistence Bean” on page 171.

4.2 Developing a CMP Bean

Typically, the development steps you would follow when using the EJB Development Environment to develop a CMP enterprise bean are:

1. Add one or more EJB groups to organize your EJB beans using one or both of the following methods:
   • Create new EJB groups. The EJB Group creation is covered in Chapter 3, “Developing a session enterprise bean” on page 19.
   • Retrieve existing EJB groups from the repository.

2. Populate your EJB groups with EJB beans using one or more of the following methods:
   • Import EJB beans from EJB jar files.
   • Create new EJB beans.
   • Retrieve existing EJB beans from the repository.

3. Add the home methods and remote (business) methods to the EJB bean class and then promote them to the home and remote interfaces. (This step can actually be done anytime after you have added your EJB beans. However, it must be done before you generate the EJB deployed classes.) In this chapter, we explain the addition of home and remote methods before the insertion of CMP fields.

4. Add, define, and map any required CMP fields:
   1. Add required fields to the EJB beans.
   2. Define CMP fields and the key field for the CMP entity EJB beans.
   3. Map the CMP fields by doing one of the following:
      • Generate a default database schema; this approach is called Top-down.
      • Map CMP fields to an existing database table (meet in the middle) by either:
         • Importing a schema from an existing database table and mapping the CMP fields to it.
         • Creating a new schema, using it to create a new table, and mapping the CMP fields to it.
5. Set your Deployment descriptors. (This step can be done anytime after you have added your EJ B beans. However, it must be done before you test/deploy your EJ B beans.)

6. Generate the EJ B deployed classes.

7. Test the EJ B beans.
   1. Create an EJ B server configuration.
   2. Start the DB2 servers (if testing entity beans).
   3. Create any required database tables.
   4. Start the Name Service and Location Daemon servers and the EJ B server.
   5. Generate an EJ B test client.
   6. Run the EJ B test client.

In this chapter we create a sample Customer EJ B bean, add methods to its remote and home interfaces, add CMP fields, create an empty DB2 database, map the CMP fields to the database tables (using the meet-in-the-middle approach), set the Deployment Descriptors, generate the deployed code and test the Bean.

Creating a CMP Bean

Use this procedure to select the EJ B group into which you want to insert your EJ B beans. To add EJ B beans to the group:

1. In the EJ Bs list, click on the ITSOEJBGroup EJ B Group.
2. Click the mouse button 2. From the pop-up menu, select Add - EJ B.
   The Create EJ B SmartGuide appears (see Figure 62).
As you can see using the Create EJB SmartGuide, you can add an EJB by either creating a new EJB or by retrieving one or more existing EJB beans from the Repository.
To add an EJB by creating a new EJB:

1. Ensure that the **Create a new Bean Class** radio button is selected.

2. In the **Bean name** field, type in **Customer**, which is the name that you want to assign to your new EJB bean.

3. Since you are creating a new EJB bean class, you need to specify the type of EJB you want to create. The Customer bean is a CMP bean, so from the **Bean type** drop-down list, choose **Entity Bean with container-managed persistence fields (CMP)**.

4. In the **Project** field, you can type the project name, modify the default name, or click the **Browse** button and select the project that will contain the EJB bean class. If you type a project name that does not exist, VisualAge for Java will not let you create your Bean, as shown in Figure 63. By default, the project to which the EJB group belongs is displayed. In our case we use the default project, which is ITSO Bank.
Figure 63. Project Not Found
5. In the **Package** field, you can type the package name, modify the default name, or click the **Browse** button to select the package that will contain your EJB bean. If you enter a package that does not exist, a new package is created. Type itso.samples.ejb.CMP in the package field.

6. In the **Class** field, either accept the default value or type in a name for the new class. We recommend that you use the default value so that all the implementation types of the EJB bean are named properly.

7. Since we do not provide any superclass for our Customer bean, we do not fill in the **Superclass** field. Your SmartGuide should look like Figure 64.
Create EJB

- Create a new EJB
  - Bean name: Customer
  - Bean type: Entity Bean with container-managed persistence fields (CMP)

- Create a new Bean class

- Use an existing Bean class
  - Project: ITSO Bank
  - Package: itso.samples.ejb.CMP
  - Class: CustomerBean
  - Superclass: 

- Add EJB(s) from the repository

Figure 64. Create EJB SmartGuide
Click on **Next >** to go to the next page of the Create EJB SmartGuide. Your screen should now look like Figure 65.

![Figure 65. EJB Class Attributes and Interfaces Definition](image)

---

Developing a Container-Managed Persistence Bean 115
8. We use the default names in our Customer example for the EJB Home Interface, EJB Remote Interface, and EJB Key Class or Field, so we do not change their values. If you wish to use your own Home Interface, Remote Interface or the KeyClass names you can change them in this window.

The **Create finder helper interface to support finder methods** checkbox tells VisualAge for Java to generate the CustomerBeanFinderHelper method, which is needed to add a special finder on the Customer beans. If you do not need any other finder than the getByPrimaryKey for your bean, you do not need to check this option.

If you need to implement specific import and/or interfaces in your bean, select them using the appropriate buttons.

The **Create method stubs which must be implemented (Recommended)** checkbox ensures that stubs are created for the methods declared in the interfaces you selected.

9. Click on **Finish** to generate the classes you need to develop the Customer implementation.

Now, your Workbench should look like Figure 66.
/**
 * This is an Entity Bean class with CMP fields
 **/
import java.rmi.RemoteException;
import java.security.Identity;
import java.util.Properties;
import java.ejb.*;
import java.lang.*;

public class CustomerBean implements javax.ejb.EntityE
private javax.ejb.EntityContext entityContext = null;
public String primaryKey;
}
As you can see in Figure 66, VisualAge for Java generates five different types for you. These types are, respectively:

- **Customer** interface (the remote interface)
- **CustomerBean** class (the Entity Bean)
- **CustomerBeanFinder** interface
- **CustomerHome** interface and
- **CustomerKey** class

VisualAge for Java automatically creates, according to the EJB specification, these five classes required for a CMP EJB bean.

**Projects, Packages and Reserved Package Organization**

Whenever you add an EJB group to the EJBs page, a reserved package is created under the project in the following form:

\(<EJB\_group\_name>EJBRreserved\) (for example, in our case, VisualAge for Java creates the reserved package \(ITSOEJBGroupEJBReserved\)).

1. Click on the **Projects** page.

2. Expand the ITSO Bank project and check that the **ITSOEJBGroupEJBReserved** package has been created. This reserved package is used to hold Java classes that represent your EJB objects and their associated meta data. Although the EJB bean is represented as a class in the reserved package, you cannot see the EJB source code because all information associated with the EJB object is saved as the meta data of the corresponding class. For this reason, you should always use the EJBs page to edit your EJB source code and should never attempt to directly delete, replace, reorganize, or otherwise manage the classes contained in the reserved package of the project.

We created another package, **itso.ejb.samples.CMP** , to contain our bean code, schemas and maps. These two packages are included in the ITSO Bank project.

3. Click on the EJBs page to get back to the EJB Development Environment.

The association between the EJB reserved package and the project is done at the creation of the EJB Group (see Figure 67).
The association between the project and the package containing our bean code, schemas, and maps is made at the creation of the EJB bean (see Figure 68).
Figure 68. Association between EJB Class Package and Project

Understanding the Generated Types

- **Customer** interface
  The Customer interface is the remote interface for the CustomerBean EJB object. The Customer bean's remote interface provides access to the business methods available in the CustomerBean class. By convention, the remote interface is named Name, where Name is the name you assign to the enterprise bean. For example, the Customer enterprise bean's remote interface is named Customer.
  The Customer remote interface is covered in detail in 4.4, “Adding Remote Interface Methods” on page 137.

- **CustomerBean** class
  The CustomerBean class represents the entity bean.
  - It must implement the business methods used to access and manipulate the data associated with the enterprise bean.
Developing a Container-Managed Persistence Bean

- It must define and implement an `ejbCreate` method for each way in which the enterprise bean can be instantiated. A corresponding `ejbPostCreate` method must be defined for each `ejbCreate` method. A thumb rule for deciding the parameters for an `ejbCreate` method: all the non-nullable fields should be initialized from the parameters so that there is no exception at the persistence store end. For example, when using RDBMs, if a row is inserted into a table with null values for columns declared as non-nullable, the RDBMs will throw an exception.

The `CustomerBean` class defines and implements the business methods of the `Customer` enterprise bean, defines and implements the methods used to create instances of the `Customer` enterprise bean, and implements the methods used by the container to inform the instances of the enterprise bean of significant events in the instance's life cycle (callbacks methods). By convention, the enterprise bean class is named `NameBean`, where `Name` is the name you assign to the enterprise bean. The enterprise bean class for the example `Customer` enterprise bean is named `CustomerBean`.

- **`CustomerBeanFinderHelper` interface:**

  The `CustomerBeanFinderHelper` interface is used by the container to generate the necessary code for querying the database on custom filters. You use the special finder methods to retrieve instances from the database, using other search criteria than the primary key value (for example, a search by customer's last name). For each finder method defined in the home interface (`CustomerHome`) other than the `create` and `findByPrimaryKey` methods, you are responsible to define the corresponding query string.

  Section 4.5, “Modifying or Adding Home Methods” on page 140 explains how to add finder methods.

- **`CustomerHome` interface:**

  The `CustomerHome` interface is the home interface for the `Customer` entity bean. An entity bean's home interface defines the methods used by clients to create new instances of the bean, find and remove existing instances, and obtain meta data information about an instance. This home interface is implemented by the EJB home class generated by the container’s deployment tool. The container registers the home interface into the naming tree, allowing EJB clients to access it using the JNDI API.

  By convention, the home interface is named `NameHome`, where `Name` is the name you assign to the enterprise bean. For example, the `Customer` enterprise bean's home interface is named `CustomerHome`. 

Developing a Container-Managed Persistence Bean 121
Section 4.5, “Modifying or Adding Home Methods” on page 140 covers the Home interface.

- **CustomerKey** class:

  Every enterprise entity bean has a unique identity within a container that is defined by using a combination of the object’s home interface name and its primary key and assigned at object creation time. If two EJB objects have the same identity, they are considered identical. The bean key class is used to encapsulate a primaryKey. By convention, the bean key class is named NameKey, where Name is the name of the enterprise bean. For example, the Customer enterprise bean’s primary key class is named CustomerKey.

  - It must be public and it must be serializable.
  - Its instance variables must be public and the variable names must match a subset of the variable names defined in the enterprise bean class.

  The CustomerKey class is an object generated by VisualAge for Java to help you creating new instances and finding existing instances of the Customer entity bean from the database. This CustomerKey object is a unique representation of the Customer. The methods from the CustomerHome interface using it are:

    - create(CustomerKey)
    - findByPrimaryKey(CustomerKey)
    - methods generated for the CustomerKey

  These methods are described in 4.3, “Understanding the Generated Methods” on page 131.

**Adding Fields to the Bean**

In order to enrich the customer bean, we add the fields: firstName and lastName.

1. In the EJBs list, select **Customer** and click on the fields icon located at the upper right corner of the Types listbox. This toggles the listbox name from Types to Fields. It also modifies the content of the listbox and displays fields for the selected EJB bean. If you do not see any, make sure that you have the Customer EJB selected. Your screen should look like Figure 69.
Figure 69. CustomerCMP Entity Bean’s Fields

VisualAge for Java automatically creates two fields:
- entityContext
- primaryKey
Now we add two additional fields firstName and lastName.

1. In the EJ B list of the EJ Bs page, select the Customer EJ B bean.
2. You need to display the classes by clicking on the Types icon.
3. Select the CustomerBean class, and click mouse button 2 in the types list. Select Add Field from the pop-up menu. The Create Field SmartGuide appears.
4. Fill in with the following information: Field Name is firstName, Field Type is java.lang.String. All the CMP Fields of a CMP Entity Bean must be public, so the Access Modifiers must be public. Mark the Access with setter and getter methods check box as checked, and select in the setter and getter group boxes the public radio buttons. Your screen should look like Figure 70.

Other Modifiers group box options are described under “Regarding the Other Modifiers (Final, Transient, Static)” on page 129.
Figure 70. Create Field SmartGuide

5. Click **Finish**.

6. Repeat steps 3 and 4 for two additional fields: lastName and dummy.

7. Click on the Fields icon to see the newly created fields: dummy, **firstName** and **lastName**, in your CustomerBean entity bean; see Figure 71.
The idea with CMP EJB beans is to let the container handle all of the database interactions involved with the creation, read, update and deletions of EJB beans. In order to achieve this, you need to identify which are the fields you want to be CMP fields.

Each of these identified CMP fields are persistent in a database record field. The container determines which fields are of CMP type by calling the getContainerManagedFields() method of the entity descriptor, which returns a Field[] array. VisualAge for Java lets you identify which fields are CMP fields.

To define dummy, firstName and lastName as CMP fields:

1. Select the dummy field, press and hold down the Ctrl key on the keyboard, and click with mouse button 1 on firstName and lastName.
2. Click on mouse button 2 and select Container Managed. An icon appears on the right side of dummy, firstName and lastName indicating that they are Container-managed persistence fields; see Figure 72.
Developing a Container-Managed Persistence Bean

Figure 72. CMP Fields Icon

The code you provide in your beans does not contain any database access, creation, and updates. VisualAge for Java generates the necessary code at deployment time for you.

VisualAge for Java also generates all the accessors (getter and setter). These methods are described in topic 4.4, “Adding Remote Interface Methods” on page 137.

Removing a Field

In this section we show you how to remove the dummy CMP field from the Customer CMP Bean.

1. Click on the dummy field from the Fields list. Click on mouse button 2 and deselect Container-Managed.
2. From the Methods list, you need to click on the getDummy() method. Press the Ctrl key on the keyboard and select the setDummy() method as well.
3. Click on mouse button 2, select Delete.
4. VisualAge for Java ask you for a deletion confirmation, click Yes.
5. You just have deleted the getter and setter methods for the field you want to delete.
6. You need to click on the Customer EJB bean from the EJBs list to see the Class definition in the source window. You need to delete the dummy field declaration from the source window; see Figure 73.
6. Press Ctrl-S to save your Customer bean. The dummy field disappears from the Fields list. Your Workbench now looks like Figure 74.
Regarding the Other Modifiers (Final, Transient, Static)

- The **final** modifier can be used on fields which will not change in their lifetime. If you select the **final** modifier, VisualAge for Java will not generate a setter method for the field, but it will generate a getter method. This means, you will be able to retrieve the value of the field, but you will be not able to set it. This final CMP field should be initialized properly in the ejbCreate method. You can use a final CMP field to store a constant value in from your database into a member attribute of your CMP entity bean.

- The **static** modifier is not supported by CMP fields.

- The **volatile** modifier can be used as follows. The Java language allows threads that access shared variables to keep private copies of the variables. These private copies need to be reconciled with the master copies in the shared memory. One way to accomplish this is by using the volatile modifier.

- The **transient** modifier can be used for local variables. The **transient** modifier basically means that the CMP member field is not persisted to the database. For example, assume a field contains the result of a long calculation. A remote method needs this field several times during its execution. Instead of calling the field's getter methods several times, we can use a CMP transient field to store its value. This way we save CPU processing time.
Working with the Key Class
When you create an entity bean, a default key generation is done. This generated key has been explained in “Understanding the Generated Types” on page 120. There are three different things you can do with this default key class:

- Use it as it is.
- Modify it by adding additional fields to it.
- Redefine it to use other field(s).

In our example, we add the lastName field to the primaryKey object.

Adding the lastName field to the CustomerKey
1. Select the customer EJB.
2. Select the lastName from the fields list.
3. Click mouse button 2 and select Key Field from the pop-up menu. Two things happen: in addition to the CMP icon, the key icon is added to the lastName field, and the lastName field is added to the CustomerKey class.
4. You then need to initialize the lastName member in the ejbCreate method. Click on the Types icon from the Fields list. Select the CustomerBean in the Types list. Select the ejbCreate(CustomerKey) from the Methods list. The following code segment is displayed in the Source window:

```java
/**
 * ejbCreate method comment for a CMP entity bean
 */
public void ejbCreate(CustomerKey key)  {
    // All CMP fields should be initialized here.
    primaryKey = key.primaryKey;
}
```
5. Add the following line:

```java
lastName = key.lastName;
```
6. Press CTRL-S to save your method.

Each time you create a Customer EJB bean using its CustomerKey object, the proper key fields (primaryKey and lastName) are initialized properly.

Redefining the Default Key
If you need to use another field than the default one generated as key, you can do it by following these tasks (be careful when you do this, because
VisualAge for Java does not provide you with an automatic mechanism to replace a key; you do have to replace some definitions and references).

1. In the EJBs pane of the EJBs page, select the bean for which you want to redefine the primaryKey.

2. Delete the field from the class definition of the bean (example: public string primaryKey;) and then save the class. This action creates temporary errors that will be fixed in the following steps.

3. Add your new primary key field (example: public int newKey) using the Create Field SmartGuide.

4. Click the Fields icon from the Types list. Go in the field list and select your newly created key field. Click mouse button 2, then select Key Field from the pop-up menu and save.

5. Toggle to the types list by clicking the Types icon.

6. Select the bean from the Types list, then the ejbCreate method from the Methods pane and modify the initialization of the key field (example: primaryKey=key.primaryKey to newKey=key.newKey). Press CTRL-S to save your method.

7. If your new key has a type different from the default key, select the key class; and from the Methods list, delete the original constructor using the old key type.

8. You can now work with your newly created key.

### 4.3 Understanding the Generated Methods

VisualAge for Java automatically creates some methods for you. These methods are mapped to the methods described in the contract for the entity beans in the CMP section of the Enterprise JavaBeans specifications. We give here the different CMP generated methods, for the following types:

- The CustomerBean class
- The CustomerHome interface
- The CustomerKey class

Note that there no generated methods for the Customer and CustomerFinderHelper methods when you create a new CMP entity bean. The methods in the Customer interface are covered in 4.4, “Adding Remote Interface Methods” on page 137; and the addition of special finder methods in the CustomerFinderHelper is covered in 4.5, “Modifying or Adding Home Methods” on page 140.
To help understanding the generated methods, we need first to look into the lifecycle of an entity bean.

**Life Cycles of Enterprise Bean Instances**

Once an enterprise bean is deployed into a container, clients can create and use instances of that bean as required. Within the container, instances of an enterprise bean go through a defined life cycle. The events in an enterprise bean's life cycle are derived from actions initiated by either the client or the container. You must understand this life cycle, because for some enterprise beans, you must write some of the code to handle the different events in the enterprise bean's life cycle.

**Creation State**

An entity bean instance's life cycle begins when the container creates that instance. After creating a new entity bean instance, the container invokes the instance's `setSessionContext` method. This method passes the bean instance a reference to an entity context interface that can be used by the instance to obtain container services and get information about the caller of the client-invoked method.

**Pooled State**

Once an entity bean instance is created, it is placed in a pool of available instances of the specified entity bean class. While the instance is in this pool, it is not associated with a specific EJB object. Every instance of the same enterprise bean class in this pool is identical. While an instance is in this pooled state, the container can use it to invoke any of the bean's finder methods.

**Ready State**

When a client needs to work with a specific entity bean instance, the container picks an instance from the pool and associates it with the EJB object initialized by the client. An entity bean instance is moved from the pooled to the ready state if there are no available instances in the ready state.

There are two events that cause an entity bean instance to be moved from the pooled state to the ready state:

- When a client invokes the create method in the bean's home interface to create a new and unique entity of the entity bean class (and a new record in the data source). As a result of this method invocation, the
Developing a Container-Managed Persistence Bean

container calls the bean instance's ejbCreate and ejbPostCreate methods and the new EJB object is associated with the bean instance.

• When a client invokes a finder method to manipulate an existing instance of the entity bean class (associated with an existing record in the data source). In this case, the container calls the bean instance's ejbActivate method to associate the bean instance with the existing EJB object.

When an enterprise bean instance is in the ready state, the container can invoke the instance's ejbLoad and ejbStore methods to synchronize the data in the instance with the corresponding data in the data source. In addition, the client can invoke the bean instance's business methods when the instance is in this state. All interactions required to handle an entity bean instance's business methods in the appropriate transactional (or non-transactional) manner are handled by the container, unless the EJB developer has decided to handle these interactions.

When a container determines that an entity bean instance in the ready state is no longer required, it moves the instance to the pooled state. This transition to the pooled state results from either of the following events:

• When the container invokes the ejbPassivate method.
• When the client invokes a remove method on the EJB object associated with the Bean instance or on the EJB home object. When one of these methods is called, the underlying entity is removed permanently from the data source.

Removal State

An entity bean instance's life cycle ends when the container invokes the unsetEntityContext method on a entity bean instance in the pooled state. Do not confuse the removal of an entity bean instance with the removal of the underlying entity whose data is stored in the data source. The former simply removes an uninitialized object; the latter removes data from the data source.

The CustomerBean's Generated Methods

You can see the generated methods in the Methods list of the CustomerBean Class.

• public void ejbActivate();

This life-cycle call-back method of the entity bean is invoked when the container assigns an instance from the pool to a specific EJB object.
identity. This signals the ready state for the enterprise bean. You can use this method to load additional resources in the bean.

If you need to do one-time initialization on non-CMP fields from your CMP Bean, you could use this callback method to do this. For example, we used the ejbActivate in our Bank inheritance implementation to initialize the Proxy object (see Chapter 11, “Inheritance” on page 271).

- public void ejbCreate(CustomerKey key);

VisualAge for Java generates a default ejbCreate(...) method which takes the EJB Key class, CustomerKey, as a parameter. This signature matches the create(...) method of the bean’s home interface. It is your responsibility to:

- Initialize the instance’s variables (like we did when we inserted the lastName field in the CustomerKey object).

It is the container responsibility to:

- Create an entry representing the entity in the persistence store.

The return value must be void for entity beans with CMP. You may have expected a Customer object being returned? In fact, a client does not access directly the ejbCreate, but instead it calls the Home interface create() method, which does return a Customer object (see “The CustomerHome Interface’s Generated Methods” on page 135).

- public void ejbLoad(), public void ejbStore()

We describe these two methods at the same time, since their utilization is quite similar. The EJB Specifications from Sun Microsystems are quite vague on these ejbLoad and ejbStore methods. They say that:

- "..., the container can invoke the ejbLoad() and ejbStore() methods one or more times, at anytime. " (section 9.3 - Instance Lifecycle, p.62)

- "The purpose of the ejbLoad method is to synchronize the state of the instance with the state of the entity in the underlying data source" (section 9.3 - Instance Lifecycle, p.62)

Let us give you a clear understanding of what these two points mean with VisualAge for Java and WebSphere. First, for all CMP fields in your CMP bean, you do not have to be concerned about the CMP fields’ state synchronization with the data store, since the container takes care of it. This means that for CMP fields, you do not need to add any code to these callback methods.

WebSphere always calls ejbLoad() after a new transaction has been started, and after you invoke a method from the remote interface. If you have any non-CMP fields in your CMP bean which always need
Developing a Container-Managed Persistence Bean

The latest state (like stock tickers), you should use the \texttt{ejbLoad()} callback method to initialize them.

The \texttt{ejbStore} is called after a transaction has been committed.

- \texttt{public void ejbPassivate();}

The EJB container invokes this method when it decides to send the instance to the pooled state based on a least-recently-used algorithm. With VisualAge for Java and WebSphere this means, your instance is disassociated from its EJB Object after a defined period (time-out).

You can use this method to release any resources that were allocated in the \texttt{ejbActivate()} method.

- \texttt{public void ejbPostCreate(CustomerKey key);}

This life-cycle callback method indicates that the EJB Object identity is available. For each \texttt{ejbCreate(...)} method, there is a matching \texttt{ejbPostCreate(...)} method with the same input parameters.

If you add an \texttt{ejbCreate()} to the entity bean, you need to add an \texttt{ejbPostCreate()} method as well, with matching parameters.

- \texttt{public void ejbRemove();}

This method is invoked by the container when the client invokes the remove method on an EJB object reference. You can use the \texttt{ejbRemove} method to implement actions you want to take place before the instance is removed from the database. The record is deleted from the database after \texttt{ejbRemove} returns. You do not have to implement the logic to remove the entity from the persistence store; the container takes care of it.

- \texttt{public void setEntityContext(EntityContext ctx);}

- \texttt{public void unsetEntityContext(EntityContext ctx);}

These two methods are explained in the “Life Cycles of Enterprise Bean Instances” on page 132.

The \texttt{CustomerHome} Interface’s Generated Methods

To see the \texttt{CustomerHome} interface’s generated methods, click on the \texttt{CustomerHome} interface from the \texttt{Types} list.

- \texttt{public Customer create ( itso.ejb.samples.CMP.CustomerKey primaryKey) throws javax.ejb.CreateException, java.rmi.RemoteException;}

A create method is used by a client to create an enterprise bean instance and insert the data associated with that instance into the data source. Each create method must be named create and it must
have the same number and types of arguments as the corresponding 
ejbCreate method in the enterprise bean class. The return types of the 
create method and its corresponding ejbCreate method are always 
different, for example, in our case the create method returns a 
Customer while the ejbCreate returns nothing.

Each create method must meet the following requirements:

• It must be named create.

• It must return the type of the enterprise bean’s remote interface. For 
  example, the return type for the create methods in the CustomerHome
  interface is Customer.

• It must have a throws clause that includes the 
  java.rmi.RemoteException exception, the javax.ejb.CreateException 
  exception, and all of the exceptions defined in the throws clause of the 
  corresponding ejbCreate and ejbPostCreate methods.

When you create a new CMP bean, VisualAge for Java creates this 
default create method for you. All of the above requirements are 
respected. VisualAge for Java automates this task for you.

•

• public Customer findByPrimaryKey
  (itso.ejb.samples.CMP.CustomerKey primaryKey) throws 
  java.rmi.RemoteException, javax.ejb.FinderException;

A finder method is used to find one or more existing entity EJB objects. 
At minimum, each home interface must define the findByPrimaryKey
method that enables a client to locate an EJB Object by using the 
primary key only. VisualAge for Java creates this default finder
tool method automatically for you.

We have in our example a findByPrimaryKey which uses a 
CustomerKey object to identify a particular entity. The 
findByPrimaryKey returns the type of the CustomerBean remote 
interface Customer.

The CustomerKey’s Generated Methods

VisualAge for Java creates four methods for the CustomerKey class: a 
default constructor CustomerKey, a constructor which takes the 
primaryKey and lastName as arguments, an equals method, and a hash
method.
A primary key class is used to create and manage the primary key for an EJB object. The primary key class must meet the following requirements:

- It must have a public default constructor, at minimum.
- It must have an equals method and a hashCode method that provide the same functionality for the primary key class as these methods provide for classes such as java.lang.String.

In the primary key class for the Customer enterprise bean, CustomerKey, we have those two methods. The hashCode method for the CustomerKey class simply invokes the corresponding hashCode method in the primaryKey and lastName member fields. In addition to the default constructor, the CustomerKey class also defines a constructor that sets the value of the primaryKey and lastName variables to specified string parameters. The equals method is called on a CustomerKey instance and is passed another CustomerKey instance as parameter. The method compares all the members of the two CustomerKey objects and return true or false. This defines whether the two Customer instances are identical or not.

### 4.4 Adding Remote Interface Methods

The business methods you want to access from your beans have to be defined in the remote interface. The remote interface describes the entity beans business methods which are accessible by a client.

Every remote interface must meet the following requirements:

1. It must extend the javax.ejb.EJBObject interface. The EJB's remote interface inherits several methods from the javax.ejb.EJBObject interface. VisualAge for Java takes care of this for you. See “Methods Inherited from javax.ejb.EJBObject” on page 139 for information on these methods.
2. Each method implemented in the enterprise bean class that needs to be accessed by a client must appear in the remote interface. VisualAge for Java automates this task.
3. The parameters and return value of each method defined in the interface must be a valid Java RMI type. For more information, see “Using Valid Parameters and Return Values for Java RMI” on page 139.
In this section we will add four methods to the remote interface: two methods
to get information from the bean, and two other methods to update
information in the bean.

VisualAge for Java generated these methods for us when we created the
fields. The steps to add them to the remote interface are:

1. Select the CustomerBean Class in the Types list; this displays its
corresponding methods.

2. Select the getFirstName method and click mouse button 2. Choose Add
To - EJB Remote Interface. The method is marked with an icon indicating that the method has been added to the Customer remote
interface.

3. To select multiple methods, press the Ctrl key on the keyboard and click
each of the methods you want to select: getLastName, setFirstName,
setLastName. Click mouse button 2 and select Add To - EJB Remote
Interface. Your Types and Methods panes should look like Figure 75.

4. To see where VisualAge for Java added your remote methods, select the
Customer interface from the Types list. You see the four methods in the
Methods list; see Figure 76.
Methods Inherited from javax.ejb.EJBObject

The remote interface inherits the following methods from the javax.ejb.EJBObject interface:

- getEJBHome—Returns the enterprise bean's home interface.
- getHandle—Returns the handle for the EJB object.
- getPrimaryKey—Returns the EJB object's primary key.
- isIdentical—Compares this EJB object with the EJB object argument to determine if they are the same.
- remove—Removes this EJB object.

These methods have the following syntax:

- public abstract EJBHome getEJBHome();
- public abstract Handle getHandle();
- public abstract Object getPrimaryKey();
- public abstract boolean isIdentical(EJBObject obj);
- public abstract void remove();

These methods are implemented by the container (WebSphere Advanced Edition 2.0) in the EJB object class.

Using Valid Parameters and Return Values for Java RMI

To be valid, a method's arguments and return value must be serializable. If you attempt to use a parameter that is not serializable, the NotSerializableException exception is thrown.
Most of the classes in the java.lang and java.util packages are already defined as serializable. You can make your own classes serializable by implementing the java.io.Serializable interface.

4.5 Modifying or Adding Home Methods

Home interface methods let you create and search for your entity beans. VisualAge for Java generates the create method for you based on the primary key object. The EJB specification says that the container provider has to generate the finder methods for the CMP entity beans. VisualAge for Java generates the finder methods at bean deployment time.

Here are the steps you need to perform to describe finder methods. For each finder method defined in the home interface, other than findByPrimaryKey method, a query string must be defined.

This query string is an SQL-prepared statement which is executed at run time by the EJB server to locate the EJB bean(s) according to the finder method’s requirements.

You have to write a description of this finder method in the <name>BeanFinderHelper interface in order to let VisualAge for Java know what to generate. We describe how to describe and generate a finder method for the Customer bean in topic 4.9, “Adding Special Finder Methods” on page 158.

4.6 Generating Database Schema and mapping

Before you can execute and test your enterprise beans, you need to map your CMP beans fields to your database structure. In the case of the Customer bean, we want to do the following mapping:

- primaryKey = CUSTID
- lastName = LNAME
- firstName = FNAME

Refer to Figure 77.
With VisualAge for Java you make an association between CMP fields and database columns by using two different browsers:

- Schema Browser
- Map Browser

With the Schema Browser you first define a schema by giving it a name (SCHEMA1). This schema must be associated to a given database (Database1). From this database you select all or a subset of tables (table1 and Table 4). Then for each column you may accept or change the converter that transforms DB2 to Java data types (see Figure 78).
After this has been done, you use the Map Browser to associate CMP fields and columns. Once again, you need to give a name to your map (MAP1). Then you associate this map to a given schema (SCHEMA1). By doing this, the mapping possibilities are limited to those defined in the schema.

You make another association between the map you are working with and a given EJB group. Then each CMP bean of this EJB group is associated to one and only one of the tables defined in the schema (CMP1 to Table 4, see Figure 79). The next operation consists in linking each field with a column in a simple or complex Map Type. In a simple Map type, a field is associated to only one column. In a complex Map Type, a field can be associated to more than one column.

Figure 79 describes the relationship between the different elements and browsers involved in CMP field-to-column mapping.
Developing a Container-Managed Persistence Bean

The EJB Development Environment from VisualAge for Java allows top-down and meet-in-the-middle mapping of EJB beans to database tables. It does not yet support the bottom-up approach. In our Customer example, we cannot use the top-down approach, since WebSphere does not support primary key objects with more than one field, and we have two fields (primaryKey and lastName) in our CustomerKey class. We will use the meet-in-middle approach.

The Top-Down Approach

The top-down approach can be used to quickly create simple table and field definitions in your database server. The top-down approach for mapping EJB beans to database assumes you have existing EJB beans and no existing database structure. For example, you may have created a new CMP bean and you want to generate a database table which corresponds to it; the CMP fields are mapped to the table field. The generated code creates SQL code in the bean class so that the default database tables can be created for the bean class. This supplies a default mapping between the bean and the database table.
If you want to use the top-down approach you need to follow these steps:

1. Select your EJB Group and click mouse button 2. From the popup window, select Generate->Deployed Code. This is going to generate all the classes required to execute the EJB in the WebSphere Advanced Edition 2.0 container.

2. Select the EJB Group for which you want to create the database tables. Click mouse button 2 and select Add To - Server Configuration.

3. Select your EJB Server from the Servers list, click on mouse button 2, and select Properties; see Figure 80.

   ![Figure 80. EJB Server Properties](image)

   Figure 80. EJB Server Properties

4. Enter the JDBC database path in the Database URL. (In this version of WebSphere Advanced Edition 2.0, the initial value displayed in this field cannot be pre-configured. You need to change it manually). The database you specify here must exist in order to get a successful tables and fields generation. Click OK.

5. Select your EJB Server from the Servers list and click on mouse button 2, select Create Database Table from the pop-up menu. When the focus returns to the EJB Server Configuration window, your database table has been created (see Figure 81). Then, you can start your server and test it.
For each bean class, a table is generated. In each table definition, a column is defined for each container-mapped field of the bean class. The container-managed fields are mapped as described in Table 1.

Table 1. Container-Managed Fields Mapping

<table>
<thead>
<tr>
<th>CMP Field's Java Type</th>
<th>Mapped Java Type</th>
<th>SQL Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang.String</td>
<td>String</td>
<td>VARCHAR(251)*</td>
</tr>
<tr>
<td>short or java.lang.Short</td>
<td>short</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>int or java.lang.Integer</td>
<td>int</td>
<td>INTEGER</td>
</tr>
<tr>
<td>float or java.lang.Float</td>
<td>float</td>
<td>FLOAT</td>
</tr>
<tr>
<td>double or java.lang.Double</td>
<td>double</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>byte or java.lang.Byte</td>
<td>short</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>long or java.lang.Long</td>
<td>String</td>
<td>VARCHAR(22)</td>
</tr>
<tr>
<td>char or java.lang.Character</td>
<td>String</td>
<td>CHAR(1)</td>
</tr>
</tbody>
</table>
Strings are limited to 251 characters in length due to DB2 restrictions dealing with varying length data types and their use in primary key types and certain SQL clauses.

** Non-primitive java types are stored as serialized instances in the database using BLOB data types.

### The Bottom-Up Approach

In a bottom-up approach, you want the tool to create Enterprise JavaBeans from an existing database. This approach is not yet supported in this version.

### The Meet-in-the-Middle Approach

The meet-in-the-middle approach for mapping EJB beans assumes that you have existing EJB beans and an existing database. To complete the mapping you:

1. Import the database schema from the database into the EJB Development Environment using the Schema Browser, describe in “Importing a Database Schema into VisualAge for Java” on page 147.

2. Define a map between the EJB beans and schema using the Map Browser which is defined in “Defining a Map between EJB Beans and the Database Schema” on page 151.

### Creating the CMP Sample Database

The utilization of Schema Maps and Database Schemas browsers is based on the usage of a DB2 database. Before we continue, we invite you to install this database, which is provided with this book. The required files can be found either in the CD-ROM or the zip file that you have downloaded from the ITSO Web site.

<table>
<thead>
<tr>
<th>CMP Field's Java Type</th>
<th>Mapped Java Type</th>
<th>SQL Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean or java.lang.Boolean</td>
<td>short</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>java.math.BigDecimal</td>
<td>java.math.BigDecimal</td>
<td>NUMERIC</td>
</tr>
<tr>
<td>java.sql.Date</td>
<td>java.sql.Date</td>
<td>DATE</td>
</tr>
<tr>
<td>java.sql.Time</td>
<td>java.sql.Time</td>
<td>TIME</td>
</tr>
<tr>
<td>java.sql.Timestamp</td>
<td>java.sql.Timestamp</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>Other</td>
<td>serialized</td>
<td>BLOB(1M)**</td>
</tr>
</tbody>
</table>
1. There are several ways to create the database and populate the corresponding tables. One way is to open a DB2 window. This is done by going to Start->Programs->DB2 for Windows NT->Command Window. From the command prompt, change to the directory where you copied the following files from the cdrom\Part1Samples\CMP:
   - CMP.DDL
   - CMP.SQL
2. Enter db2 -f CMP.DDL and hit Enter. This creates the CMPS database and CUSTOMER table.
3. When the command completes, enter db2 -f CMP.SQL and hit Enter. This populates the CUSTOMER table.
4. Type exit and hit Enter to close the DB2 command window.

Importing a Database Schema into VisualAge for Java

The Schema Browser is used to describe the schema of the relational database into which the persistent classes are stored. To import a database schema into VisualAge for Java, you first need to start the Schema Browser:
1. Select the Customer EJB bean from the EJBs list. Select the EJBs menu. Select Open To - Database Schema.

You should see the Schema Browser Window like in Figure 82.

![Figure 82. Schema Browser Window](image)
The Schema Browser window is divided in five sections: the schemas list lets you interact with schemas, the tables display a schema's tables, and the columns and foreign key relationships panes are associated with a selected table. The bottom part gives some information and statistics according to what is selected in the top panes.

To import our Customer Table schema:

1. Click the mouse button 2 in the Schemas list and select Import / Export Schema - Import Schema from the Database.

2. You then need to give a name to your schema, in our example we use the same name as the Customer EJB class name. Enter Customer then click OK.

3. VisualAge for Java will display the Database Connection Info dialog. Enter COM.ibm.db2.jdbc.app.DB2Driver as the connection type to tell it you want to use the IBM DB2 JDBC driver. Then enter the database JDBC name, jdbc:db2:CMPS. Your screen should now look like Figure 83; click OK.

4. You need to select the tables from which you want to import into. Select the CMPS, then click the Build table list. You should see your CUSTOMER table in the tables list as in Figure 84.

5. Select the CUSTOMER table and click OK. The Qualifier corresponds to the schema name defined in your DB2 database.
Figure 84. Select Tables Window

6. The Schema Browser now has one table added to the Customer schema, the CUSTOMER table. If you select the Customer Schema and then the CUSTOMER table, you will see the different columns in the columns list; see Figure 85.

Figure 85. Schema Browser
We need to change the default type converter for all Char fields to VapTrimStringConverter. If VapTrimStringConverter is not used and the default VapConverter is used with database mappings of specific lengths, then there will be persistence problems.

7. Select CUSTOMER from the Tables list. Double-click on the CUSTID column to open its definition. This brings the column editor. A data type conversion must be applied between DB2 and Java types. This is the role of Converters.

8. Select VapTrimStringConverter as the converter in the Type details group. This converter performs the correct transformation between VARCHAR and other string* to java.lang.String objects with leading and trailing blanks deleted. Your window should look like Figure 86. Click OK. Repeat this process for the FNAME and LNAME columns.

Figure 86. Column Editor
9. Select **Schemas - Save Schema**... Choose the **ITSO Bank** project and the **itso.samples.ejb.CMP** package. Click **Finish**. This creates a new class named `<schemaName>`Schema with a single static method infoString() containing information gathered by the Schema browser (see Figure 87).

![CustomerSchema class](image.png)

Figure 87. CustomerSchema class Created After Saving the Schema

10. Exit the schema browser by closing the window.

**Defining a Map between EJB Beans and the Database Schema**

Once your schema is created, you need to define a mapping between the schema and your enterprise beans fields. VisualAge for Java uses the schema mapping to generate the SQL code necessary for create, read, update and delete operations on the database columns for your CMP bean. From the window menu, select **EJBs->Open To->Schema Maps** to open the Map Browser illustrated in Figure 88.
To create a new map for your Customer schema, you need to:

1. Click the mouse button 2 in the **Datastore Maps** list, then select **New EJB Group Map** from the pop-up menu. (The choice **New datastore Map...** appears if you have added the Persistent Builder feature, but you still need to use the EJB Group Map for EJB).

2. Enter **CMPS** for the map name, this is the name VisualAge for Java will use to save the map defined. Select **ITSOEJBGroup** from the EJB Group choice list and the **Customer** schema, which we created previously. Your window should look like Figure 89. Be sure you selected the proper **EJB Group** and **Schema** because we are not able to change them after. Click **OK**.

![Map Browser Window](image)

Figure 88. Map Browser Window
3. VisualAge for Java displays the newly created **CMPS** Datastore map in the map browser with its associated persistent class **Customer**.

4. A table map define the links between your EJB beans and database tables. To create a new table map, click the mouse button 2 in the **Table Maps** list then select **New Table Map - Add Table Map with no inheritance**. Choose the **CUSTOMER** table from the combo-box (this CUSTOMER table is coming from the table associated with the Customer Database Schema) and click **OK**. Your screen should now look like Figure 90.

![Map Browser Window with Customer Table Map](image)
5. For this table map, you now need to define a property map. With this property map you tell VisualAge for Java what are the database fields the member fields of your EJB bean maps to. Select the CUSTOMER Table Map, since we need to define its properties. Click the mouse button 2 on it and select Edit Property Maps. You see a window like Figure 91.

![Property Map Editor](image)

Figure 91. Property Map Editor

6. The Property Map Editor lets you define, for each of your bean’s members, what is the corresponding field in the database. The **Class Attribute** column lists all of the CMP fields of your Enterprise bean. The **Table Column** lists the fields from the database table. From the **Map Type** column you can choose 3 values:

- **Simple**: The simple mapping type is used to map one member field to one table column. We use this type in our Customer example.
- **Complex**: The complex mapping type is used when you need to map one member field to multiple database columns. The complex mapping is described in “Using Composers” from the EJB Development Environment documentation that comes with the Enterprise Update of VisualAge for Java.
- **Not Mapped**: No mapping has been defined yet for your field.

Select the values according to what is shown in Figure 92 and click OK. You have now a map defined for the CUSTOMER table.
7. Select the **CMPS** Datastore map. Select **Datastore Maps - Save Datastore Map** from the menu. Choose the **ITSO Bank** project and the **itso.samples.ejb.CMP** package. Click **Finish**. This creates a new class named `<mapName>Map` with a single static method `infoString()` containing information gathered by the Map browser (see Figure 93).

8. Exit the schema browser by closing the window.
4.7 Adjusting the Deployment Descriptor Attributes

Since you need to define the deployment descriptor for the Customer bean, you can do this by selecting the bean and then selecting the Properties item from the EJB menu.

In the EJB Workspace:
1. Go to the EJBs list.
2. Select ITSOEJBGroup.
3. Select CustomerBean class.
4. From the menu, select EJBs->Properties.

The Properties window appears, as shown in Figure 94:

![Properties Window]

In the Bean page you can specify the following:

- In the Enter the JNDI name for BeanHome field, type the JNDI name to associate with the EJB bean in the JNDI name space. The
container binds the EJB bean’s home interface with a JNDI name that includes the name you specified. The name you specify in this field is the name that your client code provides when invoking a lookup on the initial naming context.

- In the **Transaction Attribute** drop-down box, select a transaction attribute. This attribute tells the container how to manage transaction scopes before and after the execution of the EJB method. We keep the default, **TX_REQUIRED**. The different transaction attributes are described in the Chapter 7, “Transactions” on page 217.

- In the **Isolation Level** drop-down box, select an isolation level. This level tells the container what isolation level to set on the database connections used by the EJB bean at the start of each transaction. Actually, WebSphere ignores this isolation level. Chapter 7, “Transactions” on page 217 covers this.

- In the **Run-As Mode** drop-down box, select a run-as mode. This selection tells the container the security identity to associate with the execution of the EJB method. To learn more about security, see Chapter 8, “Security” on page 241.

- The **Reentrant** check box is used as follows. By default, an entity bean instance is not re-entrant. If an instance executes a client request in a given transaction context, and another request with the same transaction context arrives at the EJB object, the container throws the java.rmi.RemoteException to the second request. This rules allows the bean developer to program the bean as single-threaded, non-reentrant code.

  Reentrant beans must be programmed and used with great caution. First, the bean programmer must code the bean with the anticipation of a loopback call. Second, since the container cannot, in general, tell a loopback from a concurrent call from a different client, the client programmer must be careful to avoid code that could lead to concurrent call in the same transaction context. Concurrent calls in the same transaction context targeted at the same EJB object are illegal, and may lead to unpredictable results. Since the container cannot, in general, distinguish between an illegal concurrent call and a legal loopback, application programmers are encouraged to avoid using loopbacks. Entity beans that do not need callbacks can be marked as non-reentrant in the deployment descriptor, allowing the container to detect and prevent illegal concurrent calls from clients.

  The Environment and Method pages are explained in Chapter 3, “Developing a session enterprise bean” on page 19.

We are now ready to generate the deployment code for the bean:
4.8 Generating Your Deployed Code

Before deploying your EJB beans you need to generated the deployed code.

1. Select your Customer bean from the EJBs list.
2. Click on the mouse button 2, select Generate - Deployed code from the popup menu. Wait until the code generation is completed.

VisualAge for Java generates all the stubs, skeletons, helper and holder classes which are needed by the RMI-IIOP communication layer. VisualAge for Java generates the finder and persister classes needed by the container for the deployment of your CMP Entity beans. We use the generated EJ S J DBCPersisterCustomerBean to help us creating the needed SQL Query string needed by the special finder method as described in the topic 4.9, “Adding Special Finder Methods” on page 158.

VisualAge for Java comes with a test client facility to help you with the testing of your EJB beans. To generate the test client for your Customer entity bean:

1. Select your Customer bean from the EJBs list.
2. Click on the mouse button 2, select Generate - Test Client from the popup menu. Wait until the code generation is completed.

4.9 Adding Special Finder Methods

A special finder method is a finder method different from the findByPrimaryKey. VisualAge for Java generates all the SQL code necessary to query the database for your special finder methods. To create a finder method to query the Customer bean on its lastName field, we need to do two things:

1. Insert a finder method declaration in the CustomerHome interface.
2. Create a query string in the CustomerBeanFinderHelper interface.

Insert a Finder Method Declaration in the CustomerHome Interface

Since we generated the deployed code, VisualAge for Java displays all the generated methods in the Types list. To hide the deployed generated methods, click on the Show Generated Types icon which is beside the Field icon in Types list header. We want to add a finder method querying
Developing a Container-Managed Persistence Bean

the Customer bean on its lastName field. To add the declaration of this finder in the CustomerHome interface:

1. Select the CustomerHome interface from the Types list.
2. Inside the Methods pane, click mouse button 2 and select Add->Method....
3. In the Create Method smart guide, enter the following declaration in the text field:
   
   public java.util.Enumeration findByLastName(java.lang.String param)

   and press Next.
4. From the Attributes window press Add... button. In the Exceptions window, in the Pattern (# = any character, * = any string) text field enter RemoteException, select it from the Type Names list and click Add. Repeat the same actions but in the Pattern (# = any character, * = any string) text field, enter FinderException, select it from the Type Names list, click Add and Close buttons. This takes you back to the Attributes window, where you can check that java.rmi.RemoteException and javax.ejb.FinderException have been added. Press Finish.
5. Save your changes.
6. The findByLastName(String) method appears in the Methods list of the CustomerHome interface.

Create a Query String in the CustomerFinderHelper Interface

We want a finder method to query the Customer Bean on its lastName field. For the finder declaration we defined in the CustomerHome interface, we need to insert a corresponding query string in the CustomerBeanFinderHelper interface.

The query string must specify the correct table name and the correct column names in the correct order. The easiest way to do this is to copy and modify the generated select statement from the persister class.

The persister class for the Customer bean is named EJSJDBCPersisterCustomer. The EJSJDBCPersisterCustomer class contain a static String constant called findByKeySqlString. Construct your query string by copying this select statement and modifying the where clause.

1. To show the deployed generated methods. Click on the Show Generated Types icon ⌚ which is beside the Field icon in Types list header. The list of generated classes appear and you can see that three classes _CustomerHomeSkeleton, _CustomerHomeStub and
EJSCustomerHome) are marked with a cross indicating an error due to the addition of a new method in the home interface. Do not worry, the next generation of deployment code done by VisualAge for Java will fix these errors.

2. Select the **EJSJ DBCPersisterCustomerBean** from the **Types** list.

3. Find the findByKeySqlString static member from the **EJSJ DBCPersisterCustomerBean** class. Copy the SQL Select statement.

4. To hide the deployed generated methods, click on the **Show Generated Types** icon which is beside the **Field** icon in **Types** list header.

5. Select the **CustomerBeanFinderHelper** interface from the **Types** list.

6. In the Source pane, paste the SQL statement from the clipboard inside the body of the method, change the method name to **findByLastName**:

```java
public static final String findByLastName = "SELECT T1.CUSTID, T1.FNAME, T1.LNAME FROM CMPS.CUSTOMER T1 WHERE (T1.CUSTID = ?) AND (T1.LASTNAME = ?) ";
```

7. Modify the where clause from the SQL String. We want a filter on the **LASTNAME** only. You code should look like:

```java
public static final String findByLastName = "SELECT T1.CUSTID, T1.FNAME, T1.LNAME FROM CMPS.CUSTOMER T1 WHERE T1.LASTNAME = ? ";
```

8. Save your interface.

9. Select the **Customer** EJB bean in the EJBs list and click the mouse button 2. Select **Generate - Deployed code**.

If you recreate a Schema map later, you will first have to regenerate the code and then copy the query string from the EJSJ DBCPersisterYourBeanName class.

The query string is referenced by the finder implementation class generated by the deployment tool. You need to follow a few guidelines when you create special finder methods.

- The name part of nameQueryString must be the name of the method.
- The number of question marks in your nameQueryString must match the number of parameters in the Finder method. When you later generate the deployment classes, the question marks in the nameQueryString are automatically replaced with the parameters defined in the Finder method.
If you use the top-down approach for creating your database schemas and maps, you will not find the `findByKeySqlString` static member, you have to look for the `loadString` static member in your EJ SJ DBCPersisteryourName class. Copy and modify the SQL select from this member.

### 4.10 Testing Your Bean

To test our Customer entity bean, we need two things:

1. The generated deployed code.
2. The generated test client.

We already generated these classes. To run the test client, you need to go in the server configuration window (see Figure 95):

1. Select the **Customer** bean in the **EJB**s list.
2. Select **EJBs - Open To - Server Configuration** from the menu.
3. Start the **Location Service Daemon**.
4. Start the **Persistent Name Server**.
5. Select the **EJB Server (server1)** from the **Servers** list. Click on the mouse button 2 and select **Start server**.
6. Switch to the VisualAge for Java Console to check if your server is started.
7. Go back to the EJB Server Configuration window. In the EJBs pane expand ITSOEJ BGroup by clicking on the "+" sign beside the **ITSOEJ BGroup** (see Figure 96).
8. Select the **Customer** EJB. Click mouse button 2 and select **Run Test Client**. You can also select Customer and click on the yellow icon 🟢. Wait to see a window like Figure 97.
9. Click on **Connect**. VisualAge for Java tries to locate the Customer EJB bean using the JNDI. When the object is located, VisualAge for Java displays the home interface of the Customer EJB bean; see Figure 98.
10. We want to find the Customer with the id equal to "104 " and whose last name is "Picon". Click on the `findByPrimaryKey(CustomerKey)` from the **Methods** list.

11. We need to create the CustomerKey object to pass as parameter. Click on **New**... To see how to create the CustomerKey, refer to Figure 99. Select the `new CustomerKey(String, String)` constructor, since we need to specify the key members value. Enter 104 as the parameter[0] value and Picon as the parameter[1] value. Click **Send** and **Done**.
12. You are now back to the Customer home interface. Click on the **Send** button to execute the **findByPrimaryKey**. After a while, VisualAge for Java returns to you with the remote interface of the Customer EJB; see Figure 100.
13. Select the `getFirstName()` method, then click the **Send** button. VisualAge for Java returns to you with the first name of the Customer EJB previously returned by the `findByPrimaryKey`; see Figure 101.

![Figure 101. Customer getName() Output](image)

14. We now want to add a new instance of Customer EJB bean. From the top of the test client Customer window, select in the Page combo box **Home interface**. Click on the **Create(CustomerKey)** method click on **New**.... Select the **new CustomerKey(String, String)** constructor and enter 999 in the [0] (String) parameter text field as the first parameter value and Dessureault in the [1] (String) text field as the second parameter value. Click **Send** and **Done**. (Refer to Figure 102).
15. Click the **Send** button from the home interface.

16. The test client does not allow you to work with the newly created instance. So, you need to Exit and restart the Test client again.

17. After the Test client is opened, press **Connect**.

18. In the Methods list of the **Remote interface**, select the **findByLastName(String)** method, enter Dessureault as the parameter value, and click on the **Send** button; see Figure 103.
19. The Test client Customer VisualAge for Java returns to the Remote Interface and an enumeration containing our Customer bean. You can now select `getLastName()` method and press `Send`. As a result you see (String) Dessureault.

20. Click on the Exit button to quit the Test client. Go to the EJB Server Configuration window and stop the Location Service Daemon, the Persistent Name Server and the EJB Server (server1).
5 Developing a Bean-Managed Persistence Bean

In this chapter you will learn how to build an Entity Bean with self-managed persistence.

5.1 Bean-Managed Persistence Entity Bean

Bean-managed persistence (BMP) implies that the task of persisting the entity is performed by the bean itself. This contrasts with container-managed persistence (CMP), where the container generates the persistence code for a deployed entity bean.

BMP has the advantage of having a greater degree of granularity, but it has the disadvantage of being targeted for only a single type of data store and hence results in a loss of reuse. The Bean developer has to develop the business logic as well as the logic to persist the bean to a data store.
You may decide to go in for this approach in the following situations:

- If the persistence store for the bean is a legacy application, a proprietary data store, or a database which uses non-standard (non-JDBC) drivers.
- If you want to use a data store which is not usually supported by Enterprise JavaBean containers. Most containers use databases as persistence store. If you prefer storage mechanisms like flat/XML files which are supported by only a few containers, it is best to use BMP.
- If you are paranoid about the code generated by the container.

5.2 The TransactionRecord Entity Bean

In this chapter we will develop a BMP TransactionRecord entity bean. The TransactionRecord represents a transaction involving a customer and a bank through his/her bank account. The TransactionRecord bean stores the following information about a particular transaction:

- transId (java.sql.Timestamp): A unique Transaction ID
- accountId (java.lang.String): The Account ID of the customer
- amount (java.math.BigDecimal): The amount transacted (a negative value for a debit and a positive one for a credit).

Creating the Entity Bean

Before continuing, ensure that you have added the IBM EJB Development Environment 1.0 feature to VisualAge for Java. To create the TransactionRecord bean:

1. In the EJBs pane, select an EJB group to contain your EJB bean, and click mouse button 2.
2. From the pop-up menu, choose Add - EJ B. The Create EJB SmartGuide appears. In the Bean name field, type in the name of the new EJB bean (TransactionRecord), as shown in Figure 104.
Figure 104. The Create EJB SmartGuide

3. From the **Bean type** drop-down list, select the **Entity bean with Bean-managed persistence fields (BMP)** item to indicate that the bean is a BMP entity bean.
4. In the **Project** field, type the project name, modify the default name, or
dick the **Browse** button and select the project that contains (or will
contain) the EJB bean class from the list of projects, then click **OK**. By
default, the project to which the EJB group belongs is displayed.

5. Click on **Finish** to create the bean. The names for the EJB home
interface, remote interface, and key class (for entity EJB beans only) will
serve as the default names.

---

### Adding and Defining BMP Fields

When you create an EJB bean in the EJB Development Environment, certain
fields are created automatically in the bean class. This topic explains how to
add the bean-specific fields to the TransactionRecord EJB bean, namely:

- `transId` (**java.sql.Timestamp**)
- `amount` (**java.math.BigDecimal**)
- `accountId` (**java.lang.String**)

These fields are declared as public and are accessible via public getter
methods.

To add a field to the TransactionRecord bean:

1. In the **Types** pane of the EJB page, select the TransactionRecord bean
class.

2. Click on the **Add Field** button to launch the Create Field SmartGuide,
shown in Figure 105.
3. Fill in the field name, type name and select the **Access with getter and setter methods**.

4. Select the **public** radio button for Getter and **private** for Setter.

5. Click **Finish** to generate the field.
Understanding the Generated Methods

When you create an EJB bean in the EJB Development Environment, a set of methods for the bean is automatically created. If you select the TransactionRecord bean in the EJBs pane, the Java methods of the bean implementation class are displayed in the methods pane (see Figure 106).

VisualAge for Java automatically creates the following methods, in addition to the getter and setter methods for the fields we defined, for a BMP entity bean:

- public void ejbActivate();
  This life-cycle callback method of the entity bean is invoked when the container assigns an instance from the pool to a specific EJB object identity. This signals the ready state for the enterprise bean. You can use this method to load additional resources in the bean.

- public TransactionRecordKey ejbCreate(TransactionRecordKey key);
  VisualAge for Java generates a default ejbCreate(...) method which takes the EJB Key class, TransactionRecordKey, as a parameter. This signature matches the create(...) method of the bean’s home interface.
You should use this method to:

- Validate the passed arguments
- Initialize the instance's variables
- Create an entry representing the entity in the persistence store

Finally, you should return the primary key for the entity at the end of this method.

- public TransactionRecordKey ejbFindByPrimaryKey (TransactionRecordKey key);

VisualAge for Java creates this default finder method, which uses a TransactionRecordKey object to identify a particular entity. You should do the following:

- Validate the client-passed arguments
- Locate the entity in the persistence store
- Refresh the state of the instance from the state in the persistence store
- Return the primary key at the end of the method

- public void ejbLoad();

This life-cycle call-back method is invoked on an instance in order to refresh its state. You should implement the logic to read the entity state from the persistent store and synchronize the instance variables accordingly.

- public void ejbPassivate();

The EJB container invokes this method when it decides to send the instance to the pooled state. In the pooled state, the instance is disassociated from its EJB object identity. You can use this method to release any resources that were allocated in the ejbActivate() method.

- public void setEntityContext(EntityContext ctx);

The EJB container uses this method to pass a EntityContext reference to the entity bean. VisualAge for Java generates the code for storing the entity context in an instance variable.

- public void ejbPostCreate(TransactionRecordKey key);

This life-cycle call-back method indicates that the EJB Object identity is available. For each ejbCreate(...) method, there is a matching ejbPostCreate(...) method with the same input parameters.

- public void ejbRemove();

Developing a Bean-Managed Persistence Bean  177
This method is invoked by the container when the client invokes the remove method. This ends the life of an EJB object. You should implement the logic to remove the entity from the persistence store.

- public void ejbStore();
  The container invokes this method on the instance when it decides to synchronize the instance's state in the persistence store with the current state. You should implement the logic to update the instance's state in the persistence store.

- public void unsetEntityContext(EntityContext ctx);
  The container invokes this method when it wants to reduce the size of the instance pool.

### Modifying the Key Class

The TransactionRecordKey class generated by VisualAge for Java uses a String field to represent the primary key. Change this field's type to java.sql.Timestamp to match the transId field in the TransactionRecord bean. This will also involve modifying the constructor of this class to accept a Timestamp object as a parameter.

It is always advantageous to use a key class that wraps the actual primary key field, because if you later change the primary key field for the bean class, you will not have to regenerate the home and remote methods.

### Adding Remote Methods

The remote interface describes the entity bean's business methods, which are accessible by a client. In this section we will add four methods to the remote interface, namely:

- public java.sql.Timestamp getTransId();
  Returns the transaction ID for the TransactionRecord.

- public String getAccountId();
  Returns the account ID of the customer of the transaction.

- public java.math.BigDecimal getAmount();
  Returns the amount transacted in the transaction. The amount may be negative if the transaction is a credit, positive for a debit, and zero for a checking operation.
public String getTransType();

Returns the type of transaction- "debit", "credit" or "checking" depending on the amount transacted.

Modifying and Adding Home Methods

VisualAge for Java creates two home methods for the TransactionRecord bean:

- A create(...) method which takes the key as input
- A findByPrimaryKey(...) method which takes the key as input

In this section, we will modify the create method to accept the customer’s account ID and the amount transacted.

1. Select the ejbCreate(...) method in the Methods pane of the EJB page. The source of this method is displayed in the Source pane.
2. Modify the method in the Source pane as below:
   - Alter the argument list of the method to include a BigDecimal argument (to pass the amount) and a String argument (to pass the account ID).
   - Perform any validation or null-check on the passed arguments in the body of the method.
   - Set the amount and accountId fields of the bean to the respective arguments.
   - Set the transId to a Timestamp representing the current time.
   - Return a TransactionRecordKey wrapping the transId.
3. Save the changes.
4. Add the method to the bean’s home interface.

Next, we add a finder method (ejbFindTransactionSummary) to the TransactionRecord bean which returns an enumeration of the last 10 transactions made by a particular customer.

1. Select TransactionRecord in the Types pane.
2. Click on the Add method button. This launches the Create Method SmartGuide.
3. Specify the signature for the new public method:
   - Set the name of the method as ejbFindTransactionSummary
   - Set the return type as java.util.Enumeration
• Specify a String input argument, accountId, which identifies the customer.
• Specify the exceptions that this method may throw: javax.ejb.FinderException and java.rmi.RemoteException.

4. Click on the Finish button in the Create Method SmartGuide to generate the method.

5. Add the method to the bean’s home interface.

We have not yet provided any logic to manage the persistence store. In the remaining portion of the chapter we describe how an entity bean can persist itself to an XML file. We start with a brief introduction to XML and developing XML based applications with Java. You can skip these portions if you know the subject already, and go directly to Chapter 5.4, “Persisting to an XML File” on page 181.

5.3 XML—The eXtensible Markup Language

Introduction to XML

Extensible Markup Language (XML) is a framework for defining document markup languages. In simple terms, a document markup language is a set of elements (frequently called tags) that have one or more of the following functions:

❑ Describe the structure of the document
❑ Describe the content of the document
❑ Control how the document is presented to the user

HTML is the most widely used markup language for Web-based documents. As the popularity of HTML increases, the limitations of the language have become more apparent. Those limitations include restricting the user to a relatively small set of tags. HTML authors cannot create their own HTML tags, because commercially available Web browsers have no knowledge of tags that are not part of the HTML standards that the browsers support.

In the XML document, the tag names convey the meaning of the data they contain. The structure of the document is easily discerned and follows a pattern. In contrast, the HTML tag names reveal little about the meaning of their content, and the structure is not particularly useful for manipulating the document and exchanging it between applications.
The Document Object Model—DOM

You can easily create, navigate, manipulate and modify XML documents in your java applications by using the XML4J parser that is included in the EJB Development feature of VisualAge for Java. The parser includes the Document Object Model (DOM) API for document operations.

DOM is a language-independent API for XML and HTML documents. The API represents XML and HTML documents as objects that can be accessed by object-oriented programs (such as Web browsers, document search engines, conversion tools, business logic, and scripting languages). By using the DOM, these programs can create, navigate, manipulate, and modify the documents.

The DOM can be used to represent an existing XML document or generate an XML document. The document is stored in computer memory. In the DOM, a document consists of a collection of Nodes that have parent/child relationships. The Node is the primary object and can be of different types (such as Document, Element, Attribute, Text, Processing Instruction, CDATA Section, and Comment).

The DOM Java classes are included in the XML4J parser. Point your browser to http://www.alphaWorks.ibm.com/formula/XML to download the XML4J documentation and http://www.w3c.org/DOM/ to learn more about DOM.

5.4 Persisting to an XML File

You should consider using XML as a persistence store when you expect a large set of low-use data, like logs of user interactions. Storing such information in a database may be an overkill because the data is used rather infrequently. XML provides a very simple storage option without the use of any database. You will, however, have to implement the bean's persistence and the logic to handle concurrent access yourself.

An entity bean represents content in a XML Document. In the XML context for a document of information, there is one bean for each child element of the document's root element. The bean acts upon the XML file using its DOM. For every entity bean there is a node in the DOM which represents the element in the XML file.
Figure 107. Mapping Entity Beans to an XML File

Figure 107 illustrates the mapping between the entity bean and the XML file. Every instance of the TransactionRecord bean corresponds to a unique transaction element. All the transaction elements are nested within a root document element named transactionDB. The attribute transId and the sub-elements, amount and accountId, within the transaction element represent the fields of the TransactionRecord bean.

The choice of XML as the persistence store for the bean stems from the fact that the number of TransactionRecords will be large and that this data will be used infrequently as it is only reference data.

Now we will illustrate the steps involved in creating the XML file to store the entity bean's instances.

Organizing the XML File

The first step involved in persisting the TransactionRecord entity bean to an XML file is to create its structure. This involves describing the following:

- The hierarchy of elements
- The attributes of its elements
- Mapping between the fields of the bean and content/markup
The map is in the form of a Document Type Definition (DTD). The DTD represents the metadata of the XML file. Creating the Document Type Definition is not important, but it will help describe the mapping process between the entity bean (TransactionRecord) and the XML file.

```
<!DOCTYPE transactionDB [ 
  <!ELEMENT transactionDB (transaction)*> 
  <!ELEMENT transaction (amount, accountId)> 
  <!ATTLIST transaction transId ID #REQUIRED> 
  <!ELEMENT amount (#PCDATA)> 
  <!ATTLIST amount class CDATA "java.math.BigDecimal"> 
  <!ELEMENT accountId (#PCDATA)> 
  <!ATTLIST accountId class CDATA "java.lang.String"> 
]> 
```

The DTD describes the hierarchy of the elements that will exist in the XML file. It has a root element `transactionDB`. Nested within the `transactionDB` element are zero or more `transaction` elements. Every instance of the `TransactionRecord` bean is represented as a `transaction` element. The `transaction` element has an attribute `transId` and two sub-elements, `amount` and `accountId`.

The fields of the `TransactionRecord` are mapped as follows:

- The `transId` field is represented as an ID attribute of the `transaction` element
- The `amount` field is stored as text within an `amount` element, which is a child of the `transaction` element
- The `accountId` field is stored as text within an `accountId` element, which is a child of the `transaction` element

You could alternatively map the `transId` field as text within a sub-element like the other fields. We use the earlier approach due to the inherent benefit of using an ID attribute in an XML file—it does not allow duplicate attributes.
Accessing the DOM

Figure 109. The Document Object Model

Figure 109 shows the DOM for a TransactionRecord Log. The Element node (folder) is the root of the tree. Each of the Element nodes (transactionDB, transaction, amount and accountId) is a tag in the document. The text (data) within each tag is a Text node, represented in the figure as rectangular nodes. Attribute nodes (dashed ovals) represent attributes within tags, like the transId of the transaction.

Table 2 on page 185 and Table 3 on page 186 show a summary of the DOM operations. You can refer to W3C’s DOM specification to get more information about the interfaces to access the DOM objects.

All the classes used in the descriptions below have the package name org.w3c.dom unless explicitly stated.
Table 2. Creating the Different DOM Nodes in the Hierarchy

<table>
<thead>
<tr>
<th><strong>A DOM Document</strong></th>
<th>Document doc = new com.ibm.xml.parser.TXDocument();</th>
</tr>
</thead>
<tbody>
<tr>
<td>Root element <strong>transactionDB</strong> for the Document</td>
<td>Element root = doc.createElement(&quot;transactionDB&quot;); doc.appendChild(root);</td>
</tr>
<tr>
<td><strong>A transaction</strong> element within the <strong>transactionDB</strong> element</td>
<td>Element transElement = doc.createElement(&quot;transaction&quot;); root.appendChild(transElement);</td>
</tr>
<tr>
<td>**A <strong>transId</strong> Attribute for the <strong>transaction</strong> element</td>
<td>transElement.setAttribute(&quot;transId&quot;, &quot;1999-02-...&quot;);</td>
</tr>
<tr>
<td><strong>An amount</strong> Node for the <strong>transaction</strong> element</td>
<td>Element amountElement = doc.createElement(&quot;amount&quot;); transElement.appendChild(amountElement);</td>
</tr>
<tr>
<td>Setting the amount data, 666.6, for the <strong>amount</strong> node</td>
<td>amountElement.appendChild(doc.createTextNode(&quot;666.6&quot;));</td>
</tr>
</tbody>
</table>

The **accountId** node can be created just like the **amount** node.

The creation of the DTD has not been described because there is no standard API for constructing the DTD nodes. Moreover, the DTD only validates the XML document that is enforced by the Persister class.
Table 3. Accessing the Different DOM Nodes

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Root element transactionDB</td>
<td>Element root = doc.getRootElement()</td>
</tr>
<tr>
<td>The transaction element whose transId is '1999-02-...':</td>
<td>NodeList nl=root.getElementsByTagName(&quot;transaction&quot;) for (int i = 0; i &lt; nl.getLength(); i++) { Element transEl = (Element) nl.item(i); if (transEl.getAttribute(&quot;transId&quot;).equals(&quot;1999-02-...&quot;)) { // this is the node we are looking for</td>
</tr>
<tr>
<td>The transId Attribute of the element</td>
<td>transElement.getAttribute(&quot;transId&quot;)</td>
</tr>
<tr>
<td>The amount Node of the transaction element</td>
<td>Element amountElement = (Element) transElement.getElementsByTagName(&quot;amount&quot;).item(0)</td>
</tr>
<tr>
<td>The amount data of the amount node</td>
<td>String amount = ((Text) amountElement.getFirstChild()).getData();</td>
</tr>
<tr>
<td>Removing a transaction element from the hierarchy</td>
<td>transElement.getParentNode().removeChild(transElement);</td>
</tr>
</tbody>
</table>

Creating the Persister Class

We now need to persist the entity bean's instances to an XML file. To accomplish this we define a helper class, TransactionRecordXMLPersister, to manage the bean's persistence. The Persistence class reads and writes an entity bean's state from the XML document described in the earlier section. The Persister class is designed as a **Singleton** class to allow all the instances of the entity bean to use the same XML document.
Fields
The TransactionRecordXMLPersister class contains the following significant fields:

- public String xmlLocation;
  - Contains the location of the XML file. This is initially null and is set to a location specified as an environment property in the deployment descriptor of the TransactionRecord bean.
- private static TransactionRecordXMLPersister persister;
  - The singleton TransactionRecordXMLPersister object.
- private Document xmlDocument;
  - The Document node of the XML file's Document Object Model. Each entity bean is contained as a unique node within the document element.

Methods
In keeping with the Singleton design pattern which the Persister class uses, we define a private no-arg constructor for the class and a public static method, getPersister(), which returns the singleton persister object.

We need to define the following book-keeping methods for this class:

- private void loadDocument();
  - This method loads the XML file specified by xmlLocation into the Document Object Model. If the specified file does not exist, a new XML file is created using the XML4J parser API.
- private void saveDocument();
  - Saves the DOM Document, xmlDocument, into the file specified by xmlLocation.

The Persister class addresses the different stages of the entity bean's lifecycle:

- Creation
  - An instance of the entity bean is created when its ejbCreate(...) method is invoked via its home interface. Within this method the Entity bean initializes the fields and invokes the create method on the Persister, passing itself as an argument. The Persister then creates a sub-tree representing the state of the Entity and appends the root element of the sub-tree to the document element.
Update

The state of the entity bean is updated in the XML file when the container invokes the `ejbStore()` method on it. Within this method, the entity bean invokes the update method on the XML file passing itself as an argument. The Persister searches for the element corresponding to the primary key of the passed entity bean and updates the sub-tree accordingly.

Refresh

The state of the entity bean is refreshed when the container invokes the `ejbLoad()` method on the bean. In this method, the entity bean invokes the refresh method on the Persister, passing itself and a primary key field as arguments. The Persister searches for the element containing the primary key field and updates the fields of the passed bean from the sub-tree of the element.

Deletion

The instance of a entity bean is destroyed when its `ejbRemove()` method is invoked via its home interface. Within this method, the entity bean invokes the remove method on the Persister, passing itself as an argument. The Persister searches for the element specified by the primary key field of the passed entity bean and deletes this node from its parent.

We include an additional method in the Persister class, `getTransactionSummary`, which returns an enumeration of IDs of the last 10 transactions made by a particular customer. The Persister traverses through all the transaction elements and checks if the accountID of the transaction matches the customer’s account ID.

The `xmlLocation` field is set in the `setEntityContext(...)` method of the `TransactionRecord` bean. Its value is specified by the value of the `transaction.file` property in the context’s environment.

5.5 Optimizing the Bean

The container invokes the `ejbLoad()` and `ejbStore()` method on the entity bean before and after every transaction on the bean. This is done in order to keep the state of the entity bean in sync with its state in the persistence store. This leads to reads/writes from/to the persistence store even when the bean’s state has not been modified.
To reduce the number of reads/writes we define a private, transient, boolean field called isModified. This flag determines whether the bean's state has been modified since the last write operation. Within the business logic, you should set the isModified flag to true only when the bean's state has been changed. Also, in the ejbLoad() and ejbStore() methods, you should:

1. Check whether the bean's state has been modified (check if isModified is true).
2. Perform the necessary read/writes from the persistence store.
3. Set the isModified flag to false.

Now, when a remote method is invoked on the bean, the container calls the ejbLoad and the ejbStore methods before and after the method which first checks whether a read/write to the persistence store is required.

The isModified flag must be set to true in the following methods:
- ejbActivate
- ejbPassivate
- setEntityContext
- ejbRemove
- Any business method which alters the bean's state

You should unset the isModified flag (set it to false) at the end of the ejbLoad, ejbStore and ejbPostCreate methods.

### 5.6 Specifying Environment Properties

Before you deploy the TransactionRecord bean, you should make an entry of the transaction.file property in the deployment descriptor. This property specifies the location of the XML file where the states of the bean will be saved and read from. Refer to Figure 110.

You can set this property by following these steps:

1. Select the TransactionRecord bean in the EJBs pane.
2. Click on the Properties menu-item of the pop-up menu. This launches the Properties dialog.
3. In the Environment page of the Properties dialog, set the transaction.file property along with the location of the XML file.
4. Click on **OK** to save the changes.

## 5.7 Transaction Management of the XML BMP

The Entity Bean, `TransactionRecord`, performs transaction management by itself. This Bean uses the auto-commit feature, that is, any change in the state of the bean is reflected immediately in the XML data store. Entity bean create and remove cannot be rolled back.

To fit into the EJB model of having the container manage the bean’s transaction, the `TransactionRecordXMLPersister` should be made transaction-aware. The persister class should be modified to register with the executing thread's current transaction and persist the bean's state only on a commit.
The entity bean cannot use the SessionSynchronization interface to be notified about transaction states like afterBegin, beforeCompletion and afterCompletion. The EJB 1.0 Specification has restricted the usage of this interface only to session beans, and although Websphere allows its use in entity beans this approach is non-standard and may not work on other servers.

5.8 Testing Your Bean

Now we can test the bean by executing the following short scenario:

- Create a first record (amount=1001, accountId=joaquin).
- Check that the file c:\temp\TransactionRecordLog.xml has been created and contains the right information. (The location of the file can be configured in the Environment Properties see Figure 110 on page 190.)
- Create a second record (amount=1002, accountId=maneesh).
- Check that the xml file TransactionRecordLog.xml has been updated and now contains two records with the right information.
- Find the second record by using its time stamp.

We do not repeat here how to start the servers to run your beans; refer to “Testing Your Bean” on page 161 if you need to refresh your memory. Once the server is started, we can start testing the TransactionRecord bean.

1. From the EJBs pane, expand ITSOGroup and select TransactionRecord.
2. Click mouse button 2, and choose Run Test Client (see Figure 111).
3. After the test client has started, when you are presented with the first window, just click on Connect. The result is the Home Interface.

4. Select `create(BigDecimal, String)` and on the Parameters listbox, click on New...

5. In the BigDecimal dialog box, select `new BigDecimal(String)` and enter in the Parameters text field the value 1001 press Send and Done (see Figure 112).
Figure 112. Create Amount
6. In the Parameters listbox, enter joaquin in the textfield (see Figure 113)

![TransactionRecord](image)

Figure 113. Create a Record: Specify Accountid

and press the **Send** button. As a result of this action, the bean instance is created and the Test client Page is now initialized with the Remote interface of the bean. All the methods declared on the remote interface become accessible from the method listbox (see Figure 114).
Figure 114. TransactionRecord Remote Interface

If you go to the c:\temp drive, you will find a new file called TransactionRecordLog.xml. Open it and check that a new transaction record has been created, as indicated in Figure 115.
7. Repeat steps 4 to 7 to create a new transaction record with the following values:
   • amount = 1002
   • accountId = maneesh

Do this as shown in Figure 116 on page 197 and Figure 117 on page 198.
Figure 116. Initialize Amount

8. After the two parameters have been initialized, press **Send** to create the second record (see Figure 117).
Figure 117. Create a Second Record
9. If you open again the c:\temp\transactionRecordLog.xml file, you can check that a second transaction record has been created (see Figure 118).

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE transactionDB [  
  <!ELEMENT transactionDB (transaction)>  
  <!ELEMENT transaction (amount,accountId)>  
  <!ATTLIST transaction transId ID #REQUIRED>  
  <!ELEMENT amount (#PCDATA)>  
  <!ATTLIST amount class CDATA #FIXED "java.math.BigDecimal">  
  <!ELEMENT accountId (#PCDATA)>  
  <!ATTLIST accountId class CDATA #FIXED "java.lang.String"> ]>
<transactionDB>
  <transaction transId="1999-04-29 14:02:04.555">
    <amount>1001</amount>
    <accountId>jcaquin</accountId>
  </transaction>
  <transaction transId="1999-04-29 14:05:05.319">
    <amount>1002</amount>
    <accountId>xeneek</accountId>
  </transaction>
</transactionDB>
```

Figure 118. TransactionRecordLog file Second Record

10. Close the Test client by pressing **Exit** and restart it again. (We have noticed a problem when working with two different bean instances created by the Test client if we do not restart it.)

11. Once you reach the Home Interface, select `findByPrimaryKey(TransactionRecordKey)`. In the Parameters listbox, click on **New** (see Figure 119).

12. In the TransactionRecordKey dialog box, select **new TransactionRecordKey(String)** and in the Parameters text field, copy from the TransactionRecordLog.xml file the timestamp corresponding to the second record, press **Send** and **Done**.

13. This brings you back to the TransactionRecord window, where you press the **Send** button to trigger the `findByPrimarykey` method. The bean instance retrieves the corresponding record from the file.
14. The bean's Remote interface is returned, select `getAccountId()` and press Send button (see Figure 120). In the Result label field, you should see (String) maneesh.
Figure 120. Retrieve Bean Information
A fundamental facility in any organized system is the means by which objects are found, given their names. This is the role provided by a Naming Service.

Many naming services are extended with a directory service. While a naming service allows you to look up an object, given its name, a directory service also allows such objects to have attributes. Therefore, in addition to lookup, you can also get an object’s attributes or search for objects, given their attributes.

Java Naming and Directory Interface (JNDI) is an API specified in the Java programming language that provides directory and naming functionality to Java applications. It is defined to be independent of any specific directory service implementation, giving access to new and existing directories through a common way.

The JNDI Service Provider Interface (SPI) provides the means by which different directories can be made accessible from applications using JNDI.

Figure 121 shows how the different components relate to each other.
By using the JNDI API, an application can get access to an initial context. From this initial context it can navigate through a naming tree composed of sub-contexts to which are attached atomic names bound to object references.

This naming tree can represent a complex system, including:

- **Naming system**
  - This is a connected set of contexts of the same type.

- **Name space**
  - This is a set of all names in a naming system.

- **Composite name**
  - This is a name that spans multiple naming systems.

As a result, a composite name can be made up of different parts belonging to different name spaces: DNS and a file system name space, for example.

The JNDI is divided into three packages:

- **javax.naming**
  - This package contains classes and interfaces for accessing naming services.

- **javax.naming.directory**
  - This package extends the javax.naming package to provide functionality for accessing directory services in addition to naming services. This package allows applications to retrieve attributes associated with objects.
Naming Services

stored in the directory and to search for objects using specified attributes. WebSphere Advanced Edition 2.0 supports naming only.

- javax.naming.spi
  This package provides the means by which naming/directory service providers can develop and hook up their implementations so that the corresponding services are accessible from applications that use the JNDI.

From this high level view of what a JNDI implementation can support, let us see what has been made available to us in WebSphere Advanced Edition 2.0.

6.1 System View

From a system level point of view, WebSphere Advanced Edition 2.0 is limited to a configuration of a single host machine running all the Enterprise Java Server processes (see Figure 122).

Figure 122. WebSphere Advanced Edition 2.0: System View
Even though different Enterprise JavaBeans servers could be running in different hosts, it is recommended to keep all EJB servers and the Persistent Name Server in the same host.

Enterprise JavaBeans servers register EJB beans with one and only one Persistent Name Server. If more than one Persistent Name Server exist in the network, each one maintains an independent name space.

### 6.2 Naming Service Components

VisualAge for Java 2.1 is bundled with the WebSphere Advanced Edition 2.0 runtime. The JNDI implementation is based on the CosNaming name service.

There are three basic components involved:
- Location Service Daemon
- Persistent Name Server
- EJB Server

**Location Service Daemon**

Any client using JNDI connects implicitly to the Location Server Daemon after an initial context is requested. The Location Server Daemon is listening on a well known port number (9000) waiting for requests.

When an object request is issued, it is forwarded to the Location Server Daemon first, which retrieves the host in which the object implementation is running and returns to the client ORB a new IOR pointing to the host running the EJB implementation.

**Persistent Name Server**

At startup, the Location Server Daemon hostname and port number are passed to the Persistent Name Server. The Persistent Name Server uses this information to inform the Location Server Daemon about its existence.

The Persistent Name Server uses the native file system to maintain the naming tree information in a persistent store. A local copy is brought into memory at startup and it is this cached information that is used during resolve() or list() operations. All other actions on the naming tree are performed against the cache and the persistent store in an atomic operation.
EJB Server

Enterprise JavaBeans are deployed and executed in an EJB server. When started, an EJB server automatically registers EJB Home objects into the naming tree. To be able to perform this operation, an EJB server is given at startup the Location Server Daemon's hostname and port number. In this way it can retrieve the naming tree's root initial context and bind EJB home objects with their JNDI names into the name tree.

6.3 Using and Configuring Naming Service

When running WebSphere Advanced Edition 2.0 inside VisualAge for Java, interactions with the naming service are limited to:

- Client looking for an object
- Changing the Persistent Name Server and EJB server's host name for EJB client running on a different machine.

Client Looking for an Object

Locating an enterprise bean's home interface is a two-step process. First, you create a javax.naming.InitialContext object. Then, you use the InitialContext object to get the EJBHome object.

Figure 123 shows the code required to create the InitialContext object. To create this object, you construct a java.util.Properties object, assign environment values to the Properties object, and then pass the object as the argument to the InitialContext constructor.
public JpPing create() {

JpPingHome pingHome = null;
javax.naming.InitialContext initContext = null;

// Get the initial context
try {
    String ProviderURL = "iiop://" + EJSHostName + ":" + EJSPort;
    java.util.Properties properties = new java.util.Properties();

    properties.put(javax.naming.Context.PROVIDER_URL, ProviderURL);
    // IBM name services
    properties.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
                   "com.ibm.jndi.CosNaming.CNInitialContextFactory");
    initContext = new javax.naming.InitialContext(properties);
}
  }
}
// endtry

Figure 123. Creating the InitialContext Object

In the ProviderURL string, EJSHostName can be set with two possible values:
  - "/" indicating that the client and the EJB are located in the same machine, or
  - "anyServerName.ibm.com" which is the fully qualified host name

EJSPort may take different values:
  - "900" if the EJB server is running inside VisualAge for Java. This value cannot be changed.
  - "9019" if the EJB server is running inside WebSphere Advanced Edition 2.0 runtime. This value is configurable from the WebSphere Advanced Edition 2.0 Application Server Administration by selecting in the left pane WebSphere Application Server->Enterprise Java Services->Global Settings and changing the Boot Port value as shown in Figure 124.
Once the InitialContext object is created, the application uses it to create the EJB home object, as shown in Figure 125.

```java
// lookup the home interface using the JNDI name
try {
    java.lang.Object o = initContext.lookup(JNDIBeanName); // this is the JNDI name
    if (o instanceof org.omg.CORBA.Object)
        pingHome = JpPingHomeHelper.narrow((org.omg.CORBA.Object) o);
} catch (javax.naming.NamingException e) {
    System.out.println("Error retrieving the home interface: "+ e.getMessage());
    System.exit(0);
} // endtry
```

Figure 125. Creating the EJB Home Object
This creation is accomplished by invoking the lookup method which takes the JNDI name of the enterprise bean in String form and returns a java.lang.Object object.

An EJB JNDI name can be modified. You change it inside VisualAge for Java by opening the EJB properties and changing the value in Enter the JNDI name for BeanHome field; see Figure 126.

![Figure 126. Changing the JNDI Name under VisualAge for Java](image)

Outside VisualAge for Java you use the WebSphere Advanced Edition 2.0's Jet tool to read the exported Jar file containing EJB beans. You start the Jet tool by running `drive:\WeBSphere\AppServer\sample\ejs\jet\jet.bat` command file. In the Input/Output tab, select **Browse** followed by the Jar file containing your EJB. According to the type of bean session or entity, the Jet tool highlights the corresponding tab. By selecting the tab, you get access to the deployment descriptor information; see Figure 127.
Figure 127. Changing the JNDI Name with Jet Tool

Once you have changed the JNDI name you select **Input/Output** tab and enter in the Output field the path and name of a new or the same Jar file to be created or modified and then press **Build** button. This updates the deployment descriptor serialized file with the new JNDI name.
The returned object of the lookup operation is filtered, using the static method narrow, to obtain an EJB Home object for the specified enterprise bean. The narrow method is contained in the home helper class generated during bean deployment. For the JpPing bean, this home helper class is named JpPingHelper.

Once a reference to the home object has been obtained, you use it to create the EJB object by using the EJB home object as shown in the following snippet of code (see Figure 128).

```java
JpPing ping = null;
try {
    ping = pingHome.create();
    System.out.println("JpPing created!");
} catch (Exception e) {
    System.out.println("Exception creating new JpPing: " + e.getMessage());
}
```

Figure 128. Creating the EJB Object

The EJB object is obtained by calling a create method or (for entity beans only) a finder method. Because the JpPing bean is a stateless session bean, the only choice is the default create method.

**Changing Host Name to Persistent Name Server and EJB Server**

If you want to run an EJB client that accesses an EJB running inside VisualAge for Java but on a different machine, you need to change the LSD name value in both the Persistent Name Server and EJB server. (See Figure 129.)
To make this change, from the workbench, select the **EJBs** tab, in the menu select **EJBs->Open To->server Configuration** as indicated in Figure 130.

This opens the EJ B Server Configuration window, as shown in Figure 131.
In the Servers pane, select **Persistent Name Server**, click mouse 2, and select **Properties**. This opens the Properties for Persistent Name Server window shown in Figure 132, giving access to three parameters:

- **Initial Root**  
  This parameter indicates where the file (root.ctx) containing the persistent naming information is located (see Figure 133).
LSD Name
This is the host name where the Location Server Daemon is running. The LSD Name is by default set to localhost and needs to be set to the fully qualified IP host name.

LSD Port
This is the port name on which the Location Server Daemon is listening. It must match the value defined during Location Server Daemon configuration.

You also need to configure the EJB server. Follow the same steps to get access to its properties (see Figure 134).
Once you have changed the localhost by the fully qualified host name in the two servers, you can start all three servers: Location Server Daemon, Persistent Name Server and the EJB server. Now your client running on a different machine can get access to the EJB running inside VisualAge for Java as long as it uses the correct fully qualified name and port number (see Figure 123 on page 208).
Nowadays, we see that a lot of developers are centralizing their business code inside objects running on a central application server. This centralization of components gives more control over the application logic to these developers. The applications developed using this kind of architecture are easier to scale as well (using such features as load balancing and database multiplexing).

In the traditional procedural approach, developers are using transactions to achieve a high degree of application integrity. A transaction is described by four properties:

- Atomic: transactions are all or nothing, they either work or they do not, they are never left incomplete.
- Consistent: transactions always let the system in a consistent state.
- Isolated: transactions execute in a safe manner, they will not fail if other transactions running on the same server are failing.
- Durable: transactions can survive system failures once they are completed and committed.

The transactional application servers combine the execution of component centralized on application servers with transactionality.
7.1 Enterprise JavaBeans and Transactions

Enterprise JavaBeans servers are transactional application servers. The transactional features of the Enterprise JavaBeans (EJB) architecture provides the developer with a good transaction service. Unlike other application programming interfaces (APIs) that support distributed transactions, the EJB API does not require enterprise bean and EJB bean application developers to write any special code to use transactions. Instead, the container manages transactions based on two deployment descriptor attributes associated with each enterprise bean, and the enterprise bean and EJB bean application developers are free to deal with the business logic of their applications.

The Enterprise JavaBeans specifications provide the developer with some mechanisms to control the transaction processing within an Enterprise JavaBeans application. Using Enterprise JavaBeans technology, a developer does not face the complexity involved with the development of a transactional application. This task is left to the Enterprise JavaBeans server provider. The Enterprise JavaBeans technology lets a developer update data into multiple databases, across multiple sites, if needed. All the transaction processing is handled by the Enterprise JavaBeans server. To achieve its transactionality, the Enterprise JavaBeans model implements things such as two-phase commit, transaction context propagation and distributed two-phase commit. It is up to the EJB Server vendors to decide which technique they use.

This is a simple protocol where the transaction manager first asks all participants if their work is finished and would like to attempt to commit their work. This is the first phase of the protocol. Once everyone agrees to commit, the second phase begins.

For a complete definition of Two-Phase commit, refer to the following Technology Review from the Software Engineering Institute at:


EJB has really the best of both worlds. If the client application developer would like to control the transactions himself (we call this: client-demarcated transactions) it is possible; see 7.8, “Client-Demarcated Transactions” on page 237. The EJB Container can manage the transaction processing on behalf of your EJB beans as well (this is called container-demarcated transactions). This means your beans do not have to explicitly demarcate transaction boundaries. The transaction control is then handled by the EJB Server. The transaction processing applies to Session and entity beans. Actually there is no support for nested transactions; the decision not to
support nested transactions was intended to allow vendors of existing transaction processing and database management systems to incorporate support for Enterprise JavaBeans technology. If these vendors provide support for nested transactions in the future, Enterprise JavaBeans technology may be enhanced to take advantage of nested transactions.

The EJB Specification describes different scenarios using transactions. We decided to illustrate the bean-managed demarcation and the container-managed demarcation. The client-demarcated transactions are explained in 7.8, “Client-Demarcated Transactions” on page 237. We were not able to test the following scenarios because they are not yet supported by the WebSphere environment:

1. Update of multiple databases using one server
2. Update of databases using multiple servers

The Bean-Managed Demarcation

Using the TX_BEAN_MANAGED transaction attribute, we are able to use the javax.jts.UserTransaction interface to demarcate transactions in our Session Bean. We then see what is the effect of the different transaction attributes on the entity beans.

If your beans are not using the TX_BEAN_MANAGED transaction attribute they will not be allowed to control their transactions. The container will not make available the javax.jts.UserTransaction to them. The javax.jts package is intended to be used both by the bean-demarcated transactions and the client-demarcated bean developer.

The Container-Managed Demarcation

Whenever a client invokes an enterprise bean, the container interposes itself on the method invocation. The interposition allows the container to control transaction demarcation declaratively as described primarily by the method's transaction attribute and in its absence by the bean's transaction attribute.

Using different transaction attributes, we see how the EJB Server, WebSphere, interposes on the method invocations and control the transactions.

The Transaction Attributes

The transaction attribute defines the transactional manner in which the container invokes enterprise bean methods. This attribute can be set for the
bean as a whole and for individual methods in a bean. The following are valid values for this attribute:

- **TX_BEAN_MANAGED**
  Notifies the container that the bean class implements the javax.jts.UserTransaction interface and that the bean can call methods to explicitly manage transaction boundaries.

- **TX_MANDATORY**
  Directs the container to always invoke the bean method within transaction context associated with the client. If the client attempts to invoke the bean method without a transaction context, the container throws the javax.jts.TransactionRequiredException exception to the client. The transaction context is passed to any enterprise bean objects or resources that are used by this bean method.

- **TX_NOT_SUPPORTED**
  Directs the container to invoke bean methods without a transaction context. If a client invokes a bean method from within a transaction context, the container suspends the association between the transaction and the current thread before invoking the method on the enterprise bean instance. The container then resumes the suspended association when the method invocation returns. The suspended transaction context is not passed to any enterprise bean objects or resources that are used by this bean method.

- **TX_REQUIRES_NEW**
  Directs the container to always invoke the bean method within a new transaction context, regardless of whether the client invokes the method within or without a transaction context. The transaction context is passed to any enterprise bean objects or resources that are used by this bean method.

- **TX_REQUIRED**
  Directs the container to invoke the bean method within a transaction context. If a client invokes a bean method from within a transaction context, the container invokes the bean method within the client transaction context. If a client invokes a bean method without a transaction context, the container creates a new transaction context and invokes the bean method from within that context. The transaction context is passed to any enterprise bean objects or resources that are used by this bean method.

- **TX_SUPPORTS**
  Directs the container to invoke the bean method within a transaction context if the client invokes the bean method within a transaction. If the client invokes the bean method without a transaction context, then the container invokes the bean method without a transaction context.
The transaction context is passed to any enterprise bean objects or resources that are used by this bean method.

**Transaction Isolation Levels**

The transaction isolation level determines how isolated one transaction is from another for read purposes only. This attribute can be set for the bean as a whole and for individual methods in a bean. The following are valid values for this attribute in decreasing order of isolation:

- **TRANSACTION_SERIALIZABLE**
  This level prohibits all of the following types of reads:
  - Dirty reads, where a transaction reads a database row containing uncommitted changes from a second transaction.
  - Nonrepeatable reads, where one transaction reads a row, a second transaction changes the same row, and the first transaction rereads the row and gets a different value.
  - Phantom reads, where one transaction reads all rows that satisfy an SQL WHERE condition, a second transaction inserts a row that also satisfies the WHERE condition, and the first transaction applies the same WHERE condition and gets the row inserted by the second transaction.

- **TRANSACTION_REPEATABLE_READ**
  This level prohibits dirty reads and nonrepeatable reads, but it allows phantom reads.

- **TRANSACTION_READ_COMMITTED**
  This level prohibits dirty reads, but allows nonrepeatable reads and phantom reads.

- **TRANSACTION_READ_UNCOMMITTED**
  This level allows dirty reads, nonrepeatable reads, and phantom reads.

These isolation levels correspond to the isolation levels defined in the Java Database Connectivity (JDBC) java.sql.Connector interface. The container uses the transaction isolation level attribute as follows:

- **Session beans and entity beans with BMP:**
  For each database connection used by the bean, the container sets the transaction isolation level at the start of each transaction.

- **Entity beans with CMP:**
  The container generates database access that achieves the specified isolation level and gets the row inserted by the second transaction.
Actually WebSphere does not support these isolation levels and will, in the presence of transactions, always open a database connection with the TRANSACTION_REPEATABLE_READ isolation level.

7.2 EJB Transactions and CORBA OTS

CORBA OTS

Enterprise JavaBeans technology supports flat transactions, modeled after the OMG Object Transaction Service 1.1 (OTS). The transaction model used in EJB technology is similar to OTS. In fact, CORBA®-compliant EJB servers must provide an OTS-compliant transaction service. Understanding how OTS works helps to understand how transactions work in EJB technology.

To better understand how OTS works, we need to look at its key components. The following components can be mapped almost directly to EJB technology, and understanding how these components work in OTS will provide a good foundation for understanding transactions. The dashed box represents a transaction. Within the transaction are all of the objects participating in the transaction. Commits and rollbacks will be applied to all the Resource objects in this group. (See Figure 135.)
The **Control** object represents the transaction. From this object we can obtain the **Coordinator** and the **Terminator**. An EJB developer never sees the **Control** object. It is used by the container to manage the transaction on behalf of the bean. The **Terminator** is also used by the container to commit or roll back the transaction when a thread returns from a bean method, where the method is described in the deployment descriptor as requiring the container to terminate the transaction upon completion of the method. When a commit or rollback is requested, all the objects in the transaction will be committed or rolled back, as appropriate.

The **Resource** is the object that has transactional state. For example, it may be a connection to a database. Calling **commit**() on this object would cause the database updates to be applied to the database. A **rollback**() would revert all changed made to the database through this connection since the start of the transaction. Once the commit or rollback completed, the corresponding rows in the database would be unlocked. The level of locking applied would have been specified in the deployment descriptor. The full set of methods for this object would show that these objects actually implement the two-phase commit protocol, allowing each object to vote as to whether the entire transaction should be committed or rolled back.
The **Synchronization** is an object that wishes to be notified upon completion of the transaction whether the transaction was committed or rolled back. Unlike the **Resource**, it does not participate in the two-phase commit protocol and has no vote as to whether the EJB bean transaction should be committed or rolled back. It plays a passive role in the transaction. This is the role that a session bean may play by implementing a special interface. The **Coordinator** is the object that makes all this work. Through this object both **Resource** objects and **Synchronization** objects are registered in the transaction. The bean does not get access to this object directly. Transaction-aware objects that are intended for use with EJB beans will transparently obtain a reference to the current transaction's **Coordinator** to register itself.

**Transactional versus Recoverable Objects**

OTS distinguishes transactional and recoverable objects. This distinction is relevant to EJB technology. While the full definition of these types is rather extensive in the OTS specification, it essentially means that recoverable objects have a **commit()** and **rollback()** method, allowing the transaction to directly manipulate their state or behavior. A transactional object does not have these methods, and cannot be influenced by the transaction. However, a transactional object is an object that has a transaction associated with it, so that recoverable objects (or **Resources**) that are allocated will be associated with the transactional object's current transaction.

An enterprise bean that is transaction enabled is a good example of a transactional object. The container will maintain a transaction on behalf of the bean. Any recoverable objects that are allocated by the bean are transparently placed into the current transaction with the help of the container. The bean does not have a **commit()** or **rollback()** method, and so the transaction cannot manipulate the bean directly. It does not really make sense to make a bean a recoverable **Resource** since this would require additional work on the part of the bean developer for every bean written, and enterprise beans rarely have internal state that should directly affect the outcome of a transaction. Perhaps this is a reason why the recoverable objects are not required by the actual version of the specifications.

They function better as managers of recoverable objects, letting the recoverable objects do the work. Note that a bean can vote to roll back a transaction prior to the container attempting a commit or rollback. A bean can still be told of the outcome of a transaction through the **SessionSynchronization** interface.
7.3 Implementing Session Synchronization

A session Bean can optionally implement the interface: javax.ejb.SessionSynchronization. This interface can provide the Bean with transaction synchronization notifications in the form of container callbacks. Session beans use these notifications to manage database data they may cache within transactions.

The afterBegin notification signals a session instance that a new transaction has begun. At this point, the instance is already in the transaction and may do any database work it requires within the scope of the transaction.

The beforeCompletion notification is issued when a session instance’s client has completed work on its current transaction but prior to committing the instance’s resources. This is the time when the instance should write out any database updates it has cached. The instance can cause the transaction to rollback by invoking the setRollbackOnly method on its session context.

The afterCompletion notification signals that the current transaction has completed. A completion status of true indicates the transaction committed; a status of false indicates a rollback occurred.

In WebSphere Advanced Edition 2.0 the entity bean implementation class can implement the SessionSynchronization interface so they can also be notified when a new transaction begins or completes. We used this in our scenario testing, since we needed to know how many transactions were involved in the test cases. Each time the afterBegin method was called in one of our beans, we printed relevant information to the console; this way we were able to know how many transactions were started by the transaction service. Please note that is not portable across different EJB container vendors.

7.4 Scenarios

Scenario Description

Our design model to test the effect of the transactions attributes has four components:

- One session bean (TransactionTest)
- Two CMP entity beans (CMPSample1 and CMPSample2)
- One client application (SimpleClient)
One scenario consists of setting the transaction attributes of the CMP beans and run the SimpleClient Java application.

The SimpleClient creates a myTransactionTest session bean. The client application then calls 3 method on this SessionBean: test1(), test2() and test3(); see Figure 136.

The myTransaction.test1() and myTransaction.test2() represent our implementation of the Bean demarcated transaction model. They show the effect of the transaction attributes within the Bean demarcated transaction model.

The myTransaction.test1() execute a committed transaction; see Figure 137.
Figure 137. MyTransactionTest.test1() method

The myTransaction.test2() represents the rolled-back transaction; see Figure 138.
Figure 138. MyTransactionTest.test2() Method
The last method the SimpleClient application calls is `test3()` . This method illustrates the effect of the transaction attributes within the **Container-demarcated transaction model**. All the transaction processing is handled by the container; see Figure 139.

![Diagram](image)

**Figure 139.** MyTransactionTest.test3() Method
The Implementation Model

The SimpleClient object retrieves the initial context from the naming service. The initial context returns the home interface of our Session Bean, myTransactionTestHome. The SimpleClient then uses it to create our Session Bean instance, myTransactionTest. The client application then calls three methods on this SessionBean (test1, test2 and test3).

**myTransactionTest.test1()**

The myTransactionTest.test1() method starts by obtaining CMPSample1 home interface. The myTransaction then gets a transaction, myTransaction, from the SessionContext object. MyTransactionTest initiates the transaction with a myTransaction.begin() (the control of transactions from our myTransactionTest EJB forced us to specify its transaction attribute as TX_BEAN_MANAGED). Our myTransaction Bean then asks for two values from the user: a key and a state (both strings). It creates a primary key object for CMPSample1 and tries to find it using this key object. If the myTransactionTest bean does not find the CMPSample1 bean, it calls create with the key object. The CMPSample1.setKey() method is called with the previously entered state as parameter. This CMPSample1.setKey() method set the value of the state member of our CMPSample1 Entity Bean.

Following this, the CMPSample1 Bean asks for other values (key and state) from the user and tries to find an instance of CMPSample2. If the CMPSample1 bean does not find it creates an instance of the CMPSample2 bean and then call the CMPSample2.setState() method.

To complete, the myTransactionTest commits the myTransaction using: myTransaction.commit().

**myTransaction.test2()**

The myTransaction.test2() method is the second method the SimpleClient application calls. The test2() method behaves like the test1() with a few differences:

- The CMPSample1 Bean does not need to get the transaction object from the SessionContext, it has already been initialized in the test1() method.
- The myTransaction object does not commit the changes to the database, it rolls them back with the myTransaction.rollback() call.
myTransaction.test3()

The last method the SimpleClient application calls is test3(). It does the same steps as test1() but without the transaction-related actions.

- The MyTransactionTest does not begin a transaction.
- The MyTransactionTest does not commit nor rollback.

All of the transaction processing is left to the container.

7.5 The Test Cases Results

The results of our different test cases are presented in Table 4 on page 232, Table 5 on page 232, Table 6 on page 233, Table 7 on page 233, and Table 8 on page 234. Each table has three columns:

- The method called tells to what method this result refers. Refer to “Scenario Description” on page 225 for a description of this method.
- The Result in Database columns tell you what happened in our database after the method call completed.
- The Comments column contains an explanation of the result; for example:
  - The number of transactions involved
  - When the transaction were started
  - Any problems encountered

The tables are presented by scenario. Here are the steps we followed for each scenario:

1. We adjusted the transaction attribute of our CMP beans (CMPSample1 and CMPSample2).
2. We generated the code.
3. We ran the SimpleClient application.

Following the tables of results, you see an explanation of the transaction attributes according to these scenarios. We used the VisualAge for Java EJB test environment with DB2 for these tests.
## Result Tables

For CMPSample1 and CMPSample2:

### Table 4. Scenario 1 - TX_REQUIRED

<table>
<thead>
<tr>
<th>Method called</th>
<th>Result in Database</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>test1()</td>
<td>Data inserted and updated in both tables</td>
<td>Only one transaction is used during this scenario, the transaction started from the myTransactioTest session bean.</td>
</tr>
<tr>
<td>test2()</td>
<td>No Data inserted or updated</td>
<td>Only one transaction is used during this scenario, the transaction started from the myTransactioTest session bean.</td>
</tr>
<tr>
<td>test3()</td>
<td>Data inserted and updated in both tables</td>
<td>Only one transaction is used during this scenario, the transaction started by the container after the CMPSample1.setState() method call and is carried on to the CMPSample2 Bean.</td>
</tr>
</tbody>
</table>

For CMPSample1 and CMPSample2:

### Table 5. Scenario 2 - TX_MANDATORY

<table>
<thead>
<tr>
<th>Method called</th>
<th>Result in Database</th>
<th>Comments a</th>
</tr>
</thead>
<tbody>
<tr>
<td>test1()</td>
<td>Data inserted and updated in both tables</td>
<td>Three transactions are involved. The transaction started by the myTransactionTest session Bean, one transaction is started by the container on the CMPSample1.setState and the third transaction is started by the container on the CMPSample2.setState.</td>
</tr>
<tr>
<td>test2()</td>
<td>No Data inserted or updated</td>
<td>Three transactions are involved. The transaction started by the myTransactionTest session Bean, one transaction is started by the container on the CMPSample1.setState and the third transaction is started by the container on the CMPSample2.setState.</td>
</tr>
</tbody>
</table>
Transactions

For CMPSample1 and CMPSample2:

Table 6. Scenario 3 TX_NOT_SUPPORTED

<table>
<thead>
<tr>
<th>Method called</th>
<th>Result in Database</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>test3( )</td>
<td>No Data, exception on client</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Since there is no transaction context available and the container needs one, the container throws one exception to the client when it tries to do the CMPSample1.setState().</td>
<td></td>
</tr>
</tbody>
</table>

a. These strange results are described in “Test Scenarios Results Explanation” on page 234.

For CMPSample1 and CMPSample2:

Table 7. Scenario 4 TX_REQUIRES_NEW

<table>
<thead>
<tr>
<th>Method called</th>
<th>Result in Database</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>test1( )</td>
<td>Data inserted in both tables not updated</td>
<td></td>
</tr>
<tr>
<td></td>
<td>One transaction is started by the myTransactionTest bean. The container suspends the transaction on the CMPSample1.setState() and restores it when the execution returns to the myTransactionTest bean.</td>
<td></td>
</tr>
<tr>
<td>test2( )</td>
<td>Not applicable</td>
<td></td>
</tr>
<tr>
<td></td>
<td>We have a ContainerObjectNotFoundException when we try to find the CMPSample1 bean.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method called</th>
<th>Result in Database</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>test1( )</td>
<td>Nothing happens, have to stop the database server.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Deadlock, the container locks after the CMPSample1.setState call.</td>
<td></td>
</tr>
</tbody>
</table>
Test Scenarios Results Explanation

Case 1: Effect of the TX_REQUIRED Attribute

The TX_REQUIRED transaction attribute means that your bean's methods require a transaction in order to interact with the database system. You want to use this attribute when your beans needs to be in a transactional context to protect their integrity. Most of the time this is the transaction attribute you use with entity beans.

In the case of the presence of a transaction context, your bean is not isolated from a client's rollback. This means if the client (either application, servlet or bean) decides to roll back the transaction, the changes in your bean will be rolled back as well.

If no transaction is available (like in myTransactionTest.test3(), the container will get one from the transaction service and provide it to the bean, before any methods are invoked on it.

The execution of the myTransactionTest.test3() made us realized something regarding the way our container modeled the transaction control. Our container, WebSphere Advanced Edition 2.0 forces all methods from the home to be called within a transaction context. This means all the finder and create methods will be executed transactionally.

WebSphere Application Server 3.0 will make the Home interfaces EJB beans, so it will be possible to describe the type of transaction attribute you need on specific finder in their deployment descriptor.

Case 2: Effect of the TX_MANDATORY Attribute

A bean with the TX_MANDATORY has to be called within the transaction context of its client (either application, servlet or bean). If the client does not have a transaction scope, the container throws the TransactionRequired exception to the client, as we saw with the myTransactionTest.test3() method.
In theory, if the calling bean or client is associated with an existing transaction context (like in myTransaction.test1() and myTransaction.test2() methods), the container invokes the bean's methods using this transaction context. This transaction context is passed to other beans when it invokes methods on it. This means if you commit or roll back the calling transaction context, all of your work done within this transaction will be committed or rolled back.

Actually, as the results of our tests with WebSphere Advanced Edition 2.0, we saw that with WebSphere Advanced Edition 2.0 more than one transaction was started. This strange behavior is an implementation detail of the transaction service WebSphere Advanced Edition 2.0.

**Case 3: Effect of the TX_NOT_SUPPORTED Attribute**

The TX_NOT_SUPPORTED is for non-transactional work. It is useful if you either do not want to deal with transactions or have something that is inherently non-transactional that you are effecting by the operations within the bean.

Since the myTransaction transaction started by the myTransactionTest session bean is suspended when the container intercepts the CMPSample1.setState method, the update of this field is not applied in the database (it is not done within a transaction context). All of the home interface methods are forced to be executed transactionally with WebSphere; this is the reason why the record is inserted in the database.

In our rollback situation method (myTransactionTest.test2()) WebSphere sent the following exception:

```java
```

This is a bug we had with WebSphere Advanced Edition 2.0.

**Case 4 and case 5: Effect of the TX_REQUIRES_NEW Attribute**

We experienced a few problems testing those scenarios. When we tried to call a method on CMPSample2 from CMPSample1, the container stayed in a "Wait for lock" mode; this meant our bean was in a deadlock situation. We had to stop the container and database to continue working.

**Case 6: Effect of the TX_SUPPORTS Attribute**

Actually, there is a bug in the original EntityDescriptor class which prevents us from using the TX_SUPPORTS transaction attribute. You then cannot specify this attribute with VisualAge for Java.
In our case, the methods myTransactionTest.test1() and myTransactionTest.test2() both have a transaction scope when they call methods on CMPSample1. However, the myTransactionTest.test3() does not have a transaction context, which caused the container to raise the TransactionRequired exception to the calling SessionBean, myTransactionTest.

### 7.6 Concurrent Access from Multiple Transactions

The enterprise bean developer does not have to worry about concurrent access from multiple transactions when writing the business methods. The enterprise bean developer writes the methods assuming that the container will ensure appropriate synchronization for entity beans that are accessed concurrently from multiple transactions. The entity container typically uses one of two implementation strategies to achieve proper synchronization:

1. The container activates multiple instances of the enterprise bean, one for each transaction in which the entity is being accessed. The transaction synchronization is performed automatically by the underlying database during the database access calls performed by the `ejbLoad`, `ejbCreate`, `ejbStore`, and `ejbRemove` methods. The database system provides all the necessary transaction synchronization; the container does not have to perform any synchronization logic.

2. The container acquires an exclusive lock on the instance’s state in the database. The container activates a single instance and serializes the access from multiple transactions to this instance.

WebSphere Advanced Edition 2.0 implements a version of option 2. Only a single active instance is managed by the container. As transactions access the instance, an exclusive lock is placed on the active instance for the duration of the transaction. This exclusive lock avoids a common deadlock that can occur as multiple transactions access multiple beans.
7.7 Transactions and Exceptions

In the case of a client-demarcated transaction which invokes an enterprise bean business method, a create method, a remove method, or a finder method, and the method returns with an exception other than javax.jts.TransactionRolledbackException, the client can assume that the transaction has not been automatically marked for rollback. The client can assume that the transaction has been marked for rollback if the exception is javax.jts.TransactionRolledbackException. It is fruitless for the client to continue the transaction because the transaction can never commit. The client should catch the javax.jts.TransactionRolledbackException exception and stop its processing.

If the client receives the java.rmi.RemoteException exception instead of the javax.jts.TransactionRolledbackException, the client, in general, does not know if the enterprise bean’s method has completed or not. Therefore, if a transactional client receives the java.rmi.RemoteException exception, the client should call the rollback method on the current transaction to prevent inconsistent data.

Note that an enterprise bean who is also a client of another enterprise bean can use the getRollbackOnly method to test if the current transaction has been marked for rollback. If the current transaction has been marked for rollback, the bean should stop its transactional processing since the transaction will be rollbacked.

On the bean side, if your bean cannot ensure at a certain period of time that the bean is in a consistent state, you should use the setRollbackOnly method on the EJBContext interface to mark a transaction for rollback. Thereafter, your bean can throw an exception to the client. This mechanism allows you to control whether a transaction should be rolled back or not within a bean method.

7.8 Client-Demarcated Transactions

The clients and beans that have to programmatically control transaction scopes should use the javax.jts.UserTransaction interface that is defined as part of the Java Transaction Service (JTS) API. The javax.jts.UserTransaction interface is the only JTS interface that the EJB container provider must implement in order to support EJB beans. Java Transaction Service (JTS) API defines all the Java programming language interfaces related to transaction management on the Java platform.
The EJB specification does not, however, define how a client application can create a transaction, in that it does not specify how to obtain a reference to the javax.jts.UserTransaction interface in a client application. In the absence of this specification, the Application Server provides a class that may be used to access the JTS CosTransactions.Current interface, which is a superset of the javax.jts.UserTransaction interface. This interface allows transactions to be created, suspended and resumed, and committed or aborted. To obtain a reference to the org.omg.CosTransactions.Current interface, the client application may simply call the static getCurrent method on the com.ibm.ejs.client.EJ Client class. For example:

```
    com.ibm.ejs.client.EJClient.getCurrent();

current.begin();
// do transactional work
current.commit(false);
```

Note: To ensure that the transactions created using the current interface that are propagated as method invocations are made on enterprise beans, the current reference should be obtained before creating a JNDI InitialContext.

For portability of client applications across different EJB implementations, developers may wish to program their clients using the javax.jts.UserTransaction interface. The following simple class implements the javax.jts.UserTransaction interface using the CosTransactions.Current interface:

```
package myPackage;

import org.omg.CosTransactions.*;

public final class UserTransaction implements javax.jts.UserTransaction{
    Current current = null;

    public UserTransaction() {
        current = com.ibm.ejs.client.EJClient.getCurrent();
    }

    public void begin() throws IllegalStateException {
        try {
            current.begin();
        } catch (SubtransactionsUnavailable e) {
            throw new IllegalStateException(e.toString());
        }
    }
}
```
    try {
        current.commit(true);
    } catch (org.omg.CosTransactions.NoTransaction nte) {
        throw new IllegalStateException("No transaction.");
    } catch (org.omg.CosTransactions.HeuristicMixed hme) {
        throw new javax.jts.HeuristicMixedException();
    } catch (org.omg.CosTransactions.HeuristicHazard hze) {
        throw new javax.jts.HeuristicRollbackException();
    }
}

public int getStatus() {
    Status stat = current.get_status();
    if (stat == Status.StatusActive)
        return javax.jts.UserTransaction.STATUS_ACTIVE;
    if (stat == Status.StatusCommitted)
        return javax.jts.UserTransaction.STATUS_COMMITTED;
    if (stat == Status.StatusCommitting)
        return javax.jts.UserTransaction.STATUS_COMMITTING;
    if (stat == Status.StatusMarkedRollback)
        return javax.jts.UserTransaction.STATUS_MARKED_ROLLBACK;
    if (stat == Status.StatusNoTransaction)
        return javax.jts.UserTransaction.STATUS_NO_TRANSACTION;
    if (stat == Status.StatusPrepared)
        return javax.jts.UserTransaction.STATUS_PREPARED;
    if (stat == Status.StatusPreparing)
        return javax.jts.UserTransaction.STATUS_PREPARING;
    if (stat == Status.StatusRolledBack)
        return javax.jts.UserTransaction.STATUS_ROLLED_BACK;
    if (stat == Status.StatusRollingBack)
        return javax.jts.UserTransaction.STATUS_ROLLING_BACK;
    return javax.jts.UserTransaction.STATUS_UNKNOWN;
}

public void rollback() throws IllegalStateException, SecurityException {
    try {
        current.rollback();
    } catch (NoTransaction nt) {
        throw new IllegalStateException("No Transaction.");
    }
}

public void setRollbackOnly() throws IllegalStateException {
    try {
        current.rollback_only();
    }
}

Transactions  239
Client-demarcated transactions have disadvantages over the container-demarcated transactions. You do not control the period of time the client will use your beans within a transaction context. This means you can waste precious server resources.

For example, a client starts a transaction to query an EJB bean, and leaves for the weekend. The client transaction will be active for all the weekend. Your server resources are then tied up for all this period. When you use client-demarcated transactions, you lose the flexibility to let the deployer adjust the transaction attribute as well. This means all your beans are TX_BEAN_MANAGED and cannot be tailored, depending on the deployment servers.

It is generally a good practice to specify your transactional beans as TX_REQUIRED because they are executed within a transactional context.

7.9 Conclusion

Enterprise JavaBeans technology simplifies the developer’s tasks with component transactionality, as we saw in this chapter. Unfortunately, the Enterprise JavaBeans specifications lacking definition details and do not explain the nature of the technology in several important areas:

- Thread pooling
- Concurrency control
- Resource management.

These details have been left to the different EJB Server vendors. This can result in proprietary EJB solutions (meaning less portable ones). These technological issues should be addressed in the Enterprise JavaBeans specifications if we want to keep one of their goals:

- Platform independence and portability
8 Security

The Internet has exposed distributed applications to possible malicious users. This has necessitated security management in these applications to ward off threats by hackers and virus attacks. The security feature of the Java language and the Virtual Machine (VM) along with the EJB specification address many of the security concerns. Enough literature is already available about the security features of the Java language and the VM. This chapter will only delve into the security features specific to EJB.

8.1 Security for EJB Beans

The EJB beans themselves provide no security. Security for EJB beans is derived from your system and is usually found within your Web Server. Application Server provides security through servlets and their relationship to the Web Server.

EJB beans require security features necessary to develop distributed transaction based applications. These are:

- Identifying and authenticating the caller
- Managing access to resource based on security policies
- Providing on-the-wire security
8.2 Identifying and Authenticating the Caller

Deploying an application on the Internet provides it access to audiences who could misuse it. It is necessary, therefore, for these applications to identify a privileged user and to grant access only on a successful authentication. (See Figure 140.)

Figure 140. Access Control

8.3 Application Server Security

Protecting Resources by Access Control Lists

Users
A user is an identity that can be authenticated by the Web server. A user can represent a person or a computer. You can name users however you like. Some common schemes include:

- Strings, such as kilroy
- Distinguished names, such as uid=kilroy
- E-mail addresses, such as kilroy@was.here.com

Groups
Groups are sets of users. Groups provide an efficient way to manage large numbers of users because an administrator can specify permissions for an entire group at one time.
Usually, group members have something in common. For example, a company might have one group for managers and another for non-managerial employees. It may be that non-managers are given access to view sales data files, and managers are allowed both to view and to edit the sales data files.

A person can be defined as both an individual user and as a member of a group. For example, the person Amy might be represented both as individual user amy and as a member of the group managers. When an individual user also belongs to a group, the individual access permissions override any group access permissions.

**Resources**

Resources are the valuable items accessible by your Web server:

- HTML files and directories, such as http://www.anycompany.com (Web pages)
- Web applications: Java servlets or CGI programs and by the IBM WebSphere Application Server:
  - Java servlets
  - Connections, sockets, files, and other resources that can be used by servlets
  - Custom servlets that enable access to enterprise resources and applications (such as databases)

---

**Note**

In the current release of WebSphere Advanced Edition 2.0 you cannot specify an EJB bean as a resource.

Each resource can be protected by establishing a single access control list (ACL) for that resource, in a single realm (discussed below). The ACL specifies which users or groups are allowed to access or modify the resource.

For each resource to protect, you will specify:

- An access control list (ACL)—a list defining who can use the resource
- A security realm—a logical security area the resource belongs to
- An authentication scheme—a way to verify users who ask for the resource

The relationship among these items will more become apparent later in this discussion.
Permissions
Permissions represent the privileges required for accessing resources. An administrator protects resources by establishing access control lists to grant permissions to users and groups.

Application Server lets you set a variety of permissions. For example, there are permissions to send and receive files, delete files, read and write files, load servlets, link to libraries, open and listen on sockets, and more. For descriptions of the available permissions, view the field help for the Access Control Lists page, the Manager page for setting permissions.

Individual user permissions take precedence over group permissions. For example, if the user amy has both read and write access to a file, but the group amy belongs to has only read access to the file, amy remains able to read and write to the file because the individual user permission overrides the more restrictive group permission. (Even if the group permission is less restrictive than the user permission, the user permission overrides the group permission).

Permissions are related to specific resources. Suppose a user has permission to view FileA and has permission to modify FileB. The user’s permission to modify FileB does not allow the user to modify FileA. Each permission applies to a particular resource and is nontransferable.

Realms
A realm is a database of users, groups, and access control lists. A user must be defined in a realm in order to access any resources belonging to that realm.

A user can belong to several realms, but a user ID cannot be duplicated within a realm. For example, the user amy can belong to the realms fileRealm and anyRealm. Each of those realms can contain only one user named amy. The user amy can be given different permissions for different resources in each realm.

A person, such as Kilroy, might have different user IDs in each realm. For example, he might be kilroy in the fileRealm, but is kilroy2 in the anyRealm because there is already a person in anyRealm using the user ID kilroy.

Realms the Application Server ships with IBM WebSphere Application Server ships with some pre-established realms:

- The defaultRealm defines how users may access resources defined locally. You can establish access control lists (discussed shortly) to determine which users and groups have access to which resources.
The **UNIX** or **NT realm** is named for your operating system and defines how users with accounts on the system the server is running on may access Application Server resources. Users and groups defined in the underlying system are shared by the IBM WebSphere Application Server, and continue to exist for as long as they exist in the underlying system. The Application Server Manager interface lets you view this realm; to change it, you must use the facilities provided by your operating system.

The **servletMgrRealm** defines how servlets may access resources defined remotely, such as remotely loaded servlets. The servletACL is the only access control list in this realm. When a remotely loaded, digitally-signed servlet tries to access a protected resource, the digital certificate in the servlet is compared with digital certificates associated with users in the servletMgrRealm. The servletACL decides whether permission is granted or denied.

For example, suppose the digital certificate of userX is packaged in the anyServlet..JAR file. If userX is added to the servletMgrRealm, any servlet containing the same digital certificate as that of userX (found in anyServlet..JAR) may execute and access resources granted to userX.

Finally, the **LDAPRealm** will be displayed if you have enabled a directory service on the Directory Management page of the Application Server Manager.

Users and groups defined in the directory service are shared by Application Server, and continue to exist for as long as they exist in the directory service or until you disable directory management support. The Application Server Manager interface lets you view this realm; to change it, you must use the facilities the LDAP server provides.

### Access Control Lists

The access control list (ACL) associated with a resource specifies which users or groups in a realm are permitted to access the resource. ACLs, realms, and resources have the following relationship:

- A realm can contain many ACLs.
- An ACL can belong to only one realm.
- A realm can contain many resources.
- A resource can belong to only one realm.
- A resource is associated with only one ACL.
- An ACL can be associated with many resources.

In some cases, a service does not require its customers to be in an access control list. For example, many Web page (HTTP) services make their
Choosing Authentication Schemes and Protocols

This section discusses authentication protocols, namely HTTP and HTTPS. Next, this section discusses authentication schemes, namely basic, digest, custom, and certificate authentication. Finally, this section discusses strategies for combining schemes and protocols.

About Authentication Protocols

If you have used a Web browser, you are most likely familiar with the Hypertext Transfer Protocol (HTTP), which determines how Web servers and Web browsers or other Web clients communicate with one another. If you have ever entered "http://..." into a browser to indicate a URL or file on the Web, you have used HTTP. HTTPS is a combination of the HTTP and SSL protocols. SSL, or Secure Socket Layer, is a network security protocol that can be used to provide the necessary security between server and client.

If using Certificate authentication, which you will learn about shortly, you must use HTTPS. Otherwise, you can choose to use HTTP or HTTPS to protect your authentication data, the data that flows between client and server to verify that the client is allowed to access the server resources:

- If you choose HTTP, authentication data receives no protection.
- If you choose HTTPS, authentication data is encrypted per the SSL protocol.

HTTPS is preferable to HTTP, unless you are operating in an already secure environment (or lack security concerns) and seek to avoid the overhead related to using HTTPS.

About Authentication Schemes

This describes and helps you choose among authentication schemes IBM WebSphere Application Server supports:

- Basic
- Digest
- Custom
- Certificate

Certificate authentication requires HTTPS; the other schemes can use either HTTP or HTTPS.
An authentication protocol defines how authentication information (such as a user ID and password) is exchanged. IBM WebSphere Application Server requires clients to authenticate themselves to the server when requesting protected resources.

IBM WebSphere Application Server supports:

- **Basic authentication**
  Uses HTTP or HTTPS to request a user name and password from the client. The information is sent to the server in plain text for verification.
  All browsers support basic authentication. Consider using it if a user ID and password provide sufficient authentication.

- **Digest authentication**
  Uses HTTP or HTTPS to request a user name and password from the client. The user name and an encrypted form of the password (using a digest) are sent to the server for verification.
  Not all browsers support digest authentication. (Currently, only the Sun HotJava browser supports it). If a browser does not support digest authentication, its user cannot access the resources protected with this protocol.

- **Custom authentication**
  Uses HTTP or HTTPS to request client information using a customized HTML form. The information is sent in plain text to the server for verification by a CGI or servlet.
  Use custom authentication when you need user authentication other than an ID and password. For example, you can request a social security number from the user for authentication.
  With this protocol, you establish HTML forms to ask the user for data. Authentication is performed by server-side code (CGIs and servlets), not by the IBM WebSphere Application Server runtime application.
  If you use custom authentication, use HTTPS to protect the data.

- **Certificate authentication**
  Uses HTTPS to request a client certificate. The SSL client authentication option must be enabled. The information is sent to the server for verification.
  Authentication using digital certificates is highly secure, and certificate authentication is often transparent to the user.
  The system or site administrator manages the client certificates. Often the task is delegated to certificate authority server software, such as the IBM Vault Registry product.
Combining Authentication Schemes and Protocols

This section helps you decide whether to use HTTP or HTTPS in combination with your chosen authentication protocol: Basic, Digest, Custom, or Certificate authentication.

As previously stated, unless you are in an environment where security is not a concern, HTTPS is usually preferred.

Possible scenarios:
- For basic security, use Basic, Digest, or Custom authentication over HTTP.
- For more security, use Basic, Digest, or Custom authentication over HTTPS.
- For maximum security, use Certificate authentication over HTTPS.
Part 2  Home Banking Application
9 Developing the Bank Application

So far, you have learned how to develop every type of EJB beans using VisualAge for Java. We still need to understand how the Enterprise JavaBeans technology can be applied to a realistic application.

For that purpose, we have defined an application supplying basic services for a banking system.

The application provides user services to customers of the home banking application through a Web browser and administration services for an internal bank application. (See Figure 141.)
As you can see in Figure 141, Internet users get access to the bank application through an HTTP server using servlets and JSPs pages. Servlets and JSPs pages deal with a session bean UserSB which implements all use cases. In a real environment, each use case could be mapped on individual session bean. This would allow the deployer to distribute session beans on different machines and provide better scalability. For each use case, UserSB session bean retrieves and invokes business methods on entity beans.

These same entity beans can also be accessed by the AdminSB session bean implementing the use cases for administration purpose. The AdminSB implements use cases to create, remove bank accounts, customers. This type of activity could also be done through the Internet using JSP pages, but seems more suitable for a Java based application taking advantage of the higher bandwidth generally available on an intranet network.
The Bank application entity beans are implemented as Container-managed persistence enterprise beans and have their persistent state managed by the container using the ITSOBANK database.

In the following chapters, we describe how we have implemented the bank application. We do not describe how to create the client applications in this book. If you want to learn more you can refer to the redbook, SG24-5423, which addresses this part of the application. It is based on the same home banking application.

Now we can continue by looking to the home banking application's class diagram.

### 9.1 Class Diagram

The class diagram contains four main objects: Bank, BankAccount, Customer, and TransactionRecord (see Figure 142). A Bank has many BankAccounts, which may be of three types: CheckingAccount, SavingsAccount and CorporateAccount. Each transaction executed on a BankAccount is recorded as a TransactionRecord. Each BankAccount has an owner: a Customer, which has his own authentication information. The CorporateAccount is used to refer to corporations that the customer does business with. For example, the gas company could represent a corporate account.
These classes are connected in different ways. A customer has a one-to-many relationship with a BankAccount. A BankAccount can be derived in CorporateAccount, SavingAccount or CheckingAccount.

Figure 142. Bank Application Class Diagram
By introducing in the application basic object oriented concepts like aggregation and inheritance, we put ourselves face to face with some of the current Enterprise JavaBeans specification limitations. It does not yet addresses these problems and the tools do not yet provide help either.

Because we believe that readers of this book will expect to find some guidance in solving these problems, we have investigated and propose a way to implement inheritance and aggregation. The solutions used in this book to solve these very difficult problems are described in the following chapters: Chapter 10, “Relationships” on page 257 and Chapter 11, “Inheritance” on page 271.
10 Relationships

10.1 Problem Description

As we have already mentioned, the Enterprise JavaBeans specification 1.0 does not define how associations must be supported. The current version 2.1 of VisualAge for Java does not support EJB associations either.

To be able to build the one-to-many relationships in our banking application, we developed a relation package (itso.bank.ejb.relation) which wraps the association complexity.

10.2 Relation Package (itso.bank.ejb.relation)

Overview

The itso.bank.ejb.relation package includes two classes (ManyLink and SingleLink) that support implementing persistent one-to-many associations between entity beans depending on storing and retrieving primary or foreign key values in a database. The one-to-many relationships are mapped to backward pointing foreign-key references.
Class ManyLink implements the multi-value association end. Depending on the relation type parameter in the constructor, ManyLink supports an aggregation association (the parent controls the live cycles of its children) or a normal association.

Class SingleLink is the single-valued inverse association end of the one-to-many relationship.

Responsibilities of the Link Classes
- Maintaining the object-level referential-integrity
  Mutators (set-methods) and collection add/remove methods automatically invoke the appropriate referential integrity maintenance behavior, such as updating the inverse association end. For aggregation associations the operations performed on the association's owner are cascaded to the member objects. For example, deletion of the association's owner object causes the member objects to be deleted: bank.remove() causes customer.remove(). Also when a member object is removed from the association the member is deleted: bank.removeCustomer(customer) causes customer.remove().
- Retrieving the member(s) at the other association end

Responsibilities of the Bean Developer
The user of the link classes is responsible for:
- Defining the primary / foreign key in the entity bean.
- Creating a customized finder method for the multi-valued association end.
- Instantiating the link object in the entity bean.
- Implementing the accessor methods in the entity bean by invoking the appropriate methods of the link classes.
- Maintaining the life cycle state of the link classes. The user has to set the state of the link to "uninitialized" in the ejbLoad() method of the entity bean.
- Invoking the appropriate link methods [ManyLink.removeAllMembers() / SingleLink.setValue(null)] in the ejbRemove() method of the entity bean.
- Defining the entity bean on both side of the association as REENTRANT (Properties pane).
Design Considerations

In this section we describe the design considerations for the one-to-many relationship implementation (aggregation or normal association)

- Persistent association technique
  There are different techniques to implement persistent association. Our implementation depends on storing and retrieving primary or foreign key values in the database.

- Usability
  Our solution hides the association complexity in link classes. Entity beans delegate the relationship maintenance and members retrieval to the relation package.

  Two link objects are involved in each one-to-many relationship: at the multi-valued association end the entity bean owns a ManyLink object, the entity bean at the single-valued association end has a SingleLink object.

  A link object gets the association relevant parameters from the entity at creation time (for example: relationship type [aggregation or normal association], role of the class, name of the primary / foreign key). To be able to retrieve and maintain the association members, the link objects use introspection (java.reflect package).

  The link objects invokes the appropriate EJB homes to retrieve the member(s).

  Example:
  The SingleLink object in a CustomerBean entity invokes bankHome.findByPrimaryKey(bankId) to get the bank reference of a customer. CustomerBean has a public instance variable which represents the bank foreign key in the customer object. SingleLink gets the foreign key value from its entity and passes the value in the findByPrimaryKey call.

  The ManyLink object in a BankBean invokes customerHome.findByBankId(bankId) to retrieve the customers of a bank and gets the parameter value from its entity key object (BankKey).

  The SingleLink object updates the foreign key instance variable of its entity. Making the foreign key value persistent is the responsibility of the EJB container (CMP entity) or the bean (BMP entity).

  The relation package supports CMP and BMP entities.
Database design
When persistent associations are based on database keys the associations are explicitly mapped to the primary-key/foreign-key relationships defined in the relational database. One-to-many relationships are mapped to backward pointing foreign-key reference (the child table refers to the parent table).

Example:
To map the one-to-many relationship between Bank and Customer the Customer table has a foreign key field (bankId) which points to the primary key field of the Bank table.

Introspection
A link object gets the association relevant parameters (for example: relationship type, role, name of primary / foreign key) at run time (constructor parameters). To be able to retrieve the members and maintain the associations, the link objects are working with introspection (package java. reflect).

EJB server compatibility
We have tested the relation package with WebSphere 2.0 Advanced. We expect that our solution runs on other EJB servers supporting relational database persistence.

Using the Link Classes

Prerequisite (package itso.bank.ejb.base)
Entity beans have to extend itso.bank.ejb.base.ITSOEntityBean, and their remote interfaces from itso.bank.ejb.base.ITSOEntity.

ITSOEntityBean maintains the entity context and the live cycle state of the bean (for example: creating, created, removing).

The entity subclass has no context instance variable and forwards the context reference to the superclass in the setEntityContext method. The ejb callback methods (ejbCreate, ejbPostCreate, setEntityContext, ejbActivate, ejbLoad, ejbStore, ejbPassivate, ejbRemove, unsetEntityContext) call the appropriate method of the superclass (first statement in the method implementation).

Naming Conventions
In this section we describe the naming conventions of the itso.bank.ejb.relation package.
**Accessor Methods**

The user of the itso.bank.ejb.relation package has to define the accessor methods in the bean class and the Remote Interface.

The naming rules for the multi-valued association end of a one-to-many relationship are:

- public void add\<inverseRole\>(EJBRemoteType)
- public void remove\<inverseRole\>(EJBRemoteType)
- public Enumeration get\<inverseRole\>s()

The `<inverseRole>` indicates the role played by the entity bean at the other end of the association. It is a parameter in the constructor of the ManyLink class and has to correspond to the Remote Interface name of the inverse entity bean. Each of the methods throws java.rmi.RemoteException.

For the BankBean class we define the accessor methods:

- public void addCustomer(Customer) throws RemoteException
- public void removeCustomer(Customer) throws RemoteException
- public Enumeration getCustomers() throws RemoteException

The naming rules for the single-valued inverse association end are:

- public void set\<inverseRole\>(EJBRemoteType)

Note: If the one-to-many association is an aggregation, no set method is needed. The parent reference has to be passed in the `ejbcreate( ...)` and `ejbPostCreate( ...)` methods of the child entity bean. The association type is a parameter in the constructor of the ManyLink class owned by the entity bean at the other end of the association.

- public void get\<inverseRole\>()

The `<inverseRole>` indicates the role played by the entity bean at the other end of the association. It is a parameter in the constructor of the SingleLink class and has to correspond to the Remote Interface name of the bean. Each method throws java.rmi.RemoteException.

The Bank - Customer relationship is an aggregation. For the CustomerBean class we define:

- public void getBank() throws RemoteException
- public void ejbCreate(CustomerKey customerKey, Bank parent, ...) throws RemoteException
- public void ejbPostCreate(CustomerKey customerKey, Bank parent, ...) throws RemoteException
Classes and Interfaces

The user of the itso.bank.ejb.relation package has to follow the standard naming rules for classes and Interfaces of the SmartGuide (Create EJB).

For the bank EJB bean, we define:

- BankBean class
- BankKey class
- Bank interface
- BankHome interface

The current implementation of the itso.bank.ejb.relation package supports bean key classes with one instance variable (no compound keys allowed). The instance variable has to be public.

For the customer EJB bean, we define:

- CustomerBean class
- CustomerKey class
- Customer interface

Using the ManyLink Class

In this section we describe how we use the ManyLink class in the link owner Bank entity bean to implement the multi-valued association end of the Bank-Customer association (one-to-many relationship).

- Defining the ManyLink class in the link owner entity bean:
  ```java
  public class BankBean extends ITSOEntityBean implements javax.ejb.EntityBean {
    private ManyLink customerLink;
  }
  ```

- Instantiating / accessing the ManyLink object
  We lazy initialize the link object when it is used for the first time.

  ```java
  private ManyLink getCustomerLink() throws java.rmi.RemoteException {
    if (customerLink == null)
      customerLink = new ManyLink(this, "Customer", "itso.bank", "findByBankId",
                                 "primaryKey", ManyLink.AGGREGATION);
    return customerLink;
  }
  ```

- Retrieving the customers

  ```java
  public Enumeration getCustomers() throws RemoteException {
    return getCustomerLink().getMembers();
  }
  ```
Adding a customer

The method is called from the SingleLink object owned by the entity of the other end of the association when a new customer is created.

```java
public void addCustomer(Customer customer) throws java.rmi.RemoteException {
    getCustomerLink().addMember(customer);
}
```

Removing a customer

The ManyLink object removes the relationship. Because we defined the association as an aggregation (constructor of ManyLink) the link object deletes the customer (child of the aggregate).

```java
public void removeCustomer(Customer customer) throws java.rmi.RemoteException {
    getCustomerLink().removeMember(customer);
}
```

Note: customer.remove() gives the same result.

Maintaining the life cycle state of the the link object

The ejbLoad() method of the link owner's entity bean sets the life cycle state of the link as "uninitialized". The ManyLink object initializes its member collection during the next accessor call by invoking the home query method of the inverse association end bean (CustomerHome).The query members method name is a parameter of the ManyLink constructor.

```java
public void ejbLoad () throws java.rmi.RemoteException {
    getCustomerLink().setUninitialized();
}
```

Removing the bank

Because we defined the association as an aggregation (constructor of ManyLink) the link object deletes all his children (customers) during the delete process of his owner entity bean.

```java
public void ejbRemove() throws java.rmi.RemoteException {
    super.ejbRemove();
    getCustomerLink().removeAllMembers();
}
```

Using the SingleLink Class

In this section we describe how we use the SingleLink class in the link owner Customer entity bean to implement the single-valued inverse association end of the Bank - Customer association (one-to-many relationship).

Defining the SingleLink class in the link owner entity bean:

```java
public class CustomerBean extends ITSOEntityBean implements javax.ejb.EntityBean {
    private SingleLink bankLink;
    public String primaryKey;
    public String bankForeignKey;
```
Instantiating / accessing the SingleLink object
We lazy initialize the the link object when it is used for the first time.

```java
private SingleLink getBankLink() throws java.rmi.RemoteException {
    if (bankLink == null)
        bankLink = new SingleLink(this, "Bank", "itso.bank", "bankForeignKey", "primaryKey");
    return bankLink;
}
```

Retrieving the bank

```java
public Bank getBank() throws RemoteException {
    return (Bank) getBankLink().getValue();
}
```

Setting the bank

```java
public void ejbCreate(CustomerKey customerKey, Bank bank, ...) throws RemoteException {
    super.ejbCreate(); //superclass maintains the entity live cycle state
    primaryKey = customerKey.primaryKey;
    getBankLink().setValue(bank);
}
```

```java
public void ejbPostCreate(CustomerKey customerKey, Bank bank, ...) throws RemoteException {
    super.ejbPostCreate(); //superclass maintains the entity live cycle state
    getBankLink().setValue(bank);
}
```

A customer is a part of a bank (Bank - Customer is an aggregation association).
In CustomerBean.ejbCreate(...) and CustomerBean.ejbPostCreate(...) we pass the reference to the aggregate of the part (Bank). CustomerBean has no setBank(Bank) method (a customer can not change the aggregate).

SingleLink.setValue(EJ BObject) called from ejbCreate(..) sets the foreign key instance variable in link owner's entity to the primary key of link owner's aggregate. Column BANKID in table CUSTOMER is a non nullable field. Not setting the foreign key in ejbCreate(..) results in an SQL error. We pass the foreign key field name and the primary key field name of the aggregate bean key in the constructor of SingleLink.

SingleLink.setValue(EJ BObject) called from ejbPostCreate(..) sets the value of the association end and updates the inverse association end by calling the appropriate add member method of the owner aggregate and passing the link owner's reference (its EJ BObject reference). The ejbObject reference is accessible in the postCreate(..) method of the link owner's entity.
Maintaining the life cycle state of the link object

The ejbLoad() method of the link owner’s entity sets the life cycle state of the link as ‘uninitialized’. The SingleLink object initializes its member value during the next accessor call by invoking the home of the inverse association end:

```
bankHome.findByPrimaryKey (primaryKeyOfBank)
```

```
public void ejbLoad () throws java.rmi.RemoteException {
    super.ejbLoad();
    getBankLink().setUninitialized();
}
```

Removing the customer

```
public void ejbRemove() throws java.rmi.RemoteException {
    super.ejbRemove(); //superclass maintains the entity live cycle state
    bank.setValue(null);
}
```

SingleLink.setValue(EJBObj) called from CustomerBean.ejbRemove(..) sets the value of the association end to null and updates the inverse association end by calling the appropriate remove member method of the owner aggregate.

Note: bank.removeCustomer(customer) gives the same result.

Description of the Link Classes

To make the implementation of the link classes the most generic possible we intensively work with the package java.reflect. Class ManyLink and SingleLink create objects, constructors, methods and fields dynamically during runtime depending on the parameters passed to the ManyLink and SingleLink constructors.

Class ManyLink

Constructor

```
ManyLink(ITSOEntityBean linkOwner,
        String inverseRole,
        String packageNameInverseAssociationEnd,
        String queryMembersMethodName,
        String primaryKeyFieldNameOfLinkOwnerBeanKey,
        int relationType) throws RemoteException
```

```
```
The constructor has six parameters:

1. Reference to the owner of the link (entity bean).

2. Inverse role (role of the entity bean at the other end of the association). The name of the role has to correspond to the Remote Interface of the inverse entity bean.

3. Package name of the inverse association end (ManyLink expects the bean key class of the link owner entity in the same package as the link owner).

4. Name of the members querying method of the inverse home class. Extensions to the home provide the services for finding the members of a many-link.

5. Name of the primary key field in the key class of the link owner entity. The primary key field has to be public. The ManyLink object gets the field from the key class and passes the value as foreign key in the members querying method (see previous parameter).

6. Relation type: Aggregation [ManyLink.AGGREGATION] controls the life cycle of the member objects or normal association [ManyLink.NORMAL_ASSOCIATION].

**public Enumeration getMembers()**

- If the member vector is not initialized yet: refreshes the vector by invoking the query members method of the inverse home (home of the entity at the other end of the association) and passes the primary key of the link owner’s entity as a parameter. The name of the query members method and the name of the primary key field of the link owner’s bean key are parameters of the ManyLink constructor.

- Converts the member vector to EJBObjectEnumeration and returns to enumeration.

**public void addMember(ITSOEntity entityToAdd)**

- Refreshes the member vector if the vector is not initialized yet (see getMembers() method).

- If the member vector does not contain the member reference: adds the reference to the vector.

- If the relation is a normal association (not an aggregation): updates the singled-value reference at the other end of the association by invoking the appropriate set method of the inverse bean and passing the EJBObject reference of the link owner’s entity as a parameter.
If the relation is an aggregation: the parent reference is passed to the child entity in the `ejbCreate(...)` and `ejbPostCreate(...)` method of the child. Both methods pass the parent reference to the `SingleLink` object by invoking `singleLink.setValue(EJBObject parentReference)`. `SingleLink` invokes the appropriate add method of the inverse multi-valued side of the association.

```java
public void removeMember(ITSOEntity entityToRemove)
```

- Refreshes the member vector if the vector is not initialized yet (see `getMembers()` method).
- If the member vector does contain the member reference: remove the reference in the vector.
- If the relation is a normal association (not an aggregation): updates the singled-value reference at the other end of the association by invoking the appropriate set method of the inverse bean and passing a null value.
- If the relation is an aggregation: deletes the child by invoking the child's `remove()` method.

```java
public void removeAllMembers()
```

- Refreshes the member vector if the vector is not initialized yet (see `getMembers()` method).
- Invokes `removeMember(entityToRemove)` for each member.

### Class SingleLink

**Constructor**

```java
SingleLink(ITSOEntityBean linkOwner,
            String inverseRole,
            String packageNameInverseAssociationEnd,
            String foreignKeyFieldName,
            String primaryKeyFieldNameOfInverseBeanKey) throws RemoteException
```

The constructor has six parameters:

1. Reference to the owner of the link (entity bean)
2. Inverse role (role of the entity bean at the other end of the association). The name of the role has to correspond to the Remote Interface of the inverse entity bean
3. Package name of the inverse association end
4. Name of the foreign key field in the link owner entity.
5. Name of the primary key field in the key class of the inverse association name of the role (Customer)
**public ITSOEntity getValue()**

- Initializes the object reference to the singled-value association end (if not initialized yet). The next paragraphs describe the initialization process:
  - Gets the foreign key value from the link owner’s entity bean. The name of the (public) foreign key field is a parameter in the SingleLink constructor.
  - Creates the inverse bean key object by invoking the constructor of the inverse bean key class (parameter: foreign key).
  - Invokes findByPrimaryKey method of the inverse bean’s home class (parameter: inverse bean key object).
  - Initializes the object reference (singled-value association end) with the result of the findByPrimaryKey method.

**public void setValue(ITSOEntity newRelation)**

- Initializes the object reference to the singled-value association end (if not initialized yet) (see getValue() method).
- Determines the type of relationship update (new, delete, update)
  - If the link owner’s entity bean is in the pre-creating state [calling method: ejbCreate(...)]: sets the foreign key instance variable of link owner’s entity bean to the primary key value of newRelation (parameter in the setValue() method).

  Note: not setting a non nullable foreign key for CMP entities during ejbCreate(...) results in an SQL error when the container tries to insert the row after ejbCreate(...) has ended.

  Updating the object references on both end of the association during ejbCreate(...) is not possible because the link owner’s entity has no identity yet.

  If the link owner’s entity bean is not in the pre-creating state [calling method: ejbPostCreate(...) or wrapper method setInverseRole(EJ BRemoteType)]: sets the foreign key instance variable of the link owner’s entity (see above), sets the value of the association end to newRelation and updates the inverse association end by calling the appropriate add member method and passing the link owner’s EJ BObject interface (also referred as the remote interface).

  Note: <inverseRole> is a parameter in the constructor of SingleLink class.
❑ Delete relationship (current relation: not null / new relation: null):
   sets the foreign key instance variable of the link owner’s entity and the
   value of the association end to null and updates the inverse association
   end by calling the appropriate remove member method and passing the
   link owner’s EJB Object interface (also referred as the remote
   interface).

❑ Update relationship (current relation: not null / new relation: not null):
   Delete old relationship, then create new association.

Limitations

The itso.bank.ejb.relation package has the following limitations:

❑ The SingleLink class implements the single-valued end of a one-to-many
   association (the entity bean at the other end of the association has to own
   a ManyLink object). One-to-one association in both directions are not
   supported.

❑ No compound keys allowed. Bean key classes have one (public) instance
   variable.
In this chapter we describe how entity EJB beans can be used to display polymorphism.

11.1 Inheritance

The main motive for implementing inheritance is to support polymorphism and to provide reuse of the objects. Polymorphism in the object-oriented sense means using a superclass variable to represent a subclass object.

For example, a bank could have different types of accounts like Savings, Checkings and Corporate. Polymorphism implies using an object (a BankAccount) to represent any type of account.

The 1.0 version of the EJB specification does not provide inheritance support between EJB beans. In this chapter we describe the programming model to support inheritance and a sample implementation using this model.
11.2 Requirements for the Inheritance

Inheritance implementation for entity beans implies providing the following functionalities:

- Specifying an is-a relationship among entity beans, like, a Saving Account is a Bank Account
- Making the findByPrimaryKey method in the Home Interface return the target class along with its subclasses, that is, a search for all the BankAccounts returns even the Saving Accounts, Checking Accounts and Corporate Accounts.
- Making relationship finders return heterogenous results, containing instances of the related class and its subclasses.

11.3 Programming Model

Inheritance affects the different components of an EJB bean: the home interface, the EJB remote interface, the EJB implementation and the primary key class. Since entity beans cannot directly inherit the properties and behavior of another entity bean, we describe a "logical" inheritance as follows:

We identify an entity EJB bean that will serve as the parent EJB bean of a set of child EJB beans. This bean is referred to as the Parent bean and is described by the following:

- Remote interface, Parent
- Home interface, ParentHome
- Bean implementation class, ParentBean
- Primary key class, ParentKey

A 'subclass' of Parent bean called Child bean would have the following characteristics:

- Remote interface, Child
- Home interface, ChildHome
- Bean implementation class ChildBean
- Primary key class, ParentKey
Remote Interface

The remote interface of the child EJB bean, Child, must extend the remote interface of the parent EJB bean, Parent. This ensures that an instance of the child EJB bean may be used where an instance of the parent EJB bean is expected, one of the important benefits of inheritance.

EJB Homes

The home of the Child bean, ChildHome, must not extend the Parent home, ParentHome. However, it is important to define the relationship between methods on the ParentHome and ChildHome interfaces.

A create method on ParentHome can only create a Parent bean instance, and a create method on ChildHome creates both a Child bean and a Parent bean instance.

A remove method on ParentHome can remove Parent bean instances only. A remove method on ChildHome removes both the Child and Parent bean instances. A custom finder method on ParentHome returns an Enumeration that may contain Parent bean instances. The determination of the actual contents of the Enumeration is described as an example in “Accessing the BankAccount Hierarchy” on page 281.

For bean implementations that are the root of an inheritance hierarchy, we define a "discriminator" field which determines that actual type of the instance. A getter method must also be defined for the discriminator and this method should be promoted to the remote interface.

For a bean that inherits from a parent class, the child bean’s implementation class must extend from a Proxy class. The Proxy superclass contains all the methods defined in the remote interface of the parent bean. An invocation on any of these methods is redirected to an instance of the parent bean.

A client must not work directly with an instance of the Parent bean but only with the Child beans. The discriminator field is useful in identifying the specific child bean to which the instance belongs to. The client then looks up the corresponding child home for obtaining the true instance.

Primary Key Class

All entity beans in an inheritance must use the same Java key class for their primary keys.
11.4 Mapping Schemes for Relational Databases

Inheritance support implies support for multiple patterns for mapping a hierarchy of types to one or more database tables.

- "Single-table" mapping: All the types in the hierarchy map to the same table, a discriminator column identifies the type of object to be instantiated from a row, tables are not fully-normalized, but the whole hierarchy can be accessed easily. (See Figure 143.)

![Figure 143. Single-table Mapping](image)

- "Root/leaf" mapping: The root of the hierarchy specifies the root table which also provides a discriminator column. Each subtype specifies the leaf table which must be joined to gather the subclass-specific columns. Tables are fully-normalized, but multiple joins can be required to read a single instance, and joins and unions are required to read a hierarchy. (See Figure 144.)

![Figure 144. Root/leaf Mapping](image)
"Distinct-table" mapping: Each class in the hierarchy maps to a separate table, which contains the complete row for that type. Tables are likely not normalized, access to a single type requires no joins, but access to a hierarchy always requires unions. Since the tables do not have to be related, key consistency problems between the tables can make this pattern very difficult to automate. (See Figure 145.)

**Note:** Our programming model supports only the root/leaf mapping.
11.5 Developing the BankAccount Hierarchy

The class diagram shown in Figure 146 has a superclass BankAccount and its subclasses- CorporateAccount, SavingsAccount and CheckingAccount. We now develop the EJB beans to match this hierarchy.

Developing the Remote Interfaces

The remote interfaces of the entity beans contain the methods of the respective classes in the hierarchy. The remote interfaces of the CorporateAccount, SavingsAccount and CheckingAccount beans extend from the BankAccount remote interface.
The BankAccount Superclass

You can develop the BankAccountBean just like any other entity bean. Create a CMP entity bean (BankAccount) having key class BankAccountKey, remote interface BankAccount and home interface BankAccountHome. Implement the business methods, defined in the class diagram, and add them to the bean's Remote interface.

We have implicitly described a discriminator field, accountType, in the superclass. This field will be used to extract the actual instance of the subclass. Based on the value of this field the appropriate Home of the child beans are looked up and the instance of the subclass obtained.

Developing Subclass EJB Beans

We implement inheritance by delegation, that is, the "subclass" has a reference to a "superclass" object rather than inheritance of its visible methods and fields. In the Bank Application, it means the subclasses, like SavingsAccountBean, define wrapper methods for the super classes methods and within these wrappers delegate responsibility to the actual BankAccount object.

In order to obtain a degree of code reuse, which is one of the design goals of inheritance, we define an abstract class BankAccountProxy, which has the wrappers and does the necessary delegation. All the subclasses—SavingsAccountBean, CheckingAccountBean and CorporateAccountBean extend from this class.

The Subclasses should use the same Key class as the one used by the superclass, namely, BankAccountKey. This is done in order to be able to identify both the superclass and the subclass with the same key.

Note

VisualAge for Java does not generate the deployment classes properly for remote interfaces which extend from other interfaces. The tool does not identify any of the methods defined in the super class. You need to manually copy all the methods from the superclass (interface) to the remote interface even though there is an inheritance to enable the tool to generate the classes without any errors.
An obvious question would be as to why SavingsAccountBean has not been directly extended from the BankAccountBean. The problem with the direct inheritance approach is that the super class bean, BankAccountBean, is not registered with the Home, BankAccountHome. Entity beans not registered with their respective homes (via create or find methods) operate just like any other Java object inaccessible to container callbacks and hence can not avail the transaction, persistence and security features.

**Proxy Superclass**

The BankAccountProxy class contains the methods that wrap the remote methods of the BankAccount bean. It also contains a method getProxy() that returns the actual BankAccount object. All the remote method calls are routed to this object. The BankAccount object is obtained after a lazy initialization. In the initialization, the Home interface, BankAccountHome, is looked up using the findByPrimaryKey method passing the same key as that of the subclass.

```java
if (proxy == null) {
    proxy = getBankAccountHome().findByPrimaryKey( getKey() );
}
```

The getKey() method is an abstract method whose implementation is provided in the subclass, where the primary key of the instance is returned as follows:

```java
if (entityContext != null)
    return (BankAccountBean) entityContext.getPrimaryKey();
```
We also define a protected method createBankAccount which is invoked within the ejbCreate method of the subclass in order to create an instance of the 'superclass' bean. In the createBankAccount method, the create(...) method is invoked on the BankAccountHome passing the necessary parameters.

```java
protected void createBankAccount(BankAccountKey key, ...) {
    proxy = getBankAccountHome().create(key, ...);
}
```

The state of the actual BankAccount object that the BankAccountProxy is using must be synchronized with the current state of the 'subclass' entity bean, like the SavingsAccount bean. This can be done by initializing/uninitializing the proxy object in the container callback methods invoked on the subclass entity bean during different stages of its lifecycle:

- **ejbPassivate**
  The container invokes this method when it decides to send the entity to the pooled state. After the completion of this method, the proxy object no longer corresponds to the state of the subclass. We uninitialize the proxy object in this method, by setting it to null.

- **ejbActivate**
  The container invokes this method when it sends the entity to the ready state. At this point the proxy object is out of synch with subclasses state. Its state will not be determined till the next ejbLoad and hence left uninitialized.

- **ejbRemove**
  When the subclass is being removed, even the superclass needs to be removed. In this method we remove the proxy object by a getProxy().remove() operation.

- **When any of the remote methods of the superclass remote methods are invoked the actual proxy object needs to be located in order to invoke the remote methods on it. This is done by a lazy initialization described earlier in this section.**

- **When a new instance of the subclass is created a new instance of the superclass entity bean must be created and the proxy objects value set to the created instance.**

We defer the loading of the super class until any of its remote methods are invoked. This way there is not unnecessary Home interface lookup which is an expensive operation.
Defining the Inheritance Mapping Scheme

In our implementation we work with the root/leaf inheritance map to represent the tables in the database. In this mapping scheme there is a 1-to-1 correspondence between the tables and the classes. The root table contains the columns matching the fields of the super class and the leaf tables contain the columns matching the fields of the specific subclasses and a foreign key matching the primary key of the root table.

![Diagram of BankAccount and SavingsAccount classes and tables]

<table>
<thead>
<tr>
<th>BankAccount Table</th>
<th>SavingsAccount Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCID</td>
<td>BANKID</td>
</tr>
<tr>
<td>101-1001</td>
<td>ITSO</td>
</tr>
<tr>
<td>101-1002</td>
<td>ITSO</td>
</tr>
<tr>
<td>102-2001</td>
<td>ITSO</td>
</tr>
<tr>
<td>102-2002</td>
<td>ITSO</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ACCID</th>
<th>MINAMT</th>
</tr>
</thead>
<tbody>
<tr>
<td>101-1002</td>
<td>100.00</td>
</tr>
<tr>
<td>102-2001</td>
<td>100.00</td>
</tr>
</tbody>
</table>

The figure above shows how the SavingsAccount and the BankAccount beans can be mapped using the root/leaf mechanism:

- All the fields of the superclass, BankAccount, are mapped to the BankAccount table as distinct columns with ACCID (accountId) being the primary key for the table.
- The fields of the SavingsAccount class are mapped similarly to the SavingsAccount table with an additional field ACCID being the primary key.
- ACCID in the BankAccount table acts as the foreign key in the SavingsAccount table.
We prefer this approach over the single table and distinct table mapping for the following reasons:

- Classes can be added to the hierarchy without altering any of the existing tables. This is not true for single table mapping where the fields of the new class in the hierarchy must be mapped to new columns in the single table.
- The resulting tables are compacter than in the distinct table mapping. In the distinct table mapping all the columns of the parent's table are duplicated in the tables of the subclasses.
- In the single table mapping, when different subclass beans share the same key class, it is possible for the findByPrimaryKey method to return an instance which does not correspond to the specified bean. For example, the findByPrimaryKey method invoked on the SavingsAccountHome passing a BankAccountKey may return a CheckingAccount.

The restriction on the type of inheritance mapping may hamper development of EJB beans from existing tables. You also cannot define any foreign key constraints (DELETE, RESTRICT) on the subclasses' table using the primary key of the superclasses' table.

### 11.6 Accessing the BankAccount Hierarchy

We now describe how different instances in the BankAccount hierarchy can be accessed by a client.

According to our programming model, every entity bean has an associated Home interface. Thus we have a SavingsAccountHome for the SavingsAccount bean, a BankAccountHome for the BankAccount bean and likewise.

Implementing inheritance by delegation results in the creation of two instances the child bean and the parent bean when a create method is invoked on the child bean's home interface. When a find<METHOD> is invoked on the parent bean's home, only the parent bean's instances are returned and not of its subclasses. The instances returned do not represent the true identity of the created entity bean but only the portion corresponding to the parent bean.

We describe a mechanism to extract the original instances, which represent the instances of the child beans.

1. A client invokes a find<METHOD> on the Parent bean's Home, BankAccountHome
2. The findMethod returns an instance or an enumeration of instances of the Parent bean.

3. The client obtains the discriminator field of an instance by invoking the getter method for the discriminator field. The value of this field is one of ‘SAVINGS’, ‘PAYEE’ or ‘CHECKING’.

4. The client determines the child beans home to be looked up based on the value of the discriminator. SavingsAccountHome for ‘SAVINGS’, CorporateAccountHome for ‘PAYEE’ and CheckingAccountHome for ‘CHECKING’.

5. The client invokes the findByPrimaryKey method on the child bean’s home passing the primary key of the parent bean’s instance.

6. The result of the above find method is the actual instance which should be used by the client.

When a client attempts to invoke methods belonging to the subclass, the object needs to be narrowed using the necessary CORBA helper class in the following way:

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

    /**
     * Find a specific BankAccount using the Key
     * If its a SavingsAccount determine the Minimum amount
     * We could have used the specific home - SavingsAccountHome but this step
     * is necessary when you are using
     * the parents Home to lookup a child bean
     */
    BankAccountHome bankAcHome = ....
    BankAccount ba = bankAcHome.findByPrimaryKey(...);

    if (ba.getAccountType().equals("Saving") ) {
        SavingsAccount sa = SavingsAccountHelper.narrow(ba);
        /*
        * we cannot directly do
        * SavingsAccount sa = bankAcHome.findByPrimaryKey("...");
        * and we also cannot type cast it directly like sa = (SavingsAccount) ba;
        */
        sa.getMinimumAmount(); // get the minimum amount
    }
}
```

**The TypeConverter Class**

We define a TypeConverter Class which performs the task of obtaining the actual instance from the parent bean’s instance.
It contains two methods to convert the instance of the parent bean to an instance of the child bean: `typeCast(BankAccount)` and `typeCastEnumeration(Enumeration)`.

The `typeCast` method implements the steps described earlier to return the actual instance from an instance of the parent bean.

```java
public static BankAccount typeCast(BankAccount parent) throws RemoteException{
    String discriminator = parent.getAccountType();
    if ("SAVINGS".equals(discriminator))
        return getSavingsHome().findByPrimaryKey( parent.getPrimaryKey());
    if ("PAYEE".equals(discriminator))
        return getCorporateHome().findByPrimaryKey(parent.getPrimaryKey());
    if ("CHECKING".equals(discriminator))
        return getCheckingHome().findByPrimaryKey(parent.getPrimaryKey());
    throw new RemoteException("Could not locate actual instance");
}
```

The `typeCastEnumeration` method returns an `Enumeration` of the actual instances from an `Enumeration` of the instances of the parent bean.

```java
public static Enumeration typeCastEnumeration(Enumeration enum) throws RemoteException{
    Vector actual = new Vector();
    while (enum.hasMoreElements()) {
        actual.addElement( typeCast((BankAccount) enum.nextElement());
    }
    return actual.elements();
}
```

The `get<ChildBean>Home` returns the home of the child beans.

### 11.7 Relationships and Inheritance

The Relationship classes, SingleLink and ManyLink, described in Chapter 10, “Relationships” on page 257 have to be enhanced when the inverse role is an entity bean at the root of an inheritance hierarchy. Such a relationship is required when the parent entity bean is expected to display polymorphic behavior.

In our examples, we use the BankAccount hierarchy with the BankAccount at the root as the inverse role of the relationships. Similar changes have to be made in regard to naming when using any other inheritance hierarchies.
**SingleLink**

In a Bank, a TransactionRecord is created by a particular BankAccount which could be a SavingsAccount, CorporateAccount or a CheckingAccount. We extend the SingleLink class to represent an association between the TransactionRecord and the BankAccount. The TransactionRecord uses the link class, BankAccountSingleLink, to set and obtain the value of the BankAccount reference.

The BankAccountSingleLink extends the SingleLink class. Please refer to “Class SingleLink” on page 267 for more information about the SingleLink class. The usage of this class remains the same except as indicated below.

**Constructor**

The Constructor takes only the linkOwner and foreignFieldName parameters. The inverseRole, packageNameInverseAssociationEnd and primaryKeyFieldNameOfInverseBeanKey are already known and their values are BankAccount, itso.bank.ejb and accountId respectively.

The super classes constructor is invoked with these parameters in the body.

```java
public BankAccount getBankAccount()
```

Within this method the actual instance of the BankAccount is determined using the TypeConverter class and returned. This method should be used in the link owners ejbCreate and ejbPostCreate methods to set the inverse role in the relationship.

**ManyLink**

In a similar Bank scenario, a Customer has an aggregation of BankAccounts. A BankAccount in the aggregation could be a SavingsAccount, CheckingAccount or CorporateAccount. We extend the ManyLink class to represent the relationship between the Customer and the BankAccount (the root of the hierarchy). The Customer uses this link class, BankAccountManyLink, to add, remove and retrieve the BankAccounts.

The BankAccountManyLink extends the ManyLink class. Please refer to “Class ManyLink” on page 265 for more information about the ManyLink class. The usage of this class remains the same except as indicated below:

**Constructor**

The Constructor takes only the linkOwner, queryMembersMethodName, primaryKeyFieldNameOfLinkOwnerBankKey and relationType.
parameters. The inverseRole and packageNameInverseAssociationEnd are already known and their values are BankAccount and itso.bank.ejb respectively.

The super classes constructor is invoked with these parameters in the body.

**public Enumeration getBankAccounts()**

Within this method a vector of the BankAccounts is constructed from the super.getMembers() method. Every member of the vector is set to the actual instance using the TypeConverter class and the elements of this vector returned as an Enumeration. This method should be used in the link owners getter method to return the members of the aggregation.

**public void addBankAccount(BankAccount acc)**

This method invokes the super.addMember() passing the BankAccount as the entityToAdd. The advantage of using a method with this signature is the compile-time checking of the BankAccount class. This method should be used by the link owner to add a BankAccount to the aggregation.

**public void removeBankAccount(BankAccount acc)**

Within this method, the original instance of the BankAccount is obtained using the TypeConverter class and passed as the entityToRemove to super.removeMember(). This method should be used by the link owner to remove a BankAccount from the aggregation.

**public void removeAllMembers()**

Within this overridden method, the getBankAccounts() method is invoked and the returned Enumeration is traversed passing every member as an entity to remove to the super.removeMember(). This method should be used in the ejbRemove method of the link owner.

---

**11.8 The Future of EJB Inheritance**

As you might have noticed, implementing inheritance is not an easy task with entity beans. There are plans afoot to support EJB inheritance in future releases of the EJB specification and VisualAge for Java/Websphere. Until then, this approach serves as a generic approach towards supporting EJB inheritance without using any vendor-specific add-ons.
In this chapter, we describe how to create the entity beans:

- Bank
- Customer
- BankAccount
- SavingAccount
- CheckingAccount
- CorporateAccount
- TransactionRecord
12.1 Developing the BankAccount Hierarchy

In the bank application, we support inheritance between the super class BankAccount and CheckingAccount, SavingsAccount, CorporateAccount subclasses.

Developing the BankAccount Bean

The BankAccount bean has been described in “The BankAccount Superclass” on page 277. We now describe how the bean should be developed in order to support relationships.

Create a new BankAccount bean in the ITSOBank group specifying the following information:

- **Bean name**: BankAccount
- **Bean type**: Entity bean with container-managed fields (CMP)
- **Project**: ITSOBank
- **Package**: itso.bank.ejb
- **Class**: BankAccountBean
- **Superclass**: itso.bank.ejb.base.ITSOEntity
- **EJB Home Interface**: BankAccountHome
- **EJB Remote Interface**: BankAccount
- **EJB Key Class or Field**: BankAccountKey

Add the CMP Fields

Add accountId(java.lang.String), accountType(java.lang.String), balance(java.math.BigDecimal), bankFK(java.lang.String) and customerFK(java.lang.String) as CMP fields to the BankAccountBean. Set accountId as the Key field. Provide getter methods for the accountId, accountType and balance fields and promote these methods to the Remote interface.

Modify the Generated Methods

We modify the ejbCreate and ejbPostCreate methods to have a formal parameter list BankAccountKey key, Customer cust, Bank bank, String accountType. The parameters represent the non-nullable fields defined in the database.
Add the Remote Methods
As in the class diagram (see Figure 142 on page 254), add the significant methods to the bean and promote them to the remote interface.

Specify the Relationships
According to the class diagram, BankAccount has an association with a Bank and Customer object and an aggregation of TransactionRecord objects. To provide such a relationship in the EJB scenario we use the SingleLink and ManyLink classes described in Chapter 10, “Relationships” on page 257 as follows:

private transient itso.bank.ejb.relation.SingleLink bankLink = null;
private transient itso.bank.ejb.relation.SingleLink customerLink = null;
private transient itso.bank.ejb.relation.ManyLink transactionRecordLink = null;

The Links are initialized within private getter methods:

bankLink = new SingleLink(this, "Bank", "itso.bank.ejb", "bankFK",
"primaryKey");
customerLink = new SingleLink(this, "Customer", "itso.bank.ejb", "custFK",
"primaryKey");
transactionRecordLink = new ManyLink(this, "TransactionRecord",
"itso.bank.ejb", "findByAccountId", "accountId", ManyLink.AGGREGATION);

Add the necessary add, remove and accessor methods to support the different relationships. Make the necessary changes within the ejbCreate, ejbPostCreate, ejbActivate, ejbPassivate, setEntityContext and unsetEntityContext methods as described in “Responsibilities of the Bean Developer” on page 258.

Developing the Children EJB Beans

The Development of the children EJB beans in the BankAccount hierarchy is described in detail in “Developing Subclass EJB Beans” on page 277.

12.2 Developing the Bank Entity Bean

Create a new Bank bean in the ITSOBank group specifying the following information:

- **Bean name**: Bank
- **Bean type**: Entity bean with container-managed fields (CMP)
- **Project:** ITSOBank
- **Package:** itso.bank.ejb
- **Class:** BankBean
- **Superclass:** itso.bank.ejb.base.ITSOEntityBean
- **EJB Home Interface:** BankHome
- **EJB Remote Interface:** Bank
- **EJB Key Class or Field:** BankKey
- **import statements:**
  - itso.bank.ejb.base.*
  - itso.bank.ejb.relation.*
  - itso.bank.ejb.inheritance.relation.*

### Add Fields

**CMP**

For directions on how to add CMP fields see “Adding Fields to the Bean” on page 122.

Table 9 to Table 11 show the values you need to enter for each field.

**Table 9. Create Field bankName**

<table>
<thead>
<tr>
<th>Field Name</th>
<th>bankName</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field Type</td>
<td>String</td>
</tr>
<tr>
<td>Access Modifiers</td>
<td>public</td>
</tr>
<tr>
<td>Other Modifiers</td>
<td>none</td>
</tr>
<tr>
<td>Access with getter and setter methods</td>
<td>checked</td>
</tr>
<tr>
<td>Getter</td>
<td>public</td>
</tr>
<tr>
<td>Setter</td>
<td>public</td>
</tr>
<tr>
<td>Container Managed</td>
<td>yes</td>
</tr>
</tbody>
</table>
**Transient**

Table 10. Create Field customerLink

<table>
<thead>
<tr>
<th>Field Name</th>
<th>customerLink</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field Type</td>
<td>itso.bank.ejb.relation.ManyLink</td>
</tr>
<tr>
<td>Access Modifiers</td>
<td>private</td>
</tr>
<tr>
<td>Other Modifiers</td>
<td>transient</td>
</tr>
<tr>
<td>Access with getter and setter methods</td>
<td>checked</td>
</tr>
<tr>
<td>Getter</td>
<td>private</td>
</tr>
<tr>
<td>Setter</td>
<td>(none)</td>
</tr>
<tr>
<td>Container Managed</td>
<td>no</td>
</tr>
</tbody>
</table>

Table 11. Create Field accountLink

<table>
<thead>
<tr>
<th>Field Name</th>
<th>accountLink</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field Type</td>
<td>itso.bank.ejb.inheritance.relation.BankAccountManyLink</td>
</tr>
<tr>
<td>Access Modifiers</td>
<td>private</td>
</tr>
<tr>
<td>Other Modifiers</td>
<td>transient</td>
</tr>
<tr>
<td>Access with getter and setter methods</td>
<td>checked</td>
</tr>
<tr>
<td>Getter</td>
<td>private</td>
</tr>
<tr>
<td>Setter</td>
<td>(none)</td>
</tr>
<tr>
<td>Container Managed</td>
<td>no</td>
</tr>
</tbody>
</table>
Add New Methods

For directions on how to add a method to an EJB see “Adding Methods to EJB Beans” on page 49.

❑ addBankAccount

public void addBankAccount(BankAccount account) throws java.rmi.RemoteException {
  traceMethod("start Bank.addBankAccount(BankAccount account)");
  try {
    getBankAccountLink().addBankAccount(account);
    traceMethod("end Bank.addBankAccount(BankAccount account)");
  }
  catch (Exception e) {
    throw wrapperNonRemoteException("BankBean.addBankAccount(BankAccount)", e);
  }
}

❑ addCustomer

public void addCustomer(Customer customer) throws java.rmi.RemoteException {
  traceMethod("start Bank.addCustomer(Customer customer)");
  try {
    getCustomerLink().addMember(customer);
    traceMethod("end Bank.addCustomer(Customer customer)");
  }
  catch (Exception e) {
    throw wrapperNonRemoteException("BankBean.addCustomer(Customer)", e);
  }
}

❑ getBankAccounts

public Enumeration getBankAccounts() throws RemoteException {
  traceMethod("getBankAccounts()");
  return getBankAccountLink().getBankAccounts();
}

❑ getBankId

public String getBankId() throws RemoteException {
  traceMethod("getBankId()");
  return primaryKey;
}

❑ getCustomers

public Enumeration getCustomers() throws RemoteException {
  traceMethod("getCustomers()");
  return getCustomerLink().getMembers();
}

❑ removeBankAccount
public void removeBankAccount(BankAccount account) throws java.rmi.RemoteException {
    traceMethod("start Bank.removeBankAccount(BankAccount account)");
    try {
        getBankAccountLink().removeBankAccount(account);
        traceMethod("end Bank.removeBankAccount(BankAccount account)");
    }
    catch(Exception e) {
        throw wrapperNonRemoteException("BankBean.removeBankAccount(BankAccount)", e);
    }
}

❑ removeCustomer

public void removeCustomer(Customer customer) throws java.rmi.RemoteException {
    traceMethod("start Bank.deleteCustomer(Customer customer)");
    try {
        getCustomerLink().removeMember(customer);
        traceMethod("end Bank.deleteCustomer(Customer customer)");
    }
    catch(Exception e) {
        throw wrapperNonRemoteException("BankBean.deleteCustomer(Customer)", e);
    }
}

❑ traceMethod

private void traceMethod(String methodNameAsString) {
    String bankKeyAsString = null;
    Object ejbObject = null;
    try {
        bankKeyAsString = ((BankKey) getPrimaryKey()).primaryKey;
        ejbObject = getEJBObject();
    }
    catch(RemoteException e) {
        bankKeyAsString = "unknown";
    }
    System.out.println(">> " + methodNameAsString + " [" + this + "] / Key " + bankKeyAsString + "[EJBObject: " + ejbObject + "]");
}

Modify the Generated Methods

❑ ejbActivate

public void ejbActivate() throws java.rmi.RemoteException {
    super.ejbActivate();
    traceMethod("ejbActivate()");
}

❑ ejbCreate
public void ejbCreate(BankKey key, String name) throws RemoteException {
    // All CMP fields should be initialized here.
    super.ejbCreate();
    primaryKey = key.primaryKey;
    bankName = name;
    traceMethod("ejbCreate(BankKey) ");
}

❑ ejbLoad
public void ejbLoad () throws java.rmi.RemoteException {
    super.ejbLoad();
    getCustomerLink().setUninitialized();
    getBankAccountLink().setUninitialized();
    traceMethod("ejbLoad() ");
}

❑ ejbPassivate
public void ejbPassivate() throws java.rmi.RemoteException {
    super.ejbPassivate();
    traceMethod("ejbPassivate() ");
}

❑ ejbPostCreate
public void ejbPostCreate(BankKey key, String name) throws java.rmi.RemoteException {
    super.ejbPostCreate();
    traceMethod("ejbPostCreate(BankKey) ");
}

❑ ejbRemove
public void ejbRemove() throws java.rmi.RemoteException {
    super.ejbRemove();
    traceMethod("ejbRemove() ");
    getCustomerLink().removeAllMembers();
    getBankAccountLink().removeAllMembers();
}

❑ ejbStore
public void ejbStore () throws java.rmi.RemoteException {
    super.ejbStore();
    traceMethod("ejbStore() ");
}

❑ getBankAccountLink
private BankAccountManyLink getBankAccountLink() throws java.rmi.RemoteException {
    if (accountLink == null)
Bank Implementation Entity Beans

//accountLink = new ManyLink(this, "BankAccount", "itso.bank.ejb", "findByBankId", "primaryKey", ManyLink.AGGREGATION);
accountLink = new BankAccountManyLink(this, "findByBankId", "primaryKey", ManyLink.AGGREGATION);
return accountLink;

❑ getCustomerLink

private ManyLink getCustomerLink() throws java.rmi.RemoteException {
    if (customerLink == null)
        customerLink = new ManyLink(this, "Customer", "itso.bank.ejb", "findByBankId", "primaryKey", ManyLink.AGGREGATION);
    return customerLink;
}

❑ setEntityContext

public void setEntityContext(EntityContext ctx) throws java.rmi.RemoteException {
    super.setEntityContext(ctx);
}

❑ unsetEntityContext

public void unsetEntityContext() throws java.rmi.RemoteException {
    super.unsetEntityContext();
    traceMethod("unsetEntityContext()");
}

Add Methods to the Remote Interface

Select the following methods and add them to the bean remote interface. For directions on how to add a method to the remote interface, see “Adding Methods to the Remote Interface” on page 54.

❑ addBankAccount
❑ addCustomer
❑ getBankAccounts
❑ getBankId
❑ getBankName
❑ getCustomers
❑ removeBankAccount
❑ removeCustomer
❑ setBankName
12.3 Developing the Customer Entity Bean

Create a new Customer bean in the ITSOBank group specifying the following information:

- **Bean name**: Customer
- **Bean type**: Entity bean with container-managed fields (CMP)
- **Project**: ITSOBank
- **Package**: itso.bank.ejb
- **Class**: CustomerBean
- **Superclass**: itso.bank.ejb.base.ITSOEntityBean
- **EJB Home Interface**: CustomerHome
- **EJB Remote Interface**: Customer
- **EJB Key Class or Field**: CustomerKey
- **import statements**:
  - itso.bank.ejb.base.*
  - itso.bank.ejb.relation.*
  - itso.bank.ejb.inheritance.relation.*

**Add Fields**

**CMP**

For directions on how to add CMP fields see “Adding Fields to EJB Beans” on page 46.

We need to add six fields with the same characteristics. Only the name of the field is different. Use the information given in Table 12, but changing Field Name accordingly, to create the following fields:

- bankForeignkey
- firstName
- lastName
- password
- title
- userid
Table 12. Create Fields for the Customer Bean

<table>
<thead>
<tr>
<th>Field Name</th>
<th>bankForeignKey</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field Type</td>
<td>String</td>
</tr>
<tr>
<td>Access Modifiers</td>
<td>public</td>
</tr>
<tr>
<td>Other Modifiers</td>
<td>none</td>
</tr>
<tr>
<td>Access with getter and setter methods</td>
<td>checked</td>
</tr>
<tr>
<td>Getter</td>
<td>public</td>
</tr>
<tr>
<td>Setter</td>
<td>public</td>
</tr>
<tr>
<td>Container Managed</td>
<td>yes</td>
</tr>
</tbody>
</table>

**Transient**

Use the information given in Table 13 and Table 14 to create two transient fields required to implement the associations between:

- Customer and Bank (SingleLink)
- Customer and BankAccount (BankAccountManyLink)

Table 13. Create Field bankAccountLink

<table>
<thead>
<tr>
<th>Field Name</th>
<th>bankAccountLink</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field Type</td>
<td>itso.bank.ejb.inheritance.relation.BankAccountManyLink</td>
</tr>
<tr>
<td>Access Modifiers</td>
<td>private</td>
</tr>
<tr>
<td>Other Modifiers</td>
<td>transient</td>
</tr>
<tr>
<td>Access with getter and setter methods</td>
<td>checked</td>
</tr>
<tr>
<td>Getter</td>
<td>private</td>
</tr>
<tr>
<td>Setter</td>
<td>(none)</td>
</tr>
<tr>
<td>Container Managed</td>
<td>no</td>
</tr>
</tbody>
</table>
Add New Methods
For directions on how to add a method to an EJB see “Adding Methods to EJB Beans” on page 49.

❑ addBankAccount

```java
public void addBankAccount(BankAccount bankAccount) throws java.rmi.RemoteException {
    getBankAccountLink().addBankAccount(bankAccount);
}
```

❑ getBankAccounts

```java
public Enumeration getBankAccounts() throws java.rmi.RemoteException {
    return getBankAccountLink().getBankAccounts();
}
```

❑ getBankLink

```java
private SingleLink getBankLink() throws java.rmi.RemoteException {
    if (bank == null)
        bank = new SingleLink(this, "Bank", "itso.bank.ejb", "bankForeignKey", "primaryKey");
    return bank;
}
```

❑ getCustomerId

```java
public String getCustomerId() {
    return primaryKey;
}
```
Modify the Generated Methods

❑ ejbActivate

```java
public void ejbActivate() throws java.rmi.RemoteException {
    super.ejbActivate();
    traceMethod("ejbActivate()");
}
```

❑ ejbCreate

```java
public void ejbCreate(CustomerKey customerKey, String customerTitle, String customerFirstName, String customerLastName, String customerUserid, String customerPassword, Bank bank) throws RemoteException {
    try {
        super.ejbCreate();
        primaryKey = customerKey.primaryKey;
        title = customerTitle;
        firstName = customerFirstName;
        lastName = customerLastName;
        userid = customerUserid;
        password = customerPassword;
        getBankLink().setValue(bank); //sets only foreign key = foreign key is defined NOT NULL in the database
    }
    catch(Exception e) {
        throw wrapperNonRemoteException("CustomerBean.ejbCreate(CustomerKey, String, String, String, Bank)", e);
    }
    traceMethod("ejbCreate(CustomerKey customerKey, String customerTitle, String customerFirstName, String customerLastName, Bank bank");
}
```

❑ ejbLoad

```java
public void ejbLoad () throws java.rmi.RemoteException {
    super.ejbLoad();
    getBankLink().setUninitialized();
    getBankAccountLink().setUninitialized();
    traceMethod("ejbLoad()");
}
```

❑ ejbPassivate
public void ejbPassivate() throws java.rmi.RemoteException {
    super.ejbPassivate();
    traceMethod("ejbPassivate()");
}

❑ ejbPostCreate

public void ejbPostCreate(CustomerKey customerKey, String customerTitle, String customerFirstName, String customerLastName, String customerUserid, String customerPassword, Bank bank) throws RemoteException {
    super.ejbPostCreate();
    getBankLink().setValue(bank); //updates the object relation
    traceMethod("ejbPostCreate(CustomerKey customerKey, String customerFirstName, String customerLastName, Bank bank");
}

❑ ejbRemove

public void ejbRemove() throws java.rmi.RemoteException {
    super.ejbRemove();
    bank.setValue(null);
    getBankAccountLink().removeAllMembers();
    traceMethod("ejbRemove()");
}

❑ ejbStore

public void ejbStore() throws java.rmi.RemoteException {
    //bankId = getBankRelation().getBankId();
    super.ejbStore();
    traceMethod("CustomerBean.ejbStore()");
}

❑ setEntityContext

public void setEntityContext(EntityContext ctx) throws java.rmi.RemoteException {
    super.setEntityContext(ctx);
    traceMethod("setEntityContext(EntityContext")
}

❑ unsetEntityContext

public void unsetEntityContext() throws java.rmi.RemoteException {
    super.unsetEntityContext();
    traceMethod("unsetEntityContext()")
}

Add Methods to the Remote Interface

Select the following methods and add them to the bean remote interface. For directions on how to add a method to the remote interface, see “Adding Methods to the Remote Interface” on page 54.
Adding Finder Methods to CustomerHome

We need to add extra finder methods to the customer home:

- findByBankId
- findByUserId

In addition to adding these methods to the CustomerHome, you need to modify the CustomerBeanFinderHelper as indicated below:

```java
public interface CustomerBeanFinderHelper {
    public static final String findByBankIdQueryString = "SELECT T1.TITLE, T1.CUSTID, T1.PASSWORD, T1.FNAME, T1.USERID, T1.BANKID, T1.LNAME FROM ITSO.CUSTOMER T1 WHERE T1.BANKID = ?";
    public static final String findByUserIdQueryString = "SELECT T1.TITLE, T1.CUSTID, T1.PASSWORD, T1.FNAME, T1.USERID, T1.BANKID, T1.LNAME FROM ITSO.CUSTOMER T1 WHERE T1.USERID = ? AND T1.PASSWORD = ?";
}
```

For additional information about how to add finder methods see 4.9, “Adding Special Finder Methods” on page 158.

12.4 Developing the TransactionRecord Entity Bean

Create a new transactionRecord bean in the ITSOBank group specifying the following information:

- **Bean name**: TransactionRecord
- **Bean type**: Entity bean with container-managed fields (CMP)
- **Project**: ITSOBank
- **Package**: itso.bank.ejb
- **Class**: TransactionRecordBean
- **Superclass**: itso.bank.ejb.base.ITSOEntityBean
- **EJB Home Interface**: TransactionRecordHome
- **EJB Remote Interface**: TransactionRecord
- **EJB Key Class or Field**: TransactionRecordKey

**import statements**:  
- itso.bank.ejb.base.*
- itso.bank.ejb.relation.*
- itso.bank.ejb.inheritance.relation.*

### Add Fields

**CMP**

For directions on how to add CMP fields see “Adding Methods to the Remote Interface” on page 54.

We need to two add fields with the same characteristics. Only the name of the field is different. Use the information given in Table 15, but changing Field Name accordingly, to create the following fields:

- accountId
- traccId

**Table 15. Create Field accountId and traccId**

<table>
<thead>
<tr>
<th>Field Name</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>accountId</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Field Type</strong></td>
<td>String</td>
</tr>
<tr>
<td><strong>Access Modifiers</strong></td>
<td>public</td>
</tr>
<tr>
<td><strong>Other Modifiers</strong></td>
<td>none</td>
</tr>
<tr>
<td><strong>Access with getter and setter methods</strong></td>
<td>unchecked</td>
</tr>
<tr>
<td><strong>Getter</strong></td>
<td>public</td>
</tr>
<tr>
<td><strong>Setter</strong></td>
<td>public</td>
</tr>
</tbody>
</table>
Add the additional fields as indicated on Table 16 to Table 18.

Table 15. Create Field accountID and transactionID

<table>
<thead>
<tr>
<th>Field Name</th>
<th>accountID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Container Managed</td>
<td>yes</td>
</tr>
</tbody>
</table>

Table 16. Create Field transamt

<table>
<thead>
<tr>
<th>Field Name</th>
<th>transamt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field Type</td>
<td>java.math.BigDecimal</td>
</tr>
<tr>
<td>Access Modifiers</td>
<td>public</td>
</tr>
<tr>
<td>Other Modifiers</td>
<td>none</td>
</tr>
<tr>
<td>Access with getter and setter methods</td>
<td>checked</td>
</tr>
<tr>
<td>Getter</td>
<td>public</td>
</tr>
<tr>
<td>Setter</td>
<td>public</td>
</tr>
<tr>
<td>Container Managed</td>
<td>yes</td>
</tr>
</tbody>
</table>

Table 17. Create Field transtype

<table>
<thead>
<tr>
<th>Field Name</th>
<th>transtype</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field Type</td>
<td>String</td>
</tr>
<tr>
<td>Access Modifiers</td>
<td>public</td>
</tr>
<tr>
<td>Other Modifiers</td>
<td>none</td>
</tr>
<tr>
<td>Access with getter and setter methods</td>
<td>checked</td>
</tr>
<tr>
<td>Getter</td>
<td>public</td>
</tr>
<tr>
<td>Setter</td>
<td>(none)</td>
</tr>
<tr>
<td>Container Managed</td>
<td>yes</td>
</tr>
</tbody>
</table>
**Transient**

Table 18. Create Field accountLink

<table>
<thead>
<tr>
<th>Field Name</th>
<th>accountLink</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field Type</td>
<td>itso.bank.ejb.inheritance.relation.BankAccountSingleLink</td>
</tr>
<tr>
<td>Access Modifiers</td>
<td>private</td>
</tr>
<tr>
<td>Other Modifiers</td>
<td>transient</td>
</tr>
<tr>
<td>Access with getter and setter methods</td>
<td>checked</td>
</tr>
<tr>
<td>Getter</td>
<td>private</td>
</tr>
<tr>
<td>Setter</td>
<td>(none)</td>
</tr>
<tr>
<td>Container Managed</td>
<td>no</td>
</tr>
</tbody>
</table>

**Add New Methods**

For directions on how to add a method to an EJB see Table “Adding Methods to EJB Beans” on page 49.

- **getBankAccount**
  
  public BankAccount getBankAccount() throws java.rmi.RemoteException {
  
    return getAccountLink().getBankAccount();
  }

- **getTransactingAccount**
  
  public BankAccount getTransactingAccount() throws java.rmi.RemoteException {
    //this.traccId = ((BankAccountKey) otherAccount.getPrimaryKey()).accountId;
    if (traccId == null) {
      return null;
    }
    
    try {
      BankAccount account =
      itso.bank.ejb.inheritance.relation.TypeConverter.getBankAccountHome().findByPrimaryKey(new BankAccountKey(traccId));
    }
  }
return itso.bank.ejb.inheritance.relation.TypeConverter.typeCast(account);
} catch (Exception exc) {
    throw new RemoteException(exc.getMessage());
}

❑ getTransactionID

public java.sql.Timestamp getTransactionID() throws java.rmi.RemoteException{
     return primaryKey;
}

❑ setTransactingAccount

public void setTransactingAccount(BankAccount otherAccount) throws java.rmi.RemoteException {
    this.traccId = ((BankAccountKey) otherAccount.getPrimaryKey()).accountId;
}

Modify the Generated Methods

❑ ejbActivate

public void ejbActivate() throws java.rmi.RemoteException {
     super.ejbActivate();
}

❑ ejbCreate

public void ejbCreate(TransactionRecordKey key, BankAccount account, java.math.BigDecimal amount, String transType) throws RemoteException {
     super.ejbCreate();
     // All CMP fields should be initialized here.
     primaryKey = key.primaryKey;
     this.transamt = amount;
     this.transtype = transType;
     getAccountLink().setValue(account);
}

❑ ejbLoad

public void ejbLoad () throws java.rmi.RemoteException {
     super.ejbLoad();
}

❑ ejbPassivate

public void ejbPassivate() throws java.rmi.RemoteException {
     super.ejbPassivate();
}

❑ ejbPostCreate
public void ejbPostCreate(TransactionRecordKey key, BankAccount account,
java.math.BigDecimal amount, String transType) throws RemoteException {
    super.ejbPostCreate();
    // All CMP fields should be initialized here.
    getAccountLink().setValue(account);
}

❑ ejbRemove

public void ejbRemove() throws java.rmi.RemoteException {
    super.ejbRemove();
    getAccountLink().setValue(null);
}

❑ ejbStore

public void ejbStore() throws java.rmi.RemoteException {
    super.ejbStore();
}

❑ setEntityContext

public void setEntityContext(javax.ejb.EntityContext ctx) throws java.rmi.RemoteException {
    super.setEntityContext(ctx);
}

❑ unsetEntityContext

public void unsetEntityContext() throws java.rmi.RemoteException {
    super.unsetEntityContext();
}

Add Methods to the Remote Interface

Select the following methods and add them to the bean remote interface. For directions on how to add a method to the remote interface, see “Adding Methods to the Remote Interface” on page 54.

❑ getBankAccount
❑ getTransactingAccount
❑ getTransactionID
❑ getTransamt
❑ getTranstype
❑ setTransamt

Adding Finder Method to TransactionRecordHome

We need to add extra finder method to the TransactionRecord home:

❑ findByAccountId
In addition to adding these methods to the TransactionRecordHome, you need to modify the TransactionRecordBeanFinderHelper as indicated below:

```java
public interface TransactionRecordBeanFinderHelper {
    public static final String findByAccountIdQueryString = 
        "SELECT T1.TRANSID, T1.TRANSTYPE, T1.TRANSAMT, T1.ACCID, T1.TRACCID
        FROM ITSO.TRANSRECORD T1 WHERE T1.ACCID = ?";
}
```

For additional information about how to add finder methods see 4.9, “Adding Special Finder Methods” on page 158.

### 12.5 Creating the Bank Schemas

In the bank application we use Container-managed persistence beans. Normally, a bean developer does not need to specify how the bean’s fields are mapped to a given persistent store. This is the deployer role.

Remember that VisualAge for Java, in addition to a complete Enterprise JavaBeans development environment, also provides deployment and test facilities.

We use these additional capabilities to create the Container-managed persistence field mapping to a persistent store. For our bank application, the persistent store is based on an existing database called ITSOBANK. Details about the definition of the ITSOBANK database can be found in Appendix C, “ITSOBANK Database” on page 369.

For directions on how to create a schema see “Importing a Database Schema into VisualAge for Java” on page 147.

In order to create this schema, you need to provide the following information given in Table 19:

| Enter the names for the schema | Bank |
| Database Connection type       | COM.ibm.db2.jdbc.app.DB2Driver |
| Database Connection Data source | jdbc:db2::ITSOBANK |

Table 19. Create Bank Schema
Table 19. Create Bank Schema

<table>
<thead>
<tr>
<th>Enter the names for the schema</th>
<th>Bank</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select Tables Qualifiers</td>
<td>ITSO</td>
</tr>
<tr>
<td>Tables</td>
<td>select them all</td>
</tr>
</tbody>
</table>

The next step consists of using the schema to create a datastore map.

12.6 Creating the Bank Maps

For directions on how to create a schema see “Defining a Map between EJB Beans and the Database Schema” on page 151.

For the creation of this map you need the following information given on Table 20:

Table 20. Create Bank Datastore Map

<table>
<thead>
<tr>
<th>New Datastore Map Name</th>
<th>Bank</th>
</tr>
</thead>
<tbody>
<tr>
<td>New Datastore Map EJB group</td>
<td>ITSOBank</td>
</tr>
</tbody>
</table>

In the Map browser, you need to specify for each of the Persistent Classes the corresponding Table Maps as described in Table 21:

Table 21. Bank Table Maps

<table>
<thead>
<tr>
<th>Persistent Classes</th>
<th>Table Maps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bank</td>
<td>BANK</td>
</tr>
<tr>
<td>BankAccount</td>
<td>ACCOUNT</td>
</tr>
<tr>
<td>CheckingAccount</td>
<td>CHECKING</td>
</tr>
<tr>
<td>CorporateAccount</td>
<td>PAYEE</td>
</tr>
<tr>
<td>Customer</td>
<td>CUSTOMER</td>
</tr>
<tr>
<td>SavingsAccount</td>
<td>SAVINGS</td>
</tr>
<tr>
<td>TransactionRecord</td>
<td>TRANSRECORD</td>
</tr>
</tbody>
</table>
For each Table Maps item, you indicate the relationship between the class attribute and a table column. In addition, you also give the map type as indicated in Table 22:

Table 22. Bank Property Map Relationship

<table>
<thead>
<tr>
<th>Persistent Classes</th>
<th>Class Attribute</th>
<th>Map Type</th>
<th>Table Column</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bank</td>
<td>primaryKey</td>
<td>Simple</td>
<td>BANKID</td>
</tr>
<tr>
<td></td>
<td>bankName</td>
<td>Simple</td>
<td>BANKNAME</td>
</tr>
<tr>
<td>BankAccount</td>
<td>accountId</td>
<td>Simple</td>
<td>ACCID</td>
</tr>
<tr>
<td></td>
<td>accountType</td>
<td>Simple</td>
<td>ACCTYPE</td>
</tr>
<tr>
<td></td>
<td>balance</td>
<td>Simple</td>
<td>BALANCE</td>
</tr>
<tr>
<td></td>
<td>customerFK</td>
<td>Simple</td>
<td>CUSTID</td>
</tr>
<tr>
<td></td>
<td>bankFK</td>
<td>Simple</td>
<td>BANKID</td>
</tr>
<tr>
<td>CheckingAccount</td>
<td>overdraft</td>
<td>Simple</td>
<td>OVERDRAF</td>
</tr>
<tr>
<td></td>
<td>accountId</td>
<td>Simple</td>
<td>ACCID</td>
</tr>
<tr>
<td>CorporateAccount</td>
<td>billPaymentTitle</td>
<td>Simple</td>
<td>TITLE</td>
</tr>
<tr>
<td></td>
<td>accountId</td>
<td>Simple</td>
<td>ACCID</td>
</tr>
<tr>
<td>SavingAccount</td>
<td>accountId</td>
<td>Simple</td>
<td>ACCID</td>
</tr>
<tr>
<td></td>
<td>minAmount</td>
<td>Simple</td>
<td>MINAMT</td>
</tr>
<tr>
<td>Customer</td>
<td>title</td>
<td>Simple</td>
<td>TITLE</td>
</tr>
<tr>
<td></td>
<td>primaryKey</td>
<td>Simple</td>
<td>CUSTID</td>
</tr>
<tr>
<td></td>
<td>firstName</td>
<td>Simple</td>
<td>FNAME</td>
</tr>
<tr>
<td></td>
<td>password</td>
<td>Simple</td>
<td>PASSWORD</td>
</tr>
<tr>
<td></td>
<td>userid</td>
<td>Simple</td>
<td>USERID</td>
</tr>
<tr>
<td></td>
<td>bankForeignKey</td>
<td>Simple</td>
<td>BANKID</td>
</tr>
<tr>
<td></td>
<td>lastName</td>
<td>Simple</td>
<td>LNAME</td>
</tr>
</tbody>
</table>
We have finished with the implementation of the bank application entity beans. Another part of the business logic is implemented as session beans. We have defined a UserSB which implements customer use cases and an AdminSB which implements administrative use cases.

These two session beans are the subjects of the following chapters.
In this chapter we describe the UserSB session bean, which implements all use cases as seen by the end user.

Customers use the home banking application to perform a set of banking operations from home through the Internet. This application includes making funds transfers, checking account balances, paying bills and more.

It addresses three kinds of accounts: checking, savings and payee. Checking accounts are used to manage current expenses, savings accounts are used to manage personal savings, and payee accounts are used to manage the corporations to which the customer pays bills.

All possible operations are described by sequence diagrams in the following sections.
13.1 The UserSB Sequence Diagrams

To better illustrate the interactions between UserSB and its client, we provide the corresponding sequence diagrams.

UserSB Customer Authentication Sequence Diagram

Before a user can start using any secured system, he is always asked to identify and authenticate himself. For that purpose, the client application of our bank application first must create an instance of UserSB, and then must invoke the authenticate method. (Information about how to create a session bean can be found in Chapter 3, “Developing a Session Bean” on page 19).
The authenticate method uses a user defined finder method on the CustomerHome in order to get an enumeration of customers with the same userId (see Figure 149). Since each userId is unique, only one customer should be returned. Once the customer is found, we store the customerLogin password and object reference for later use.
UserSB Customer GetAccounts Sequence Diagram

When the client asks for a list of the accounts, the getAccounts method delegates the request to the customer object by invoking getBankAccounts shown in Figure 150. The getAccounts() method returns an array of data used by the client for display purposes only. In this case, we return the accountId and the accountType description for each account.

Figure 150. UserSB GetAccounts Sequence Diagram
UserSB Customer Deposit Sequence Diagram

Once the client gets the list of accounts he can perform the usual bank operations. Basically, the UserSB needs to find the corresponding BankAccount instance from the BankAccountHome identified by its key. For information about how to build the key, refer to "Working with the Key Class" on page 130. We do not include the steps for creating the keys in the sequence diagram because of lack of space, but of course you need to create a key first.

As you can see from Figure 151, the deposit business action is delegated to the BankAccount object, while the UserSB is responsible to create the TransactionRecord associated to this bank transaction.

Figure 151. UserSB Deposit Sequence Diagram
**UserSB Customer Withdraw Sequence Diagram**

The flow (Figure 152) here is very similar to the one of the deposit action.

![UserSB Customer Withdraw Sequence Diagram](image)

Figure 152. UserSB Withdraw Sequence Diagram

**UserSB Customer GetBalance Sequence Diagram**

The GetBalance sequence diagram is very simple. When the UserSB receives the getBalance request from the client, it asks to the BankAccountHome to find the right BankAccount corresponding to the specified key. Then it invokes the getBalance() method on the BankAccount instance (see Figure 153).
The client can select an account and ask for the account history by specifying a starting date and an ending date.

For example, the customer wants to see all the account movement starting from March 1, 1999 until March 12, 1999.

UserSB looks for the right account reference. It invokes `getTransactionRecords` method and performs a filtering operation based on the given starting and ending dates before returning the result (see Figure 154).
UserSB Customer Transfer Sequence Diagram

So far, most of the services provided by the UserSB result in delegating the operation to a single entity bean. With the transfer method, the use of a session bean makes more sense. It implements the logic to deal with several entity beans as shown in Figure 155.
The payBill action implies two BankAccounts: the source (the customer account) and the target (the payee account). For example, the customer wants to pay a phone bill to the phone company. To authorize a payBill action we need to authenticate the customer.

You can see from the sequence diagram in Figure 156 that the payBill action is actually a transfer where the target account is always a payee account.
Now we are ready to develop the UserSB.

13.2 Creating the UserSB Session Bean

UserSB is a stateful session bean because it maintains the customer login status information. We assume that now you are familiar with the environment, so we are skipping some details in the following steps:

1. Start VisualAge for Java.
2. Select the ITSOBank project.
3. Create a new package called itso.bank.ejb.sb.user.
4. Create a new EJB called UserSB in the ITSOBank EJB group. Use itso.bank.ejb.sb.user as package name (see Figure 157).
5. Once the UserSB is created, go to the Properties and select \#STATEFUL in the State Management Attribute combo box.
6. Select the UserSBBean class and add the following import statements:

```java
import itso.bank.ejb.*;
import itso.bank.ejb.base.*;
import itso.bank.ejb.relation.*;
import itso.bank.ejb.inheritance.relation.*;
```

7. The UserSB declaration should look like this:

```java
public class UserSBBean extends ITSOSessionBean implements javax.ejb.SessionBean
```

We extend from ITSOSessionBean, which extends itself from itso.bank.ejb.base.ITSOEnterpriseBean. This upper class, with the help of the Homefactory class, manages to find and cache the homes. In this way we save the processing time required to go to the naming services and find the homes.

![Properties](image)

Figure 157. Properties for the UserSB

### Add Fields

The UserSB session bean needs to maintain some information like the login password. We also need values like the reference to the customer who logs in.
In this way, we do not have to get it every time we need it because, since the UserSB is alive, it will be tied to that particular customer session.

We also cache the Home objects for all the classes involved in our use case. In this way we avoid doing a lookup each time we need to deal with a class we have already used.

Use the information provided in Table 23 to Table 31 to create the required fields.

Table 23. Create Field bankAccountHome

<table>
<thead>
<tr>
<th>Field Name</th>
<th>bankAccountHome</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field Type</td>
<td>itso.bank.ejb.BankAccountHome</td>
</tr>
<tr>
<td>Access Modifiers</td>
<td>private</td>
</tr>
<tr>
<td>Other Modifiers</td>
<td>none</td>
</tr>
<tr>
<td>Access with getter and setter methods</td>
<td>checked</td>
</tr>
<tr>
<td>Getter</td>
<td>private</td>
</tr>
<tr>
<td>Setter</td>
<td>(none)</td>
</tr>
<tr>
<td>Container Managed</td>
<td>no</td>
</tr>
</tbody>
</table>

Table 24. Create Field bankHome

<table>
<thead>
<tr>
<th>Field Name</th>
<th>bankHome</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field Type</td>
<td>itso.bank.ejb.BankAccountHome</td>
</tr>
<tr>
<td>Access Modifiers</td>
<td>private</td>
</tr>
<tr>
<td>Other Modifiers</td>
<td>none</td>
</tr>
<tr>
<td>Access with getter and setter methods</td>
<td>checked</td>
</tr>
<tr>
<td>Getter</td>
<td>private</td>
</tr>
<tr>
<td>Setter</td>
<td>(none)</td>
</tr>
<tr>
<td>Container Managed</td>
<td>no</td>
</tr>
</tbody>
</table>
### Table 25. Create Field custKey

<table>
<thead>
<tr>
<th>Field Name</th>
<th>custKey</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filed Type</td>
<td>itso.bank.ejb.CustomerKey</td>
</tr>
<tr>
<td>Access Modifiers</td>
<td>private</td>
</tr>
<tr>
<td>Other Modifiers</td>
<td>none</td>
</tr>
<tr>
<td>Access with getter and setter methods</td>
<td>checked</td>
</tr>
<tr>
<td>Getter</td>
<td>private</td>
</tr>
<tr>
<td>Setter</td>
<td>private</td>
</tr>
<tr>
<td>Container Managed</td>
<td>no</td>
</tr>
</tbody>
</table>

### Table 26. Create Field customerHome

<table>
<thead>
<tr>
<th>Field Name</th>
<th>customerHome</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filed Type</td>
<td>itso.bank.ejb.CustomerHome</td>
</tr>
<tr>
<td>Access Modifiers</td>
<td>private</td>
</tr>
<tr>
<td>Other Modifiers</td>
<td>none</td>
</tr>
<tr>
<td>Access with getter and setter methods</td>
<td>checked</td>
</tr>
<tr>
<td>Getter</td>
<td>private</td>
</tr>
<tr>
<td>Setter</td>
<td>private</td>
</tr>
<tr>
<td>Container Managed</td>
<td>no</td>
</tr>
</tbody>
</table>
Table 27. Create Field customerRef

<table>
<thead>
<tr>
<th>Field Name</th>
<th>customerRef</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filed Type</td>
<td>itso.bank.ejb.Customer</td>
</tr>
<tr>
<td>Access Modifiers</td>
<td>private</td>
</tr>
<tr>
<td>Other Modifiers</td>
<td>none</td>
</tr>
<tr>
<td>Access with getter and setter methods</td>
<td>checked</td>
</tr>
<tr>
<td>Getter</td>
<td>private</td>
</tr>
<tr>
<td>Setter</td>
<td>private</td>
</tr>
<tr>
<td>Container Managed</td>
<td>no</td>
</tr>
</tbody>
</table>

Table 28. Create Field initialCtx

<table>
<thead>
<tr>
<th>Field Name</th>
<th>initialCtx</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filed Type</td>
<td>javax.naming.Context</td>
</tr>
<tr>
<td>Access Modifiers</td>
<td>private</td>
</tr>
<tr>
<td>Other Modifiers</td>
<td>none</td>
</tr>
<tr>
<td>Access with getter and setter methods</td>
<td>checked</td>
</tr>
<tr>
<td>Getter</td>
<td>private</td>
</tr>
<tr>
<td>Setter</td>
<td>private</td>
</tr>
<tr>
<td>Container Managed</td>
<td>no</td>
</tr>
</tbody>
</table>
Table 29. CreateField loginPassword

<table>
<thead>
<tr>
<th>Field Name</th>
<th>loginPassword</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filed Type</td>
<td>String</td>
</tr>
<tr>
<td>Access Modifiers</td>
<td>private</td>
</tr>
<tr>
<td>Other Modifiers</td>
<td>none</td>
</tr>
<tr>
<td>Access with getter and setter methods</td>
<td>checked</td>
</tr>
<tr>
<td>Getter</td>
<td>private</td>
</tr>
<tr>
<td>Setter</td>
<td>private</td>
</tr>
<tr>
<td>Container Managed</td>
<td>no</td>
</tr>
</tbody>
</table>

Table 30. CreateField loginStatus

<table>
<thead>
<tr>
<th>Field Name</th>
<th>loginStatus</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filed Type</td>
<td>boolean</td>
</tr>
<tr>
<td>Access Modifiers</td>
<td>private</td>
</tr>
<tr>
<td>Other Modifiers</td>
<td>none</td>
</tr>
<tr>
<td>Access with getter and setter methods</td>
<td>checked</td>
</tr>
<tr>
<td>Getter</td>
<td>private</td>
</tr>
<tr>
<td>Setter</td>
<td>private</td>
</tr>
<tr>
<td>Container Managed</td>
<td>no</td>
</tr>
</tbody>
</table>
Add New Methods

For directions on how to add a method to an EJB see “Adding Methods to EJB Beans” on page 49.

authenticate

public boolean authenticate(String user, String password) throws java.rmi.RemoteException {

    Customer tempCust = null;
    CustomerHome custHome = getCustomerHome();
    java.util.Enumeration custList = null;
    int numberElements = 0;
    try {
        custList = custHome.findByUserId(user, password);
    } catch (javax.ejb.FinderException e) {

        // Wrong userId and/or bad password, authentication failed
        setLoginState(false);
        return loginStatus;
    }

    while (custList.hasMoreElements()) {
        numberElements++;
        tempCust = (Customer) custList.nextElement();
    }

    if (numberElements > 1) {

        // More than one customer record exist in the database with this user id and password
        // Authentication failed
        setLoginState(false);
    }

    return true;
}

Table 31.  Create Field transactionRecordHome

<table>
<thead>
<tr>
<th>Field Name</th>
<th>transactionRecordHome</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filed Type</td>
<td>itso.bank.ejb.TransactionHome</td>
</tr>
<tr>
<td>Access Modifiers</td>
<td>private</td>
</tr>
<tr>
<td>Other Modifiers</td>
<td>none</td>
</tr>
<tr>
<td>Access with getter and setter methods</td>
<td>checked</td>
</tr>
<tr>
<td>Getter</td>
<td>private</td>
</tr>
<tr>
<td>Setter</td>
<td>private</td>
</tr>
<tr>
<td>Container Managed</td>
<td>no</td>
</tr>
</tbody>
</table>
return loginStatus;
} else
    if (numberElements == 0) {
        // Enumeration is empty, no Customer found
        setLoginState(false);
        return loginStatus;
    }

    // User Id and password are valid, authentication is OK. Now we need to check
    // if only one instance has been returned from the database
    setLoginState(true);
    loginPassword = password;
    setCustomerRef(tempCust);
    return getLoginState();
}

❑ buildBankAccountHome
private void buildBankAccountHome() throws java.rmi.RemoteException {
    bankAccountHome = (BankAccountHome) getHome("BankAccount");
}

❑ buildBankHome
private void buildBankHome() throws RemoteException {
    bankHome = (BankHome) getHome("Bank");
}

❑ buildCustomerHome
private void buildCustomerHome() throws RemoteException {
    customerHome = (CustomerHome) getHome("Customer");
}

❑ buildTransactionRecordHome
private void buildTransactionRecordHome() throws java.rmi.RemoteException {
    transactionRecordHome = (TransactionRecordHome) getHome("TransactionRecord");
}

❑ deposit
public void deposit(String accId, java.math.BigDecimal amt) {
    TransactionRecordKey txKey = null;
    // Get the home interface
    try {
        // Create the BankAccountKey from the ID
        BankAccountKey bAcKey = new BankAccountKey(accId);
        // Find the right BankAccount instance with from the key
        BankAccount
        account = TypeConverter.typeCast(getBankAccountHome().findByPrimaryKey(bAcKey));
        // call the deposit method on the account
        account = TypeConverter.typeCast(getBankAccountHome().findByPrimaryKey(bAcKey));
        // call the deposit method on the account
    }
account.deposit(amt);
// Since the deposit action is a transaction made on the account we need to create a
TransactionRecord
// and store i in the DataBase.

TransactionRecordHome trHome =
(TransactionRecordHome) getHome("TransactionRecord");

try {
    // Now that the withdraw has been processed we need to create an instance of
TransactionRecord
    java.sql.Timestamp transId = new java.sql.Timestamp(System.currentTimeMillis());
txKey = new TransactionRecordKey(transId);
    TransactionRecord newRecord = trHome.create(txKey, account, amt,"C");
} catch (javax.ejb.CreateException e) {
    setRollbackFlag();

} catch (BankTransactionException e1) {
    System.out.println(e1.getMessage());
    // setRollbackFlag();

} catch (Exception e) {
    e.printStackTrace(System.out);
// endtry
}

❑ getBalance
public java.math.BigDecimal getBalance( String accountId) {
    try {
        //this is only needed for testing.
        BankAccountKey bAcKey = new BankAccountKey(accountId);
        BankAccount account=
        TypeConverter.typeCast(getBankAccountHome().findByPrimaryKey(bAcKey));
        return account.getBalance();
    }
    catch(Exception except) {
        except.printStackTrace(System.out);
        //throw wrapperNonRemoteException("UserSB.getCustomer(String)", except);
        return null;
    }
}

❑ getCustomer
public Customer getCustomer(String customerId) throws RemoteException {
    CustomerHome cHome =null;
}

328 Enterprise JavaBeans Development Using VisualAge for Java
try {
    //this is only needed for testing.
    CustomerKey cKey = new CustomerKey(customerId);
    setCustomerRef(getCustomerHome().findByPrimaryKey(cKey));
    return (customerRef);
}

//this is only needed for testing.
CustomerKey cKey = new CustomerKey(customerId);
setCustomerRef(getCustomerHome().findByPrimaryKey(cKey));
return (customerRef);

} catch(Exception except) {
    except.printStackTrace(System.out);
    //throw wrapperNonRemoteException("UserSB.getCustomer(String)", except);
    return null;
}

} getHistory

public String[][] getHistory(String accId, java.util.Date startDate, java.util.Date endDate, String type) {

    // Get the home interface for the Bank Account
    TransactionRecordHome trHome =null;
    try {
        // Create the BankAccountKey from the ID
        BankAccountKey bAcKey = new BankAccountKey(accId);
        //Find the right BankAccount instance with from the key
        BankAccount account =
            TypeConverter.typeCast(getBankAccountHome().findByPrimaryKey(bAcKey));
        // Vector to contain the right transaction to return
        java.util.Vector v = new java.util.Vector();
        //call the getTransactionRecords to get all transaction related to this account
        java.util.Enumeration transEnum = account.getTransactionRecords();

        while (transEnum.hasMoreElements()) {
            TransactionRecord tRecord = (TransactionRecord)transEnum.nextElement();
            if (startDate != null){
                if ((java.util.Date)(tRecord.getTransactionID())).before( startDate){
                    continue;
                }
            }
            if (endDate != null){
                if ((java.util.Date)(tRecord.getTransactionID())).after( endDate){
                    continue;
                }
            }
            if (type != null){
                if (!type.equals( tRecord.getTranstype())){
                    continue;
                }
            }
            v.addElement( tRecord);
        }
        return v.toArray(new TransactionRecord[v.size()]);
    }
    return null;
}

Bank Implementation UserSB  329
```java
String[][] transDataArray = new String[v.size()][3];
if (v.size() > 0) {
    for (int i = 0; i < v.size(); i++) {
        transDataArray[i][0] = ((java.util.Date)(((TransactionRecord)(v.elementAt(i))).getTransactionID())).toString();
        transDataArray[i][1] = (((TransactionRecord)(v.elementAt(i))).getTranstype());
        transDataArray[i][2] = (((TransactionRecord)(v.elementAt(i))).getTransamt()).toString();
    }
}
return transDataArray;

/**********
catch(Exception except) {
    except.printStackTrace(System.out);
    return null;
}
}   

❑ paybill

public void paybill(String source, String destination, java.math.BigDecimal amount, String password) throws java.rmi.RemoteException {
    CustomerHome custHome = getCustomerHome();
    BankAccountKey keyObject = null;
    // First, we need to verify the validity of the password
    if (!(password.equals(this.loginPassword))) {
        // throw new BankException();
    } else {
        // Proceed with the transfer
        transfer(source, destination, amount);
    }
}

❑ transfer

public void transfer(String source, String destination, java.math.BigDecimal amount) throws java.rmi.RemoteException {
    BankAccountHome bankAccountHome = getBankAccountHome();
    TransactionRecordHome trxHome = getTransactionRecordHome();
    BankAccountKey sourceKeyObject = null;
    BankAccountKey destKeyObject = null;
    TransactionRecordKey txKey = null;
```
try {
    sourceKeyObject = new BankAccountKey(source);
    BankAccount sourceAccount =
    TypeConverter.typeCast(bankAccountHome.findByPrimaryKey(sourceKeyObject));
    destKeyObject = new BankAccountKey(destination);
    BankAccount targetAccount =
    TypeConverter.typeCast(bankAccountHome.findByPrimaryKey(destKeyObject));

    // Use the next line with inheritance implemented
    sourceAccount.withdraw(amount);

    // Use the next block if no inheritance is implemented (Using one object and one table)
    // sourceAccount.withdraw(amount, sourceAccount.getAccountType());
    try {
        // Now that the withdraw has been processed we need to create an instance of
        TransactionRecord
        java.sql.Timestamp transId = new java.sql.Timestamp(System.currentTimeMillis());
        txKey = new TransactionRecordKey(transId);
        TransactionRecord newRecord = trxHome.create(txKey, sourceAccount, amount,"T");
        newRecord.setTransactingAccount(targetAccount);
    } catch (javax.ejb.CreateException e) {
        // setRollbackFlag();
    }

    targetAccount.deposit(amount);
    try {
        // Now that the withdraw has been processed we need to create an instance of
        TransactionRecord
        java.sql.Timestamp transId = new java.sql.Timestamp(System.currentTimeMillis());
        txKey = new TransactionRecordKey(transId);
        TransactionRecord newRecord = trxHome.create(txKey, targetAccount, amount,"F");
        newRecord.setTransactingAccount(sourceAccount);
    } catch (javax.ejb.CreateException e) {
        // setRollbackFlag();
    }
} catch (FinderException e) {
    // setRollbackFlag();
}
catch (BankTransactionException e1) {
    System.out.println(e1.getMessage());
    // setRollbackFlag();
}
catch (Exception ex) {
    System.out.println(ex.getMessage());
}
}

withdraw
public void withdraw(String account, java.math.BigDecimal amount) throws java.rmi.RemoteException {
    BankAccountHome bankAccountHome = getBankAccountHome();
    TransactionRecordHome trxHome = getTransactionRecordHome();
    BankAccountKey keyObject = null;
    TransactionRecordKey txKey = null;
    try {
        keyObject = new BankAccountKey(account);
        BankAccount sourceAccount = TypeConverter.typeCast(bankAccountHome.findByPrimaryKey(keyObject));
    } catch (FinderException e) {
        setRollbackFlag();
        return;
    } catch (BankTransactionException e1) {
        System.out.println(e1.getMessage());
        setRollbackFlag();
        return;
    }
    // Use the next line with inheritance implemented
    sourceAccount.withdraw(amount);
    // Use the next block if no inheritance is implemented (Using one object and one table)
    // sourceAccount.withdraw(amount, sourceAccount.getAccountType());
    try {
        // Now that the withdraw has been processed we need to create an instance of
        TransactionRecord
        java.sql.Timestamp transId = new java.sql.Timestamp(System.currentTimeMillis());
        txKey = new TransactionRecordKey(transId);
        TransactionRecord newRecord = trxHome.create(txKey, sourceAccount, amount, "D");
    } catch (javax.ejb.CreateException e) {
        // setRollbackFlag();
        return;
    } catch (FinderException e) {
        // setRollbackFlag();
        return;
    } catch (BankTransactionException e1) {
        System.out.println(e1.getMessage());
        // setRollbackFlag();
        return;
    }
}

Modify the Generated Methods

❑ getBank

private Bank getBank(String bankId) throws RemoteException {
    try {
        return getBankHome().findByPrimaryKey(new BankKey(bankId));
    } catch (Exception except) {
        throw wrapperNonRemoteException("UseCaseSB.getBank(String)", except);
    }
}

332 Enterprise JavaBeans Development Using VisualAge for Java
getBankHome

private BankHome getBankHome() throws RemoteException {
    if (bankHome == null)
        buildBankHome();
    return bankHome;
}

getCheckingAccount

private CheckingAccount getCheckingAccount(String bankingAccountId) throws RemoteException {
    try {
        return getCheckingAccountHome().findByPrimaryKey(new BankAccountKey(bankingAccountId));
    }
    catch(Exception except) {
        throw wrapperNonRemoteException("UseCaseSB.getCheckingAccount(String bankingAccountId)", except);
    }
}

getCheckingAccountHome

private CheckingAccountHome getCheckingAccountHome() throws RemoteException {
    if (checkingAccountHome == null)
        buildCheckingAccountHome();
    return checkingAccountHome;
}

getCorporateAccount

private CorporateAccount getCorporateAccount(String bankingAccountId) throws RemoteException {
    try {
        return getCorporateAccountHome().findByPrimaryKey(new BankAccountKey(bankingAccountId));
    }
    catch(Exception except) {
        throw wrapperNonRemoteException("UseCaseSB.getCorporateAccount(String bankingAccountId)", except);
    }
}

getCorporateAccountHome

private CorporateAccountHome getCorporateAccountHome() throws RemoteException {
    if (corporateAccountHome == null)
        buildCorporateAccountHome();
    return corporateAccountHome;
}

customer

private BankImplementation UserSB
private Customer getCustomer(String customerId) throws RemoteException {
    try {
        return getCustomerHome().findByPrimaryKey(new CustomerKey(customerId));
    } catch (Exception except) {
        throw wrapperNonRemoteException("UseCaseSB.getCustomer(String)", except);
    }
}

❑ getCustomerHome

private CustomerHome getCustomerHome() throws RemoteException {
    if (customerHome == null)
        buildCustomerHome();
    return customerHome;
}

❑ getSavingsAccount

private SavingsAccount getSavingsAccount(String bankingAccountId) throws RemoteException {
    try {
        return getSavingsAccountHome().findByPrimaryKey(new BankAccountKey(bankingAccountId));
    } catch (Exception except) {
        throw wrapperNonRemoteException("UseCaseSB.getSavingsAccount(String bankingAccountId)", except);
    }
}

❑ getSavingsAccountHome

private SavingsAccountHome getSavingsAccountHome() throws RemoteException {
    if (savingsAccountHome == null)
        buildSavingsAccountHome();
    return savingsAccountHome;
}

Add Methods to the Remote Interface

Select the following methods and add them to the bean remote interface. For directions on how to add a method to the remote interface, see “Adding Methods to the Remote Interface” on page 54.

❑ createBank
❑ createCheckingAccount
❑ createCorporateAccount
❑ createCustomer
Generate Deployed Code

Once you are done you can generate the deployed code. You may notice that in addition to the classes generated so far (see “Generate Deployment Code inside VisualAge for Java” on page 65), there are some new classes in the Types panes (see Figure 158).

![Image of Workbench showing UserSB generated code]

Figure 158. UserSB Generated Code
The StringX?ArrayHelper and the StringX?ArrayHolder have been generated because we have used arrays of string as return type for getAccounts and getHistory methods. These classes are used to marshal the arrays before sending them through the ORB.

The UserSB is now ready for use by a client application. You can also use the test client to verify the behavior of the UserSB. (See “Running the Generated Test Client” on page 73.) You need to start with the first business method: authenticate.
14 Bank Implementation AdminSB

As we have explained in previous chapters, the bank application implementation covers two different aspects: user and administration interactions.

In this chapter, we look to the administration part. The session bean AdminSB offers to client applications a set of methods based on use cases required to manage the bank.
14.1 Developing the AdminSB Session Bean

Create a new AdminSB session bean in the ITSOBank group, specifying the following information:

- **Bean name**: AdminSB
- **Bean type**: Session Bean
- **Project**: ITSOBank
- **Package**: itso.bank.ejb.sb.admin
- **Class**: AdminSBBean
- **Superclass**: itso.bank.ejb.base.ITSOSessionBeanEJB
- **Home Interface**: AdminSBHome
- **EJB Remote Interface**: AdminSB
- **EJB Key Class or Field**: none
- **import statements**:
  - itso.bank.ejb.*
  - itso.bank.ejb.base.*

Add Fields

The AdminSB requires the creation of the following fields:

- customerHome
- bankHome
- savingsAccountHome
- checkingAccountHome
- checkingAccountHome

with the same characteristics as given in Table 32.
Add New Methods

❑ getBank

private Bank getBank(String bankId) throws RemoteException {
    try {
        return getBankHome().findByPrimaryKey(new BankKey(bankId));
    } catch (Exception except) {
        throw wrapperNonRemoteException("UseCaseSB.getBank(String)", except);
    }
}

❑ getCheckingAccount

private CheckingAccount getCheckingAccount(String bankingAccountId) throws RemoteException {
    try {
        return getCheckingAccountHome().findByPrimaryKey(new BankAccountKey(bankingAccountId));
    } catch (Exception except) {
        throw wrapperNonRemoteException("UseCaseSB.getCheckingAccount(String)", except);
    }
}
getCorporateAccount

private CorporateAccount getCorporateAccount(String bankingAccountId) throws RemoteException {
    try {
        return getCorporateAccountHome().findByPrimaryKey(new BankAccountKey(bankingAccountId));
    } catch(Exception except) {
        throw wrapperNonRemoteException("UseCaseSB.getCorporateAccount(String bankingAccountId)", except);
    }
}

getCustomer

private Customer getCustomer(String customerId) throws RemoteException {
    try {
        return getCustomerHome().findByPrimaryKey(new CustomerKey(customerId));
    } catch(Exception except) {
        throw wrapperNonRemoteException("UseCaseSB.getCustomer(String)", except);
    }
}

getSavingsAccount

private SavingsAccount getSavingsAccount(String bankingAccountId) throws RemoteException {
    try {
        return getSavingsAccountHome().findByPrimaryKey(new BankAccountKey(bankingAccountId));
    } catch(Exception except) {
        throw wrapperNonRemoteException("UseCaseSB.getSavingsAccount(String bankingAccountId)", except);
    }
}

buildBankHome

private void buildBankHome() throws RemoteException {
    bankHome = (BankHome) getHome("Bank");
}

buildCheckingAccountHome

private void buildCheckingAccountHome() throws RemoteException {
    checkingAccountHome = (CheckingAccountHome) getHome("CheckingAccount");
}

buildCorporateAccountHome
private void buildCorporateAccountHome() throws RemoteException {
    corporateAccountHome = (CorporateAccountHome) getHome("CorporateAccount");
}

private void buildCustomerHome() throws RemoteException {
    customerHome = (CustomerHome) getHome("Customer");
}

private void buildSavingsAccountHome() throws RemoteException {
    savingsAccountHome = (SavingsAccountHome) getHome("SavingsAccount");
}

public void createBank(String bankId, String bankName) throws java.rmi.RemoteException {
    try {
        getBankHome().create(new BankKey(bankId), bankName);
    } catch(Exception e) {
        throw wrapperNonRemoteException("AdminSBBean.createBank(String bankId, String
bankName)", e);
    }
}

public void createCheckingAccount(String accountId, String customerId, BigDecimal
overdraftLimit) throws java.rmi.RemoteException {
    try {
        Customer customerOfCheckingAccount = getCustomer(customerId);
        Bank bankOfCustomer = customerOfCheckingAccount.getBank();
        CheckingAccount newCheckingAccount = getCheckingAccountHome().create(new
BankAccountKey(accountId), customerOfCheckingAccount, bankOfCustomer);
        newCheckingAccount.setOverDraftLimit(overdraftLimit);
    } catch(Exception e) {
        throw wrapperNonRemoteException("AdminSBBean.createCheckingAccount(String
accountId, String customerId, BigDecimal overdraftLimit)", e);
    }
}

public void createCorporateAccount(String accountId, String customerId, String billPaymentTitle) throws java.rmi.RemoteException {
    try {
        Customer customerOfCorporateAccount = getCustomer(customerId);
        Bank bankOfCustomer = customerOfCorporateAccount.getBank();
        CorporateAccount newCorporateAccount = getCorporateAccountHome().create(new
BankAccountKey(accountId), customerOfCorporateAccount, bankOfCustomer);
    }
}
newCorporateAccount.setBillPaymentTitle(billPaymentTitle);
}
catch(Exception e) {
    throw wrapperNonRemoteException("AdminSBBean.createCorporateAccount(String accountId, String customerId, String billPaymentTitle)", e);
}
}

❑ createCustomer

public void createCustomer(String customerId, String customerTitle, String customerFirstName,
String customerLastName, String customerUserid, String customerPassword, String bankId)
throws RemoteException {
    try {
        Customer newCustomer = null;
        Bank bankOfCustomer = getBank(bankId);
        newCustomer = getCustomerHome().create(new CustomerKey(customerId),
customerTitle, customerFirstName, customerLastName, customerUserid, customerPassword,
bankOfCustomer);
    } catch(Exception e) {
        throw wrapperNonRemoteException("AdminSBBean.createCustomer(String customerId,
String customerTitle, String customerFirstName, String customerLastName, String customerUserid, String customerPassword, String bankId)", e);
    }
}

❑ createSavingsAccount

public void createSavingsAccount(String accountId, String customerId, BigDecimal minimumBalance) throws java.rmi.RemoteException {
    try {
        Customer customerOfSavingsAccount = getCustomer(customerId);
        Bank bankOfCustomer = customerOfSavingsAccount.getBank();
        SavingsAccount newSavingsAccount = getSavingsAccountHome().create(new BankAccountKey(accountId), customerOfSavingsAccount, bankOfCustomer);
        newSavingsAccount.setMinimumBalance(minimumBalance);
    } catch(Exception e) {
        throw wrapperNonRemoteException("AdminSBBean.createSavingsAccount(String accountId, String customerId, BigDecimal minimumBalance)", e);
    }
}

❑ deleteBank

public void deleteBank(String bankId) throws java.rmi.RemoteException {
    try {
        Bank bankToDelete = getBank(bankId);
        bankToDelete.remove();
    } catch(Exception e) {

deleteCheckingAccount

public void deleteCheckingAccount(String bankAccountId) throws java.rmi.RemoteException {
    try {
        CheckingAccount checkingAccountToDelete = getCheckingAccount(bankAccountId);
        checkingAccountToDelete.remove();
    } catch(Exception e) {
        throw wrapperNonRemoteException("AdminSBBean.deleteCheckingAccount(String bankAccountId)", e);
    }
}

deleteCorporateAccount

public void deleteCorporateAccount(String bankAccountId) throws java.rmi.RemoteException {
    try {
        CorporateAccount corporateAccountToDelete = getCorporateAccount(bankAccountId);
        corporateAccountToDelete.remove();
    } catch(Exception e) {
        throw wrapperNonRemoteException("AdminSBBean.deleteCorporateAccount(String bankAccountId)", e);
    }
}

deleteCustomer

public void deleteCustomer(String customerId) throws java.rmi.RemoteException {
    try {
        Customer customerToDelete = getCustomer(customerId);
        customerToDelete.remove();
    } catch(Exception e) {
        throw wrapperNonRemoteException("AdminSBBean.deleteCustomer(String customerId)", e);
    }
}

deleteSavingsAccount

public void deleteSavingsAccount(String bankAccountId) throws java.rmi.RemoteException {
    try {
        SavingsAccount savingsAccountToDelete = getSavingsAccount(bankAccountId);
        savingsAccountToDelete.remove();
    } catch(Exception e) {
        throw wrapperNonRemoteException("AdminSBBean.deleteSavingsAccount(String bankAccountId)", e);
    }
}
throw wrapperNonRemoteException("AdminSBBean.deleteSavingsAccount(String bankAccountId)", e);
}

Modify the Generated Methods

❑ getBankHome

private BankHome getBankHome() throws RemoteException {
    if (bankHome == null)
        buildBankHome();
    return bankHome;
}

❑ getCheckingAccountHome

private CheckingAccountHome getCheckingAccountHome() throws RemoteException {
    if (checkingAccountHome == null)
        buildCheckingAccountHome();
    return checkingAccountHome;
}

❑ getCorporateAccountHome

private CorporateAccountHome getCorporateAccountHome() throws RemoteException {
    if (corporateAccountHome == null)
        buildCorporateAccountHome();
    return corporateAccountHome;
}

❑ getCustomerHome

private CustomerHome getCustomerHome() throws RemoteException {
    if (customerHome == null)
        buildCustomerHome();
    return customerHome;
}

❑ getSavingsAccountHome

private SavingsAccountHome getSavingsAccountHome() throws RemoteException {
    if (savingsAccountHome == null)
        buildSavingsAccountHome();
    return savingsAccountHome;
}
Add Methods to the Remote Interface

Select the following methods and add them to the bean remote interface. For directions on how to add a method to the remote interface, see “Adding Methods to the Remote Interface” on page 54.

- createBank
- createCheckingAccount
- createCorporateAccount
- createCustomer
- createSavingsAccount
- deleteBank
- deleteCheckingAccount
- deleteCorporateAccount
- deleteCustomer
- deleteSavingsAccount

The AdminSB is almost ready for testing the bank administration.

Generate Deployed Code

You still need to generate the deployed code. The next step is to run and test the application. You can use the test client to verify the behavior of the AdminSB. (See “Running the Generated Test Client” on page 73.)

The very last step will be to package the application and deploy it on any EJB server.

If we make the assumption that the administration client application is used in the intranet, then it could be developed as an EJB client application using JNDI and RMI or a pure CORBA application. The larger network bandwidth should allow the development of a GUI highly interactive based on swing classes for example.
A Hints and Tips

A.1 Importing EJB Beans from a Jar File

Once you have added one or more EJB groups to the EJBs page, you can add EJB beans to the EJB groups by:

- Importing EJB beans from EJB jar files
- Creating new EJB beans
- Retrieving existing EJB beans from the repository

This topic discusses how to add EJB beans by importing EJB beans from a jar file. Information about importing EJB beans into a repository is found in A.2, “Moving or Copying EJB Beans between Repositories” on page 353.

Note that if an EJB bean you plan to import already exists in the EJBs page, it will not overwrite the existing EJB bean. Instead, you will receive an error message indicating that the EJB bean already exists.

To import EJB beans from a jar file:
1. In the EJBs pane of the EJBs page, select the EJB group that you want to contain the imported EJB beans.

2. Click mouse button 2, then select **Import EJBs** from the pop-up menu (see Figure 159). The **Import from an EJB Jar File** SmartGuide appears.

3. Beside the **Filename** field, type the name of the jar file you want to import.

   Alternatively, click the **Browse** button. A dialog appears. In the dialog, navigate to the jar file, select it, and click **OK**. The SmartGuide marks for importation all of the types and beans found in the jar file.

   In the CDROM \Part1Samples\SessionBean we provide a .jar file called CreateCustomerInfo.jar. This is an EJB jar file, so it does not
contain an EJS jar file. Select it to try to import an EJB jar file (see Figure 160).

Figure 160. Import from CreateCustomerInfo EJB Jar File

4. To select individual file types to import, click one or more of the following check boxes:
   - **beans**—Import all the beans in the jar file.
   - **.class**—Import all the bytecode files in the jar file.
   - **.java**—Import all the source code files.
   - **resource**—Import all the non-.java and non-.class files
• **Details**—By default, all the selected file types in the jar file are imported. Click the **Details** button next to a file type to see a list of the files in the jar file, and to specify individual files to import.

5. If needed, specify additional options using the **Options** check boxes as follows:

• **Create new/scratch editions of versioned projects/packages**—If you are importing into a versioned EJB group, and you want to create a new or scratch edition of the versioned EJB group, click this check box. If you haven’t done a version of our ITSO BANK project you will not need this option.

• **Overwrite existing resource files without warning**—If you do not want to be warned about overwriting existing resource files in the EJB group into which you are importing, click this check box.

• **Version imported classes and new editions of packages/projects**—If you want to automatically version imported classes and the projects and packages they change, click this check box. Also, specify either automatic naming for the new version or a name of your own choosing by selecting either the **Name automatically** (recommended) radio button or the **Version name** radio button.

6. Click Finish.

At this point, your Ejb workspace should look like in Figure 161:
Since we have imported an EJB jar file in order to test it you have to generate the deployed code first. To do this see “Generate deployment code inside VisualAge for Java” on page 55.

A.2 Moving or Copying EJB Beans between Repositories

To move or copy EJBs from one repository to another, do the following:

1. Go to the Projects page in the Workbench.
2. Select the project that contains the packages associated with your EJB group or EJB bean. These packages are:
   - Reserved package
   - Package containing the EJB classes, interfaces, schemas, and maps

Figure 161. EJB Workspace after Importing and EJB .jar
Note that if these packages are spread across more than one project, you can expand these projects and do a multiple selection of the packages. However, we recommend that these associated packages be kept in a single project.

7. Version the project and export it as a .dat file into the target repository. (To open the export dialog, click on File - Export in the Packages page.)

Note: If the target repository is on a different machine, once you export the project to the .dat file, move the .dat file to the other machine. Then, import the .dat file into the repository where you want to locate the project. (To open the import dialog, click on File - Import in the Packages page.)

### A.3 Updating WebSphere Class Path

In the WebSphere Application server administration left pane, choose **Setup-Java Engine** (see Figure 162). Select the **Path** tab and add the **C:\customerObject.jar** in the **Application Server Classpath** field.

Figure 162. WebSphere Application Server Java Engine Setup
For these changes to take effect, you must stop and start WebSphere Advanced Edition 2.0.

A.4 Starting and Stopping WebSphere Advanced Edition 2.0

- To stop WebSphere:
  - Shut down the Web server (from Start - Settings - Control Panel - Services and select Lotus Domino Go Webserver, then click the Stop button.)
  - Shut down the WebSphere ServletService to stop Application Server. (Select the WebSphere Servlet Service from Start - Settings - Control Panel - Services and push the Stop button.)

- To start WebSphere:
  - The server starts automatically when you start your Web server (from Start - Settings - Control Panel - Services and select Lotus Domino Go Webserver, then click the Start button.)
B Transaction Samples Appendix

B.1 Installing and Running the Transaction Samples

B.1.1 Creating the DB2 Database

1. You need to copy the following file from the CD-ROM to your SQLLIB\BIN directory (this is the directory where the DB2 executable files reside):
   • TRX.DDL
   These files come with the sample code provided on the CD-ROM.

2. Click on Start - Programs - DB2 - Command Window to start the DB2 command interpreter.

3. Type:
   
   db2 -f TRX.DDL

   Then hit Enter. This creates the SampleDB database.
Type:
exit

Then hit Enter to close the DB2 command interpreter.

B.1.2 Importing the transaction.dat Repository

1. In the Projects page of VisualAge for Java, click the mouse button 2. Select Import....
2. VisualAge for Java displays the Import SmartGuide. Select Repository as the import source. Click on the Next > button.
3. Select Local repository, then click on the Browse button. Select the your CD-Rom drive. From this drive go to the Part1Samples\transaction directory. Select the transaction.dat file, then click on the Open button.
4. Select the Projects radio button, from What do you want to import ?.
5. Click on the Details button. Choose the TransactionSamples project and click OK.
6. Click on the Finish button from the Import SmartGuide. VisualAge for Java imports all the classes.

B.1.3 Adding the TransactionSamples Project to the Workspace

1. In the Projects page of VisualAge for Java click the mouse button 2. Select Add - Project....
2. VisualAge for Java displays the Add Project SmartGuide. Click on Add projects from the repository radio button.
3. Select the TransactionSamples project. Click on the Finish button.
4. The TransactionSamples project now appears in the Projects page of your workspace.
5. Click on the EJBs page. You should see the ITSOEJBSamples EJB Group with three EJB beans in it, as shown in Figure 163
B.1.4 Generating the Deployed Code and Running the Samples

1. Click on the **EJBs** page. You should see the ITSOEJBSamples EJB Group with three 
EJB beans in it, as shown in Figure 163.

2. Select the **ITSOEJBSamples** EJB Group from the **EJBs** list.

3. Click on the mouse button 2, select **Generate - Deployed code** from the popup 
menu. Wait until the code generation is completed.

4. Select the **ITSOEJBSamples** EJB Group from the **EJBs** list.

5. Select **EJBs - Add To - Server Configuration** from the menu.

6. Start the **Location Service Daemon**.

7. Select the **EJB Server (server1)** from the **Servers** list. Click on the mouse button 2 
and select **Start server**.

8. Switch to the VisualAge for Java Console to check if your server is started.

9. Switch back to your workspace. Click on the **Projects** page. Expand the 
**com.alain.cmpl** package. Select the **SimpleClient** class and click on the **Run** icon 
from the toolbar.

10. Wait for a moment and VisualAge for Java’s console will appear and ask you for 
some input.

11. The output done from the SimpleClient and the Beans can be seen on the Console.
B.2 Output Samples

Scenario 1—myTransactionTest.test1() Sample Output

TransactionTest Output: -------Starting Client-demarcated commited transaction Scenario...
TransactionTest Output: -------Retrieving initial context...
TransactionTest Output: -------Set properties to: javax.naming.Context.PROVIDER_URL, iiop://cobalt:900
TransactionTest Output: -------Set properties to: com.ibm.jndi.CosNaming.CNInitialContextFactory
TransactionTest Output: -------Invoke: javax.naming.InitialContext(properties)
TransactionTest Output: Retrieving the home interface...
TransactionTest Output: Retrieving User Transaction...
TransactionTest Output: Transaction Status after mySessionCtx.getUserTransaction() is: 6
TransactionTest Output: ******************* TransactionTest: afterBegin() **************88
TransactionTest Output: Transaction Status after myTransaction.begin() is: 0
TransactionTest Output: CMPSample1 Object...
TransactionTest Output: Enter key value:
TransactionTest Output: Enter state value:
TransactionTest Output: Searching for existing object with this key: 1
TransactionTest Output: Object 1 found.
CMPSample1 Output: Enter key value for CMP2 Entity Bean:
CMPSample1 Output: Enter state value:
CMPSample1 Output: -------Retrieving initial context...
CMPSample1 Output: -------Set properties to: javax.naming.Context.PROVIDER_URL, iiop://cobalt:900
CMPSample1 Output: -------Set properties to: com.ibm.jndi.CosNaming.CNInitialContextFactory
CMPSample1 Output: -------Invoke: javax.naming.InitialContext(properties)
CMPSample1 Output: CMPSample2Bean
CMPSample1 Output: Retrieving the home interface...
CMPSample1 Output: Searching for existing object with this key: 1
CMPSample1 Output: Object 1 found.
CMPSample1 Output: State updated.
TransactionTest Output: State updated.
TransactionTest Output: ********************* TransactionTest: beforeCompletion() **************
CMPSample1 Output: ********************* CMPSample1: beforeCompletion() **************
CMPSample2 Output: ********************* CMPSample2: beforeCompletion() **************
TransactionTest Output: ************************** TransactionTest: afterCompletion() ************
CMPSample1 Output: ************************** CMPSample1: afterCompletion() ************
CMPSample2 Output: ************************** CMPSample2: afterCompletion() ************
TransactionTest Output: Transaction Status after myTransaction.commit() is :6
### Scenario 1—myTransactionTest.test2() Sample Output

Transaction Test Output: **Starting Client-demarcated rolled back transaction Scenario...**
Transaction Test Output: **Retrieving initial context...**
Transaction Test Output: **Set properties to: javax.naming.Context.PROVIDER_URL, iiop://cobalt:900**
Transaction Test Output: **Set properties to: com.ibm.jndi.CosNaming.CNInitialContextFactory**
Transaction Test Output: **Invoke: javax.naming.InitialContext(properties)**
Transaction Test Output: **Retrieving the home interface...**
Transaction Test Output: **Transaction Test: afterBegin()**
Transaction Test Output: **Transaction Status after myTransaction.begin() is: 0**
Transaction Test Output: **CMPSample1 Object...**
  Transaction Test Output: **Enter key value:**
  Transaction Test Output: **Enter state value:**
  Transaction Test Output: **Searching for existing object with this key: 1**
  Transaction Test Output: **Object 1 found.**
CMPSample1 Output: **Enter key value for CMP Entity Bean:**
CMPSample1 Output: **Enter state value:**
  CMPSample1 Output: **Searching for existing object with this key: 1**
  CMPSample1 Output: **Object 1 found.**
CMPSample1 Output: **State updated.**
Transaction Test Output: **Transaction Status after myTransaction.rollback() is: 6**

---

**Transaction Samples Appendix**

361
Scenario 1—myTransactionTest.test3() Sample Output
Scenario 2—myTransactionTest.test1() Sample Output

Transaction Test Output: -----Starting Client-demarcated committed transaction Scenario...
Transaction Test Output: -----Retrieving initial context...
Transaction Test Output: -----Set properties to: javax.naming.Context.PROVIDER_URL, iiop://cobalt:900
Transaction Test Output: -----Set properties to: com.ibm.jndi.CosNaming.CNInitialContextFactory
Transaction Test Output: -----Invoke: javax.naming.InitialContext(properties)
Transaction Test Output: Retrieving the home interface...
Transaction Test Output: Retrieving User Transaction...
Transaction Test Output: Transaction Status after mySessionCtx.getUserTransaction() is: 6
Transaction Test Output: ******************* TransactionTest: afterBegin() **************88
Transaction Test Output: Transaction Status after myTransaction.begin() is: 0
Transaction Test Output: CMPSample1 Object...
Transaction Test Output: Enter key value:
Transaction Test Output: Enter state value:
Transaction Test Output: Searching for existing object with this key: 1
Transaction Test Output: Object 1 found.
CMPSample1 Output: ******************* CMPSample1: afterBegin() **************88
CMPSample1 Output: Enter key value for CMP2 Entity Bean:
CMPSample1 Output: Enter state value:
CMPSample1 Output: -------Retrieving initial context...
CMPSample1 Output: -------Set properties to: javax.naming.Context.PROVIDER_URL, iiop://cobalt:900
CMPSample1 Output: -------Set properties to: com.ibm.jndi.CosNaming.CNInitialContextFactory
CMPSample1 Output: -------Invoke: javax.naming.InitialContext(properties)
CMPSample1 Output: CMPSample2Bean
CMPSample1 Output: Retrieving the home interface...
CMPSample1 Output: Searching for existing object with this key: 1
CMPSample1 Output: Object 1 found.
CMPSample2 Output: ******************* CMPSample2: afterBegin() **************88
CMPSample1 Output: State updated.
Transaction Test Output: State updated.
Transaction Test Output: ********************* TransactionTest: beforeCompletion() **************
CMPSample1 Output: ********************* CMPSample1: beforeCompletion() **************
CMPSample2 Output: ********************* CMPSample2: beforeCompletion() **************
Transaction Test Output: Transaction Status after myTransaction.commit() is: 6
Scenario 2—myTransactionTest.test2() Sample Output
Scenario 2—myTransactionTest.test2() Sample Output
Scenario 3—Sample Output

TransactionTest Output: --------Starting Client-demarcated committed transaction Scenario...
TransactionTest Output: --------Retrieving initial context...
TransactionTest Output: --------Set properties to: com.ibm.jndi.CosNaming.CNInitialContextFactory
TransactionTest Output: --------Invoke: javax.naming.InitialContext(properties)
TransactionTest Output: Retrieving the home interface...
TransactionTest Output: Retrieving User Transaction...
TransactionTest Output: Transaction Status after mySessionCtx.getUserTransaction() is: 6
TransactionTest Output: ***************************************** TransactionTest: afterBegin() *****************************************
TransactionTest Output: Transaction Status after myTransaction.begin() is: 0
TransactionTest Output: CMPSample1 Object...
TransactionTest Output: Enter key value: 
TransactionTest Output: Enter state value: 
TransactionTest Output: Searching for existing object with this key: 1
TransactionTest Output: Object not found. Creating a new instance...
TransactionTest Output: New Object instance created!
CMPSample1 Output: Enter key value for CMP2 Entity Bean: 
CMPSample1 Output: Enter state value: 
CMPSample1 Output: ------Retrieving initial context...
CMPSample1 Output: ------Set properties to: javax.naming.Context.PROVIDER_URL, iiop://cobalt:900
CMPSample1 Output: ------Set properties to: com.ibm.jndi.CosNaming.CNInitialContextFactory
CMPSample1 Output: ------Invoke: javax.naming.InitialContext(properties)
CMPSample1 Output: CMPSample2Bean
CMPSample1 Output: Retrieving the home interface...
CMPSample1 Output: Searching for existing object with this key: 1
CMPSample1 Output: New Object instance created!
CMPSample1 Output: State updated.
TransactionTest Output: State updated.
TransactionTest Output: ***************************************** TransactionTest: beforeCompletion() *****************************************
TransactionTest Output: ***************************************** TransactionTest: afterCompletion() *****************************************
TransactionTest Output: Transaction Status after myTransaction.commit() is :6
TransactionTest Output: --------Starting Client-demarcated rollbacked transaction Scenario...
TransactionTest Output: --------Retrieving initial context...
TransactionTest Output: --------Set properties to: com.ibm.jndi.CosNaming.CNInitialContextFactory
TransactionTest Output: --------Invoke: javax.naming.InitialContext(properties)
TransactionTest Output: Retrieving the home interface...
TransactionTest Output: ***************************************** TransactionTest: afterBegin() *****************************************
TransactionTest Output: Transaction Status after myTransaction.begin() is: 0
TransactionTest Output: CMPSample1 Object...
TransactionTest Output: Enter key value: 
TransactionTest Output: Enter state value: 
TransactionTest Output: Searching for existing object with this key: 9
066.434 00000992 EJSEntityHome < findOrActivateFromKey: object not found
com.ibm.ejs.container.ContainerObjectNotFoundException
java.lang.Throwable()
java.lang.Exception()
java.io.IOException()
java.rmi.RemoteException()
com.ibm.ejs.EJSException()
com.ibm.ejs.container.ContainerException()
com.ibm.ejs.container.ContainerObjectNotFoundException()
Scenario 4—Sample Output

Transaction Test Output: ------Starting Client-demarcated committed transaction Scenario...
Transaction Test Output: ------Retrieving initial context...
Transaction Test Output: ------Set properties to: javax.naming.Context.PROVIDER_URL, iiop://cobalt:900
Transaction Test Output: ------Set properties to: com.ibm.jndi.CosNaming.CNInitialContextFactory
Transaction Test Output: ------Invoke: javax.naming.InitialContext(properties)
Transaction Test Output: Retrieving the home interface...
Transaction Test Output: Retrieving User Transaction...
Transaction Test Output: Transaction Status after mySessionCtx.getUserTransaction() is: 6
Transaction Test Output: **************************************** TransactionTest: afterBegin() **********88
Transaction Test Output: Transaction Status after myTransaction.begin() is: 0
Transaction Test Output: CMPSample1 Object...
Transaction Test Output: Enter key value:
Transaction Test Output: Enter state value:
Transaction Test Output: Searching for existing object with this key: q
Transaction Test Output: Object not found. Creating a new instance...
Transaction Test Output: New Object instance created!
CMPSample1 Output: **************************************** CMPSample1: afterBegin() **********88
044.000 000058bd Transactional > lock
   com.transarc.encina.js.ts.CoordinatorImpl@26444#tid=3
   IWRITE
044.051 000058bd Mutex     E Mutex acquired forThread[Thread-10,5,main]
044.054 000058bd Transactional > conflict
   com.transarc.encina.js.ts.CoordinatorImpl@26444#tid=3
   IWRITE
044.075 000058bd Transactional < conflict, true
044.077 000058bd Transactional > addWaiter, com.transarc.jmon.lock.Locker@7315
044.078 000058bd Transactional > isHolder
044.081 000058bd Transactional < isHolder, false, com.transarc.encina.js.ts.CoordinatorImpl@26444#tid=3
044.083 000058bd Transactional < addWaiter
044.084 000058bd Mutex     E Mutex released
044.085 000058bd Transactional E Waiting for lock
Scenario 4—Sample Output
C ITSOBANK
Database

C.1 Database Definition DDL

We use the command line processor commands and SQL statements to create the database and the information within it. Figure 164 and Figure 165 show the definitions of the tables. Figure 166 shows the foreign key relationships and the grant statements.
echo --- create the ITSOBANK database ---
echo CREATE DATABASE ITSOBANK

echo --- connect to ITSOBANK database ---
CONNECT TO ITSOBANK

echo --- drop all tables ---
DROP TABLE ITSO.BANK
DROP TABLE ITSO.CUSTOMER
DROP TABLE ITSO.CARD
DROP TABLE ITSO.POLICY
DROP TABLE ITSO.ACCOUNT
DROP TABLE ITSO.SAVINGS
DROP TABLE ITSO.CHECKING
DROP TABLE ITSO.PAYEE
DROP TABLE ITSO.TRANSRECORD
DROP TABLE ITSO.CARDACCOUNT
DROP SYNONYM ITSO.TRANS

echo --- creating tables ---
CREATE TABLE ITSO.BANK (     
  bankid  CHAR( 4) NOT NULL,  
  bankname CHAR(30) NOT NULL,  
  PRIMARY KEY (BANKID)  
)

CREATE TABLE ITSO.CUSTOMER (    
  custid CHAR( 4) NOT NULL,  
  title CHAR( 3) NOT NULL,  
  fname CHAR(30) NOT NULL,  
  lname CHAR(30) NOT NULL,  
  userid CHAR( 8),  
  password CHAR( 8),  
  bankid CHAR( 4) NOT NULL,  
  PRIMARY KEY (CUSTID)  
)
CREATE TABLE ITSO.ACCOUNT (  
accid       CHAR(8)  NOT NULL,  
bankid      CHAR(4),  
custid      CHAR(4)  NOT NULL,  
acctype     CHAR(10)  NOT NULL,  
balance     DEC(8,2)  NOT NULL,  
PRIMARY KEY (ACCID)  
)  

CREATE TABLE ITSO.SAVINGS (  
accid CHAR(8) NOT NULL,  
minamt DECIMAL(8,2) NOT NULL,  
PRIMARY KEY (accid)  
)  

CREATE TABLE ITSO.PAYEE (  
accid CHAR(8) NOT NULL,  
title CHAR(20),  
PRIMARY KEY (accid)  
)  

CREATE TABLE ITSO.CHECKING (  
accid CHAR(8) NOT NULL,  
overdraf DECIMAL(8,2) NOT NULL,  
PRIMARY KEY (accid)  
)  

CREATE TABLE ITSO.TRANSRECORD (  
transid     TIMESTAMP NOT NULL,  
accid       CHAR(8)  NOT NULL,  
transtype   CHAR(1)  NOT NULL,  
transamt    DEC(8,2)  NOT NULL,  
traccid     CHAR(8),  
PRIMARY KEY (TRANSID)  
)  

Figure 165. ITSOBANK Database Data Definition Language (Part 2)
CREATE SYNONYM ITSO.TRANS FOR ITSO.TRANSRECORD

echo --- referential integrity ---
ALTER TABLE ITSO.TRANSRECORD
  ADD CONSTRAINT "AccountTransrecord" FOREIGN KEY (ACCID)
  REFERENCES ITSO.ACCOUNT ON DELETE RESTRICT

ALTER TABLE ITSO.ACCOUNT
  ADD CONSTRAINT "CustomerAccount" FOREIGN KEY (CUSTID)
  REFERENCES ITSO.CUSTOMER ON DELETE RESTRICT

ALTER TABLE ITSO.ACCOUNT
  ADD CONSTRAINT "BankAccount" FOREIGN KEY (BANKID)
  REFERENCES ITSO.BANK

ALTER TABLE ITSO.CUSTOMER
  ADD CONSTRAINT "BankCustomer" FOREIGN KEY (BANKID)
  REFERENCES ITSO.BANK ON DELETE RESTRICT

echo --- execute GRANT statements ---
GRANT BINDADD ON DATABASE         TO PUBLIC
GRANT CONNECT ON DATABASE         TO PUBLIC
GRANT ALL     ON ITSO.BANK        TO PUBLIC
GRANT ALL     ON ITSO.CUSTOMER    TO PUBLIC
GRANT ALL     ON ITSO.ACCOUNT     TO PUBLIC
GRANT ALL     ON ITSO.SAVINGS     TO PUBLIC
GRANT ALL     ON ITSO.CHECKING    TO PUBLIC
GRANT ALL     ON ITSO.PAYEE       TO PUBLIC
GRANT ALL     ON ITSO.TRANSRECORD TO PUBLIC
GRANT ALL     ON ITSO.TRANS       TO PUBLIC

echo --- connect reset ---
CONNECT RESET

Figure 166. ITSOBANK Database Data Definition Language (Part 3)
Execute the following command from a DB2 command line processor:

```
db2 -f itsobank.ddl
```

You can either create the file itsobank.ddl from the above tables or get it from the CDROM or zip file that you can download from the ITSO Web site in the subdirectory:

CdRom\Part2BankApplication

## C.2 Sample Data of ITSOBANK Tables

The database tables are initialized with the data described in itsobank.sql described in the following Figure 167, Figure 168, Figure 169, and Figure 170.

```sql
connect to ITSOBANK

echo --- insert into BANK table ---

INSERT INTO ITSO.BANK (bankid, bankname) VALUES ('ITSO', 'THE ITSO BANK')

echo --- insert into CUSTOMER table ---

INSERT INTO ITSO.CUSTOMER (custid, title, fname, lname, userid, password, bankid) VALUES
('101', 'Mr.', 'John', 'Akerley', 'cust101', 'JA', 'ITSO'),
('102', 'Mr.', 'Pat', 'McCarthy', 'cust102', 'PM', 'ITSO'),
('103', 'Mr.', 'Markus', 'Muetschard', 'cust103', 'MM', 'ITSO'),
('104', 'Mr.', 'Joaquin', 'Picon', 'cust104', 'JP', 'ITSO'),
('105', 'Ms.', 'Unknown', 'Lady', null, null, 'ITSO'),
('106', 'Mr.', 'Ueli', 'Wahli', 'cust106', 'UW', 'ITSO'),
('901', 'THE', 'XYZ CORPORATION', null, null, 'ITSO')
```

Figure 167. ITSOBANK Data (Part 1)
Figure 168. ITSOBANK Data (Part 2)
echo --- insert into CHECKING table ---

```
INSERT INTO ITSO.CHECKING
    (accid, overdraf) VALUES
    ('101-1001', 200.00),
    ('102-2002', 200.00),
    ('103-3002', 200.00),
    ('104-4001', 200.00),
    ('104-4002', 200.00),
    ('105-5001', 200.00),
    ('106-6001', 200.00),
    ('106-6004', 200.00)
```

echo --- insert into PAYEE table ---

```
INSERT INTO ITSO.PAYEE
    (accid, title ) VALUES
    ('901-9001', 'XYZ VISA')
```

Figure 169. ITSOBANK Data (Part 3)
Figure 170. ITSOBANK Data (Part 4)
D Samples and BANK Application

D.1 Content of the CDROM

A CD-ROM is included with this redbook, it provides VisualAge for Java repositories and Java source files (see Figure 171). You find under Part1Samples directory the samples developed for the first part of the book.

The home banking application is located under the directory Part2BankApplication. It contains the enterprise beans implementation as well as the client part including HTML, servlets and JSP.
The sample code associated with this redbook is also available in softcopy on the Internet from the redbooks Web server:

Point your Web browser to:

ftp://www.redbooks.ibm.com/redbooks/SG245429

You can also go to the following URL and select Additional Materials

http://www.redbooks.ibm.com

### D.2 Running the ITSOBANK application

The Enterprise JavaBeans application can be tested by using the test client and by invoking use cases made available through the UserSB session bean.

However, we also provide a sample client application. This client part is completely described in SG245423 redbook and has been implemented by using servlets, JSP, HTML pages and a specific bank implementation. We have removed the specific bank implementation and have replaced it by the
Enterprise JavaBeans bank implementation. Being able to replace one implementation by another one was one of our goal when we defined the two projects SG245423 and the present one SG245429.

You can run the complete (client and server) bank application in VisualAge for Java or in the WebSphere Advanced Edition 2.0.2 runtime. Two different directories contain the required files:

- EJBBankCode\Part2bankApplication\VisualAge for Java
- EJBBankCode\Part2bankApplication\WebSphere Advanced 2.0.2

Obviously, you can also have a mixed configuration by running one part inside VisualAge for Java and another inside WebSphere Advanced Edition 2.0.2.

**D.2.1 Inside VisualAge for Java**

1. Create ITSOBANK database.
2. Use the itsobank.ddl and itsobank.sql to create and populate the tables required by the application.
3. Import inside VisualAge for Java the repositories files:
   - bank506.dat (project: ITSOBank, version=5.0.6)
   - WebSphere Bank Application.dat (project: WebSphere Bank Application, version: HBA_EJB_002)
4. From the repository, add the projects:
   - ITSOBank 5.0.6 and
   - WebSphere Bank Application, version: HBA_EJB_002
5. create the directory c:\bank\config and copy the file bank.properties from CdRom\Part2BankApplication\VisualAge for Java directory.
6. Edit c:\bank\config\bank.properties and replace
   `atlantic.almaden.ibm.com:900` by your fully qualified hostname.
7. Edit the file servlets.properties located in
   `drive:\IBM\javade\project_resources\IBM WebSphere Test Environment\properties\server\servlet\servletservice`
   - change the `servlets.startup=invoker` line to:
     `- servlets.startup=invoker BankServlet`
   - add the following lines below #Servlets added by the user:
     `- #Servlets added by the user`
     `- servlet.BankServlet.initArgs=implementation=itso.bank.ejbimpl`
8. Edit doc.properties file in drive:\IBM\ava\ide\project_resources\IBM WebSphere Test Environment\properties\server\servlet\httpservice and ensure that the default document root is correct. (example: doc.root=p:\WWW\HTML).


10. Select **Program** tab and in the Command line arguments, add ".\IBM WebSphere Test Environment"

11. Select **Class Path** tab and click **Edit** from Project Path: and select:
   - IBM Servlet IDE Utility class libraries
   - IBM WebSphere Test Environment
   - ITSOBank
   - WebSphere Bank Application

12. Click on the EJBs tab and select ITSOBank. Click mouse button 2 and select **Add To -> Server Configuration**

13. In the EJB Server Configuration window, select **Persistent Name Server** and click mouse button 2 and select **Properties**.

14. In LSD Name, change localhost to your fully qualified hostname. In this way you will be able to access your EJB beans from clients running in different machines.

15. In the EJB Server Configuration window, select **EJB Server (server1)**, click mouse button 2 and select **Properties**.

16. Change Database URL from jdbc:db2:sampleDB to jdbc:db2:ITSOBANK

17. In LSD Name, change localhost to your fully qualified hostname. In this way you will be able to access your EJB beans from clients running in different machines.

18. Ensure DB2 is running and that you set up in Window->Options the Workspace class path to points to drive:\SQLLIB\java\db2java.zip, then start:
   1. Location Service Daemon
   2. Persistent Name Server
   3. EJB Server (server1)

19. In the Console window, select your EJS server and in the Standard Out panel, you should see in the last 2 lines:
20. Go back to the Projects, select **SERunner** and start it.

### D.2.2 Inside WebSphere Advanced Edition 2.0.2

The client part and the Enterprise JavaBeans implementation have been successfully tested with the WebSphere Advanced Edition 2.0.2 runtime. We used two different machines. On host1 we deployed the client part (HTML/servlets/JSP) and on host2 we deployed the Enterprise JavaBeans.

#### D.2.2.1 Deployment of the client part

22. Unzip **site.zip** file in your root document directory. If using Lotus Go Web server, you find the document root directory by opening `drive:\winnt\httpd.cnf` file and look for the statement `Pass/*`. (ex: `Pass /* W:\WWW\HTML\*` defines `W:\WWW\HTML` as the directory where you unzip the site.zip file).

23. Copy the file **bank.jar** to `drive:\WebSphere\AppServer\classes`
24. Copy the file **itsobank.jar** to `drive:\WebSphere\AppServer\classes`
25. Unzip **servlets.zip** into `drive:\WebSphere\AppServer\servlets`
26. Edit the file `servlets.properties` located in `drive:\WebSphere\AppServer\properties\server\servlet\servletservice` and

    - change the `servlets.startup=invoker` line to:
      
      ```
      servlets.startup=invoker BankServlet
      ```

    - add the following lines below `# Servlets added by the user`:
      
      ```
      # Servlets added by the user
      servlet.BankServlet.code=itso.bank.servlet.BankServlet
      servlet.BankServlet.initArgs=implementation=itso.bank.ejbimpl.atlantic.almaden.ibm.com UserSB 9019
      ```

27. Edit the bootstrap.properties file in `drive:\WebSphereAppServer\properties` and add:

    - `bank.jar` and `itsobank.jar` files to the CLASSPATH. You must use the 8 char naming convention for this, e.g.,
      
      ```
      f:\WEBSPH~1\APPSE~1\classes\bank.jar;
      ```
D.2.2.2 Deployment of the Enterprise JavaBeans implementation

1. Create ITSOBANK database.
2. Use the itsobank.ddl and itsobank.sql to create and populate the tables required by the application.
3. Copy itsobank.jar to drive:\WebSphere\AppServer\classes
4. Copy bankEB.jar and bankSB.jar to the directories:
   - drive:\WebSphere\AppServer\deployableEJBs and
   - drive:\WebSphere\AppServer\deployedEJBs
5. Copy bank.properties into c:\bank\config directory and indicate the fully qualified hostname on which the EJB implementation is running. If running in WebSphere Advanced Edition 2.0.2 use port 9019, if running inside VisualAge for Java use port 900.
6. Edit the bootstrap.properties file in drive:\WebSphereAppServer\properties and add:
   - itsobank.jar files to the CLASSPATH.
   You must use the 8 char naming convention for this, e.g., f:\WEBSPH~1\APPSER~1\classes\bank.jar;
7. Edit c:\bank\config\bank.properties and replace atlantic.almaden.ibm.com:9019 by your fully qualified hostname.
8. Edit IBMNameServiceConfig.properties file in drive:\WebSphereAppServer\properties\ejs and change the default hostname by your fully qualified hostname.
9. Edit ejs.properties file in drive:\WebSphereAppServer\properties\ejs and change
   - localhost to your fully qualified domain name
   - hostnamenameservice.hostname = atlantic.almaden.ibm.com
   - add/modify bankSB.jar, bankEB.jar, dbPassword, dbUser according to your db2 definitions:
     - defaultSessionContainer.jarFileDirectory=deployedEJBs
     - defaultSessionContainer.jarFiles=bankSB.jar
     - defaultSessionContainer.dbUrl=
     - defaultSessionContainer.implClass=com.ibm.ejs.container.EJSSessionContainer
     - defaultEntityContainer.implClass=com.ibm.ejs.container.EJSEntityContainer
     - defaultEntityContainer.jarFileDirectory=deployedEJBs
Samples and BANK Application

10. Restart WebSphere and the Web server.

11. A possible test scenario is described in Appendix D.2.3, “Testing a bank scenario” on page 383.

**D.2.3 Testing a bank scenario**

To test the application, open your browser and enter the URL:

http://hostname:8080/index.html if running inside VisualAge for Java (don't forget, the HTTP server running inside VisualAge for Java is listening on port 8080) or http://hostname/index.html if running inside WebSphere Advanced Edition 2.0.2.

You should get the Welcome screen (Figure 172):
Welcome...

Are you ready to bank at home? We are dedicated to providing our customers the best services available for on-line banking. Use the navigation bar at the top to navigate your way through the site. You must login into the system by going through the Login page before you are allowed entrance into the sections to the right of Login on the navigation bar. If you want to use the online facilities of the bank please login into the system.

Figure 172. Welcome Page

By clicking on Login menu item, you are prompted to identify/authenticate yourself (Figure 173).
The userid and password are set by default to cust101 and JA. Valid userids and passwords are defined in the CUSTOMER table of the ITSOBANK database by the columns USERID and PASSWORD. If you want to keep the default values, click on Login. If you are successfully identified, you should get the next panel (Figure 174).
Figure 174. Account Page

To see account information, click on **Display account information** (Figure 174).
Figure 175. Account Information Page

By clicking on Account Balance (Figure 175) you should get (Figure 176):
Clicking **Account**, on the left side, takes you back to the Account page (see Figure 174 on page 386). By Clicking on **Transfer funds between accounts** you should see the following page (Figure 177):
Samples and BANK Application

Figure 177. Transfer Funds page

Keep the default selection in Transfer From and select **SAVINGS 101-1002** in Transfer To combo box. Enter an amount in the Amount textfield and enter the corresponding password for cust101 which is **JA**. The application is checking again the user identity before performing a fund transfer. Click on **Transfer** button and you should get the next page Funds Transferred (Figure 178).
Figure 178. Fund Transferred Page

This is the end of our scenario. You can keep playing with it but be aware that **Pay your Bills** (Appendix 174, “Account Page” on page 386) has not been implemented. You can still click on it and see the exception thrown!
Special Notices

This publication is intended to help application developers using VisualAge for Java to create distributed objects applications based on the Enterprise JavaBeans architecture. The information in this publication is not intended as the specification of any programming interfaces that are provided by VisualAge for Java Enterprise Edition. See the PUBLICATIONS section of the IBM Programming Announcement for VisualAge for Java Enterprise Edition for more information about what publications are considered to be product documentation.

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM’s product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM’s intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you
any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY 10594 USA.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Corporation, Dept. 600A, Mail Drop 1329, Somers, NY 10589 USA.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

Any pointers in this publication to external Web sites are provided for convenience only and do not in any manner serve as an endorsement of these Web sites.

The following terms are trademarks of the International Business Machines Corporation in the United States and/or other countries:

IBM
DB2
S/390
ThinkPad

The following terms are trademarks of other companies:

C-bus is a trademark of Corollary, Inc. in the United States and/or other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and/or other countries.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of
Microsoft Corporation in the United States and/or other countries.

PC Direct is a trademark of Ziff Communications Company in the United States and/or other countries and is used by IBM Corporation under license.

ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks of Intel Corporation in the United States and/or other countries.

UNIX is a registered trademark in the United States and/or other countries licensed exclusively through X/Open Company Limited.

SET and the SET logo are trademarks owned by SET Secure Electronic Transaction LLC.

Other company, product, and service names may be trademarks or service marks of others.
Related Publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

F.1 International Technical Support Organization Publications

For information on ordering these ITSO publications see “How To Get Itso Redbooks” on page 399.

- Developing an e-Business Application for the WebSphere Application Server, SG24-5423
- VisualAge for Java Enterprise Version 2: Persistence Builder with GUIs, Servlets, and Java Server Pages, SG24-5426
- IBM Component Broker Connector Overview, SG24-2022
- IBM CBCConnector Cookbook Collection: First Steps, SG24-2033
- IBM CBCConnector Cookbook Collection: CBCConnector Bank Implementation, SG24-5119
- Application Development with VisualAge for Java Enterprise, SG24-5081
F.2 Redbooks on CD-ROMs

Redbooks are also available on CD-ROMs. Order a subscription and receive updates 2-4 times a year at significant savings.

<table>
<thead>
<tr>
<th>CD-ROM Title</th>
<th>Subscription Number</th>
<th>Collection Kit Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>System/390 Redbooks Collection</td>
<td>SBOF-7201</td>
<td>SK2T-2177</td>
</tr>
<tr>
<td>Networking and Systems Management Redbooks Collection</td>
<td>SBOF-7370</td>
<td>SK2T-6022</td>
</tr>
<tr>
<td>Transaction Processing and Data Management Redbook</td>
<td>SBOF-7240</td>
<td>SK2T-8038</td>
</tr>
<tr>
<td>Lotus Redbooks Collection</td>
<td>SBOF-6899</td>
<td>SK2T-8039</td>
</tr>
<tr>
<td>Tivoli Redbooks Collection</td>
<td>SBOF-6898</td>
<td>SK2T-8044</td>
</tr>
<tr>
<td>AS/400 Redbooks Collection</td>
<td>SBOF-7270</td>
<td>SK2T-2849</td>
</tr>
<tr>
<td>RS/6000 Redbooks Collection (HTML, BkMgr)</td>
<td>SBOF-7230</td>
<td>SK2T-8040</td>
</tr>
<tr>
<td>RS/6000 Redbooks Collection (PostScript)</td>
<td>SBOF-7205</td>
<td>SK2T-8041</td>
</tr>
<tr>
<td>RS/6000 Redbooks Collection (PDF Format)</td>
<td>SBOF-8700</td>
<td>SK2T-8043</td>
</tr>
<tr>
<td>Application Development Redbooks Collection</td>
<td>SBOF-7290</td>
<td>SK2T-8037</td>
</tr>
</tbody>
</table>

F.3 Other Publications

These publications are also relevant as further information sources:
- Design Patterns: Elements of Reusable Object Oriented Software
  SR28-5629 (Addison-Wesley Professional Computing Series)
How To Get Itso Redbooks

This section explains how both customers and IBM employees can find out about ITSO redbooks, CD-ROMs, workshops, and residencies. A form for ordering books and CD-ROMs is also provided.

This information was current at the time of publication, but is continually subject to change. The latest information may be found at http://www.redbooks.ibm.com/.

How IBM Employees Can Get ITSO Redbooks

Employees may request ITSO deliverables (redbooks, BookManager BOOKs, and CD-ROMs) and information about redbooks, workshops, and residencies in the following ways:

- **Redbooks Web Site on the World Wide Web**
  http://w3.itso.ibm.com/

- **PUBORDER** – to order hardcopies in the United States

- **Tools Disks**
  To get LIST3820s of redbooks, type one of the following commands:
  
  TOOLCAT REDPRINT
  TOOLS SENDTO EHONE4 TOOLS2 REDPRINT GET SG24xxxx PACKAGE
  TOOLS SENDTO CANVM2 TOOLS REDPRINT GET SG24xxxx PACKAGE (Canadian users only)

  To get BookManager BOOKs of redbooks, type the following command:
  
  TOOLCAT REDBOOKS

  To get lists of redbooks, type the following command:
  
  TOOLS SENDTO USDIST MKTTOOLS MKTTOOLS GET ITSOCAT TXT

  To register for information on workshops, residencies, and redbooks, type the following command:
  
  TOOLS SENDTO WTSCPOK TOOLS ZDISK GET ITSOREGI 1998

- **REDBOOKS Category on INEWS**

- **Online** – send orders to: USIB6FPL at IBMMAIL or DKIBMBSH at IBMMAIL

---

For information so current it is still in the process of being written, look at "Redpieces" on the Redbooks Web Site (http://www.redbooks.ibm.com/redpieces.html). Redpieces are redbooks in progress; not all redbooks become redpieces, and sometimes just a few chapters will be published this way. The intent is to get the information out much quicker than the formal publishing process allows.
How Customers Can Get ITSO Redbooks

Customers may request ITSO deliverables (redbooks, BookManager BOOKs, and CD-ROMs) and information about redbooks, workshops, and residencies in the following ways:

- **Online Orders** – send orders to:
  
  **IBMMAIL**
  - In United States: usib6fp1 at ibmmail
  - In Canada: caibmbkz at ibmmail
  - Outside North America: dkibmbsh at ibmmail

  **Internet**
  - usib6fp1@ibmmail.com
  - Imannix@vnet.ibm.com
  - bookshop@dk.ibm.com

- **Telephone Orders**
  - United States (toll free): 1-800-879-2755
  - Canada (toll free): 1-800-IBM-4YOU

- **Mail Orders** – send orders to:
  
  IBM Publications
  Publications Customer Support
  P.O. Box 29570
  Raleigh, NC 27626-0570
  USA

- **Fax** – send orders to:
  
  United States (toll free): 1-800-445-9269
  Canada: 1-800-267-4455
  Outside North America: (+45) 48 14 2207 (long distance charge)

- **1-800-IBM-4FAX (United States) or (+1) 408 256 5422 (Outside USA)** – ask for:
  
  Index #4421 Abstracts of new redbooks
  Index #4422 IBM redbooks
  Index #4420 Redbooks for last six months

- **On the World Wide Web**
  
  Redbooks Web Site: http://www.redbooks.ibm.com
For information so current it is still in the process of being written, look at "Redpieces" on the Redbooks Web Site (http://www.redbooks.ibm.com/redpieces.html). Redpieces are redbooks in progress; not all redbooks become redpieces, and sometimes just a few chapters will be published this way. The intent is to get the information out much quicker than the formal publishing process allows.
IBM Redbook Order Form

Please send me the following:

<table>
<thead>
<tr>
<th>Title</th>
<th>Order</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

First name: 
Last name: 
Company: 
Address: 
City: Postal code: Country: 
Telephone number: Telefax number: VAT number: 

- [ ] Invoice to customer: 
- [ ] Credit card number: 

Credit card expiration: Card issued to: Signature: 

We accept American Express, Diners, Eurocard, Master Card, and Visa. Payment by credit card not available in all countries. Signature mandatory for credit card payment.
Glossary

This glossary defines terms and abbreviations that are used in this book. If you do not find the term you are looking for, refer to the IBM Dictionary of Computing, New York: McGraw-Hill, 1994.

This glossary includes terms and definitions from the American National Standard Dictionary for Information Systems, ANSI X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI). Copies may be purchased from the American National Standards Institute, 1430 Broadway, New York, New York 10018.

A

abstract class. A class that provides common behavior across a set of subclasses but is not itself designed to have instances that work. An abstract class represents a concept; classes derived from it represent implementations of the concept. See also base class.

access application. Generated by the Data Access Builder for each schema mapping, an executable GUI that provides access to the database using the other classes generated for the mapping.

accessor methods. Methods that an object provides to define the interface to its instance variables. The accessor method to return the value of an instance variable is called a get method or getter method, and the accessor method to assign a value to an instance variable is called a set method or setter method.

applet. A Java program designed to run within a Web browser. Contrast with application.

application. In Java programming, a self-contained, stand-alone Java program that includes a main() method. Contrast with applet.

application server. A server program that allows the installation of application specific software components, in a manner so that they can be remotely invoked, usually by some form of remote object method call.

argument. A data element, or value, included as a bean in a method call. Arguments provide additional information that the called method can use to perform the requested operation.

attribute. A specification of a property of a bean. For example, a customer bean could have a name attribute and an address attribute. An attribute can itself be a bean with its own behavior and attributes. In the Data Access Builder, the aspect of a schema mapping that represents a column in a database table.

B

base class. A class from which other classes or beans are derived. A base class may itself be derived from another base class. See also abstract class.

bean. A definition or instance of a JavaBeans component. See also JavaBeans.

BeanInfo. (1) A companion class for a bean that defines a set of methods that can be accessed to retrieve information on the bean’s properties, events, and methods. (2) In the VisualAge for Java IDE, a page in the class browser that provides bean information.

bean-managed persistence. When an Enterprise JavaBeans performs its own long-term state management.

beans palette. In the Visual Composition Editor, a two-column pane that contains prefabricated beans that you can select and manipulate to create programs. The left column contains categories of beans, and the right column contains beans for the selected category. The default set of beans generally represents JDK AWT components. You can add your own categories and beans to the beans palette.

break point. A point in a computer program where the execution can be halted.

© Copyright IBM Corp. 1999
browser. (1) In VisualAge for Java, a window that provides information on program elements. There are browsers for projects, packages, classes, methods, and interfaces. (2) An Internet-based tool that lets users browse Web sites.

C

C++ Access Builder. A VisualAge for Java Enterprise tool that generates beans to access C and C++ DLLs.

category. In the Visual Composition Editor, a selectable grouping of beans represented by an icon in the left-most column. Selecting a category displays the beans belonging to that category in the next column. See also beans palette.

CICS Access Builder. A VisualAge for Java Enterprise tool that generates beans to access CICS transactions through the CICS Gateway for Java and CICS Client.

CICS Client. A server program that processes CICS ECI calls, forwarding transaction requests to a CICS program running on a host.

CICS ECI. An API that provides C and C++ programs with procedural access to transactions.

CICS Gateway for Java. A server program that processes Java ECI calls and forwards CICS ECI calls to the CICS Client.

class. An aggregate that defines properties, operations, and behavior for all instances of that aggregate.

class hierarchy. The relationships between classes that share a single inheritance. All Java classes inherit from the Object class.

class library. A collection of classes.

class method. See method.

CLASSPATH. In your deployment environment, the environment variable that specifies the directories in which to look for class and resource files.

client/server. The model of interaction in distributed data processing where a program at one location sends a request to a program at another location and awaits a response. The requesting program is called a client, and the answering program is called a server.

client-side server proxy. Generated by the RMI Access Builder, a local representative of a remote bean. This proxy provides access to the operations of the server bean, allowing a Java client to work with it as if it were the server bean. See also proxy bean and server-side server proxy.

Class Browser. In the VisualAge for Java IDE, a tool used to browse the classes loaded in the workspace.

collection. A set of features in which each feature is an object.

commit. The operation that ends a unit of work and updates the database such that other processes can access any changes made.

Common Object Request Broker Architecture (CORBA). A middleware specification which defines a software bus—the Object Request Broker (ORB)—that provides the infrastructure.

communications area (COMMAREA). In a CICS transaction program, a group of records that describes both the format and volume of data used.

component model. An architecture and an API that allows developers to define reusable segments of code that can be combined to create a program. VisualAge for Java uses the JavaBeans component model.

composite bean. A bean that is composed of a bean and one or more subbeans. A composite bean can contain visual beans, nonvisual beans, or both. See also nonvisual bean, bean, and visual bean.

congrete class. A subclass of an abstract class that is a specialization of the abstract class.

connection. In the Visual Composition Editor, a visual link between two components that represents the relationship between the components. Each connection has a source, a target, and other properties. See also
event-to-method connection, event-to-property connection, parameter connection, property-to-method connection, and property-to-property connection.

**console.** In VisualAge for Java, the window that acts as the standard input (System.in) and standard output (System.out) device for programs running in the VisualAge for Java IDE.

**container-managed persistence.** When an Enterprise JavaBeans server manages a bean's long term state.

**construction from parts.** A software development technology in which applications are assembled from existing and reusable software components, known as parts. In VisualAge for Java, parts are called beans.

**constructor.** A special class method that has the same name as the class and is used to construct and possibly initialize objects of its class type.

**container.** A component that can hold other components. In Java, examples of containers include applets, frames, and dialogs. In the Visual Composition Editor, containers can be graphically represented and generated.

**current edition.** The edition of a program element that is currently in the workspace. See also current edition, open edition, and versioned edition.

**cursor.** A database control structure used by the Data Access Builder to point to a specific row within some ordered set of rows and to retrieve rows from a set, possibly making updates or deletions.

**data abstraction.** A data type with a private representation and a public set of operations. The Java language uses the concept of classes to implement data abstraction.

**Data Access Builder.** A VisualAge for Java Enterprise tool that generates beans to access and manipulate the content of JDBC/ODBC-compliant relational databases.

**DB2 for MVS/ESA.** An IBM relational database management system for the MVS operating system.

**double-byte character set (DBCS).** A set of characters in which each character is represented by 2 bytes. Languages such as Japanese, Chinese, and Korean, which contain more symbols than can be represented by 256 code points, require double-byte character sets. Compare with single-byte character set.

**Distributed Computing Environment (DCE).** Adopted by the computer industry as a de facto standard for distributed computing. DCE allows computers from a variety of vendors to communicate transparently and share resources such as computing power, files, printers, and other objects in the network.

**Distributed Component Object Model (DCOM).** A protocol that enables software components to communicate directly over a network in a reliable, secure, and efficient manner. Previously called "Network OLE," DCOM is designed for use across multiple network transports, including Internet protocols such as HTTP. DCOM is based on the Open Software Foundation's DCE-RPC specification and works with both Java applets and ActiveX components through its use of the Component Object Model (COM).

**dynamic link library (DLL).** A file containing executable code and data bound to a program at run time rather than at link time. The C++ Access Builder generates beans and C++ wrappers that let your Java programs access C++ DLLs.


**encapsulation.** The hiding of a software object's internal representation. The object provides an interface that queries and manipulates the data without exposing its underlying structure.
enterprise access builders. In VisualAge for Java Enterprise, a set of code-generation tools. See also C++ Access Builder, CICS Access Builder, Data Access Builder, and RMI Access Builder.

**Enterprise JavaBeans.** A server component developed by SUN Microsystems.

**event.** An action by a user program, or a specification of a notification that may trigger specific behavior. In JDK 1.1, events notify the relevant listener classes to take appropriate actions.

**event-to-method connection.** A connection from an event generated by a bean to a method of another bean. When the connected event occurs, the method is executed. See also connection.

**event-to-property connection.** A connection that changes the value of a property when a certain event occurs. See also connection.

**F**

**feature.** (1) A major component of a software product that can be installed separately. (2) In VisualAge for Java, a method, field, or event that is available from a bean's interface and to which other beans can connect.

**field.** A data object in a class. For example, a customer class could have a name field and an address field. A field can itself be an object with its own behavior and fields. By default, a field, in contrast to a property, does not support event notification.

**free-form surface.** The large open area of the Visual Composition Editor where you can work with visual and nonvisual beans. You add, remove, and connect beans on the free-form surface.

**framework.** A set of cooperative classes with strong connections that provide a template for development.

**G**

**garbage collection.** A Smalltalk process for periodically identifying unreferenced objects and deallocating their memory.

**gateway.** A host computer that connects networks that communicate in different languages. For example, a gateway connects a company's LAN to the Internet.

**graphical user interface (GUI).** A type of interface that enables users to communicate with a program by manipulating graphical features, rather than by entering commands. Typically, a graphical user interface includes a combination of graphics, pointing devices, menu bars and other menus, overlapping windows, and icons.

**H**

**hypertext.** Text in a document that contains a hidden link to other text. You can click a mouse on a hypertext word and it will take you to the text designated in the link. Hypertext is used in Windows help programs and CD encyclopedias to jump to related references elsewhere within the same document. Hypertext can link—using HTTP over the Web—to any Web document in the world, with only a single mouse click.

**Hypertext Markup Language (HTML).** The basic language that is used to build hypertext documents on the World Wide Web. It is used in basic, plain ASCII-text documents, but when those documents are interpreted (rendered) by a Web browser such as Netscape, the document can display formatted text, color, a variety of fonts, graphics images, special effects, hypertext jumps to other Internet locations, and information forms.

**Hypertext Transfer Protocol (HTTP).** The protocol for moving hypertext files across the Internet. Requires an HTTP client program on one end, and an HTTP server program on the other end.
inheritance. (1) A mechanism by which an object class can use the attributes, relationships, and methods defined in more abstract classes related to it (its base classes). (2) An object-oriented programming technique that allows you to use existing classes as bases for creating other classes.

instance. Synonym for object, a particular instantiation of a data type.

Integrated Development Environment (IDE). In VisualAge for Java, the set of windows that provide the user with access to development tools. The primary windows are Workbench, Log, Console, Debugger, and Repository Explorer.

interchange file. A file that you can export from VisualAge for Java that contains information about selected projects or packages. This file can then be imported into any VisualAge for Java session.

interface. A set of methods that can be accessed by any class in the class hierarchy. The Interface page in the Workbench lists all interfaces in the workspace.

Internet. The vast collection of interconnected networks that use TCP/IP and evolved from the ARPANET of the late 1960s and early 1970s.

intranet. A private network, inside a company or organization, that uses the same kinds of software that you would find on the public Internet. Many of the tools used on the Internet are being used in private networks; for example, many companies have Web servers that are available only to employees.


IP number. An Internet address that is a unique number consisting of four parts separated by dots, sometimes called a dotted quad (for example: 198.204.112.1). Every Internet computer has an IP number, and most computers also have one or more domain names that are plain language substitutes for the dotted quad.

Java. A programming language invented by Sun Microsystems that is specifically designed for writing programs that can be safely downloaded to your computer through the Internet and immediately run without fear of viruses or other harm to your computer or files. Using small Java programs (called applets), Web pages can include functions such as animation, calculators, and other fancy tricks. We can expect to see a huge variety of features added to the Web through Java, because you can write a Java program to do almost anything a regular computer program can do and then include that Java program in a Web page.

Java archive (JAR). A platform-independent file format that groups many files into one. JAR files are used for compression, reduced download time, and security. Because the JAR format is written in Java, JAR files are fully extensible.

JavaBeans. In JDK 1.1, the specification that defines the platform-neutral component model used to represent parts. Instances of JavaBeans (often called beans) may have methods, properties, and events.

Java Database Connectivity (JDBC). In JDK 1.1, the specification that defines an API that enables programs to access databases that comply with this standard.

Java Naming and Directory Interface. The Java standard API for accessing directory services, such as LDAP, COS Naming, and others.

Java Native Interface (JNI). In JDK 1.1, the specification that defines a standard naming and calling convention so that the Java virtual machine can locate and invoke methods written in a language different from Java. See also native method.

JTA. Java transaction API.

JTS. The Java Transaction Service based on the CORBA Transaction Service which provides a way for middleware vendors to build interoperable transactional middleware.
**K**

**keyword.** A predefined word, reserved for Java, that cannot be used as an identifier.

**L**


**legacy code.** Existing code that a user might have. Legacy applications often have character-based, non-graphical user interfaces. Usually they are written in a non-object-oriented language, such as C or COBOL.

**listener.** In JDK 1.1, a class that receives and handles events.

**local area network (LAN).** A computer network located on a user's establishment within a limited geographical area. A LAN typically consists of one or more server machines providing services to a number of client workstations.

**log.** In VisualAge for Java, the window that displays messages and warnings during development.

**M**

**mapping.** See schema mapping.

**member.** (1) A data object in a structure or a union. (2) In Java, classes and structures can also contain functions and types as members.

**method.** A fragment of Java code within a class that can be invoked and passed a set of parameters to perform a specific task.

**method call.** A communication from one object to another that requests the receiving object to execute a method. A method call consists of a method name that indicates the requested method and the arguments to be used in executing the method. The method call always returns some object to the requesting object as the result of performing the method. Synonym for message.

**message.** A request from one object that the receiving object implement a method. Because data is encapsulated and not directly accessible, a message is the only way to send data from one object to another. Each message specifies the name of the receiving object, the method to be implemented, and any arguments the method needs for implementation. Synonym for method call.

**model.** A nonvisual bean that represents the state and behavior of an object, such as a customer or an account. Contrast with view.

**N**

**native method.** Method written in a language other than Java that can be called by a Java object through the JNI specification.

**named package.** In the VisualAge for Java IDE, a package that has been explicitly named and created.

**nonvisual bean.** In the Visual Composition Editor, a bean that has no visual representation at run time. A nonvisual bean typically represents some real-world object that exists in the business environment. Compare with model. Contrast with view and visual bean.

**notification framework.** In JDK 1.1, a set of classes that implement the notifier/listener protocol. The notification framework is the base of the construction from beans technology (Visual Composition Editor).

**O**

**object.** (1) A computer representation of something that a user can work with to perform a task. An object can appear as text or an icon. (2) A collection of data and methods that operate on that data, which together represent a logical entity in the system. In object-oriented programming, objects are grouped into classes that share common data definitions and methods. Each object in the class is said to be an instance of the class. (3) An instance of an object class consisting of attributes, a data structure, and operational methods. It can represent a
person, place, thing, event, or concept. Each instance has the same properties, attributes, and methods as other instances of the object class, although it has unique values assigned to its attributes.

**object class.** A template for defining the attributes and methods of an object. An object class can contain other object classes. An individual representation of an object class is called an object.

**object factory.** A nonvisual bean capable of dynamically creating new instances of a specified bean. For example, during the execution of an application, an object factory can create instances of a new class to collect the data being generated.

**object-oriented programming (OOP).** A programming approach based on the concepts of data abstraction and inheritance. Unlike procedural programming techniques, object-oriented programming concentrates on those data objects that constitute the problem and how they are manipulated, not on how something is accomplished.

**Object Request Broker (ORB).** A CORBA term designating the means by which objects transparently make requests and receive responses from objects, whether they are local or remote.

**ODBC driver.** A DLL that implements ODBC function calls and interacts with a data source.

**Open Database Connectivity (ODBC).** A Microsoft developed C database API that allows access to database management systems calling callable SQL, which does not require the use of an SQL preprocessor. In addition, ODBC provides an architecture that allows users to add modules (database drivers) that link the application to their choice of database management systems at run time. Applications no longer need to be directly linked to the modules of all the database management systems that are supported.

**open edition.** An edition of a program element that can still be modified; that is, the edition has not been versioned. An open edition may reside in the workspace as well as in the repository.

**operation.** A method or service that can be requested of an object.

**overloading.** An object-oriented programming technique that allows redefinition of methods when the methods are used with class types.

P

**package.** A program element that contains related classes and interfaces.

**palette.** See beans palette

**parameter connection.** A connection that satisfies a parameter of an action or method by supplying either a property’s value or the return value of an action, method, or script. The parameter is always the source of the connection. See also connection.

**parent class.** The class from which another bean or class inherits data, methods, or both.

**part.** An existing, reusable software component. In VisualAge for Java, all parts created with the Visual Composition Editor conform to the JavaBeans component model and are referred to as beans. See also nonvisual bean and visual bean. Compare with Class Editor and Composition Editor.

**primitive bean.** A basic building block of other beans. A primitive bean can be relatively complex in terms of the function it provides.

**private.** In Java, an access modifier associated with a class member. It allows only the class itself to access the member.

**process.** A collection of code, data, and other system resources, including at least one thread of execution, that performs a data processing task.

**program.** In VisualAge for Java, a term that refers to both Java applets and applications.

**project.** In VisualAge for Java, the topmost kind of program element. A project contains Java packages.

**promote features.** Make features of a subbean available to be used for making connections. This applies to subbeans that are to be included in other beans, for example, a subbean consisting of
three push buttons on a panel. If this sample subbean is placed in a frame, the features of the push buttons would have to be promoted to make them available from within the frame.

**property.** An initial setting or characteristic of a bean; for example, a name, font, text, or positional characteristic.

**property sheet.** In the Visual Composition Editor, a set of name-value pairs that specify the initial appearance and other bean characteristics. A bean's property sheet can be viewed from the Properties secondary window.

**property-to-method connection.** A connection that calls a method whenever a property's value changes. It is similar to an event-to-method connection because the property's event ID is used to notify the method when the value of the property changes. See also connection.

**property-to-property connection.** A connection from a property of one bean to a property of another bean. When one property is updated, the other property is updated automatically. See also connection.

**protected.** In Java, an access modifier associated with a class member. It allows the class itself, subclasses, and all classes in the same package to access the member.

**protocol.** (1) The set of all messages to which an object will respond. (2) Specification of the structure and meaning (the semantics) of messages that are exchanged between a client and a server. (3) Computer rules that provide uniform specifications so that computer hardware and operating systems can communicate. It is similar to the way that mail, in countries around the world, is addressed in the same basic format so that postal workers know where to find the recipient’s address, the sender’s return address, and the postage stamp. Regardless of the underlying language, the basic protocols remain the same.

**prototype.** A method declaration or definition that includes both the return type of the method and the types of its arguments.

**proxy bean.** A group of client-side and server-side objects that represent a remote server bean. The top-level class that implements the proxy bean is the client-side server proxy. See also client-side server proxy and server-side server proxy.

**R**

Remote Method Invocation (RMI). In JDK 1.1, the API that enables you to write distributed Java programs, allowing methods of remote Java objects to be accessed from other Java virtual machines.

**remote object instance manager.** Creates and manages instances of RMI server beans through their associated server-side server proxies.

**repository.** In VisualAge for Java, the storage area, separate from the workspace, that contains all editions (both open and versioned) of all program elements that have ever been in the workspace, including the current editions that are in the workspace. You can add editions of program elements to the workspace from the repository.

**Repository Explorer.** In VisualAge for Java, the window from which you can view and compare editions of program elements that are in the repository.

**resource file.** A noncode file that can be referred to from your Java program in VisualAge for Java. Examples include graphics and audio files.

**RMI Access Builder.** A VisualAge for Java Enterprise tool that generates proxy beans and associated classes and interfaces so you can distribute code for remote access, enabling Java-to-Java solutions.

**RMI compiler.** The compiler that generates stub and skeleton files that facilitate RMI communication. This compiler can be automatically invoked by the RMI Access Builder or from the Tools menu item.
**RMI registry.** A server program that allows remote clients to get a reference to a server bean.

**rollback.** The process of restoring data changed by SQL statements to the state at its last commit point.

**S**

**schema.** In the Data Access Builder, the representation of the database that will be mapped.

**schema mapping.** In the Data Access Builder, a set of definitions for all attributes matching all columns for your database table, view, or SQL statement. The mapping contains the information required by the Data Access Builder to generate Java classes.

**Scrapbook.** In VisualAge for Java, the window from which you can write and test fragments of code, without having to define an encompassing class or method.

**server.** A computer that provides services to multiple users or workstations in a network; for example, a file server, a print server, or a mail server.

**server bean.** The bean that is distributed using RMI services and deployed on a server.

**server-side server proxy.** Generated by the RMI Access Builder, a companion class to the client-side server proxy, facilitating client-side server proxy communication over RMI. See also client-side server proxy and proxy bean.

**service.** A specific behavior that an object is responsible for exhibiting.

**single-byte character set.** A set of characters in which each character is represented by a 1-byte code.

**SmartGuide.** In IBM software products, an interface that guides you through performing common tasks.

**SQL predicate.** The conditional part of an SQL statement.

**sticky.** In the Visual Composition Editor, the mode that enables an application developer to add multiple beans of the same class (for example, three push buttons) without going back and forth between the beans palette and the free-form surface.

**stored procedure.** A procedure that is part of a relational database. The Data Access Builder can generate Java code that accesses stored procedures.

**superclass.** See abstract class and base class.

**T**

**Transmission Control Protocol/Internet Protocol (TCP/IP).** The basic programming foundation that carries computer messages around the globe through the Internet. The suite of protocols that defines the Internet. Originally designed for the UNIX operating system, TCP/IP software is now available for every major kind of computer operating system. To be truly on the Internet, your computer must have TCP/IP software.

**tear-off property.** A property that a developer has exposed to work with as though it were a stand-alone bean.

**thread.** A unit of execution within a process.

**tool bar.** The strip of icons along the top of the free-form surface. The tool bar contains tools to help an application developer construct composite beans.

**transaction.** In a CICS program, an event that queries or modifies a database that resides on a CICS server.

**type.** In VisualAge for Java, a generic term for a class or interface.

**U**

**Unicode.** A character coding system designed to support the interchange, processing, and display of the written texts of the diverse languages of the modern world. Unicode characters are normally encoded using 16-bit integral unsigned numbers.
uniform resource locator (URL). A standard identifier for a resource on the World Wide Web, used by Web browsers to initiate a connection. The URL includes the communications protocol to use, the name of the server, and path information identifying the objects to be retrieved on the server. A URL looks like this: http://www.matisse.net/seminars.html or telnet://well.sf.ca.us.br or news:new.newusers.question.br

user interface (UI). (1) The hardware, software, or both that enables a user to interact with a computer. (2) The visual presentation and its underlying software with which a user interacts.

V

variable. (1) A storage place within an object for a data feature. The data feature is an object, such as number or date, stored as an attribute of the containing object. (2) A bean that receives an identity at run time. A variable by itself contains no data or program logic; it must be connected such that it receives run-time identity from a bean elsewhere in the application.

versioned edition. An edition that has been versioned and can no longer be modified.

versioning. The act of making an open edition a versioned edition; that is, making the edition read-only.

view. (1) A visual bean, such as a window, push button, or entry field. (2) A visual representation that can display and change the underlying model objects of an application. Views are both the end result of developing an application and the basic unit of composition of user interfaces. Compare with visual bean. Contrast with model.

visual bean. In the Visual Composition Editor, a bean that is visible to the end user in the graphical user interface. Compare with view. Contrast with nonvisual bean.

visual programming tool. A tool that provides a means for specifying programs graphically. Application programmers write applications by manipulating graphical representations of components.

Visual Composition Editor. In VisualAge for Java, the tool where you can create graphical user interfaces from prefabricated beans and define relationships (connections) between both visual and nonvisual beans. The Visual Composition Editor is a page in the class browser.

W

Workbench. In VisualAge for Java, the main window from which you can manage the workspace, create and modify code, and open browsers and other tools.

workspace. The work area that contains all the code you are currently working on (that is, current editions). The workspace also contains the standard Java class libraries and other class libraries.

Your glossary term, acronym or abbreviation. Term definition
List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANSI</td>
<td>American National Standards Institute</td>
</tr>
<tr>
<td>API</td>
<td>application programming interface</td>
</tr>
<tr>
<td>ATM</td>
<td>automated teller machine</td>
</tr>
<tr>
<td>AWT</td>
<td>Abstract Windowing Toolkit</td>
</tr>
<tr>
<td>CAE</td>
<td>Client Access Enabler</td>
</tr>
<tr>
<td>CB</td>
<td>Component Broker</td>
</tr>
<tr>
<td>CICS</td>
<td>Customer Information Control System</td>
</tr>
<tr>
<td>CLI</td>
<td>call level interface</td>
</tr>
<tr>
<td>COM</td>
<td>Component Object Model</td>
</tr>
<tr>
<td>CORBA</td>
<td>Common Object Request Broker Architecture</td>
</tr>
<tr>
<td>COS</td>
<td>CORBA object services</td>
</tr>
<tr>
<td>DB2</td>
<td>DATABASE 2</td>
</tr>
<tr>
<td>DBCS</td>
<td>double-byte character set</td>
</tr>
<tr>
<td>DBMS</td>
<td>database management system</td>
</tr>
<tr>
<td>DL/I</td>
<td>Data Language/I</td>
</tr>
<tr>
<td>DLL</td>
<td>dynamic link library</td>
</tr>
<tr>
<td>DNS</td>
<td>domain name server</td>
</tr>
<tr>
<td>DRDA</td>
<td>Distributed Relational Database Architecture</td>
</tr>
<tr>
<td>ECD</td>
<td>edit-compile-debug</td>
</tr>
<tr>
<td>ECI</td>
<td>external call interface</td>
</tr>
<tr>
<td>FTP</td>
<td>File Transfer Protocol</td>
</tr>
<tr>
<td>GUI</td>
<td>graphical user interface</td>
</tr>
<tr>
<td>HTML</td>
<td>Hypertext Markup Language</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol</td>
</tr>
<tr>
<td>IBM</td>
<td>International Business Machines Corporation</td>
</tr>
<tr>
<td>IDE</td>
<td>integrated development environment</td>
</tr>
<tr>
<td>IDL</td>
<td>interface definition language</td>
</tr>
<tr>
<td>IIOP</td>
<td>Internet Inter-ORB Protocol</td>
</tr>
<tr>
<td>IMS</td>
<td>Information Management System</td>
</tr>
<tr>
<td>IOR</td>
<td>interoperable object reference</td>
</tr>
<tr>
<td>ITSO</td>
<td>International Technical Support Organization</td>
</tr>
<tr>
<td>JAR</td>
<td>Java archive</td>
</tr>
<tr>
<td>JDBC</td>
<td>Java Database Connectivity</td>
</tr>
<tr>
<td>JDK</td>
<td>Java Developer’s Kit</td>
</tr>
<tr>
<td>JNI</td>
<td>Java Native Interface</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>LAN</td>
<td>local area network</td>
</tr>
<tr>
<td>MOFW</td>
<td>managed object framework</td>
</tr>
<tr>
<td>MVS</td>
<td>Multiple Virtual Storage</td>
</tr>
<tr>
<td>NLS</td>
<td>National Language Support</td>
</tr>
<tr>
<td>NT</td>
<td>new technology</td>
</tr>
<tr>
<td>ODBC</td>
<td>Open Database Connectivity</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>OMT</td>
<td>object modeling technique</td>
</tr>
<tr>
<td>OO</td>
<td>object-oriented</td>
</tr>
<tr>
<td>OOA</td>
<td>object-oriented analysis</td>
</tr>
<tr>
<td>OOD</td>
<td>object-oriented design</td>
</tr>
<tr>
<td>ORB</td>
<td>Object Request Broker</td>
</tr>
<tr>
<td>OS/2</td>
<td>Operating System/2</td>
</tr>
<tr>
<td>OTS</td>
<td>object transaction service</td>
</tr>
<tr>
<td>PIN</td>
<td>personal identification number</td>
</tr>
<tr>
<td>RAD</td>
<td>rapid application development</td>
</tr>
<tr>
<td>RDBMS</td>
<td>relational database management system</td>
</tr>
<tr>
<td>RMI</td>
<td>Remote Method Invocation</td>
</tr>
<tr>
<td>SBCS</td>
<td>single-byte character set</td>
</tr>
<tr>
<td>SDK</td>
<td>Software Developer’s Kit</td>
</tr>
<tr>
<td>SQL</td>
<td>structured query language</td>
</tr>
<tr>
<td>TCP/IP</td>
<td>Transmission Control Protocol/Internet Protocol</td>
</tr>
</tbody>
</table>

© Copyright IBM Corp. 1999
| **TP** | transaction processing |
| **UOW** | unit of work |
| **URL** | uniform resource locator |
| **WWW** | World Wide Web |
Index

A
Access Modifiers
CMP  124
AccountHomeHelper
SB
lookup 28
afterBegin
BMP 191
SB 21
afterCompletion
BMP 191
SB 22
arguments
SB
ejbCreate 28
associations
EJB specification 257
authentication
security 241

B
Bean
class
SB 19
Bean-managed persistence
see BMP
beforeCompletion
BMP 191
BMP
afterBegin 191
afterCompletion 191
beforeCompletion 191
DOM 181
DTD 183
ejbActivate 176
ejbCreate 176
ejbFindByPrimaryKey 177
ejbLoad 177
ejbPassivate 177
ejbPostCreate 177
ejbRemove 177
ejbStore 178
isModified 189
legacy application  172
refresh state  188
SessionSynchronization 21, 191
setEntityContext 177, 188
singleton  186
unsetEntityContext 178
XML  172, 180
XML4J parser  181
bottom-up mapping
CMP  143
business logic
separation 9
business method
SB
remote interface  26
transactional  21
business transaction services  4

c
changing localhost
JNDI  216
client
business method invocation
SB  21
CMP
Access Modifiers  124
bottom-up mapping  143
COM.ibm.db2.jdbc.app.DB2Driver  148
complex mapping  142, 154
cornerstone-mapped field  145
Create Database Table  144
creating new EJB  111
database interactions  126
Database URL  144
ejbActivate 133, 134, 135
ejbCreate  121
association EJB object and bean instance  133
ejbLoad  133
ejbPassivate  133
ejbPostCreate  121
ejbRemove  135
ejbStore  133
findByKeySqlString  159
finder method  140
FinderHelper  121, 140
generated skeletons  158

© Copyright IBM Corp. 1999
persistence 14
CORBA 8
DCOM 8
Deployment Descriptor 13
entity beans 14
PrimaryKey 15
provider 12
reserved package 118
SanFrancisco 7
server provider 13
session beans 13
stand-alone application 3
system administrator 13
EJB beans page
VisualAge for Java 34
EJB group
organize EJB beans 36
what is an 36
EJB server
Database URL 71
password 71
user ID 71
ejbActivate
BMP 176
CMP 133, 134, 135
SB 30
passivate 22
ejbCreate
BMP 176
CMP 121
SB 21, 24
container 28
stateless 24, 28
ejbFindByPrimaryKey
BMP 177
ejbLoad
BMP 177
CMP 133
ejbLoad()
relationship 258
ejbPassivate
BMP 177
CMP 133
SB 22, 30
ejbPostCreate
BMP 177
CMP 121
SB 28
ejbRemove
BMP 177
CMP 135
relationship 258
SB 22, 30
ejbStore
BMP 178
CMP 133
enterprise bean class
definition 26
SB
remote interface 24
entity bean
specify the type 41
entity beans
inheritance 272
F
fields
SB 46
fields synchronization 107
findByPrimaryKey
CMP 159
findByPrimaryKey
SB 26
finder method
CMP 140
finder methods
SB 31
FinderException
SB 26
FinderHelper
CMP 121, 140
G
generated code 43
generated skeletons
CMP 158
generated stubs
CMP 158
getContainerManagedFields
CMP 107, 126
getInitialContext
SB 28
H
Home interface
SB 20
home interface
  registration in the naming tree 121

I
infoString
  CMP 151
inheritance
  BankAccountManyLink 284
  BankAccountSingleLink 284
  child bean 281
  discriminator 273, 282
  Distinct-table 275
  ejbActivate 279
  ejbCreate 279
  ejbPassivate 279
  ejbRemove 279
  Entity beans 272
  findByPrimaryKey 272
  getKey() 278
  getProxy() 278
  inverseRole 285
  is-a relationship 272
  parent bean 281
  polymorphism 271
  preferred type of mapping 281
  Proxy class 273
  remote interface extending remote interface 277
  Root/leaf 274
  Single-table 274
  TypeConverter 282
initial context
  naming service 204
Initial Root
  naming service 214
InitialContext
  relationship 259
isModified
  BMP 189
Isolation Level
  CMP 157
  SB 58

J
Java Naming and Directory Interface
  see JNDI
java.rmi.RemoteException
  SB 29
javax.ejb.EJBHome
  SB 31
javax.ejb.EJBObject
  SB 31
  remote interface 33
javax.ejb.Session
  SB 44
javax.ejb.SessionBean
  SB 44
javax.ejb.SessionContext
  SB 46
javax.ejb.SessionSynchronization
  SB 24
javax.naming
  JNDI 204
javax.naming.Context.INITIAL_CONTEXT_FACTORY 82
javax.naming.directory
  JNDI 204, 205
jdbc
  db2
    CMPS
      CMP 148
JNDI
  changing localhost 216
  changing the JNDI name 211
javax.naming 204
javax.naming.directory 204, 205
ProviderURL 208
JNDI name
  CMP 156
  SB 58

K
key class
  SB 43
key generation
  CMP 130

L
legacy application
  BMP 172
life cycle
  SB 21
localhost 75
Location Service Daemon 69
  Port 69
lookup 83
SB 28
LSD
  port number 206
LSD Name
  JNDI 215
LSD Port
  JNDI 215

M
ManyLink
  relationship 257
Map Browser
  CMP 142
meet-in-the-middle mapping
  CMP 143
Method
  adding methods to EJBs 49
multiple database columns
  CMP 154

N
nameQueryString
  CMP 160
naming convention
  SB 24
    home interface 31
    remote interface 33
naming service
  initial context 204
narrow 83
SB
  lookup 28
nontransactional business method
  SB 22

O
on-the-wire
  security 241

P
passivate
  SB
    stateless 22
    transaction 22
permissions
  security 244

Persistent Name Server 69
LSD Name 70
LSD Port 70
Persistent Name Server Initial Root 70
policies
  security 241
polymorphism
  inheritance 271
pool
  SB 22
pool instance
  CMP 132
port number
  LSD 206
primary key
  CMP 122, 137
  SB 20, 23
property map
  CMP 154
property name 56
ProviderURL
  JNDI 208

R
ready state
  SB 21
realm
  security 244
Reentrant
  CMP 157
reentrant
  relationship 258
refresh state
  BMP 188
regenerate a schema map
  CMP 160
relationship
  aggregation 259, 267
  backward pointing foreign-key reference 260
  backward pointing foreign-key references 257
  compound keys 269
  customized finder 258
  dynamically construction 265
  ejbLoad() 258
    uninitialized 263
ejbRemove 258
getValue() 268
introspection 259
inverse association update 264
inverse end package name 266
inverse role 266
inverseRole 261
ITSOEntity 260
ITSOEntityBean 260
lazy initialization 262
ManyLink 257, 262
ManyLink.AGGREGATION 266
ManyLink.NORMAL_ASSOCIATION 266
multi-valued association end 258
multi-valued end 259
naming rules 261
normal association 259, 266
one-to-many 257
one-to-one association 269
owner of the link 266
primary key field 266
reentrant 258
relation package 259
reentrant 258
setBank(Bank) 264
setValue(ITSOEntity newRelation) 268
SingleLink 257
single-value end 259
Remote interface
SB 20
remote interface
dclient invocation 26
SB 45
business method 26
container 26
RemoteException
SB 26
reserved package 118
rollback
SB 22
rolling back
SB 24
Run-As Mode
CMP 157

S
San Francisco
CORBA 6
SanFrancisco
EJB 7
SB

AccountHomeHelper
lookup 28
afterBegin 21
afterCompletion 22
Bean class 19
container-generated classes 20
create
home interface 28
data 23
across methods 20
conversational state 20
longer lived 20
shorter lived 20
stateless 21
ejbActivate 30
passivate 22
ejbCreate 21, 24
arguments 28
container 28
stateful 28
stateless 24
ejbPassivate 22, 30
ejbPostCreate 28
ejbRemove 22, 30
fields 46
findByPrimaryKey 26
finder methods 31
FinderException 26
getInitialContext 28
Home Interface 20
Isolation Level 58
java.rmi.RemoteException 29
javax.ejb.EJBHome 31
javax.ejb.Session 44
javax.ejb.SessionBean 24, 29
javax.ejb.SessionContext 46
JNDI name 58
key class 43
life cycle 21
lookup 28
narrow 28
naming convention 24
home interface 31
remote interface 33
nontransactional business method 22
pool 22
primary key 20, 23
ready state 21
Remote Interface 20
remote interface 45
correlation 26
enterprise bean class 24
correlation 26
correlation 26
RemoteException 26
rollback 22
rolling back 24
security Run-As Mode 59
setState 21, 30
specify the type 41
stateful 20, 24
transactional 24
stateful setting 56
stateless 24
stateless setting 56
timeout 57
transaction 24
Transaction Attribute 58
transactional context 24
transient 19
SBconversation 24
SBjavax.ejb.SessionSynchronization 24
Schema Browser
CMP 141
security
access control list 243, 245
authentication 241
Basic authentication 247
Certificate 246
Certificate authentication 247
Custom authentication 247
defaultRealm 244
Digest authentication 247
grant permissions 244
identification 241
identification 241
LDAPRealm 245
on-the-wire 241
permissions 244
policies 241
realm 243, 244
servletMgrRealm 245
UNIX or NT realm 245
security Run-As Mode
SB 59
server status 73
Session Bean
see SB
Session bean
specification 19
SessionSynchronization
BMP 21, 191
setEntityContext
BMP 177, 188
CMP 135
setSessionContext
CMP 132
SB 21, 30
simple mapping
CMP 142
CMP
simple mapping 154
SingleLink
relationship 257
singleton
BMP 186
SQL to Java Types
CMP 145
stateful
SB 20, 24
ejbCreate 28
stateful setting
SB 56
stateless
data
SB 21
SB 24
ejbCreate 28
passivate 22
stateless setting
SB 56
T
testing your EJB 66
timeout
SB 57
top-down mapping
CMP 143
transaction
afterBegin 225
afterCompletion 225
attributes 218
beforeCompletion 225
client-demarcated 218
com.ibm.ejs.container.ContainerObjectNot-
FoundException 235
concurrent access 236
container-demarcated 218
context propagation 218
CosTransactions.Current 238
distributed two-phase commit 218
EJBContext setRollbackOnly 237
getCurrent 238
getRollbackOnly 237
implementation strategies 236
initial context 230
java.rmi.RemoteException 237
javax.ejb.SessionSynchronization 225
javax.jts.TransactionRolledbackException 237
javax.jts.UserTransaction 219, 237
multiple instances 236
nested 218
OMG Object Transaction Service 1.1 (OTS) 222
programmatically control 237
SB 24
   passivate 22
SessionSynchronization 225
setRollbackOnly 237
TRANSACTION_READ_COMMITTED 221
TRANSACTION_READ_UNCOMMITTED 221
TRANSACTION_REPEATABLE_READ 221
TRANSACTION.Serializable 221
TransactionRequired exception 234
two-phase commit 218
TX_BEAN_MANAGED 220
TX_MANDATORY 220
TX_NOT_SUPPORTED 220
TX_REQUIRED 220
TX_REQUIRES_NEW 220
TX_SUPPORTS 220
Transaction Attribute
   CMP 157
   SB 58
transactional
   SB
   business method 21
   stateful 24
transactional application servers 218
transactional context
   SB 24
transient
   SB 19
type converter
ITSO Redbook Evaluation

Developing Enterprise JavaBeans with VisualAge for Java
SG24-5429-00

Your feedback is very important to help us maintain the quality of ITSO redbooks. Please complete this questionnaire and return it using one of the following methods:

- Use the online evaluation form found at http://www.redbooks.ibm.com
- Fax this form to: USA International Access Code +1 914 432 8264
- Send your comments in an Internet note to redbook@us.ibm.com

Which of the following best describes you?
- Customer
- Business Partner
- Solution Developer
- IBM employee
- None of the above

Please rate your overall satisfaction with this book using the scale:
(1 = very good, 2 = good, 3 = average, 4 = poor, 5 = very poor)

Overall Satisfaction

Please answer the following questions:

Was this redbook published in time for your needs? Yes___ No___

If no, please explain:

__________________________________________________________________________
__________________________________________________________________________
__________________________________________________________________________

What other redbooks would you like to see published?

__________________________________________________________________________
__________________________________________________________________________
__________________________________________________________________________

Comments/Suggestions: (THANK YOU FOR YOUR FEEDBACK!)

__________________________________________________________________________
__________________________________________________________________________
__________________________________________________________________________

© Copyright IBM Corp. 1999