Sun Microsystems

JSR 220: Enterprise JavaBeans™, Version 3.0

EJB Core Contracts and Requirements

EJB 3.0 Expert Group

Specification Lead:
Linda DeMichiel, Sun Microsystems
Michael Keith, Oracle Corporation

Please send comments to: ejb3-pfd-feedback@sun.com

Version 3.0, Proposed Final Draft

December 18, 2005

Specification: JSR-000220 Enterprise JavaBeans(tm) v.3.0 ("Specification")

Status: Pre-FCS, Proposed Final Draft
Release: December 21, 2005

THE SPECIFICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION THEREIN; THESE CHANGES WILL BE INCORPORATED INTO NEW VERSIONS OF THE SPECIFICATION, IF ANY. SUN MAY MAKE IMPROVEMENTS AND/OR CHANGES TO THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THE SPECIFICATION AT ANY TIME. Any use of such changes in the Specification will be governed by the then-current license for the applicable version of the Specification.

LIMITATION OF LIABILITY: TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL SUN OR ITS LICENSORS BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION, LOST REVENUE, PROFITS OR DATA, OR FOR SPECIAL, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF OR RELATED TO ANY FURNISHING, PRACTICING, MODIFYING OR ANY USE OF THE SPECIFICATION, EVEN IF SUN AND/OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You will hold Sun (and its licensors) harmless from any claims based on your use of the Specification for any purposes other than the limited right of evaluation as described above, and from any claims that later versions or releases of any Specification furnished to you are incompatible with the Specification provided to you under this license.

RESTRICTED RIGHTS LEGEND: If this Software is being acquired by or on behalf of the U.S. Government or by a U.S. Government prime contractor or subcontractor (at any tier), then the Government's rights in the Specification and accompanying documentation shall be only as set forth in this license; this is in accordance with 48 C.F.R. 227.7201 through 227.7202-4 (for Department of Defense (DoD) acquisitions) and with 48 C.F.R. 2.101 and 12.212 (for non-DoD acquisitions).

REPORT: You may wish to report any ambiguities, inconsistencies or inaccuracies you may find in connection with your evaluation of the Specification ("Feedback"). To the extent that you provide Sun with any Feedback, you hereby: (i) agree that such Feedback is provided on a non-proprietary and non-confidential basis, and (ii) grant Sun a perpetual, non-exclusive, worldwide, fully paid-up, irrevocable license, with the right to sublicense through multiple levels of sublicensees, to incorporate, disclose, and use without limitation the Feedback for any purpose related to the Specification and future versions, implementations, and test suites thereof.

GENERAL TERMS: Any action related to this Agreement will be governed by California law and controlling U.S. federal law. The U.N. Convention for the International Sale of Goods and the choice of law rules of any jurisdiction will not apply.

The Specification is subject to U.S. export control laws and may be subject to export or import regulations in other countries. Licensee agrees to comply strictly with all such laws and regulations and acknowledges that it has the responsibility to obtain such licenses to export, re-export or import as may be required after delivery to Licensee.

Neither party may assign or otherwise transfer any of its rights or obligations under this Agreement, without the prior written consent of the other party, except that Sun may assign this Agreement to an affiliated company.

This Agreement is the parties' entire agreement relating to its subject matter. It supersedes all prior or contemporaneous oral or written communications, proposals, conditions, representations and warranties and prevails over any conflicting or additional terms of any quote, order, acknowledgment, or other communication between the parties relating to its subject matter during the term of this Agreement. No modification to this Agreement will be binding, unless in writing and signed by an authorized representative of each party.

(Sun.pre-FCS.Spec.license.11.14.2003)

Table of Contents

Chapter 1 Introduction ... 23
1.1 Target Audience ... 23
1.2 What is New in EJB 3.0 ... 23
1.3 EJB 3.0 Expert Group ... 24
1.4 Organization of the Specification Documents ... 24
1.5 Document Conventions ... 25

Chapter 2 Overview ... 27
2.1 Overall Goals ... 27
2.2 EJB Roles ... 28
2.2.1 Enterprise Bean Provider ... 29
2.2.2 Application Assembler ... 29
2.2.3 Deployer ... 29
2.2.4 EJB Server Provider ... 30
2.2.5 EJB Container Provider ... 30
2.2.6 Persistence Provider ... 31
2.2.7 System Administrator ... 31
2.3 Enterprise Beans ... 32
2.3.1 Characteristics of Enterprise Beans ... 32
2.3.2 Flexible Model ... 32
2.4 Session, Entity, and Message-Driven Objects ... 33
2.4.1 Session Objects ... 33
2.4.2 Message-Driven Objects ... 34
2.4.3 Entity Objects ... 34
2.5 Standard Mapping to CORBA Protocols ... 34
2.6 Mapping to Web Service Protocols ... 35

Chapter 3 Client View of a Session Bean ... 37
3.1 Overview ... 37
3.2 Local, Remote, and Web Service Client Views ... 39
3.2.1 Remote Clients ... 39
3.2.2 Local Clients ... 39
3.2.3 Choosing Between a Local or Remote Client View ... 40
3.2.4 Web Service Clients ... 41
3.3 EJB Container ... 42
3.4 Client View of Session Beans Written to the EJB 3.0 Simplified API ... 42
3.4.1 Obtaining a Session Bean's Business Interface ... 42
3.4.2 Session Bean's Business Interface ... 42
3.4.3 Client View of Session Object's Life Cycle ... 43
3.4.4 Example of Obtaining and Using a Session Object ... 43
3.4.5 Session Object Identity ... 45
3.4.5.1 Stateful Session Beans ... 45

3.4.5.2 Stateless Session Beans ... 45
3.5 The Web Service Client View of a Stateless Session Bean ... 46
3.5.1 JAX-WS Web Service Clients ... 47
3.5.2 JAX-RPC Web Service Clients ... 48
3.6 Remote and Local Client View of Session Beans Written to the EJB 2.1 Client View API 48
3.6.1 Locating a Session Bean's Home Interface ... 49
3.6.2 Session Bean's Remote Home Interface ... 49
3.6.2.1 Creating a Session Object ... 50
3.6.2.2 Removing a Session Object ... 51
3.6.3 Session Bean's Local Home Interface ... 51
3.6.3.1 Creating a Session Object ... 51
3.6.3.2 Removing a Session Object ... 52
3.6.4 EJBObject and EJBLocalObject ... 52
3.6.5 Object Identity ... 53
3.6.6 Client view of Session Object's Life Cycle ... 53
3.6.6.1 References to Session Object Remote Interfaces ... 54
3.6.6.2 References to Session Object Local Interfaces ... 55
3.6.7 Creating and Using a Session Object ... 55
3.6.8 Object Identity ... 57
3.6.8.1 Stateful Session Beans ... 57
3.6.8.2 Stateless Session Beans ... 57
3.6.8.3 getPrimaryKey() ... 58
3.6.9 Type Narrowing ... 58

Chapter 4 Session Bean Component Contract ... 59
4.1 Overview ... 59
4.2 Conversational State of a Stateful Session Bean ... 60
4.2.1 Instance Passivation and Conversational State ... 61
4.2.2 The Effect of Transaction Rollback on Conversational State ... 63
4.3 Protocol Between a Session Bean Instance and its Container ... 63
4.3.1 Required Session Bean Metadata ... 64
4.3.2 Dependency Injection ... 64
4.3.3 The SessionContext Interface ... 64
4.3.4 Session Bean Lifecycle Callback Interceptor Methods ... 65
4.3.5 The Optional SessionBean Interface ... 66
4.3.6 Use of the MessageContext Interface by Stateless Session Beans ... 67
4.3.7 The Optional SessionSynchronization Interface for Stateful Session Beans 67
4.3.8 Timeout Callbacks for Stateless Session Beans ... 68
4.3.9 Business Method Delegation ... 68
4.3.10 Session Bean Creation ... 68
4.3.10.1 Stateful Session Beans ... 69
4.3.10.2 Stateless Session Beans ... 69
4.3.11 Business Method Interceptor Methods for Session Beans ... 70
4.3.12 Serializing Session Bean Methods ... 70
4.3.13 Transaction Context of Session Bean Methods ... 71
4.4 Stateful Session Bean State Diagram ... 71

4.4.1 Operations Allowed in the Methods of a Stateful Session Bean Class75
4.4.2 Dealing with Exceptions ... 79
4.4.3 Missed PreDestroy Calls ... 79
4.4.4 Restrictions for Transactions ... 80
4.5 Stateless Session Beans ... 81
4.5.1 Stateless Session Bean State Diagram ... 82
4.5.2 Operations Allowed in the Methods of a Stateless Session Bean Class84
4.5.3 Dealing with Exceptions ... 88
4.6 The Responsibilities of the Bean Provider ... 89
4.6.1 Classes and Interfaces ... 89
4.6.2 Session Bean Class ... 89
4.6.3 Lifecycle Callback Interceptor Methods ... 90
4.6.4 ejbCreate<METHOD> Methods ... 91
4.6.5 Business Methods ... 91
4.6.6 Session Bean's Business Interface ... 92
4.6.7 Session Bean's Remote Interface ... 93
4.6.8 Session Bean's Remote Home Interface ... 93
4.6.9 Session Bean's Local Interface ... 94
4.6.10 Session Bean's Local Home Interface ... 95
4.6.11 Session Bean's Web Service Endpoint Interface ... 95
4.7 The Responsibilities of the Container Provider ... 96
4.7.1 Generation of Implementation Classes ... 97
4.7.2 Generation of WSDL ... 97
4.7.3 Session Business Interface Implementation Class ... 97
4.7.4 Session EJBHome Class ... 98
4.7.5 Session EJBObject Class ... 98
4.7.6 Session EJBLocalHome Class ... 98
4.7.7 Session EJBLocalObject Class ... 98
4.7.8 Web Service Endpoint Implementation Class ... 98
4.7.9 Handle Classes ... 99
4.7.10 EJBMetaData Class ... 99
4.7.11 Non-reentrant Instances ... 99
4.7.12 Transaction Scoping, Security, Exceptions ... 99
4.7.13 JAX-WS and JAX-RPC Message Handlers for Web Service Endpoints99
4.7.14 SessionContext ... 100

Chapter 5 Message-Driven Bean Component Contract ... 101
5.1 Overview ... 101
5.2 Goals ... 102
5.3 Client View of a Message-Driven Bean ... 102
5.4 Protocol Between a Message-Driven Bean Instance and its Container ... 104
5.4.1 Required MessageDrivenBean Metadata ... 104
5.4.2 The Required Message Listener Interface ... 104
5.4.3 Dependency Injection ... 105
5.4.4 The MessageDrivenContext Interface ... 105
5.4.5 Message-Driven Bean Lifecycle Callback Interceptor Methods ... 106
5.4.6 The Optional MessageDrivenBean Interface ... 106
5.4.7 Timeout Callbacks ... 107

5.4.8 Message-Driven Bean Creation ... 107
5.4.9 Message Listener Interceptor Methods for Message-Driven Beans ... 107
5.4.10 Serializing Message-Driven Bean Methods ... 108
5.4.11 Concurrency of Message Processing ... 108
5.4.12 Transaction Context of Message-Driven Bean Methods ... 108
5.4.13 Activation Configuration Properties ... 108
5.4.14 Message Acknowledgment for JMS Message-Driven Beans ... 109
5.4.15 Message Selectors for JMS Message-Driven Beans ... 109
5.4.16 Association of a Message-Driven Bean with a Destination or Endpoint 109
5.4.16.1 JMS Message-Driven Beans ... 110
5.4.17 Dealing with Exceptions ... 110
5.4.18 Missed PreDestroy Callbacks ... 111
5.4.19 Replying to a JMS Message ... 111
5.5 Message-Driven Bean State Diagram ... 111
5.5.1 Operations Allowed in the Methods of a Message-Driven Bean Class 113
5.6 The Responsibilities of the Bean Provider ... 115
5.6.1 Classes and Interfaces ... 115
5.6.2 Message-Driven Bean Class ... 115
5.6.3 Message Listener Method ... 116
5.6.4 Lifecycle Callback Interceptor Methods ... 116
5.7 The Responsibilities of the Container Provider ... 117
5.7.1 Generation of Implementation Classes ... 117
5.7.2 Deployment of JMS Message-Driven Beans ... 117
5.7.3 Request/Response Messaging Types ... 117
5.7.4 Non-reentrant Instances ... 117
5.7.5 Transaction Scoping, Security, Exceptions ... 117

Chapter 6 Entity Beans and Persistence ... 119

Chapter 7 Client View of an EJB 2.1 Entity Bean ... 121
7.1 Overview ... 121
7.2 Remote Clients ... 122
7.3 Local Clients ... 123
7.4 EJB Container ... 123
7.4.1 Locating an Entity Bean’s Home Interface ... 124
7.4.2 What a Container Provides ... 124
7.5 Entity Bean’s Remote Home Interface ... 125
7.5.1 Create Methods ... 126
7.5.2 Finder Methods ... 127
7.5.3 Remove Methods ... 128
7.5.4 Home Methods ... 128
7.6 Entity Bean’s Local Home Interface ... 129
7.6.1 Create Methods ... 129
7.6.2 Finder Methods ... 130
7.6.3 Remove Methods ... 130
7.6.4 Home Methods ... 131

7.7 Entity Object's Life Cycle 131
7.7.1 References to Entity Object Remote Interfaces 133
7.7.2 References to Entity Object Local Interfaces 133
7.8 Primary Key and Object Identity 134
7.9 Entity Bean's Remote Interface 135
7.10 Entity Bean's Local Interface 136
7.11 Entity Bean's Handle 137
7.12 Entity Home Handles 138
7.13 Type Narrowing of Object References 138

Chapter 8 EJB 2.1 Entity Bean Component Contract for Container-Managed Persistence 139
8.1 Overview 140
8.2 Container-Managed Entity Persistence and Data Independence 140
8.3 The Entity Bean Provider's View of Container-Managed Persistence 142
8.3.1 The Entity Bean Provider's Programming Contract 143
8.3.2 The Entity Bean Provider's View of Persistent Relationships 145
8.3.3 Dependent Value Classes 145
8.3.4 Remove Protocols 146
8.3.4.1 Remove Methods 146
8.3.4.2 Cascade-delete 147
8.3.5 Identity of Entity Objects 147
8.3.6 Semantics of Assignment for Relationships 148
8.3.6.1 Use of the java.util.Collection API to Update Relationships 148
8.3.6.2 Use of Set Accessor Methods to Update Relationships 150
8.3.7 Assignment Rules for Relationships 151
8.3.7.1 One-to-one Bidirectional Relationships 152
8.3.7.2 One-to-one Unidirectional Relationships 153
8.3.7.3 One-to-many Bidirectional Relationships 154
8.3.7.4 One-to-many Unidirectional Relationships 158
8.3.7.5 Many-to-one Unidirectional Relationships 161
8.3.7.6 Many-to-many Bidirectional Relationships 163
8.3.7.7 Many-to-many Unidirectional Relationships 167
8.3.8 Collections Managed by the Container 170
8.3.9 Non-persistent State 170
8.3.10 The Relationship Between the Internal View and the Client View 171
8.3.10.1 Restrictions on Remote Interfaces 171
8.3.11 Mapping Data to a Persistent Store 171
8.3.12 Example 172
8.3.13 The Bean Provider's View of the Deployment Descriptor 175
8.4 The Entity Bean Component Contract 179
8.4.1 Runtime Execution Model of Entity Beans 179
8.4.2 Container Responsibilities 181
8.4.2.1 Container-Managed Fields 181
8.4.2.2 Container-Managed Relationships 181
8.5 Instance Life Cycle Contract Between the Bean and the Container 182
8.5.1 Instance Life Cycle 183
8.5.2 Bean Provider's Entity Bean Instance's View 185

8.5.3 Container's View 189
8.5.4 Read-only Entity Beans 193
8.5.5 The EntityContext Interface 194
8.5.6 Operations Allowed in the Methods of the Entity Bean Class 195
8.5.7 Finder Methods 197
8.5.7.1 Single-Object Finder Methods 197
8.5.7.2 Multi-Object Finder Methods 198
8.5.8 Select Methods 199
8.5.8.1 Single-Object Select Methods 200
8.5.8.2 Multi-Object Select Methods 200
8.5.9 Timer Notifications 201
8.5.10 Standard Application Exceptions for Entities 201
8.5.10.1 CreateException 201
8.5.10.2 DuplicateKeyException 202
8.5.10.3 FinderException 202
8.5.10.4 ObjectNotFoundException 202
8.5.10.5 RemoveException 203
8.5.11 Commit Options 203
8.5.12 Concurrent Access from Multiple Transactions 205
8.5.13 Non-reentrant and Re-entrant Instances 206
8.6 Responsibilities of the Enterprise Bean Provider 207
8.6.1 Classes and Interfaces 207
8.6.2 Enterprise Bean Class 207
8.6.3 Dependent Value Classes 208
8.6.4 ejbCreate<METHOD> Methods 208
8.6.5 ejbPostCreate<METHOD> Methods 209
8.6.6 ejbHome<METHOD> Methods 210
8.6.7 ejbSelect<METHOD> Methods 210
8.6.8 Business Methods 210
8.6.9 Entity Bean's Remote Interface 211
8.6.10 Entity Bean's Remote Home Interface 211
8.6.11 Entity Bean's Local Interface 212
8.6.12 Entity Bean's Local Home Interface 213
8.6.13 Entity Bean's Primary Key Class 214
8.6.14 Entity Bean's Deployment Descriptor 214
8.7 The Responsibilities of the Container Provider 214
8.7.1 Generation of Implementation Classes 215
8.7.2 Enterprise Bean Class 215
8.7.3 ejbFind<METHOD> Methods 216
8.7.4 ejbSelect<METHOD> Methods 216
8.7.5 Entity EJBHome Class 217
8.7.6 Entity EJBObject Class 217
8.7.7 Entity EJBLocalHome Class 217
8.7.8 Entity EJBLocalObject Class 218
8.7.9 Handle Class 218
8.7.10 Home Handle Class 218
8.7.11 Metadata Class 219
8.7.12 Instance's Re-entrance 219
8.7.13 Transaction Scoping, Security, Exceptions 219

8.7.14 Implementation of Object References ... 219
8.7.15 EntityContext ... 219
8.8 Primary Keys ... 220
8.8.1 Primary Key That Maps to a Single Field in the Entity Bean Class . 220
8.8.2 Primary Key That Maps to Multiple Fields in the Entity Bean Class220
8.8.3 Special Case: Unknown Primary Key Class ... 220

Chapter 9 EJB 2.1 Entity Bean Component Contract for Bean-Managed Persistence ... 223
9.1 Overview of Bean-Managed Entity Persistence ... 224
9.1.1 Entity Bean Provider's View of Persistence ... 224
9.1.2 Runtime Execution Model ... 225
9.1.3 Instance Life Cycle ... 227
9.1.4 The Entity Bean Component Contract ... 229
9.1.4.1 Entity Bean Instance's View ... 229
9.1.4.2 Container's View ... 233
9.1.5 Read-only Entity Beans ... 236
9.1.6 The EntityContext Interface ... 236
9.1.7 Operations Allowed in the Methods of the Entity Bean Class ... 237
9.1.8 Caching of Entity State and the ejbLoad and ejbStore Methods ... 239
9.1.8.1 ejbLoad and ejbStore with the NotSupported Transaction Attribute 240
9.1.9 Finder Method Return Type ... 241
9.1.9.1 Single-Object Finder ... 241
9.1.9.2 Multi-Object Finders ... 242
9.1.10 Timer Notifications ... 243
9.1.11 Standard Application Exceptions for Entities ... 243
9.1.11.1 CreateException ... 244
9.1.11.2 DuplicateKeyException ... 244
9.1.11.3 FinderException ... 245
9.1.11.4 ObjectNotFoundException ... 245
9.1.11.5 RemoveException ... 245
9.1.12 Commit Options ... 245
9.1.13 Concurrent Access from Multiple Transactions ... 246
9.1.14 Non-reentrant and Re-entrant Instances ... 248
9.2 Responsibilities of the Enterprise Bean Provider ... 249
9.2.1 Classes and Interfaces ... 249
9.2.2 Enterprise Bean Class ... 249
9.2.3 ejbCreate<METHOD> Methods ... 250
9.2.4 ejbPostCreate<METHOD> Methods ... 251
9.2.5 ejbFind Methods ... 251
9.2.6 ejbHome<METHOD> Methods ... 252
9.2.7 Business Methods ... 252
9.2.8 Entity Bean's Remote Interface ... 253
9.2.9 Entity Bean's Remote Home Interface ... 254
9.2.10 Entity Bean's Local Interface ... 255
9.2.11 Entity Bean's Local Home Interface ... 255
9.2.12 Entity Bean's Primary Key Class ... 256
9.3 The Responsibilities of the Container Provider ... 257

9.3.1 Generation of Implementation Classes 257
9.3.2 Entity EJBHome Class 258
9.3.3 Entity EJBObject Class 258
9.3.4 Entity EJBLocalHome Class 258
9.3.5 Entity EJBLocalObject Class 259
9.3.6 Handle Class 259
9.3.7 Home Handle Class 259
9.3.8 Metadata Class 260
9.3.9 Instance's Re-entrance 260
9.3.10 Transaction Scoping, Security, Exceptions 260
9.3.11 Implementation of Object References 260
9.3.12 EntityContext 260

Chapter 10 EJB 1.1 Entity Bean Component Contract for Container-Managed Persistence 261
10.1 EJB 1.1 Entity Beans with Container-Managed Persistence 261
10.1.1 Container-Managed Fields 262
10.1.2 ejbCreate, ejbPostCreate 263
10.1.3 ejbRemove 264
10.1.4 ejbLoad 264
10.1.5 ejbStore 264
10.1.6 Finder Hethods 265
10.1.7 Home Methods 265
10.1.8 Create Methods 265
10.1.9 Primary Key Type 265
10.1.9.1 Primary Key that Maps to a Single Field in the Entity Bean Class 265
10.1.9.2 Primary Key that Maps to Multiple Fields in the Entity Bean Class 266
10.1.9.3 Special Case: Unknown Primary Key Class 266

Chapter 11 Interceptors 267
11.1 Overview 267
11.2 Interceptor Life Cycle 268
11.3 Business Method Interceptors 269
11.3.1 Multiple Business Method Interceptor Methods 269
11.3.2 Exceptions 270
11.4 Interceptors for LifeCycle Event Callbacks 270
11.4.1 Multiple Callback Interceptor Methods for a Life Cycle Callback Event 271
11.4.2 Exceptions 272
11.5 InvocationContext 273
11.6 Default Interceptors 274
11.7 Method-level Interceptors 274
11.8 Specification of Interceptors in the Deployment Descriptor 275
11.8.1 Specification of Interceptors 275
11.8.2 Specification of the Binding of Interceptors to Beans 276
11.8.2.1 Examples 278

Chapter 12 Support for Transactions 281
12.1 Overview 281
12.1.1 Transactions 281
12.1.2 Transaction Model 282
12.1.3 Relationship to JTA and JTS 283
12.2 Sample Scenarios 283
12.2.1 Update of Multiple Databases 283
12.2.2 Messages Sent or Received Over JMS Sessions and Update of Multiple Databases284
12.2.3 Update of Databases via Multiple EJB Servers 286
12.2.4 Client-Managed Demarcation 287
12.2.5 Container-Managed Demarcation 288
12.3 Bean Provider's Responsibilities 289
12.3.1 Bean-Managed Versus Container-Managed Transaction Demarcation289
12.3.1.1 Non-Transactional Execution 289
12.3.2 Isolation Levels 290
12.3.3 Enterprise Beans Using Bean-Managed Transaction Demarcation .. 290
12.3.3.1 getRollbackOnly and setRollbackOnly Methods 295
12.3.4 Enterprise Beans Using Container-Managed Transaction Demarcation296
12.3.4.1 javax.ejb.SessionSynchronization Interface 297
12.3.4.2 javax.ejb.EJBContext.setRollbackOnly Method 298
12.3.4.3 javax.ejb.EJBContext.getRollbackOnly method 298
12.3.5 Use of JMS APIs in Transactions 298
12.3.6 Specification of a Bean's Transaction Management Type 298
12.3.7 Specification of the Transaction Attributes for a Bean's Methods.... 299
12.3.7.1 Specification of Transaction Attributes with Metadata Annotations 302
12.3.7.2 Specification of Transaction Attributes in the Deployment Descriptor303
12.4 Application Assembler's Responsibilities 305
12.5 Deployer's Responsibilities 306
12.6 Container Provider Responsibilities 306
12.6.1 Bean-Managed Transaction Demarcation 306
12.6.2 Container-Managed Transaction Demarcation for Session and Entity Beans 309
12.6.2.1 NOT_SUPPORTED 309
12.6.2.2 REQUIRED 310
12.6.2.3 SUPPORTS 310
12.6.2.4 REQUIRES_NEW 310
12.6.2.5 MANDATORY 311
12.6.2.6 NEVER 311
12.6.2.7 Transaction Attribute Summary 311
12.6.2.8 Handling of setRollbackOnly Method 312
12.6.2.9 Handling of getRollbackOnly Method 313
12.6.2.10 Handling of getUserTransaction Method 313
12.6.2.11 javax.ejb.SessionSynchronization Callbacks 313
12.6.3 Container-Managed Transaction Demarcation for Message-Driven Beans 313
12.6.3.1 NOT_SUPPORTED 314

12.6.3.2 REQUIRED 314
12.6.3.3 Handling of setRollbackOnly Method 314
12.6.3.4 Handling of getRollbackOnly Method 315
12.6.3.5 Handling of getUserTransaction Method 315
12.6.4 Local Transaction Optimization 315
12.6.5 Handling of Methods that Run with “an unspecified transaction context” 315
12.7 Access from Multiple Clients in the Same Transaction Context 316
12.7.1 Transaction “Diamond” Scenario with an Entity Object 317
12.7.2 Container Provider’s Responsibilities 318
12.7.3 Bean Provider’s Responsibilities 318
12.7.4 Application Assembler and Deployer’s Responsibilities 318
12.7.5 Transaction Diamonds involving Session Objects 318

Chapter 13 Exception Handling 321
13.1 Overview and Concepts 321
13.1.1 Application Exceptions 321
13.1.2 Goals for Exception Handling 322
13.2 Bean Provider’s Responsibilities 322
13.2.1 Application Exceptions 322
13.2.2 System Exceptions 323
13.2.2.1 javax.ejb.NoSuchEntityException 325
13.3 Container Provider Responsibilities 325
13.3.1 Exceptions from a Session Bean’s Business Interface Methods 326
13.3.2 Exceptions from Method Invoked via Session or Entity Bean’s 2.1 Client View or through Web Service Client View 328
13.3.3 Exceptions from PostConstruct and PreDestroy Methods of a Stateless Session Bean with Web Service Client View 331
13.3.4 Exceptions from Message-Driven Bean Message Listener Methods 332
13.3.5 Exceptions from PostConstruct and PreDestroy Methods of a Message-Driven Bean 333
13.3.6 Exceptions from an Enterprise Bean’s Timeout Callback Method ... 333
13.3.7 Exceptions from Other Container-invoked Callbacks 334
13.3.8 javax.ejb.NoSuchEntityException 335
13.3.9 Non-existing Stateful Session or Entity Object 336
13.3.10 Exceptions from the Management of Container-Managed Transactions 336
13.3.11 Release of Resources 336
13.3.12 Support for Deprecated Use of java.rmi.RemoteException 337
13.4 Client’s View of Exceptions 337
13.4.1 Application Exception 338
13.4.1.1 Local and Remote Clients 338
13.4.1.2 Web Service Clients 338
13.4.2 java.rmi.RemoteException and javax.ejb.EJBException 338
13.4.2.1 javax.ejb.EJBTransactionRolledbackException, javax.ejb.TransactionRolledbackLocalException, and javax.transaction.TransactionRolledbackException 339
13.4.2.2 javax.ejb.EJBTransactionRequiredException, javax.ejb.TransactionRequiredLocalException, and javax.transaction.TransactionRequiredException 340
13.4.2.3 javax.ejb.NoSuchEJBException, javax.ejb.NoSuchObjectLocalEx-

ception, and java.rmi.NoSuchObjectException340
13.5 System Administrator's Responsibilities 340

Chapter 14 Support for Distributed Interoperability 341
14.1 Support for Distribution 341
14.1.1 Client-Side Objects in a Distributed Environment 342
14.2 Interoperability Overview 342
14.2.1 Interoperability Goals 343
14.3 Interoperability Scenarios 344
14.3.1 Interactions Between Web Containers and EJB Containers for E-Commerce Applications344
14.3.2 Interactions Between Application Client Containers and EJB Containers Within an Enterprise's Intranet345
14.3.3 Interactions Between Two EJB Containers in an Enterprise's Intranet346
14.3.4 Intranet Application Interactions Between Web Containers and EJB Containers347
14.4 Overview of Interoperability Requirements 347
14.5 Remote Invocation Interoperability 348
14.5.1 Mapping Java Remote Interfaces to IDL 349
14.5.2 Mapping Value Objects to IDL 349
14.5.3 Mapping of System Exceptions 349
14.5.4 Obtaining Stub and Client View Classes 350
14.5.5 System Value Classes 350
14.5.5.1 HandleDelegate SPI 351
14.6 Transaction Interoperability 352
14.6.1 Transaction Interoperability Requirements 352
14.6.1.1 Transaction Context Wire Format 352
14.6.1.2 Two-Phase Commit Protocol 352
14.6.1.3 Transactional Policies of Enterprise Bean References 354
14.6.1.4 Exception Handling Behavior 354
14.6.2 Interoperating with Containers that do not Implement Transaction Interoperability354
14.6.2.1 Client Container Requirements 355
14.6.2.2 EJB container requirements 355
14.7 Naming Interoperability 357
14.8 Security Interoperability 358
14.8.1 Introduction 358
14.8.1.1 Trust Relationships Between Containers, Principal Propagation359
14.8.1.2 Application Client Authentication 360
14.8.2 Securing EJB Invocations 360
14.8.2.1 Secure Transport Protocol 361
14.8.2.2 Security Information in IORs 362
14.8.2.3 Propagating Principals and Authentication Data in IIOP Messages 362
14.8.2.4 Security Configuration for Containers 364
14.8.2.5 Runtime Behavior 364

Chapter 15 Enterprise Bean Environment 367
15.1 Overview 367
15.2 Enterprise Bean's Environment as a JNDI Naming Context 369
15.2.1 Sharing of Environment Entries 369
15.2.2 Annotations for Environment Entries 370
15.2.3 Annotations and Deployment Descriptors 371
15.3 Responsibilities by EJB Role 372
15.3.1 Bean Provider's Responsibilities 372
15.3.2 Application Assembler's Responsibility 373
15.3.3 Deployer's Responsibility 373
15.3.4 Container Provider Responsibility 373
15.4 Simple Environment Entries 373
15.4.1 Bean Provider's Responsibilities 374
15.4.1.1 Injection of Simple Environment Entries Using Annotations 374
15.4.1.2 Programming Interfaces for Accessing Simple Environment Entries 374
15.4.1.3 Declaration of Simple Environment Entries in the Deployment Descriptor 375
15.4.2 Application Assembler's Responsibility 379
15.4.3 Deployer's Responsibility 379
15.4.4 Container Provider Responsibility 379
15.5 EJB References 380
15.5.1 Bean Provider's Responsibilities 380
15.5.1.1 Injection of EJB References 380
15.5.1.2 EJB Reference Programming Interfaces 381
15.5.1.3 Declaration of EJB References in Deployment Descriptor 382
15.5.2 Application Assembler's Responsibilities 384
15.5.2.1 Overriding Rules 386
15.5.3 Deployer's Responsibility 386
15.5.4 Container Provider's Responsibility 387
15.6 Web Service References 387
15.7 Resource Manager Connection Factory References 388
15.7.1 Bean Provider's Responsibilities 388
15.7.1.1 Injection of Resource Manager Connection Factory References 388
15.7.1.2 Programming Interfaces for Resource Manager Connection Factory References 389
15.7.1.3 Declaration of Resource Manager Connection Factory References in Deployment Descriptor 390
15.7.1.4 Standard Resource Manager Connection Factory Types 392
15.7.2 Deployer's Responsibility 393
15.7.3 Container Provider Responsibility 393
15.7.4 System Administrator's Responsibility 395
15.8 Resource Environment References 395
15.8.1 Bean Provider's Responsibilities 395
15.8.1.1 Injection of Resource Environment References 395
15.8.1.2 Resource Environment Reference Programming Interfaces 395
15.8.1.3 Declaration of Resource Environment References in Deployment Descriptor 396
15.8.2 Deployer's Responsibility 396

15.8.3 Container Provider's Responsibility 397
15.9 Message Destination References 397
15.9.1 Bean Provider's Responsibilities 397
15.9.1.1 Injection of Message Destination References 397
15.9.1.2 Message Destination Reference Programming Interfaces .. 398
15.9.1.3 Declaration of Message Destination References in Deployment Descriptor399
15.9.2 Application Assembler's Responsibilities 400
15.9.3 Deployer's Responsibility 403
15.9.4 Container Provider's Responsibility 403
15.10 Persistence Unit References 404
15.10.1 Bean Provider's Responsibilities 404
15.10.1.1 Injection of Persistence Unit References 404
15.10.1.2 Programming Interfaces for Persistence Unit References .. 404
15.10.1.3 Declaration of Persistence Unit References in Deployment Descriptor406
15.10.2 Application Assembler's Responsibilities 407
15.10.3 Deployer's Responsibility 408
15.10.4 Container Provider Responsibility 408
15.10.5 System Administrator's Responsibility 409
15.11 Persistence Context References 409
15.11.1 Bean Provider's Responsibilities 409
15.11.1.1 Injection of Persistence Context References 409
15.11.1.2 Programming Interfaces for Persistence Context References410
15.11.1.3 Declaration of Persistence Context References in Deployment Descriptor411
15.11.2 Application Assembler's Responsibilities 413
15.11.3 Deployer's Responsibility 413
15.11.4 Container Provider Responsibility 414
15.11.5 System Administrator's Responsibility 414
15.12 UserTransaction Interface 414
15.12.1 Bean Provider's Responsibility 416
15.12.2 Container Provider's Responsibility 416
15.13 ORB References 416
15.13.1 Bean Provider's Responsibility 417
15.13.2 Container Provider's Responsibility 417
15.14 TimerService References 417
15.14.1 Bean Provider's Responsibility 417
15.14.2 Container Provider's Responsibility 417
15.15 EJBContext References 418
15.15.1 Bean Provider's Responsibility 418
15.15.2 Container Provider's Responsibility 418
15.16 Deprecated EJBContext.getEnvironment Method 418

Chapter 16 Security Management 421
16.1 Overview 421
16.2 Bean Provider's Responsibilities 423
16.2.1 Invocation of Other Enterprise Beans 423

16.2.2 Resource Access ... 423
16.2.3 Access of Underlying OS Resources ... 424
16.2.4 Programming Style Recommendations ... 424
16.2.5 Programmatic Access to Caller's Security Context ... 424
16.2.5.1 Use of getCallerPrincipal ... 425
16.2.5.2 Use of isCallerInRole ... 427
16.2.5.3 Declaration of Security Roles Referenced from the Bean's Code 427
16.3 Responsibilities of the Bean Provider and/or Application Assembler ... 429
16.3.1 Security Roles ... 430
16.3.2 Method Permissions ... 432
16.3.2.1 Specification of Method Permissions with Metadata Annotations 432
16.3.2.2 Specification of Method Permissions in the Deployment Descriptor 433
16.3.2.3 Unspecified Method Permissions ... 437
16.3.3 Linking Security Role References to Security Roles ... 437
16.3.4 Specification of Security Identities in the Deployment Descriptor ... 438
16.3.4.1 Run-as ... 438
16.4 Deployer's Responsibilities ... 439
16.4.1 Security Domain and Principal Realm Assignment ... 439
16.4.2 Assignment of Security Roles ... 440
16.4.3 Principal Delegation ... 440
16.4.4 Security Management of Resource Access ... 440
16.4.5 General Notes on Deployment Descriptor Processing ... 441
16.5 EJB Client Responsibilities ... 441
16.6 EJB Container Provider's Responsibilities ... 441
16.6.1 Deployment Tools ... 441
16.6.2 Security Domain(s) ... 442
16.6.3 Security Mechanisms ... 442
16.6.4 Passing Principals on EJB Calls ... 442
16.6.5 Security Methods in javax.ejb.EJBContext ... 443
16.6.6 Secure Access to Resource Managers ... 443
16.6.7 Principal Mapping ... 443
16.6.8 System Principal ... 443
16.6.9 Runtime Security Enforcement ... 444
16.6.10 Audit Trail ... 445
16.7 System Administrator's Responsibilities ... 445
16.7.1 Security Domain Administration ... 445
16.7.2 Principal Mapping ... 445
16.7.3 Audit Trail Review ... 445

Chapter 17 Timer Service ... 447
17.1 Overview ... 447
17.2 Bean Provider's View of the Timer Service ... 448
17.2.1 The Timer Service Interface ... 449
17.2.2 Timeout Callbacks ... 449
17.2.3 The Timer and TimerHandle Interfaces ... 451

17.2.4 Timer Identity ... 451
17.2.5 Transactions ... 451
17.3 Bean Provider's Responsibilities ... 452
17.3.1 Enterprise Bean Class ... 452
17.3.2 TimerHandle ... 452
17.4 Container's Responsibilities ... 452
17.4.1 TimerService, Timer, and TimerHandle Interfaces ... 452
17.4.2 Timer Expiration and Timeout Callback Method ... 453
17.4.3 Timer Cancellation ... 453
17.4.4 Entity Bean Removal ... 453

Chapter 18 Deployment Descriptor ... 455
18.1 Overview ... 455
18.2 Bean Provider's Responsibilities ... 456
18.3 Application Assembler's Responsibility ... 459
18.4 Container Provider's Responsibilities ... 462
18.5 Deployment Descriptor XML Schema ... 462

Chapter 19 Ejb-jar File ... 505
19.1 Overview ... 505
19.2 Deployment Descriptor ... 506
19.3 Ejb-jar File Requirements ... 506
19.4 The Client View and the ejb-client JAR File ... 507
19.5 Requirements for Clients ... 507
19.6 Example ... 508

Chapter 20 Runtime Environment ... 509
20.1 Bean Provider's Responsibilities ... 509
20.1.1 APIs Provided by Container ... 509
20.1.2 Programming Restrictions ... 511
20.2 Container Provider's Responsibility ... 513
20.2.1 Java 2 APIs Requirements ... 514
20.2.2 EJB 3.0 Requirements ... 515
20.2.3 JNDI Requirements ... 516
20.2.4 JTA 1.0.1 Requirements ... 516
20.2.5 JDBC™ 3.0 Extension Requirements ... 516
20.2.6 JMS 1.1 Requirements ... 517
20.2.7 Argument Passing Semantics ... 517
20.2.8 Other Requirements ... 518

Chapter 21 Responsibilities of EJB Roles ... 519
21.1 Bean Provider's Responsibilities ... 519
21.1.1 API Requirements ... 519
21.1.2 Packaging Requirements ... 519

21.2 Application Assembler's Responsibilities 520
21.3 EJB Container Provider's Responsibilities 520
21.4 Persistence Provider's Responsibilities 520
21.5 Deployer's Responsibilities 520
21.6 System Administrator's Responsibilities 520
21.7 Client Programmer's Responsibilities 521

Chapter 22 Related Documents 523

Appendix A Revision History 525
A.1 Public Draft 525
A.2 Proposed Final Draft 525

List of Figures

Figure 1 Session Bean Example Objects 44
Figure 2 Web Service Client View of Stateless Session Beans Deployed in a Container 47
Figure 3 Life Cycle of a Session Object. 54
Figure 4 Session Bean Example Objects 56
Figure 5 Life Cycle of a Stateful Session Bean Instance 72
Figure 6 Life Cycle of a Stateless Session Bean 82
Figure 7 Client view of Message-Driven Beans Deployed in a Container 103
Figure 8 Life Cycle of a Message-Driven Bean. 112
Figure 9 Client View of Entity Beans Deployed in a Container 125
Figure 10 Client View of Entity Object Life Cycle 132
Figure 11 View of Underlying Data Sources Accessed Through Entity Bean 142
Figure 12 Relationship Example 172
Figure 13 Overview of the Entity Bean Runtime Execution Model 180
Figure 14 Life Cycle of an Entity Bean Instance. 183
Figure 15 Multiple Clients Can Access the Same Entity Object Using Multiple Instances 205
Figure 16 Multiple Clients Can Access the Same Entity Object Using Single Instance 206
Figure 17 Client View of Underlying Data Sources Accessed Through Entity Bean 224
Figure 18 Overview of the Entity Bean Runtime Execution Model 226
Figure 19 Life Cycle of an Entity Bean Instance. 227
Figure 20 Multiple Clients Can Access the Same Entity Object Using Multiple Instances 247
Figure 21 Multiple Clients Can Access the Same Entity Object Using Single Instance 248
Figure 22 Updates to Simultaneous Databases 284
Figure 23 Message Sent to JMS Queue and Updates to Multiple Databases 285
Figure 24 Message Sent to JMS Queue Serviced by Message-Driven Bean and Updates to Multiple Databases 286
Figure 25 Updates to Multiple Databases in Same Transaction 286
Figure 26 Updates on Multiple Databases on Multiple Servers 287
Figure 27 Update of Multiple Databases from Non-Transactional Client 288
Figure 28 Transaction Diamond Scenario with Entity Object 317
Figure 29 Transaction Diamond Scenario with a Session Bean 319
Figure 30 Location of EJB Client Stubs. 342
Figure 31 Heterogeneous EJB Environment 343
Figure 32 Transaction Context Propagation 353

List of Tables

Table 1 Operations Allowed in the Methods of a Stateful Session Bean 77
Table 2 Operations Allowed in the Methods of a Stateless Session Bean 86
Table 3 Operations Allowed in the Methods of a Message-Driven Bean 114
Table 4 Operations Allowed in the Methods of an Entity Bean 195
Table 5 Comparison of Finder and Select Methods 200
Table 6 Summary of Commit-Time Options 204
Table 7 Operations Allowed in the Methods of an Entity Bean 237
Table 8 Summary of Commit-Time Options 246
Table 9 Container's Actions for Methods of Beans with Bean-Managed Transaction 308
Table 10 Transaction Attribute Summary 311
Table 11 Handling of Exceptions Thrown by a Business Interface Method of a Bean with Container-Managed Transaction Demarcation 326
Table 12 Handling of Exceptions Thrown by a Business Interface Method of a Session Bean with Bean-Managed Transaction Demarcation 328
Table 13 Handling of Exceptions Thrown by Methods of Web Service Client View or EJB 2.1 Client View of a Bean with Container-Managed Transaction Demarcation 329
Table 14 Handling of Exceptions Thrown by a Business Method of a Session Bean with Bean-Managed Transaction Demarcation 331
Table 15 Handling of Exceptions Thrown by a PostConstruct or PreDestroy Method of a Stateless Session Bean with Web Service Client View. 331
Table 16 Handling of Exceptions Thrown by a Message Listener Method of a Message-Driven Bean with Container-Managed Transaction Demarcation. 332
Table 17 Handling of Exceptions Thrown by a Message Listener Method of a Message-Driven Bean with Bean-Managed Transaction Demarcation. 333
Table 18 Handling of Exceptions Thrown by a PostConstruct or PreDestroy Method of a Message-Driven Bean. . 333
Table 19 Handling of Exceptions Thrown by the Timeout Callback Method of an Enterprise Bean with Container-Managed Transaction Demarcation. 334
Table 20 Handling of Exceptions Thrown by the Timeout Callback Method of an Enterprise Bean with Bean-Managed Transaction Demarcation. 334
Table 21 Java 2 Platform Security Policy for a Standard EJB Container 515

Chapter 1 Introduction

This is the specification of the Enterprise JavaBeans™ architecture. The Enterprise JavaBeans architecture is a architecture for the development and deployment of component-based business applications. Applications written using the Enterprise JavaBeans architecture are scalable, transactional, and multi-user secure. These applications may be written once, and then deployed on any server platform that supports the Enterprise JavaBeans specification.

1.1 Target Audience

The target audiences for this specification are the vendors of transaction processing platforms, vendors of enterprise application tools, vendors or object/relational mapping products, and other vendors who want to support the Enterprise JavaBeans (EJB) technology in their products.

Many concepts described in this document are system-level issues that are transparent to the Enterprise JavaBeans application programmer.

1.2 What is New in EJB 3.0

The Enterprise JavaBeans 3.0 architecture presented in this document extends Enterprise JavaBeans to include the following new functionality and simplifications to the earlier EJB APIs:

- *Definition of the Java language metadata annotations that can be used to annotate EJB applications. These metadata annotations are targeted at simplifying the developer's task, at reducing the number of program classes and interfaces the developer is required to implement, and at eliminating the need for the developer to provide an EJB deployment descriptor.*
- *Specification of programmatic defaults, including for metadata, to reduce the need for the developer to specify common, expected behaviors and requirements on the EJB container. A "configuration by exception" approach is taken whenever possible.*
- *Encapsulation of environmental dependencies and JNDI access through the use of annotations, dependency injection mechanisms, and simple lookup mechanisms.*
- *Simplification of the enterprise bean types.*
- *Elimination of the requirement for EJB component interfaces for session beans. The required business interface for a session bean can be a plain Java interface rather than an EJBObject, EJBLocalObject, or java.rmi.Remote interface.*

- *Elimination of the requirement for home interfaces for session beans.*
- *Simplification of entity bean persistence. Support for light-weight domain modeling, including inheritance and polymorphism.*
- *Elimination of all required interfaces for persistent entities[2].*
- *Specification of Java language metadata annotations and XML deployment descriptor elements for the object/relational mapping of persistent entities [2].*
- *Enhancements to EJB QL to provide additional functionality. Addition of projection, explicit inner and outer join operations, bulk update and delete, subqueries, and group-by. Addition of a dynamic query capability and support for native SQL queries.*
- *An interceptor facility for session beans and message-driven beans.*
- *Reduction of the requirements for usage of checked exceptions.*
- *Elimination of the requirement for the implementation of callback interfaces.*

1.3 EJB 3.0 Expert Group

The EJB 3.0 specification work is being conducted as part of JSR-220 under the Java Community Process Program. This specification is the result of the collaborative work of the members of the EJB 3.0 Expert Group. These include the following present and former expert group members: Apache Software Foundation: Jeremy Boynes; BEA: Seth White; Borland: Jishnu Mitra; E.piphany: Karthik Kothandaraman; Fujitsu-Siemens: Anton Vorsamer; Google: Cedric Beust; IBM: Jim Knutson, Randy Schnier; IONA: Conrad O'Dea; Ironflare: Hani Suleiman; JBoss: Gavin King, Bill Burke, Marc Fleury; Macromedia: Hemant Khandelwal; Nokia: Vic Zaroukian; Novell: YongMin Chen; Oracle: Michael Keith, Debu Panda, Olivier Caudron; Pramati: Deepak Anupalli; SAP: Steve Winkler, Umit Yalcinalp; SAS Institute: Rob Saccoccio; SeeBeyond: Ugo Corda; SolarMetric: Patrick Linskey; Sun Microsystems: Linda DeMichiel, Mark Reinhold; Sybase: Evan Ireland; Tibco: Shivajee Samdarshi; Tmax Soft: Woo Jin Kim; Versant: David Tinker; Xcalia: Eric Samson; Reza Behforooz; Emmanuel Bernard; Wes Biggs; David Blevins; Scott Crawford; Geoff Hendrey; Oliver Ihns; Oliver Kamps; Richard Monson-Haefel; Dirk Reinshagen; Carl Rosenberger; Suneet Shah.

1.4 Organization of the Specification Documents

This specification is organized into the following three documents:

- EJB 3.0 Simplified API
- EJB Core Contracts and Requirements
- Java Persistence

The document "*EJB 3.0 Simplified API*" provides an overview of the simplified API that is introduced by the Enterprise JavaBeans 3.0 release.

The document "*Java Persistence*" is the specification of the new API for the management of persistence together with the full specification of EJB QL. It provides the definition of the persistence API that is required to be supported under the Enterprise JavaBeans 3.0 release as well as the definition of how Java Persistence is supported for use in Java SE environments.

This document, "*EJB Core Contracts and Requirements*", defines the contracts and requirements for the use and implementation of Enterprise JavaBeans. These contracts include those for the EJB 3.0 Simplified API, as well as for the EJB 2.1 API, which is also required to be supported in this release. The contracts and requirements for implementations of this specification also include, by reference, those defined in the "*Java Persistence*" document [2].

1.5 Document Conventions

The regular Times font is used for information that is prescriptive by the EJB specification.

The italic Times font is used for paragraphs that contain descriptive information, such as notes describing typical use, or notes clarifying the text with prescriptive specification.

`The Courier font is used for code examples.`

Chapter 2

Overview

2.1 Overall Goals

The Enterprise JavaBeans (EJB) architecture has the following goals:

- *The Enterprise JavaBeans architecture will be the standard component architecture for building object-oriented business applications in the Java™ programming language.*
- *The Enterprise JavaBeans architecture will be the standard component architecture for building distributed business applications in the Java™ programming language.*
- *The Enterprise JavaBeans architecture will support the development, deployment, and use of web services.*
- *The Enterprise JavaBeans architecture will make it easy to write applications: application developers will not have to understand low-level transaction and state management details, multi-threading, connection pooling, or other complex low-level APIs.*
- *Enterprise JavaBeans applications will follow the Write Once, Run Anywhere™ philosophy of the Java programming language. An enterprise bean can be developed once, and then deployed on multiple platforms without recompilation or source code modification.*

- *The Enterprise JavaBeans architecture will address the development, deployment, and runtime aspects of an enterprise application's life cycle.*
- *The Enterprise JavaBeans architecture will define the contracts that enable tools from multiple vendors to develop and deploy components that can interoperate at runtime.*
- *The Enterprise JavaBeans architecture will make it possible to build applications by combining components developed using tools from different vendors.*
- *The Enterprise JavaBeans architecture will provide interoperability between enterprise beans and Java Platform, Enterprise Edition (Java EE) components as well as non-Java programming language applications.*
- *The Enterprise JavaBeans architecture will be compatible with existing server platforms. Vendors will be able to extend their existing products to support Enterprise JavaBeans.*
- *The Enterprise JavaBeans architecture will be compatible with other Java programming language APIs.*
- *The Enterprise JavaBeans architecture will be compatible with the CORBA protocols.*

The purpose of the EJB 3.0 release is both to continue to achieve these goals and to improve the EJB architecture by reducing its complexity from the enterprise application developer's point of view.

2.2 EJB Roles

The Enterprise JavaBeans architecture defines seven distinct roles in the application development and deployment life cycle. Each EJB Role may be performed by a different party. The EJB architecture specifies the contracts that ensure that the product of each EJB Role is compatible with the product of the other EJB Roles. The EJB specification focuses on those contracts that are required to support the development and deployment of ISV-written enterprise beans.

> *In some scenarios, a single party may perform several EJB Roles. For example, the Container Provider and the EJB Server Provider may be the same vendor. Or a single programmer may perform the two EJB Roles of the Enterprise Bean Provider and the Application Assembler.*

The following sections define the seven EJB Roles.

2.2.1 Enterprise Bean Provider

The Enterprise Bean Provider (Bean Provider for short) is the producer of enterprise beans. His or her output is an ejb-jar file that contains one or more enterprise beans. The Bean Provider is responsible for the Java classes that implement the enterprise beans' business methods; the definition of the beans' client view interfaces; and declarative specification of the beans' metadata. The beans' metadata may take the form of metadata annotations applied to the bean classes and/or an external XML deployment descriptor. The beans' metadata—whether expressed in metadata annotations or in the deployment descriptor—includes the structural information of the enterprise beans and declares all the enterprise beans' external dependencies (e.g. the names and types of resources that the enterprise beans use).

The Enterprise Bean Provider is typically an application domain expert. The Bean Provider develops reusable enterprise beans that typically implement business tasks or business entities.

The Bean Provider is not required to be an expert at system-level programming. Therefore, the Bean Provider usually does not program transactions, concurrency, security, distribution, or other services into the enterprise beans. The Bean Provider relies on the EJB container for these services.

A Bean Provider of multiple enterprise beans often performs the EJB Role of the Application Assembler.

2.2.2 Application Assembler

The Application Assembler combines enterprise beans into larger deployable application units. The input to the Application Assembler is one or more ejb-jar files produced by the Bean Provider(s). The Application Assembler outputs one or more ejb-jar files that contain the enterprise beans along with their application assembly instructions.

The Application Assembler can also combine enterprise beans with other types of application components when composing an application.

The EJB specification describes the case in which the application assembly step occurs *before* the deployment of the enterprise beans. However, the EJB architecture does not preclude the case that application assembly is performed *after* the deployment of all or some of the enterprise beans.

The Application Assembler is a domain expert who composes applications that use enterprise beans. The Application Assembler works with the enterprise bean's metadata annotations and/or deployment descriptor and the enterprise bean's client-view contract. Although the Assembler must be familiar with the functionality provided by the enterprise bean's client-view interfaces, he or she does not need to have any knowledge of the enterprise bean's implementation.

2.2.3 Deployer

The Deployer takes one or more ejb-jar files produced by a Bean Provider or Application Assembler and deploys the enterprise beans contained in the ejb-jar files in a specific operational environment. The operational environment includes a specific EJB server and container.

The Deployer must resolve all the external dependencies declared by the Bean Provider (e.g. the Deployer must ensure that all resource manager connection factories used by the enterprise beans are present in the operational environment, and he or she must bind them to the resource manager connection factory references declared in the metadata annotations or deployment descriptor), and must follow the application assembly instructions defined by the Application Assembler. To perform his or her role, the Deployer uses tools provided by the EJB Container Provider.

The Deployer's output is a set of enterprise beans (or an assembled application that includes enterprise beans) that have been customized for the target operational environment, and that are deployed in a specific EJB container.

The Deployer is an expert at a specific operational environment and is responsible for the deployment of enterprise beans. For example, the Deployer is responsible for mapping the security roles defined by the Bean Provider or Application Assembler to the user groups and accounts that exist in the operational environment in which the enterprise beans are deployed.

The Deployer uses tools supplied by the EJB Container Provider to perform the deployment tasks. The deployment process is typically two-stage:

- *The Deployer first generates the additional classes and interfaces that enable the container to manage the enterprise beans at runtime. These classes are container-specific.*
- *The Deployer performs the actual installation of the enterprise beans and the additional classes and interfaces into the EJB container.*

In some cases, a qualified Deployer may customize the business logic of the enterprise beans at their deployment. Such a Deployer would typically use the Container Provider's tools to write relatively simple application code that wraps the enterprise bean's business methods.

2.2.4 EJB Server Provider

The EJB Server Provider is a specialist in the area of distributed transaction management, distributed objects, and other lower-level system-level services. A typical EJB Server Provider is an OS vendor, middleware vendor, or database vendor.

The current EJB architecture assumes that the EJB Server Provider and the EJB Container Provider roles are the same vendor. Therefore, it does not define any interface requirements for the EJB Server Provider.

2.2.5 EJB Container Provider

The EJB Container Provider (Container Provider for short) provides:

- The deployment tools necessary for the deployment of enterprise beans.
- The runtime support for the deployed enterprise bean instances.

From the perspective of the enterprise beans, the container is a part of the target operational environment. The container runtime provides the deployed enterprise beans with transaction and security management, network distribution of remote clients, scalable management of resources, and other services that are generally required as part of a manageable server platform.

The "EJB Container Provider's responsibilities" defined by the EJB architecture are meant to be requirements for the implementation of the EJB container and server. Since the EJB specification does not architect the interface between the EJB container and server, it is left up to the vendor how to split the implementation of the required functionality between the EJB container and server.

The expertise of the Container Provider is system-level programming, possibly combined with some application-domain expertise. The focus of a Container Provider is on the development of a scalable, secure, transaction-enabled container that is integrated with an EJB server. The Container Provider insulates the enterprise bean from the specifics of an underlying EJB server by providing a simple, standard API between the enterprise bean and the container. This API is the Enterprise JavaBeans component contract.

The Container Provider typically provides support for versioning the installed enterprise bean components. For example, the Container Provider may allow enterprise bean classes to be upgraded without invalidating existing clients or losing existing enterprise bean objects.

The Container Provider typically provides tools that allow the System Administrator to monitor and manage the container and the beans running in the container at runtime.

2.2.6 Persistence Provider

The expertise of the Persistence Provider is in object/relational mapping, query processing, and caching. The focus of the Persistence Provider is on the development of a scalable, transaction-enabled runtime environment for the management of persistence.

The Persistence Provider provides the tools necessary for the object/relational mapping of persistent entities to a relational database, and the runtime support for the management of persistent entities and their mapping to the database.

The Persistence Provider insulates the persistent entities from the specifics of the underlying persistence substrate, providing a standard API between the persistent entities and the object/relational runtime.

The Persistence Provider may be the same vendor as the EJB Container vendor or the Persistence Provider may be a third-party vendor that provides a pluggable persistence environment as described in [2].

2.2.7 System Administrator

The System Administrator is responsible for the configuration and administration of the enterprise's computing and networking infrastructure that includes the EJB server and container. The System Administrator is also responsible for overseeing the well-being of the deployed enterprise beans applications at runtime.

2.3 Enterprise Beans

Enterprise JavaBeans is an architecture for component-based transaction-oriented enterprise applications.

2.3.1 Characteristics of Enterprise Beans

The essential characteristics of an enterprise bean are:

- An enterprise bean typically contains business logic that operates on the enterprise's data.
- An enterprise bean's instances are managed at runtime by a container.
- An enterprise bean can be customized at deployment time by editing its environment entries.
- Various service information, such as transaction and security attributes, may be specified together with the business logic of the enterprise bean class in the form of metadata annotations, or separately, in an XML deployment descriptor. This service information may be extracted and managed by tools during application assembly and deployment.
- Client access is mediated by the container in which the enterprise bean is deployed.
- If an enterprise bean uses only the services defined by the EJB specification, the enterprise bean can be deployed in any compliant EJB container. Specialized containers can provide additional services beyond those defined by the EJB specification. An enterprise bean that depends on such a service can be deployed only in a container that supports that service.
- An enterprise bean can be included in an assembled application without requiring source code changes or recompilation of the enterprise bean.
- The Bean Provider defines a client view of an enterprise bean. The Bean Provider can manually define the client view or it can be generated automatically by application development tools. The client view is unaffected by the container and server in which the bean is deployed. This ensures that both the beans and their clients can be deployed in multiple execution environments without changes or recompilation.

2.3.2 Flexible Model

The enterprise bean architecture is flexible enough to implement the following:

- An object that represents a stateless service.
- An object that represents a stateless service and that implements a web service endpoint.
- An object that represents a stateless service and whose invocation is asynchronous, driven by the arrival of messages.

- An object that represents a conversational session with a particular client. Such session objects automatically maintain their conversational state across multiple client-invoked methods.
- An entity object that represents a fine-grained persistent object.

Enterprise beans that are remotely accessible components are intended to be relatively coarse-grained business objects (e.g. shopping cart, stock quote service). Fine-grained objects (e.g. employee record, line items on a purchase order) should be modeled as light weight persistent entities, as described in [2], not as remotely accessible components.

Although the state management protocol defined by the Enterprise JavaBeans architecture is simple, it provides an enterprise bean developer great flexibility in managing a bean's state.

2.4 Session, Entity, and Message-Driven Objects

The Enterprise JavaBeans architecture defines the following types of enterprise bean objects:

- A session object.
- A message-driven object.
- An entity object.

2.4.1 Session Objects

A typical session object has the following characteristics:

- *Executes on behalf of a single client.*
- *Can be transaction-aware.*
- *Updates shared data in an underlying database.*
- *Does not represent directly shared data in the database, although it may access and update such data.*
- *Is relatively short-lived.*
- *Is removed when the EJB container crashes. The client has to re-establish a new session object to continue computation.*

A typical EJB container provides a scalable runtime environment to execute a large number of session objects concurrently.

*The EJB specification defines both **stateful** and **stateless session beans**. There are differences in the API between stateful session beans and stateless session beans.*

2.4.2 Message-Driven Objects

A typical message-driven object has the following characteristics:

- *Executes upon receipt of a single client message.*
- *Is asynchronously invoked.*
- *Can be transaction-aware.*
- *May update shared data in an underlying database.*
- *Does not represent directly shared data in the database, although it may access and update such data.*
- *Is relatively short-lived.*
- *Is stateless.*
- *Is removed when the EJB container crashes. The container has to re-establish a new message-driven object to continue computation.*

A typical EJB container provides a scalable runtime environment to execute a large number of message-driven objects concurrently.

2.4.3 Entity Objects

A typical entity object has the following characteristics:

- *Is part of a domain model, providing an object view of data in the database.*
- *Can be long-lived (lives as long as the data in the database).*
- *The entity and its primary key survive the crash of the EJB container. If the state of an entity was being updated by a transaction at the time the container crashed, the entity's state is restored to the state of the last committed transaction when the entity is next retrieved.*

A typical EJB container and server provide a scalable runtime environment for a large number of concurrently active entity objects.

2.5 Standard Mapping to CORBA Protocols

To help interoperability for EJB environments that include systems from multiple vendors, the EJB specification requires compliant implementations to support the interoperability protocol based on CORBA/IIOP for remote invocations from Java EE clients. Implementations may support other remote invocation protocols in addition to IIOP.

Chapter 14 summarizes the requirements for support for distribution and interoperability.

2.6 Mapping to Web Service Protocols

To support web service interoperability, the EJB specification requires compliant implementations to support XML-based web service invocations using WSDL and SOAP over HTTP in conformance with the requirements of the JAX-WS [32], JAX-RPC [25], Web Services for Java EE [31], and Web Services Metadata for the Java Platform [30] specifications.

Chapter 3 Client View of a Session Bean

This chapter describes the client view of a session bean. The session bean itself implements the business logic. The bean's container provides functionality for remote access, security, concurrency, transactions, and so forth.

While classes implemented by the container provide the client view of the session bean, the container itself is transparent to the client.

3.1 Overview

For a client, a session object is a non-persistent object that implements some business logic running on the server. One way to think of a session object is as a logical extension of the client program that runs on the server. A session object is not shared among multiple clients.

A client never directly accesses instances of the session bean's class. A client accesses a session object through the session bean's client view interface(s).

The client of a session bean may be a local client, a remote client, or a web service client, depending on the interface provided by the bean and used by the client.

A remote client of an session bean can be another enterprise bean deployed in the same or different container; or it can be an arbitrary Java program, such as an application, applet, or servlet. The client view of a session bean can also be mapped to non-Java client environments, such as CORBA clients that are not written in the Java programming language.

The interface used by a remote client of a session bean is implemented by the container as a remote business interface (or a remote EJBObject interface), and the remote client view of a session bean is location-independent. A client running in the same JVM as the session object uses the same API as a client running in a different JVM on the same or different machine.

Terminology note*: This specification uses the term* ***remote business interface*** *to refer to the business interface of an EJB 3.0 session bean that supports remote access. The term* ***remote interface*** *is used to refer to the remote component interface of the EJB 2.1 client view. The term* ***local business interface*** *refers to the local business interface of an EJB 3.0 session bean that supports local access. The term* ***local interface*** *is used to refer to the local component interface of the EJB 2.1 client view.*

Use of a session bean's local interface(s) entails the collocation of the local client and the session. The local client of an enterprise bean must be collocated in the same container as the bean. The local client view is not location-independent.

The client of a stateless session bean may be a web service client. Only a *stateless* session bean may provide a web service client view. A web service client makes use of the enterprise bean's web service client view, as described by a WSDL document. The bean's client view web service endpoint is in terms of a JAX-WS endpoint [32] or JAX-RPC endpoint interface [25]. Web service clients are discussed in Sections 3.2.4 and 3.5.

While it is possible to provide more than one client view for a session bean, typically only one will be provided.

The considerations that should be taken into account in determining the client view to be used for a session bean are further described in Section 3.2, "Local, Remote, and Web Service Client Views".

From its creation until destruction, a session object lives in a container. The container provides security, concurrency, transactions, swapping to secondary storage, and other services for the session object transparently to the client.

Each session object has an identity which, in general, *does not* survive a crash and restart of the container, although a high-end container implementation can mask container and server crashes to a remote or web service client.

Multiple enterprise beans can be installed in a container. The container allows the clients of session beans that provide local or remote client views to obtain the business interfaces and/or home interfaces of the installed enterprise beans through dependency injection or to look them up via JNDI.

The client view of a session object is independent of the implementation of the session bean and the container.

3.2 Local, Remote, and Web Service Client Views

This section describes some of the considerations the Bean Provider should take into account in determining the client view to provide for an enterprise bean.

3.2.1 Remote Clients

In EJB 3.0, a remote client accesses a session bean through the bean's remote business interface. For a session bean client and component written to the EJB 2.1 and earlier APIs, the remote client accesses the session bean through the session bean's remote home and remote component interfaces.

Compatibility Note: The EJB 2.1 and earlier API required that a remote client access the session bean by means of the session bean's remote home and remote component interfaces. These interfaces remain available for use with EJB 3.0, and are described in Section 3.6.

The remote client view of an enterprise bean is location independent. A client running in the same JVM as a bean instance uses the same API to access the bean as a client running in a different JVM on the same or different machine.

The arguments and results of the methods of the remote business interface are passed by value.

3.2.2 Local Clients

Session beans may have local clients. A local client is a client that is collocated in the same JVM with the session bean that provides the local client view and which may be tightly coupled to the bean. A local client of a session bean may be another enterprise bean or a web component.

Access to an enterprise bean through the local client view requires the collocation in the same JVM of both the local client and the enterprise bean that provides the local client view. The local client view therefore does not provide the location transparency provided by the remote client view.

In EJB 3.0, a local client accesses a session bean through the bean's local business interface. For a session bean or entity bean client and component written to the EJB 2.1 and earlier APIs, the local client accesses the enterprise bean through the bean's local home and local component interfaces. The container object that implements a local business interface is a local Java object.

Compatibility Note: The EJB 2.1 and earlier API required that a local client access the session bean by means of the session bean's local home and local component interfaces. These interfaces remain available for use with EJB 3.0, and are described in Section 3.6.

The arguments and results of the methods of the local business interface are passed "by reference"[1]. Enterprise beans that provide a local client view should therefore be coded to assume that the state of any Java object that is passed as an argument or result is potentially shared by caller and callee.

[1] More literally, references are passed by value in the JVM: an argument variable of primitive type holds a value of that primitive type; an argument variable of a reference type hold a reference to the object. See [28].

The Bean Provider must be aware of the potential sharing of objects passed through local interfaces. In particular, the Bean Provider must be careful that the state of one enterprise bean is not assigned as the state of another. In general, the references that are passed across local interfaces cannot be used outside of the immediate call chain and must never be stored as part of the state of another enterprise bean. The Bean Provider must also exercise caution in determining which objects to pass across local interfaces. This caution applies particularly in the case where there is a change in transaction or security context.

3.2.3 Choosing Between a Local or Remote Client View

The following considerations should be taken into account in determining whether a local or remote access should be used for an enterprise bean.

- The remote programming model provides location independence and flexibility with regard to the distribution of components in the deployment environment. It provides a loose coupling between the client and the bean.
- Remote calls involve pass-by-value. This copy semantics provides a layer of isolation between caller and callee, and protects against the inadvertant modification of data. The client and the bean may be programmed to assume this parameter copying.
- Remote calls are potentially expensive. They involve network latency, overhead of the client and server software stacks, argument copying, etc. Remote calls are typically programmed in a coarse-grained manner with few interactions between the client and bean.
- The objects that are passed as parameters on remote calls must be serializable.
- When the EJB 2.1 and earlier remote home and remote component interfaces are used, the narrowing of remote types requires the use of `javax.rmi.PortableRemoteObject.narrow` rather than Java language casts.
- Remote calls may involve error cases due to communication, resource usage on other servers, etc., which are not expected in local calls. When the EJB 2.1 and earlier remote home and remote component interfaces are used, the client has to explicitly program handlers for handling the `java.rmi.RemoteException`.
- Because of the overhead of the remote programming model, it is typically used for relatively coarse-grained component access.
- Local calls involve pass-by-reference. The client and the bean may be programmed to rely on pass-by-reference semantics. For example, a client may have a large document which it wants to pass on to the bean to modify, and the bean further passes on. In the local programming model the sharing of state is possible. On the other hand, when the bean wants to return a data structure to the client but the bean does not want the client to modify it, the bean explicitly copies the data structure before returning it, while in the remote programming model the bean does not copy the data structure because it assumes that the system will do the copy.
- Because local calls involve pass-by-reference, the local client and the enterprise bean providing the local client view are collocated.

- The collocation entailed by the local programming model means that the enterprise bean cannot be deployed on a node different from that of its client—thus restricting the distribution of components.

- Because the local programming model provides more lightweight access to a component, it better supports more fine-grained component access.

Note that although collocation of the remote client and the enterprise bean may allow the container to reduce the overhead of calls through a remote business interface or remote component interface, such calls are still likely to be less efficient than calls made using a local interface because any optimizations based on collocation must be done transparently.

The choice between the local and the remote programming model is a design decision that the Bean Provider makes when developing the enterprise bean.

While it is possible to provide both a remote client view and a local client view for an enterprise bean, more typically only one or the other will be provided.

3.2.4 Web Service Clients

Stateless session beans may have web service clients.

A web service client accesses a stateless session bean through the web service client view. The web service client view is described by the WSDL document for the web service that the bean implements. WSDL is an XML format for describing a web service as a set of endpoints operating on messages. The abstract description of the service is bound to an XML based protocol (SOAP [27]) and underlying transport (HTTP or HTTPS) by means of which the messages are conveyed between client and server. (See references [25], [26], [30], [31], [32]).

The web service methods of a stateless session bean provide the basis of the web service client view of the bean that is exported through WSDL. See references [30] and [25] for a description of how Java language metadata annotations may be used to specify a stateless session bean's web services client view.

Compatibility Note: EJB 2.1 required the Bean Provider to define a web service endpoint interface for a stateless session bean when he or she wished to expose the functionality of the bean as a web service endpoint through WSDL. This requirement to define the web service endpoint interface is removed in EJB 3.0. See [30].

A bean's web service client view may be initially defined by a WSDL document and then mapped to a web service endpoint that conforms to this, or an existing bean may be adapted to provide a web service client view. Reference [31] describes various design-time scenarios that may be used for EJB web service endpoints.

The web service client view of an enterprise bean is location independent and remotable.

Web service clients may be Java clients and/or clients not written in the Java programming language. A web service client that is a Java client accesses the web service by means of the JAX-WS or JAX-RPC client APIs. Access through web service clients occurs through SOAP 1.1 or 1.2 and HTTP(S).

3.3 EJB Container

An EJB container (container for short) is a system that functions as the “container” for enterprise beans. Multiple enterprise beans can be deployed in the same container. The container is responsible for making the business interfaces and/or home interfaces of its deployed enterprise beans available to the client through dependency injection and/or through lookup in the JNDI namespace.

3.4 Client View of Session Beans Written to the EJB 3.0 Simplified API

The EJB 3.0 local or remote client of a session bean written to the EJB 3.0 API accesses a session bean through its business interface. The business interface of an EJB 3.0 session bean is an ordinary Java interface, regardless of whether local or remote access is provided for the bean. In particular, the EJB 3.0 session bean business interface is not one of the interface types required by earlier versions of the EJB specification (i.e., EJBObject or EJBLocalObject interface).

3.4.1 Obtaining a Session Bean’s Business Interface

A client can obtain a session bean’s business interface through dependency injection or lookup in the JNDI namespace.

For example, the business interface `Cart` for the `CartBean` session bean may be obtained using dependency injection as follows:

```
@EJB Cart cart;
```

The `Cart` business interface could also be looked up using JNDI as shown in the following code segment using the `lookup` method provided by the EJBContext interface. In this example, a reference to the client bean’s SessionContext object is obtained through dependency injection:

```
@Resource SessionContext ctx;
Cart cart = (Cart)ctx.lookup(“cart”);
```

In both cases, the syntax used in obtaining the reference to the `Cart` business interface is independent of whether the business interface is local or remote. In the case of remote access, the actual location of a referenced enterprise bean and EJB container are, in general, transparent to the client using the remote business interface of the bean.

3.4.2 Session Bean’s Business Interface

The session bean’s interface is an ordinary Java interface. It contains the business methods of the session bean.

A reference to a session bean's business interface may be passed as a parameter or return value of a business interface method. If the reference is to a session bean's local business interface, the reference may only be passed as a parameter or return value of a local business interface method.

The business interface of a stateful session bean typically contains a method to initialize the state of the session object and a method to indicate that the client has finished using the session object and that it can be removed. See Chapter 4, "Session Bean Component Contract".

It is invalid to reference a session object that does not exist. If a stateful session bean has been removed, attempted invocations on the stateful session bean business interface result in the `javax.ejb.NoSuchEJBException`.[2]

The container provides an implementation of a session bean's business interface such that when the client invokes a method on the instance of the business interface, the business method on the session bean instance and any interceptor methods are invoked as needed.

The container makes the session bean's business interface available to the EJB 3.0 client through dependency injection and through lookup in the JNDI namespace. Section 15.5 describes in further detail how clients can obtain references to EJB business interfaces.

3.4.3 Client View of Session Object's Life Cycle

From the point of view of the client, a session object exists once the client has obtained a reference to its business interface—whether through dependency injection or from lookup of the business interface in JNDI.

A client that has a reference to a session object's business interface can then invoke business methods on the interface and/or pass the reference as a parameter or return value of a business interface method.[3]

A client may remove a stateful session bean by invoking a method of its business interface designated as a `Remove` method.

The lifecycle of a stateless session bean does not require that it be removed by the client. Removal of a stateless session bean instance is performed by the container, transparently to the client.

The contracts for session bean lifecycle are described in Chapter 4, "Session Bean Component Contract".

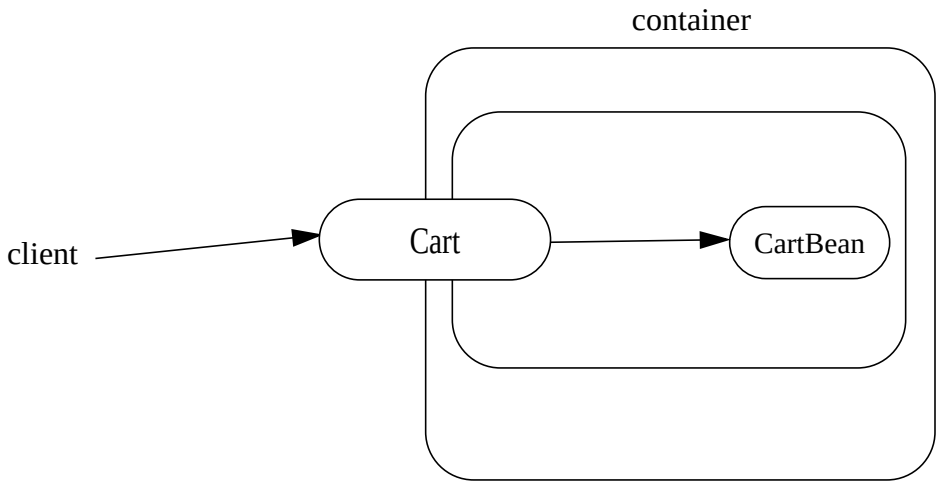
3.4.4 Example of Obtaining and Using a Session Object

An example of the session bean runtime objects is illustrated by the following diagram:

[2] This may not apply to stateless session beans; see Section 4.5.

[3] Note that the EJB 3.0 session bean business interface is not an EJBObject. It is not valid to pass a reference to the remote business interface through a bean's remote component interface.

Figure 1 Session Bean Example Objects



A client obtains a reference to a `Cart` session object, which provides a shopping service, by means of dependency injection or using JNDI lookup. The client then uses this session object to fill the cart with items and to purchase its contents. `Cart` is a stateful session.

In this example, the client obtains a reference to the Cart's business interface through dependency injection. The client then uses the business interface to initialize the session object and add a few items to it. The `startShopping` method is a business method that is provided for the initialization of the session object.

```
@EJB Cart cart;
...
cart.startShopping();
cart.addItem(66);
cart.addItem(22);
```

Finally the client purchases the contents of the shopping cart, and finishes the shopping activity.[4]

```
cart.purchase();
cart.finishShopping();
```

[4] It is part of the logic of an application designed using stateful session beans to designate a method that causes the removal of the stateful session (and thus allows for the reclamation of resources used by the session bean). This example assumes that the finishShopping method is such a Remove method. See Section 4.4 for further discussion.

3.4.5 Session Object Identity

A client can test two session bean business interface references for identity by means of the `Object.equals` and `Object.hashCode` methods.

3.4.5.1 Stateful Session Beans

A stateful session object has a unique identity that is assigned by the container at the time the object is created. A client of the stateful session bean business interface can determine if two business interface references refer to the same session object by use of the `equals` method.

For example,

```
@EJB Cart cart1;
@EJB Cart cart2;
...
if  (cart1.equals(cart1))  { // this test must return true
  ...
}
...
if  (cart1.equals(cart2))  { // this test must return false
  ...
}
```

All stateful session bean references to the same business interface for the same stateful session bean instance will be equal. Stateful session bean references to different interface types or to different session bean instances will not have the same identity.

3.4.5.2 Stateless Session Beans

All business object references of the same interface type for the same stateless session bean have the same object identity, which is assigned by the container.

For example,

```
@EJB Cart cart1;
@EJB Cart cart2;
...
if  (cart1.equals(cart1))  { // this test must return true
  ...
}
if  (cart1.equals(cart2))  { // this test must also return true
  ...
}
```

The `equals` method always returns true when used to compare references to the same business interface type of the same session bean. Session bean references to either different business interface types or different session beans will not be equal.

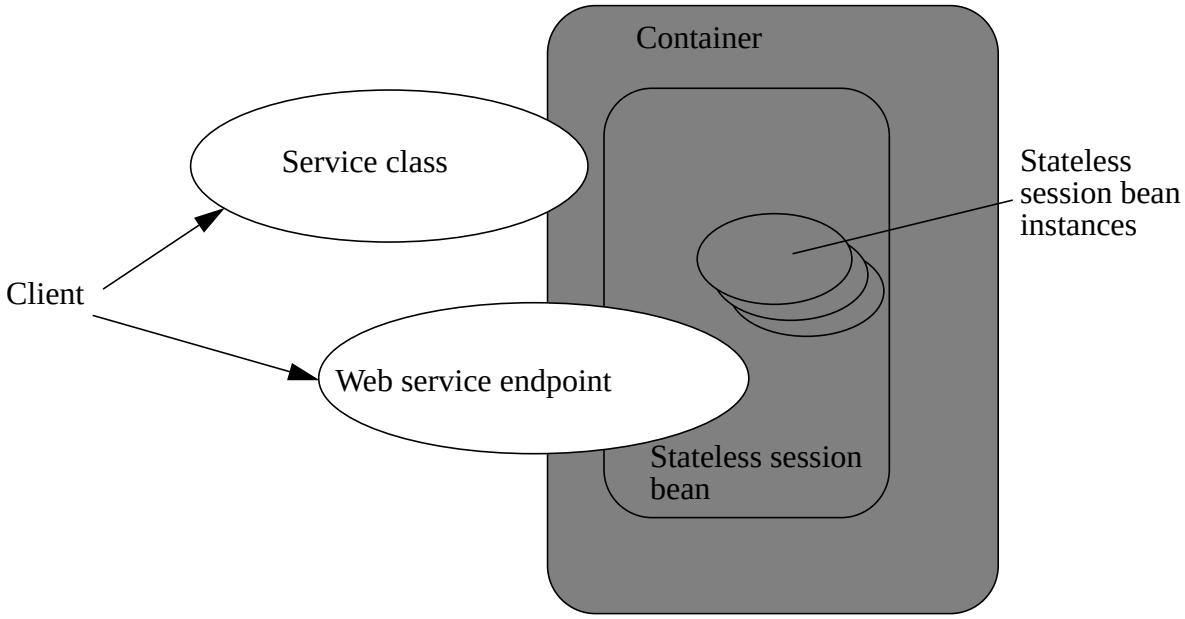
3.5 The Web Service Client View of a Stateless Session Bean

From the perspective of the client, the existence of the stateless session bean is completely hidden behind the web service endpoint that the bean implements.

The web service client's access to the web service functionality provided by a stateless session bean occurs through a web service endpoint. In the case of Java clients, this endpoint is accessed as a JAX-WS or JAX-RPC service endpoint using the JAX-WS or JAX-RPC client view APIs, as described in [32] and [25].

The following diagram illustrates the view that is provided to Java EE web service clients of a stateless session bean through the JAX-WS client view APIs.

Figure 2 Web Service Client View of Stateless Session Beans Deployed in a Container



3.5.1 JAX-WS Web Service Clients

The Java EE web service client obtains a reference to the service instance of the `javax.xml.ws.Service` class through dependency injection or using JNDI. The service class can be a generic `javax.xml.ws.Service` class or a generated service class which extends the `javax.xml.ws.Service` class. The service instance is then used to obtain a port object for the web service endpoint. The mechanisms and APIs for client web service access are described in the JAX-WS specification [32] and in the Web Services for Java EE specification [31].

The following example illustrates how a JAX-WS client obtains a reference to a web service endpoint, obtains a port object for the web service endpoint, and invokes a method on that endpoint.

```
@WebServiceRef
public StockQuoteService stockQuoteService;
...
StockQuoteProvider sqp =
     stockQuoteService.getStockQuoteProviderPort();
float quotePrice = sqp.getLastTradePrice("ACME");
...
```

The use of service references and the `@WebServiceRef` annotation are described in further detail in [32].

3.5.2 JAX-RPC Web Service Clients

The JAX-RPC web service client obtains a reference to the service object that implements the `javax.xml.rpc.Service` interface through dependency injection or using JNDI. The service interface can be a generic `javax.xml.rpc.Service` interface or a generated service interface which extends the `javax.xml.rpc.Service` interface. The service interface is then used to obtain a stub or proxy that implements the stateless session bean's web service endpoint interface. The mechanisms and APIs for client web service access are described in the JAX-RPC specification [25] and in the Web Services for Java EE specification [31].

The following example illustrates how a Java EE client looks up a web service in JNDI using a logical name called a service reference (specified using the `service-ref` element), obtains a stub instance for a web service endpoint, and invokes a method on that endpoint.

```
Context ctx = new InitialContext();
com.example.StockQuoteService sqs = (com.example.StockQuoteService)
     ctx.lookup("java:comp/env/service/StockQuoteService");
com.example.StockQuoteProvider sqp =
     sqs.getStockQuoteProviderPort();
float quotePrice = sqp.getLastTradePrice("ACME");
...
```

The use of service references and the `service-ref` deployment descriptor element is described in further detail in [31].

3.6 Remote and Local Client View of Session Beans Written to the EJB 2.1 Client View API

The remainder of this chapter describes the Session Bean client view defined by the EJB 2.1 and earlier specifications. Support for the definition and use of these earlier client interfaces is required to be provided by implementations of the EJB 3.0 specification.

3.6.1 Locating a Session Bean's Home Interface

The EJB 2.1 and earlier specifications require that the client first obtain a reference to a session bean's home interface, and then use the home interface to obtain a reference to the bean's component interface. This earlier programming model continues to be supported in the EJB 3.0. Both dependency injection and use of the EJBContext `lookup` method may be used as an alternative to the JNDI APIs to obtain a reference to the home interface.

For example, an EJB 3.0 client might obtain a reference to a bean's home interface as follows:

```
@EJB CartHome cartHome;
```

This home interface could be looked up in JNDI using the EJBContext `lookup` method as shown in the following code segment:

```
@Resource SessionContext ctx;
CartHome cartHome = (CartHome)ctx.lookup("cartHome");
```

When the EJBContext `lookup` method is used to look up a home interface, the use of `javax.rmi.PortableRemoteObject.narrow` is not required.

The following code segments illustrate how the home interface is obtained when the JNDI APIs are used directly, as was required in the EJB 2.1 programming model. For example, the remote home interface for the `Cart` session bean can be located using the following code segment:

```
Context initialContext = new InitialContext();
CartHome cartHome = (CartHome)javax.rmi.PortableRemoteObject.narrow(
     initialContext.lookup("java:comp/env/ejb/cart"),
     CartHome.class);
```

If the `Cart` session bean provides a local client view instead of a remote client view and `CartHome` is a local home interface, this lookup might be as follows:

```
Context initialContext = new InitialContext();
CartHome cartHome = (CartHome)
     initialContext.lookup("java:comp/env/ejb/cart");
```

3.6.2 Session Bean's Remote Home Interface

This section is specific to session beans that provide a remote client view using the remote and remote home interfaces.

This was the only way of providing a remote client view in EJB 2.1 and earlier releases. The remote client view provided by the business interface in EJB 3.0 as described in Section 3.4 is now to be preferred.

The container provides the implementation of the remote home interface for each session bean that defines a remote home interface that is deployed in the container. The object that implements a session bean's remote home interface is called a session EJBHome object. The container makes the session bean's remote home interface available to the client through dependency injection or through lookup in the JNDI namespace.

The remote home interface allows a client to do the following:

- Create a new session object.
- Remove a session object.
- Get the `javax.ejb.EJBMetaData` interface for the session bean. The `javax.ejb.EJBMetaData` interface is intended to allow application assembly tools to discover information about the session bean, and to allow loose client/server binding and client-side scripting.
- Obtain a handle for the remote home interface. The home handle can be serialized and written to stable storage. Later, possibly in a different JVM, the handle can be deserialized from stable storage and used to obtain back a reference of the remote home interface.

The life cycle of the distributed object implementing the remote home interface (the EJBHome object) or the local Java object implementing the local home interface (the EJBLocalHome object) is container-specific. A client application should be able to obtain a home interface, and then use it multiple times, during the client application's lifetime.

A client can pass a remote home object reference to another application. The receiving application can use the home interface in the same way that it would use a remote home object reference obtained via JNDI.

3.6.2.1 Creating a Session Object

A home interface defines one or more `create<METHOD>` methods, one for each way to create a session object. The arguments of the `create` methods are typically used to initialize the state of the created session object.

The return type of a `create<METHOD>` method on the remote home interface is the session bean's remote interface.

The following example illustrates a remote home interface that defines two `create<METHOD>` methods:

```
public interface CartHome extends javax.ejb.EJBHome {
     Cart create(String customerName, String account)
          throws RemoteException, BadAccountException,
               CreateException;
     Cart createLargeCart(String customerName, String account)
          throws RemoteException, BadAccountException,
               CreateException;
}
```

The following example illustrates how a client creates a new session object using a `create<METHOD>` method of the `CartHome` interface:

```
cartHome.create(“John”, “7506”);
```

3.6.2.2 Removing a Session Object

A remote client may remove a session object using the `remove()` method of the `javax.ejb.EJBObject` interface, or the r`emove(Handle handle)` method of the `javax.ejb.EJBHome` interface.

Because session objects do not have primary keys that are accessible to clients, invoking the `javax.ejb.EJBHome.remove(Object primaryKey)` method on a session results in a `javax.ejb.RemoveException`.

3.6.3 Session Bean’s Local Home Interface

This section is specific to session beans that provide a local client view using the local and local home interfaces.

> *This was the only way of providing a local client view in EJB 2.1 and earlier releases. The local client view provided by the business interface in EJB 3.0 as described in Section 3.4 is now to be preferred.*

The container provides the implementation of the local home interface for each session bean that defines a local home interface that is deployed in the container. The object that implements a session bean’s local home interface is called a session EJBLocalHome object. The container makes the session bean’s local home interface available to the client through JNDI.

The local home interface allows a local client to do the following:

- Create a new session object.
- Remove a session object.

A client can pass a local home object reference to another application through its local interface. A local home object reference cannot be passed as an argument or result of a method on an enterprise bean’s remote home or remote interface.

3.6.3.1 Creating a Session Object

A local home interface defines one or more `create<METHOD>` methods, one for each way to create a session object. The arguments of the `create` methods are typically used to initialize the state of the created session object.

The return type of a `create<METHOD>` method on the local home interface is the session bean’s local interface.

The following example illustrates a local home interface that defines two `create<METHOD>` methods:

```
public interface CartHome extends javax.ejb.EJBLocalHome {
     Cart create(String customerName, String account)
          throws BadAccountException, CreateException;
     Cart createLargeCart(String customerName, String account)
          throws BadAccountException, CreateException;
}
```

The following example illustrates how a client creates a new session object using a `create<METHOD>` method of the `CartHome` interface:

```
cartHome.create(“John”, “7506”);
```

3.6.3.2 Removing a Session Object

A local client may remove a session object using the `remove()` method of the `javax.ejb.EJBLocalObject` interface.

Because session objects do not have primary keys that are accessible to clients, invoking the `javax.ejb.EJBLocalHome.remove(Object primaryKey)` method on a session results in a `javax.ejb.RemoveException`.

3.6.4 EJBObject and EJBLocalObject

A remote or local client that uses the EJB 2.1 client view APIs uses the session bean’s component interface to access a session bean instance. The class that implements the session bean’s component interface is provided by the container. Instances of a session bean’s remote interface are called session **EJBObjects**. Instances of a session bean’s local interface are called session **EJBLocalObjects**.

A session EJBObject supports:

- The business logic methods of the object. The session EJBObject delegates invocation of a business method to the session bean instance.
- The methods of the `javax.ejb.EJBObject` interface. These methods allow the client to:
  - Get the session object’s remote home interface.
  - Get the session object’s handle.
  - Test if the session object is identical with another session object.
  - Remove the session object.

A session EJBLocalObject supports:

- The business logic methods of the object. The session EJBLocalObject delegates invocation of a business method to the session bean instance.
- The methods of the `javax.ejb.EJBLocalObject` interface. These methods allow the client to:

- Get the session object's local home interface.
- Test if the session object is identical with another session object.
- Remove the session object.

The implementation of the methods defined in the `javax.ejb.EJBObject` and `javax.ejb.EJBLocalObject` interfaces is provided by the container. They are not delegated to the instances of the session bean class.

3.6.5 Object Identity

Session objects are intended to be private resources used only by the client that created them. For this reason, session objects, from the client's perspective, appear anonymous. In contrast to entity objects, which expose their identity as a primary key, session objects hide their identity. As a result, the `EJBObject.getPrimaryKey()` method results in a `java.rmi.RemoteException` and the `EJBLocalObject.getPrimaryKey()` method results in a `javax.ejb.EJBException`, and the `EJBHome.remove(Object primaryKey)` and the `EJBLocalHome.remove(Object primaryKey)` methods result in a `javax.ejb.RemoveException` if called on a session bean. If the `EJBMetaData.getPrimaryKeyClass()`method is invoked on a `EJBMetaData` object for a session bean, the method throws the `java.lang.RuntimeException`.

Since all session objects hide their identity, there is no need to provide a finder for them. The home interface of a session bean must not define any finder methods.

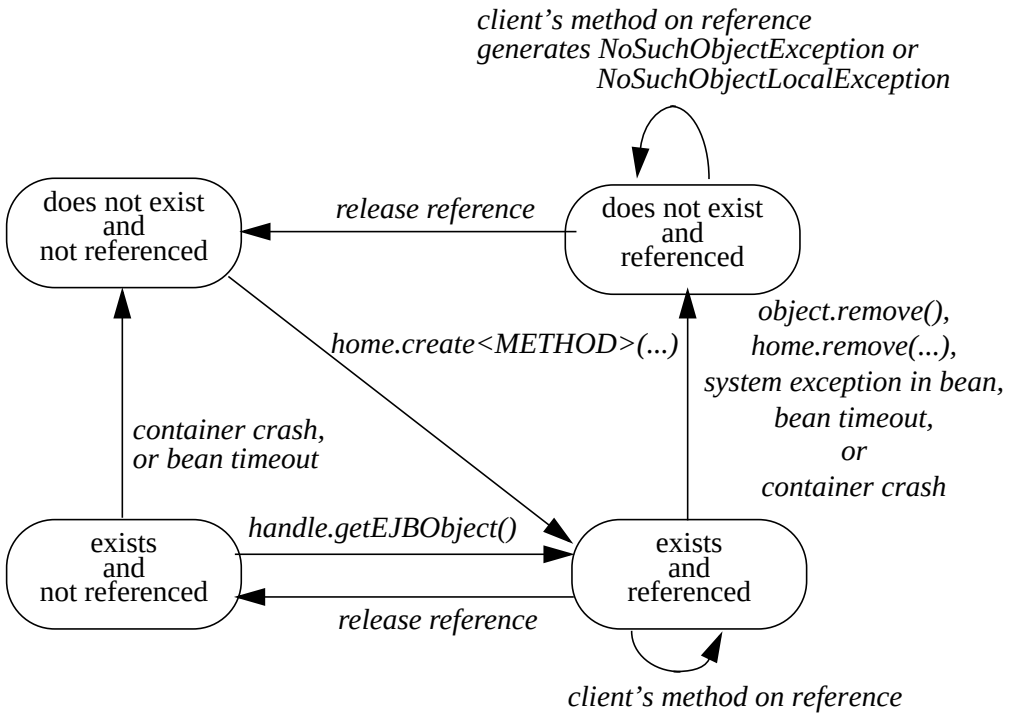
A session object handle can be held beyond the life of a client process by serializing the handle to persistent storage. When the handle is later deserialized, the session object it returns will work as long as the session object still exists on the server. (An earlier timeout or server crash may have destroyed the session object.)

A handle is not a capability, in the security sense, that would automatically grant its holder the right to invoke methods on the object. When a reference to a session object is obtained from a handle, and then a method on the session object is invoked, the container performs the usual access checks based on the caller's principal.

3.6.6 Client view of Session Object's Life Cycle

From the point of view of a local or remote client using the EJB 2.1 and earlier client view API, the life cycle of a session object is illustrated below.

Figure 3 Life Cycle of a Session Object.



A session object does not exist until it is created. When a client creates a session object, the client has a reference to the newly created session object's component interface.

3.6.6.1 References to Session Object Remote Interfaces

A client that has a reference to a session object's remote interface can then do any of the following:

- Invoke business methods defined in the session object's remote interface.
- Get a reference to the session object's remote home interface.
- Get a handle for the session object.
- Pass the reference as a parameter or return value within the scope of the client.
- Remove the session object. A container may also remove the session object automatically when the session object's lifetime expires.

It is invalid to reference a session object that does not exist. Attempted remote invocations on a stateful session object that does not exist result in a `java.rmi.NoSuchObjectException`.[5]

3.6.6.2 References to Session Object Local Interfaces

A client that has a reference to a session object's local interface can then do any of the following:

- Invoke business methods defined in the session object's local interface.
- Get a reference to the session object's local home interface.
- Pass the reference as a parameter or return value of a local interface method.
- Remove the session object. A container may also remove the session object automatically when the session object's lifetime expires.

It is invalid to reference a session object that does not exist. Attempted invocations on a stateful session object that does not exist result in `javax.ejb.NoSuchObjectLocalException`.[6]

A client can pass a local object reference or local home object reference to another application through its local interface. A local object reference or local home object reference cannot be passed as an argument or result of a method on an enterprise bean's remote home or remote interface.

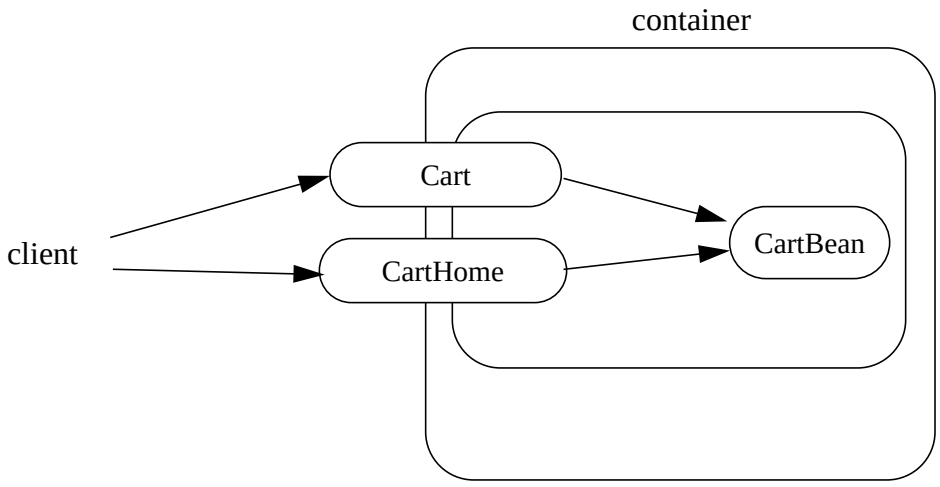
3.6.7 Creating and Using a Session Object

An example of the session bean runtime objects is illustrated by the following diagram:

[5] This may not apply to stateless session beans; see Section 4.5.

[6] This may not apply to stateless session beans; see Section 4.5.

Figure 4 Session Bean Example Objects



A client creates a remote `Cart` session object, which provides a shopping service, using a `create<METHOD>` method of the `Cart`'s remote home interface. The client then uses this session object to fill the cart with items and to purchase its contents.

Suppose that the end-user wishes to start the shopping session, suspend the shopping session temporarily for a day or two, and later complete the session. The client might implement this feature by getting the session object's handle, saving the serialized handle in persistent storage, and using it later to reestablish access to the original `Cart`.

For the following example, we start by looking up the `Cart`'s remote home interface in JNDI. We then use the remote home interface to create a `Cart` session object and add a few items to it:

```
CartHome cartHome = (CartHome)javax.rmi.PortableRemoteObject.narrow(
    initialContext.lookup(...), CartHome.class);
Cart cart = cartHome.createLargeCart(...);
cart.addItem(66);
cart.addItem(22);
```

Next we decide to complete this shopping session at a later time so we serialize a handle to this cart session object and store it in a file:

```
Handle cartHandle = cart.getHandle();
// serialize cartHandle, store in a file...
```

Finally we deserialize the handle at a later time, re-create the reference to the cart session object, and purchase the contents of the shopping cart:

```
Handle cartHandle = ...; // deserialize from a file...
Cart cart = (Cart)javax.rmi.PortableRemoteObject.narrow(
              cartHandle.getEJBObject(), Cart.class);
cart.purchase();
cart.remove();
```

3.6.8 Object Identity

3.6.8.1 Stateful Session Beans

A stateful session object has a unique identity that is assigned by the container at create time.

A remote client can determine if two remote object references refer to the same session object by invoking the `isIdentical(EJBObject otherEJBObject)` method on one of the references. A local client can determine if two local object references refer to the same session object by invoking the `isIdentical(EJBLocalObject otherEJBLocalObject)` method.

The following example illustrates the use of the `isIdentical` method for a stateful session object.

```
FooHome fooHome = ...; // obtain home of a stateful session bean
Foo foo1 = fooHome.create(...);
Foo foo2 = fooHome.create(...);

if (foo1.isIdentical(foo1)) {// this test must return true
    ...
}

if (foo1.isIdentical(foo2)) {// this test must return false
    ...
}
```

3.6.8.2 Stateless Session Beans

All session objects of the same stateless session bean within the same home have the same object identity, which is assigned by the container. If a stateless session bean is deployed multiple times (each deployment results in the creation of a distinct home), session objects from different homes will have a different identity.

The `isIdentical(EJBObject otherEJBObject)` and `isIdentical(EJBLocalObject otherEJBLocalObject)` methods always returns true when used to compare object references of two session objects of the same stateless session bean.

The following example illustrates the use of the `isIdentical` method for a stateless session object.

```
FooHome fooHome = ...; // obtain home of a stateless session bean
Foo foo1 = fooHome.create();
Foo foo2 = fooHome.create();

if (foo1.isIdentical(foo1)) {// this test returns true
    ...
}

if (foo1.isIdentical(foo2)) {// this test returns true
    ...
}
```

3.6.8.3 getPrimaryKey()

The object identifier of a session object is, in general, opaque to the client. The result of `getPrimaryKey()` on a session EJBObject reference results in `java.rmi.RemoteException`. The result of `getPrimaryKey()` on a session EJBLocalObject reference results in `javax.ejb.EJBException`.

3.6.9 Type Narrowing

A client program that is intended to be interoperable with all compliant EJB container implementations must use the `javax.rmi.PortableRemoteObject.narrow` method to perform type-narrowing of the client-side representations of the remote home and remote interfaces.[7]

Note: Programs using the cast operator for narrowing the remote and remote home interfaces are likely to fail if the container implementation uses RMI-IIOP as the underlying communication transport.

[7] Use of javax.rmi.PortableRemoteObject.narrow is not needed when the EJBContext lookup method is used to look up the remote home interface.

Chapter 4 Session Bean Component Contract

This chapter specifies the contract between a session bean and its container. It defines the life cycle of the session bean instances.

This chapter defines the developer's view of session bean state management and the container's responsibilities for managing session bean state.

4.1 Overview

A session bean instance is an instance of the session bean class. It holds the session object's state.

A session bean instance is an extension of the client that creates it:

- In the case of a stateful session bean, its fields contain a **conversational state** on behalf of the session object's client. This state describes the conversation represented by a specific client/session object pair.
- It typically reads and updates data in a database on behalf of the client.
- In the case of a stateful session bean, its lifetime is controlled by the client.

A container may also terminate a session bean instance's life after a deployer-specified time-out or as a result of the failure of the server on which the bean instance is running. For this reason, a client should be prepared to recreate a new session object if it loses the one it is using.

Typically, a session object's conversational state is not written to the database. A session bean developer simply stores it in the session bean instance's fields and assumes its value is retained for the lifetime of the instance. A developer may use an extended persistence context to store a stateful session bean's persistent conversational state. See the document "*Java Persistence*" of this specification [2].

A session bean that does not make use of Java Persistence must explicitly manage cached database data. A session bean instance must write any cached database updates prior to a transaction completion, and it must refresh its copy of any potentially stale database data at the beginning of the next transaction. A session bean must also refresh any `java.sql` Statement objects before they are used in a new transaction context. Java Persistence provides a session bean with automatic management of database data, including the automatic flushing of cached database updates upon transaction commit. See [2].

The container manages the life cycle of the session bean instances. It notifies the instances when bean action may be necessary, and it provides a full range of services to ensure that the session bean implementation is scalable and can support a large number of clients.

A session bean may be either:

- *stateless*—the session bean instances contain no conversational state between methods; any instance can be used for any client.
- *stateful*—the session bean instances contain conversational state which must be retained across methods and transactions.

4.2 Conversational State of a Stateful Session Bean

The conversational state of a *stateful* session object is defined as the session bean instance's field values, its associated interceptors and their instance field values, plus the transitive closure of the objects from these instances' fields reached by following Java object references.

To efficiently manage the size of its working set, a session bean container may need to temporarily transfer the state of an idle *stateful* session bean instance to some form of secondary storage. The transfer from the working set to secondary storage is called instance *passivation*. The transfer back is called *activation*.

In advanced cases, a session object's conversational state may contain open resources, such as open sockets and open database cursors. A container cannot retain such open resources when a session bean instance is passivated. A developer of a stateful session bean must close and open the resources in the `PrePassivate` and `PostActivate` lifecycle callback interceptor methods.[8]

[8] Note that this requirement does not apply to the EntityManager and EntityManagerFactory objects.

A container may only passivate a stateful session bean instance when the instance is *not* in a transaction.

A container must not passivate a stateful session bean with an extended persistence context unless the following conditions are met:[9]

- All the entities in the persistence context are serializable.
- The EntityManager is serializable.

A stateless session bean is never passivated.

4.2.1 Instance Passivation and Conversational State

The Bean Provider is required to ensure that the `PrePassivate` method leaves the instance fields and the fields of its associated interceptors ready to be serialized by the container. The objects that are assigned to the instance's non-`transient` fields and the non-`transient` fields of its interceptors after the `PrePassivate` method completes must be one of the following. Any interceptor classes associated with the stateful session bean must be Serializable.

- A serializable object[10].
- A `null`.
- A reference to an enterprise bean's business interface.
- A reference to an enterprise bean's remote interface, even if the stub class is not serializable.
- A reference to an enterprise bean's remote home interface, even if the stub class is not serializable.
- A reference to an entity bean's local interface, even if it is not serializable.
- A reference to an entity bean's local home interface, even if it is not serializable.
- A reference to the `SessionContext` object, even if it is not serializable.
- A reference to the environment naming context (that is, the `java:comp/env` JNDI context) or any of its subcontexts.
- A reference to the `UserTransaction` interface.
- A reference to a resource manager connection factory.
- A reference to an `EntityManager` object, even if it is not serializable.

[9] The container is not permitted to destroy a stateful session bean instance because it does not meet these requirements.

[10] Note that the Java Serialization protocol dynamically determines whether or not an object is serializable. This means that it is possible to serialize an object of a serializable subclass of a non-serializable declared field type.

- A reference to an `EntityManagerFactory` object, even if it is not serializable.
- A reference to a `javax.ejb.Timer` object.
- An object that is not directly serializable, but becomes serializable by replacing the references to an enterprise bean's business interface, an enterprise bean's home and component interfaces, the references to the `SessionContext` object, the references to the `java:comp/env` JNDI context and its subcontexts, the references to the `UserTransaction` interface, and the references to the `EntityManager` and/or `EntityManagerFactory` by serializable objects during the object's serialization.

This means, for example, that the Bean Provider must close all JDBC™ connections in the `PrePassivate` *method and assign the instance's fields storing the connections to* `null`.

The last bulleted item covers cases such as storing Collections of component interfaces in the conversational state.

The Bean Provider must assume that the content of transient fields may be lost between the `PrePassivate` and `PostActivate` notifications. Therefore, the Bean Provider should not store in a `transient` field a reference to any of the following objects: `SessionContext` object; environment JNDI naming context and any its subcontexts; business interfaces; home and component interfaces; `EntityManager` interface; `EntityManagerFactory` interface; `UserTransaction` interface.

The restrictions on the use of transient fields ensure that containers can use Java Serialization during passivation and activation.

The following are the requirements for the container.

The container performs the Java programming language Serialization (or its equivalent) of the instance's state after it invokes the `PrePassivate` method on the instance.

The container must be able to properly save and restore the reference to the business interfaces and home and component interfaces of the enterprise beans stored in the instance's state even if the classes that implement the object references are not serializable.

The container must be able to properly save and restore references to timers stored in the instance's state even if the classes that implement the timers are not serializable.

The container may use, for example, the object replacement technique that is part of the `java.io.ObjectOutputStream` *and* `java.io.ObjectInputStream` *protocol to externalize the home and component references.*

If the session bean instance stores in its conversational state an object reference to the `javax.ejb.SessionContext` interface, the container must be able to save and restore the reference across the instance's passivation. The container can replace the original `SessionContext` object with a different and functionally equivalent `SessionContext` object during activation.

If the session bean instance stores in its conversational state an object reference to the `java:comp/env` JNDI context or its subcontext, the container must be able to save and restore the object reference across the instance's passivation. The container can replace the original object with a different and functionally equivalent object during activation.

If the session bean instance stores in its conversational state an object reference to the `UserTransaction` interface, the container must be able to save and restore the object reference across the instance's passivation. The container can replace the original object with a different and functionally equivalent object during activation.

If the session bean instance stores in its conversational state an object reference to the `EntityManager` or `EntityManagerFactory` interface, the container must be able to save and restore the object reference across the instance's passivation.

The container may destroy a session bean instance if the instance does not meet the requirements for serialization after `PrePassivate`.

While the container is not required to use the Serialization protocol for the Java programming language to store the state of a passivated session instance, it must achieve the equivalent result. The one exception is that containers are not required to reset the value of `transient` fields during activation[11]. Declaring the session bean's fields as `transient` is, in general, discouraged.

4.2.2 The Effect of Transaction Rollback on Conversational State

A session object's conversational state is not transactional. It is not automatically rolled back to its initial state if the transaction in which the object has participated rolls back.

If a rollback could result in an inconsistency between a session object's conversational state and the state of the underlying database, the bean developer (or the application development tools used by the developer) must use the `afterCompletion` notification to manually reset its state.

4.3 Protocol Between a Session Bean Instance and its Container

Containers themselves make no actual service demands on the session bean instances. The container makes calls on a bean instance to provide it with access to container services and to deliver notifications issued by the container.

[11] This is to allow the container to swap out an instance's state through techniques other than the Java Serialization protocol. For example, the container's Java Virtual Machine implementation may use a block of memory to keep the instance's variables, and the container swaps the whole memory block to the disk instead of performing Java Serialization on the instance.

4.3.1 Required Session Bean Metadata

A session bean must be annotated or denoted in the deployment descriptor as a stateless or stateful session bean. A stateless session bean must be annotated with the `Stateless` annotation or denoted in the deployment descriptor as a stateless session bean. A stateful session bean must be annotated with the `Stateful` annotation or denoted in the deployment descriptor as a stateful session bean.

4.3.2 Dependency Injection

A session bean may use dependency injection mechanisms to acquire references to resources or other objects in its environment (see Chapter 15, "Enterprise Bean Environment"). If a session bean makes use of dependency injection, the container injects these references after the bean instance is created, and before any business methods are invoked on the bean instance. If a dependency on the `SessionContext` is declared, or if the bean class implements the optional `SessionBean` interface (see Section 4.3.5), the `SessionContext` is also injected at this time. If dependency injection fails, the bean instance is discarded.

Under the EJB 3.0 API, the bean class may acquire the `SessionContext` *interface through dependency injection without having to implement the* `SessionBean` *interface. In this case, the* `Resource` *annotation (or* `resource-ref` *deployment descriptor element) is used to denote the bean's dependency on the* `SessionContext`. *See Chapter 15, "Enterprise Bean Environment".*

4.3.3 The SessionContext Interface

If the bean specifies a dependency on the `SessionContext` interface (or if the bean class implements the `SessionBean` interface), the container must provide the session bean instance with a `SessionContext`. This gives the session bean instance access to the instance's context maintained by the container. The `SessionContext` interface has the following methods:

- The `getCallerPrincipal` method returns the `java.security.Principal` that identifies the invoker.
- The `isCallerInRole` method tests if the session bean instance's caller has a particular role.
- The `setRollbackOnly` method allows the instance to mark the current transaction such that the only outcome of the transaction is a rollback. Only instances of a session bean with container-managed transaction demarcation can use this method.
- The `getRollbackOnly` method allows the instance to test if the current transaction has been marked for rollback. Only instances of a session bean with container-managed transaction demarcation can use this method.
- The `getUserTransaction` method returns the `javax.transaction.UserTransaction` interface. The instance can use this interface to demarcate transactions and to obtain transaction status. Only instances of a session bean with bean-managed transaction demarcation can use this method.

- The `getTimerService` method returns the `javax.ejb.TimerService` interface. Only stateless session beans can use this method. Stateful session beans cannot be timed objects.
- The `getMessageContext` method returns the `javax.xml.rpc.handler.MessageContext` interface of a stateless session bean that implements a JAX-RPC web service endpoint. Only stateless session beans with web service endpoint interfaces can use this method.
- The `getBusinessObject(Class businessInterface)` method returns the session bean's business interface. Only session beans with an EJB 3.0 business interface can call this method.
- The `getInvokedBusinessInterface` method returns the session bean business interface through which the bean was invoked.
- The `getEJBObject` method returns the session bean's remote interface. Only session beans with a remote EJBObject interface can call this method.
- The `getEJBHome` method returns the session bean's remote home interface. Only session beans with a remote home interface can call this method.
- The `getEJBLocalObject` method returns the session bean's local interface. Only session beans with a local EJBLocalObject interface can call this method.
- The `getEJBLocalHome` method returns the session bean's local home interface. Only session beans with a local home interface can call this method.
- The `lookup` method enables the session bean to look up its environment entries in the JNDI naming context.

4.3.4 Session Bean Lifecycle Callback Interceptor Methods

The following lifecycle event callbacks are supported for session beans. Lifecycle callback interceptor methods may be defined directly on the bean class or on a separate interceptor class. See Section 4.6.3 and Chapter 11.

- `PostConstruct`
- `PreDestroy`
- `PostActivate`
- `PrePassivate`

The `PostConstruct` callback invocations occur before the first business method invocation on the bean. This is at a point after which any dependency injection has been performed by the container.

The `PostConstruct` lifecycle callback interceptor methods execute in an unspecified transaction and security context.

The `PreDestroy` callback notification signals that the instance is in the process of being removed by the container. In the `PreDestroy` lifecycle callback interceptor methods, the instance typically releases the resources that it has been holding.

The `PreDestroy` lifecycle callback interceptor methods execute in an unspecified transaction and security context.

The `PrePassivate` and `PostActivate` lifecycle callback interceptor methods are only called on stateful session bean instances.

The `PrePassivate` callback notification signals the intent of the container to passivate the instance. The `PostActivate` notification signals the instance it has just been reactivated. Because containers automatically maintain the conversational state of a stateful session bean instance when it is passivated, these notifications are not needed for most session beans. Their purpose is to allow stateful session beans to maintain those open resources that need to be closed prior to an instance's passivation and then reopened during an instance's activation.

The `PrePassivate` and `PostActivate` lifecycle callback interceptor methods execute in an unspecified transaction and security context.

4.3.5 The Optional SessionBean Interface

The session bean class is not required to implement the `SessionBean` interface or the `Serializable` interface.

Compatibility Note: The SessionBean interface was required to be implemented by the session bean class in earlier versions of the Enterprise JavaBeans specification. In EJB 3.0, the functionality previously provided by the SessionBean interface is available to the bean class through selective use of dependency injection (of the SessionContext) and optional lifecycle callback interceptor methods.

The `SessionBean` interface defines four methods: `setSessionContext`, `ejbRemove`, `ejbPassivate`, and `ejbActivate`.

The `setSessionContext` method is called by the bean's container to associate a session bean instance with its context maintained by the container. Typically a session bean instance retains its session context as part of its state.

The `ejbRemove` notification signals that the instance is in the process of being removed by the container. In the `ejbRemove` method, the instance typically releases the same resources that it releases in the `ejbPassivate` method.

> *Under the EJB 3.0 API, the bean class may optionally define a* `PreDestroy` *lifecycle callback interceptor method for notification of the container's removal of the bean instance.*

The `ejbPassivate` notification signals the intent of the container to passivate the instance. The `ejbActivate` notification signals the instance it has just been reactivated. Their purpose is to allow stateful session beans to maintain those open resources that need to be closed prior to an instance's passivation and then reopened during an instance's activation. The `ejbPassivate` and `ejbActivate` methods are only called on stateful session bean instances.

Under the EJB 3.0 API, the bean class may optionally define `PrePassivate` *and/or* `PostActivate` *lifecycle callback interceptor methods for notification of the passivation/activation of the bean instance.*

This specification requires that the `ejbRemove`, `ejbActivate`, and `ejbPassivate` methods of the `SessionBean` interface, and the `ejbCreate` method of a stateless session bean be treated as `PreDestroy`, `PostActivate`, `PrePassivate` and `PostConstruct` life cycle callback interceptor methods, respectively.

If the session bean implements the `SessionBean` interface, the `PreDestroy` annotation can only be applied to the `ejbRemove` method; the `PostActivate` annotation can only be applied to the `ejbActivate` method; the `PrePassivate` annotation can only be applied to the `ejbPassivate` method. Similar requirements apply to use of deployment descriptor metadata as an alternative to the use of annotations.

4.3.6 Use of the MessageContext Interface by Stateless Session Beans

A stateless session bean that implements a web service endpoint using the JAX-RPC APIs contracts access the JAX-RPC `MessageContext` interface by means of the `SessionContext.getMessageContext` method. The `MessageContext` interface allows the stateless session bean instance to see the SOAP message for the web service endpoint, as well as the properties set by the JAX-RPC SOAP message handlers, if any. The stateless session bean may use the `MessageContext` interface to set properties for the JAX-RPC message response handlers, if any.

A stateless session bean that implements a web service endpoint using the JAX-WS contracts should use the JAX-WS `WebServiceContext`, which can be injected by use of the `Resource` annotation. The `WebServiceContext` interface allows the stateless session bean instance to see the SOAP message for the web service endpoint, as well as the properties set by the JAX-WS message handlers, if any. The stateless session bean may use the `WebServiceContext` interface to set properties for the JAX-WS message response handlers, if any. See [32].

4.3.7 The Optional SessionSynchronization Interface for Stateful Session Beans

A stateful session bean class can optionally implement the `javax.ejb.SessionSynchronization` interface. This interface provides the session bean instances with transaction synchronization notifications. The instances can use these notifications, for example, to manage database data they may cache within transactions—e.g., if Java Persistence is not used.

The `afterBegin` notification signals a session bean instance that a new transaction has begun. The container invokes this method before the first business method within a transaction (which is not necessarily at the beginning of the transaction). The `afterBegin` notification is invoked with the transaction context. The instance may do any database work it requires within the scope of the transaction.

The `beforeCompletion` notification is issued when a session bean instance's client has completed work on its current transaction but prior to committing the resource managers used by the instance. At this time, the instance should write out any database updates it has cached. The instance can cause the transaction to roll back 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 that the transaction has committed. A status of `false` indicates that a rollback has occurred. Since a session bean instance's conversational state is not transactional, it may need to manually reset its state if a rollback occurred.

All container providers must support `SessionSynchronization`. It is optional only for the bean implementor. If a bean class implements `SessionSynchronization`, the container must invoke the `afterBegin`, `beforeCompletion`, and `afterCompletion` notifications as required by the specification.

Only a stateful session bean with container-managed transaction demarcation may implement the `SessionSynchronization` interface. A stateless session bean must not implement the `SessionSynchronization` interface.

There is no need for a session bean with bean-managed transaction demarcation to rely on the synchronization call backs because the bean is in control of the commit—the bean knows when the transaction is about to be committed and it knows the outcome of the transaction commit.

4.3.8 Timeout Callbacks for Stateless Session Beans

A stateless session bean can be registered with the EJB Timer Service for time-based event notifications if it provides a timeout callback method. The container invokes the bean instance's timeout callback method when a timer for the bean has expired. See Chapter 17, "Timer Service". Stateful session beans cannot be registered with the EJB Timer Service, and therefore should not implement timeout callback methods.

4.3.9 Business Method Delegation

The session bean's business interface, component interface, or web service endpoint defines the business methods callable by a client.

The container classes that implement these are generated by the container tools. The class that implements the session bean's business interface and the class that implements a session bean's component interface delegate an invocation of a business method to the matching business method that is implemented in the session bean class. The class that handles requests to the web service endpoint invokes the stateless session bean method that matches the web service method corresponding to the SOAP request.

4.3.10 Session Bean Creation

The container creates an instance of a session bean as follows. First, the container calls the bean class's `newInstance` method to create a new session bean instance. Second, the container injects the bean's `SessionContext`, if applicable, and performs any other dependency injection as specified by metadata annotations on the bean class or by the deployment descriptor. Third, the container calls the `PostConstruct` lifecycle callback interceptor methods for the bean, if any. The additional steps described below apply if the session bean is invoked through the EJB 2.1 client view APIs.

4.3.10.1 Stateful Session Beans

If the bean is a stateful session bean and the client has used one of the `create<METHOD>` methods defined in the session bean's home or local home interface to create the bean, the container then calls the instance's initialization method whose signature matches the signature of the `create<METHOD>` invoked by the client, passing to the method the input parameters sent from the client. If the bean class was written to the EJB 3.0 API, and has been adapted for use with an earlier client view, this initialization method is a matching `Init` method, as designated by use of the `Init` annotation or `init-method` deployment descriptor element. If the bean class was written to the EJB 2.1 or earlier API, this initialization method is a matching `ejbCreate<METHOD>` method, as described in Section 4.6.4.

Each stateful session bean class that has a home interface must have at least one such initialization method. The number and signatures of a session bean's initialization methods are specific to each session bean class. Since a stateful session bean represents a specific, private conversation between the bean and its client, its initialization parameters typically contain the information the client uses to customize the bean instance for its use.

4.3.10.2 Stateless Session Beans

A stateless session bean that has an EJB 2.1 local or remote client view has a single `create` method on its home interface. In this case, EJB 2.1 required the stateless session bean class to have a single `ejbCreate` method have no arguments. Under EJB 3.0, it is not required that a stateless session bean have an `ejbCreate` method, even when it has a home interface. An EJB 3.0 stateless session bean class may have a `PostConstruct` method, as described in Section 4.3.4.

If the stateless session bean instance has an `ejbCreate` method, the container treats the `ejbCreate` method as the instance's `PostConstruct` method, and, in this case, the `PostConstruct` annotation (or deployment descriptor metadata) can only be applied to the bean's `ejbCreate` method.

Since stateless session bean instances are typically pooled, the time of the client's invocation of the `create` *method need not have any direct relationship to the container's invocation of the* `PostConstruct`/`ejbCreate` *method on the stateless session bean instance.*

A stateless session bean that provides only a web service client view has no `create` method. If the `ejbCreate` method required by EJB 2.1 is present, it is likewise treated by the container as the instance's `PostConstruct` method, and is invoked when the container needs to create a new session bean instance in order to service a client request.

4.3.11 Business Method Interceptor Methods for Session Beans

The `AroundInvoke` interceptor methods are supported for session beans. These interceptor methods may be defined on the bean class and/or on interceptor classes, and apply to the handling of the invocation of the business methods of the bean's business interface, component interface, and/or web service endpoint.

For stateful session beans that implement the `SessionSynchronization` interface, `afterBegin` occurs before any `AroundInvoke` method invocation, and `beforeCompletion` after all `AroundInvoke` invocations are finished.

Interceptors are described in Chapter 11, "Interceptors".

4.3.12 Serializing Session Bean Methods

The container serializes calls to each session bean instance. Most containers will support many instances of a session bean executing concurrently; however, each instance sees only a serialized sequence of method calls. Therefore, a session bean does not have to be coded as reentrant.

The container must serialize all the container-invoked callbacks (that is, the business method interceptor methods, lifecycle callback interceptor methods, timeout callback methods, `beforeCompletion`, and so on), and it must serialize these callbacks with the client-invoked business method calls.

Clients are not allowed to make concurrent calls to a stateful session object. If a client-invoked business method is in progress on an instance when another client-invoked call, from the same or different client, arrives at the same instance of a stateful session bean class, if the second client is a client of the bean's business interface, the concurrent invocation may result in the second client receiving the `javax.ejb.ConcurrentAccessException`[12]. If the EJB 2.1 client view is used, the container may throw the `java.rmi.RemoteException` if the second client is a remote client, or the `javax.ejb.EJBException` if the second client is a local client. This restriction does not apply to a stateless session bean because the container routes each request to a different instance of the session bean class.

> *In certain special circumstances (e.g., to handle clustered web container architectures), the container may instead queue or serialize such concurrent requests. Clients, however, cannot rely on this behavior.*

[12] The `javax.ejb.ConcurrentAccessException` is a subclass of the `javax.ejb.EJBException`. If the business interface is a remote business interface that extends `java.rmi.Remote`, the client will receive the `java.rmi.RemoteException` instead.

4.3.13 Transaction Context of Session Bean Methods

The implementation of a method defined in a session bean's business interface or component interface, a web service method, or a timeout callback method is invoked in the scope of a transaction determined by the transaction attribute specified in the bean's metadata annotations or deployment descriptor.

A session bean's `afterBegin` and `beforeCompletion` methods are always called with the same transaction context as the business methods executed between the `afterBegin` and `beforeCompletion` methods.

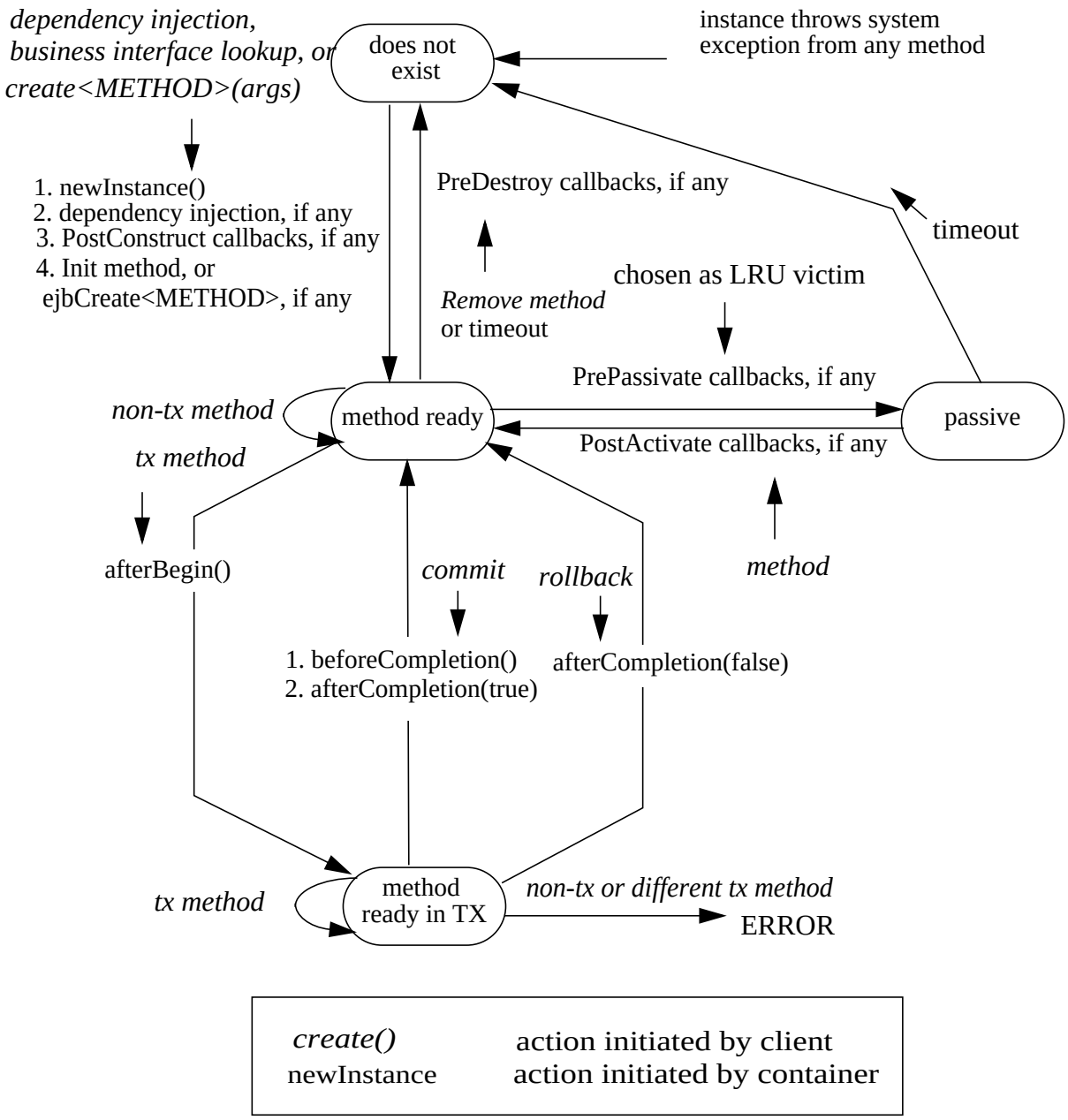
A session bean's `newInstance`, `setSessionContext`, other dependency injection methods, life cycle callback interceptor methods, and `afterCompletion` methods are called with an unspecified transaction context. Refer to section 12.6.5 for how the container executes methods with an unspecified transaction context.

For example, it would be wrong to perform database operations within a session bean's `PostConstruct` *or* `PreDestroy` *lifecycle callback interceptor methods and to assume that the operations are part of the client's transaction. The* `PostConstruct` *and* `PreDestroy` *methods are not controlled by a transaction attribute because handling rollbacks in these methods would greatly complicate the session instance's state diagram.*

4.4 Stateful Session Bean State Diagram

The following figure illustrates the life cycle of a stateful session bean instance.

Figure 5 Life Cycle of a Stateful Session Bean Instance



The following steps describe the life cycle of a stateful session bean instance:

- A session bean instance's life starts when a client obtains a reference to a stateful session bean instance through dependency injection or JNDI lookup, or when the client invokes a `cre-`

ate<METHOD> method on the session bean's home interface. This causes the container to invoke newInstance on the session bean class to create a new session bean instance. Next, the container injects the bean's SessionContext, if applicable, and performs any other dependency injection as specified by metadata annotations on the bean class or by the deployment descriptor. The container then calls the PostConstruct lifecycle callback interceptor method(s) for the bean, if any. Finally, if the session bean was written to the EJB 2.1 client view, the container invokes the matching ejbCreate<METHOD> or Init method on the instance. The container then returns the session object reference to the client. The instance is now in the method ready state.

NOTE: When a stateful session bean is looked up or otherwise obtained through the explicit JNDI lookup mechanisms, the container must provide a new stateful session bean instance, as required by the Java EE specification (Section "Java Naming and Directory Interface (JNDI) Naming Context" [12]).

- The session bean instance is now ready for client's business methods. Based on the transaction attributes in the session bean's metadata annotations and/or deployment descriptor and the transaction context associated with the client's invocation, a business method is executed either in a transaction context or with an unspecified transaction context (shown as "tx method" and "non-tx method" in the diagram). See Chapter 12 for how the container deals with transactions.

- A non-transactional method is executed while the instance is in the method ready state.

- An invocation of a transactional method causes the instance to be included in a transaction. When the session bean instance is included in a transaction, the container issues the afterBegin method on it. The afterBegin method is invoked on the instance before any business method or business method interceptor method is executed as part of the transaction. The instance becomes associated with the transaction and will remain associated with the transaction until the transaction completes.

- Session bean methods invoked by the client in this transaction can now be delegated to the bean instance. An error occurs if a client attempts to invoke a method on the session object and the bean's metadata annotations and/or deployment descriptor for the method requires that the container invoke the method in a different transaction context than the one with which the instance is currently associated or in an unspecified transaction context.

- If a transaction commit has been requested, the transaction service notifies the container of the commit request before actually committing the transaction, and the container issues a beforeCompletion on the instance. When beforeCompletion is invoked, the instance should write any cached updates to the database. If a transaction rollback had been requested instead, the rollback status is reached without the container issuing a beforeCompletion. The container may not call the beforeCompletion method if the transaction has been marked for rollback (nor does the instance write any cached updates to the database).

- The transaction service then attempts to commit the transaction, resulting in either a commit or rollback.

- When the transaction completes, the container issues afterCompletion on the instance, specifying the status of the completion (either commit or rollback). If a rollback occurred, the

bean instance may need to reset its conversational state back to the value it had at the beginning of the transaction.

- The container's caching algorithm may decide that the bean instance should be evicted from memory. (This could be done at the end of each method, or by using an LRU policy). The container invokes the `PrePassivate` lifecycle callback interceptor method(s) for the bean instance, if any. After this completes, the container saves the instance's state to secondary storage. A session bean can be passivated only between transactions, and not within a transaction.

- While the instance is in the passivated state, the container may remove the session object after the expiration of a timeout specified by the Deployer. All object references and handles for the session object become invalid. If a client attempts to invoke a method on the bean's business interface, the container will throw the `javax.ejb.NoSuchEJBException`[13]. If the EJB 2.1 client view is used, the container will throw the `java.rmi.NoSuchObjectException` if the client is a remote client, or the `javax.ejb.NoSuchObjectLocalException` if the client is a local client.

- If a client invokes a session object whose session bean instance has been passivated, the container will activate the instance. To activate the session bean instance, the container restores the instance's state from secondary storage and invokes the `PostActivate` method for the instance, if any.

- The session bean instance is again ready for client methods.

- When the client calls a business method of the bean that has been designated as a `Remove` method, or a `remove` method on the home or component interface, the container invokes `PreDestroy` lifecycle callback interceptor method(s) (if any) for the bean instance after the `Remove` method completes.[14] This ends the life of the session bean instance and the associated session object. If a client subsequently attempts to invoke a method on the bean's business interface, the container will throw the `javax.ejb.NoSuchEJBException`[15]. If the EJB 2.1 client view is used, any subsequent attempt causes the `java.rmi.NoSuchObjectException` to be thrown if the client is a remote client, or the `javax.ejb.NoSuchObjectLocalException` if the client is a local client. (The `java.rmi.NoSuchObjectException` is a subclass of the `java.rmi.RemoteException`; the `javax.ejb.NoSuchObjectLocalException` is a subclass of the `javax.ejb.EJBException`). Note that a container can also invoke the `PreDestroy` method on the instance without a client call to remove the session object after the lifetime of the EJB object has expired.

The container must call the `afterBegin`, `beforeCompletion`, and `afterCompletion` methods if the session bean class implements, directly or indirectly, the `SessionSynchronization` interface. The container does not call these methods if the session bean class does not implement the `SessionSynchronization` interface.

---

[13] If the business interface is a remote business interface that extends `java.rmi.Remote`, the `java.rmi.NoSuchObjectException` is thrown to the client instead.

[14] If the `Remove` annotation specifies the value of `retainIfException` as `true`, and the `Remove` method throws an exception, the instance is not removed (and the `PreDestroy` lifecycle callback interceptor methods are not invoked).

[15] If the business interface is a remote business interface that extends `java.rmi.Remote`, the `java.rmi.NoSuchObjectException` is thrown to the client instead.

4.4.1 Operations Allowed in the Methods of a Stateful Session Bean Class

Table 1 defines the methods of a stateful session bean class from which the session bean instances can access the methods of the `javax.ejb.SessionContext` interface, the `java:comp/env` environment naming context, resource managers, `Timer` methods, the EntityManager and other enterprise beans.

If a session bean instance attempts to invoke a method of the `SessionContext` interface, and that access is not allowed in Table 1, the container must throw the `java.lang.IllegalStateException.`

If a session bean instance attempts to access a resource manager, an enterprise bean, or the EntityManager, and that access is not allowed in Table 1, the behavior is undefined by the EJB architecture.

If a session bean instance attempts to invoke a method of the `Timer` interface and the access is not allowed in Table 1, the container must throw the `java.lang.IllegalStateException`.

Table 1 Operations Allowed in the Methods of a Stateful Session Bean

Bean method	**Bean method can perform the following operations**	
	Container-managed transaction demarcation	**Bean-managed transaction demarcation**
constructor	-	-
dependency injection methods (e.g., setSessionContext)	SessionContext methods: *getEJBHome, getEJBLocalHome, lookup* JNDI access to java:comp/env	SessionContext methods: *getEJBHome, getEJBLocalHome, lookup* JNDI access to java:comp/env
PostConstruct, PreDestroy, PrePassivate, PostActivate lifecycle callback interceptor methods	SessionContext methods: *getBusinessObject, getEJBHome, getEJBLocalHome, getCallerPrincipal, isCallerInRole, getEJBObject, getEJBLocalObject, lookup* JNDI access to java:comp/env Resource manager access Enterprise bean access EntityManagerFactory access	SessionContext methods: *getBusinessObject, getEJBHome, getEJBLocalHome, getCallerPrincipal, isCallerInRole, getEJBObject, getEJBLocalObject, getUserTransaction, lookup* UserTransaction methods JNDI access to java:comp/env Resource manager access Enterprise bean access EntityManagerFactory access
business method from business interface or from component interface; business method interceptor method	SessionContext methods: *getBusinessObject, getEJBHome, getEJBLocalHome, getCallerPrincipal, getRollbackOnly, isCallerInRole, setRollbackOnly, getEJBObject, getEJBLocalObject, getInvokedBusinessInterface, lookup* JNDI access to java:comp/env Resource manager access Enterprise bean access EntityManagerFactory access EntityManager access Timer methods	SessionContext methods: *getBusinessObject, getEJBHome, getEJBLocalHome, getCallerPrincipal, isCallerInRole, getEJBObject, getEJBLocalObject, getInvokedBusinessInterface, getUserTransaction, lookup* UserTransaction methods JNDI access to java:comp/env Resource manager access Enterprise bean access EntityManagerFactory access EntityManager access Timer methods

Table 1 Operations Allowed in the Methods of a Stateful Session Bean

Bean method	Bean method can perform the following operations	
	Container-managed transaction demarcation	**Bean-managed transaction demarcation**
afterBegin beforeCompletion	SessionContext methods: *getBusinessObject, getEJBHome, getEJBLocalHome, getCallerPrincipal, getRollbackOnly, isCallerInRole, setRollbackOnly, getEJBObject, getEJBLocalObject, lookup* JNDI access to java:comp/env Resource manager access Enterprise bean access EntityManagerFactory access EntityManager access Timer methods	N/A (a bean with bean-managed transaction demarcation cannot implement the SessionSynchronization interface)
afterCompletion	SessionContext methods: *getBusinessObject, getEJBHome, getEJBLocalHome, getCallerPrincipal, isCallerInRole, getEJBObject, getEJBLocalObject, lookup* JNDI access to java:comp/env	

Notes:

- The `PostConstruct`, `PreDestroy`, `PrePassivate`, `PostActivate`, `Init`, and/or `ejbCreate<METHOD>`, `ejbRemove`, `ejbPassivate`, and `ejbActivate` methods of a session bean with container-managed transaction demarcation execute with an unspecified transaction context. Refer to Subsection 12.6.5 for how the container executes methods with an unspecified transaction context.

Additional restrictions:

- The `getRollbackOnly` and `setRollbackOnly` methods of the `SessionContext` interface should be used only in the session bean methods that execute in the context of a transaction. The container must throw the `java.lang.IllegalStateException` if the methods are invoked while the instance is not associated with a transaction.

The reasons for disallowing the operations in Table 1 follow:

- Invoking the `getBusinessObject` method is disallowed if the session bean does not define an EJB 3.0 business interface.

- Invoking the `getInvokedBusinessInterface` method is disallowed if the session bean does not define an EJB 3.0 business interface.

- Invoking the `getEJBObject` and `getEJBHome` methods is disallowed if the session bean does not define a remote client view.
- Invoking the `getEJBLocalObject` and `getEJBLocalHome` methods is disallowed if the session bean does not define a local client view.
- Invoking the `getRollbackOnly` and `setRollbackOnly` methods is disallowed in the session bean methods for which the container does not have a meaningful transaction context, and to all session beans with bean-managed transaction demarcation.
- Accessing resource managers, enterprise beans, and the EntityManager is disallowed in the session bean methods for which the container does not have a meaningful transaction context and/or client security context.
- The `UserTransaction` interface is unavailable to enterprise beans with container-managed transaction demarcation.
- The `TimerService` interface is unavailable to stateful session beans.
- Invoking the `getMessageContext` method is disallowed for stateful session beans.
- Invoking the `getEJBObject` and `getEJBLocalObject` methods is disallowed in the session bean methods in which there is no session object identity established for the instance.

4.4.2 Dealing with Exceptions

A `RuntimeException` thrown from any method of the session bean class (including the business methods and the lifecycle callback interceptor methods invoked by the container) results in the transition to the "does not exist" state. Exception handling is described in detail in Chapter 13. See section 11.4.2 for the rules pertaining to lifecycle callback interceptor methods when more than one such method applies to the bean class.

From the client perspective, the corresponding session object does not exist any more. If a client subsequently attempts to invoke a method on the bean's business interface, the container will throw the `javax.ejb.NoSuchEJBException`[16]. If the EJB 2.1 client view is used, the container will throw the `java.rmi.NoSuchObjectException` if the client is a remote client, or the `javax.ejb.NoSuchObjectLocalException` if the client is a local client.

4.4.3 Missed `PreDestroy` Calls

The Bean Provider cannot assume that the container will always invoke the `PreDestroy` lifecycle callback interceptor method(s) (or `ejbRemove` method) for a session bean instance. The following scenarios result in the `PreDestroy` lifecycle callback interceptor method(s) not being called for an instance:

[16] If the business interface is a remote business interface that extends `java.rmi.Remote`, the `java.rmi.NoSuchObjectException` is thrown to the client instead.

- A crash of the EJB container.
- A system exception thrown from the instance's method to the container.
- A timeout of client inactivity while the instance is in the `passive` state. The timeout is specified by the Deployer in an EJB container implementation-specific way.

If resources are allocated in a `PostConstruct` lifecycle callback interceptor method (or `ejbCreate<METHOD>` method) and/or in the business methods, and normally released in a `PreDestroy` lifecycle callback interceptor method, these resources will not be automatically released in the above scenarios. The application using the session bean should provide some clean up mechanism to periodically clean up the unreleased resources.

For example, if a shopping cart component is implemented as a session bean, and the session bean stores the shopping cart content in a database, the application should provide a program that runs periodically and removes "abandoned" shopping carts from the database.

4.4.4 Restrictions for Transactions

The state diagram implies the following restrictions on transaction scoping of the client invoked business methods. The restrictions are enforced by the container and must be observed by the client programmer.

- A session bean instance can participate in at most a single transaction at a time.
- If a session bean instance is participating in a transaction, it is an error for a client to invoke a method on the session object such that the transaction attribute specified in the bean's metadata annotations and/or the deployment descriptor would cause the container to execute the method in a different transaction context or in an unspecified transaction context. In such a case, the `javax.ejb.EJBException` will be thrown to a client of the bean's business interface[17]. If the EJB 2.1 client view is used, the container throws the `java.rmi.RemoteException` to the client if the client is a remote client, or the `javax.ejb.EJBException` if the client is a local client.
- If a session bean instance is participating in a transaction, it is an error for a client to invoke the `remove` method on the session object's home or component interface object. The container must detect such an attempt and throw the `javax.ejb.RemoveException` to the client. The container should not mark the client's transaction for rollback, thus allowing the client to recover.

[17] If the business interface is a remote business interface that extends `java.rmi.Remote`, the `java.rmi.RemoteException` is thrown to the client instead.

4.5 Stateless Session Beans

Stateless session beans are session beans whose instances have no conversational state. This means that all bean instances are equivalent when they are not involved in servicing a client-invoked method.

The term "stateless" signifies that an instance has no state for a specific client. However, the instance variables of the instance can contain the state across client-invoked method calls. Examples of such state include an open database connection and an object reference to an enterprise bean object.

> *The Bean Provider must exercise caution if retaining any application state across method calls. In particular, references to common bean state should not be returned through multiple local interface method calls.*

Because all instances of a stateless session bean are equivalent, the container can choose to delegate a client-invoked method to any available instance. This means, for example, that the container may delegate the requests from the same client within the same transaction to different instances, and that the container may interleave requests from multiple transactions to the same instance.

A container only needs to retain the number of instances required to service the current client load. Due to client "think time," this number is typically much smaller than the number of active clients. Passivation is not needed or used for stateless sessions. The container creates another stateless session bean instance if one is needed to handle an increase in client work load. If a stateless session bean is not needed to handle the current client work load, the container can destroy it.

Because stateless session beans minimize the resources needed to support a large population of clients, depending on the implementation of the container, applications that use stateless session beans may scale somewhat better than those using stateful session beans. However, this benefit may be offset by the increased complexity of the client application that uses the stateless beans.

Compatibility Note: Local and remote clients using the EJB 2.1 client view interfaces use the `create` *and* `remove` *methods on the home interface of a stateless session bean in the same way as on a stateful session bean. To the EJB 2.1 client, it appears as if the client controls the life cycle of the session object. However, the container handles the* `create` *and* `remove` *calls without necessarily creating and removing an EJB instance. The home interface of a stateless session bean must have one* `create` *method that takes no arguments. The* `create` *method of the remote home interface must return the session bean's remote interface. The* `create` *method of the local home interface must return the session bean's local interface. There can be no other* `create` *methods in the home interface.*

There is no fixed mapping between clients and stateless instances. The container simply delegates a client's work to any available instance that is method-ready.

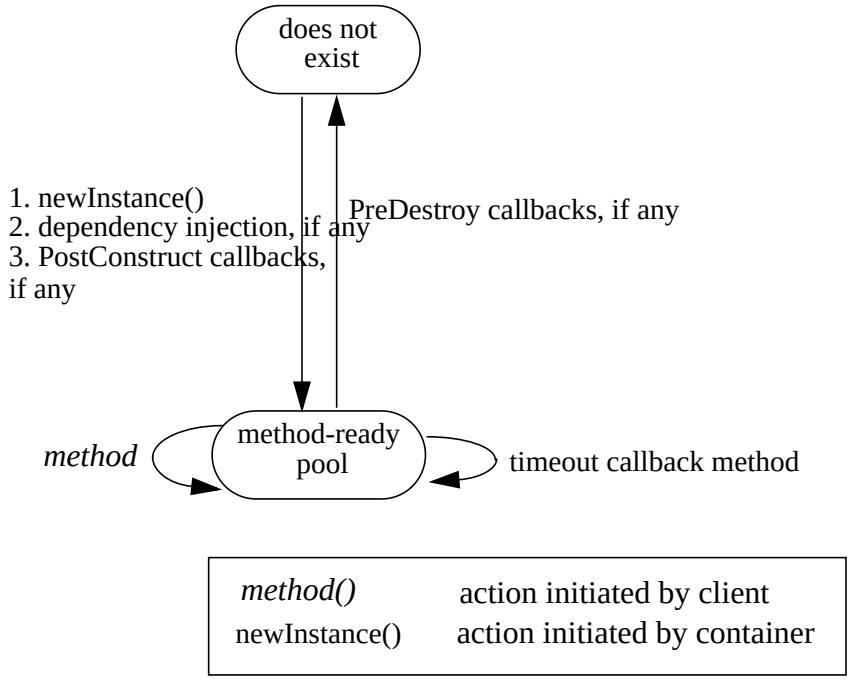
A stateless session bean must not implement the `javax.ejb.SessionSynchronization` interface.

4.5.1 Stateless Session Bean State Diagram

When a client calls a method on a stateless session object or invokes a method on a stateless session bean through its web service client view, the container selects one of its method-ready instances and delegates the method invocation to it.

The following figure illustrates the life cycle of a *stateless* session bean instance.

Figure 6 Life Cycle of a Stateless Session Bean



The following steps describe the life cycle of a session bean instance:

- A stateless session bean instance's life starts when the container invokes the `newInstance` method on the session bean class to create a new session bean instance. Next, the container injections the bean's `SessionContext`, if applicable, and performs any other dependency injection as specified by metadata annotations on the bean class or by the deployment descriptor. The container then calls the `PostConstruct` lifecycle callback interceptor methods for the bean, if any. The container can perform the instance creation at any time—there is no direct relationship to a client's invocation of a business method or the `create` method.

- The session bean instance is now ready to be delegated a business method call from any client or a call from the container to the timeout callback method.

- When the container no longer needs the instance (usually when the container wants to reduce the number of instances in the method-ready pool), the container invokes the `PreDestroy` lifecycle callback interceptor methods for it, if any. This ends the life of the stateless session bean instance.

4.5.2 Operations Allowed in the Methods of a Stateless Session Bean Class

Table 2 defines the methods of a stateless session bean class in which the session bean instances can access the methods of the `javax.ejb.SessionContext` interface, the `java:comp/env` environment naming context, resource managers, `TimerService` and `Timer` methods, the EntityManager, and other enterprise beans.

If a session bean instance attempts to invoke a method of the `SessionContext` interface, and the access is not allowed in Table 2, the container must throw the `java.lang.IllegalStateException.`

If a session bean instance attempts to invoke a method of the `TimerService` or `Timer` interface and the access is not allowed in Table 2, the container must throw the `java.lang.IllegalStateException`.

If a session bean instance attempts to access a resource manager, an enterprise bean, or the EntityManager, and the access is not allowed in Table 2, the behavior is undefined by the EJB architecture.

.

Table 2 Operations Allowed in the Methods of a Stateless Session Bean

Bean method	**Bean method can perform the following operations**	
	Container-managed transaction demarcation	**Bean-managed transaction demarcation**
constructor	-	-
dependency injection-methods (e.g., setSessionContext)	SessionContext methods: *getEJBHome, getEJBLocalHome, lookup* JNDI access to java:comp/env	SessionContext methods: *getEJBHome, getEJBLocalHome, lookup* JNDI access to java:comp/env
PostConstruct, PreDestroy lifecycle callback interceptor methods	SessionContext methods: *getBusinessObject, getEJBHome, getEJBLocalHome, getEJBObject, getEJBLocalObject, getTimerService, lookup* JNDI access to java:comp/env	SessionContext methods: *getBusinessObject, getEJBHome, getEJBLocalHome, getEJBObject, getEJBLocalObject, getUserTransaction, getTimerService, lookup* JNDI access to java:comp/env
business method from business interface or component interface; business method interceptor method	SessionContext methods: *getBusinessObject, getEJBHome, getEJBLocalHome, getCallerPrincipal, isCallerInRole, getRollbackOnly, setRollbackOnly, getEJBObject, getEJBLocalObject, getTimerService, getInvokedBusinessInterface, lookup* JNDI access to java:comp/env Resource manager access Enterprise bean access EntityManagerFactory access EntityManager access TimerService and Timer methods	SessionContext methods: *getBusinessObject, getEJBHome, getEJBLocalHome, getCallerPrincipal, isCallerInRole, getEJBObject, getEJBLocalObject, getUserTransaction, getTimerService, getInvokedBusinessInterface, lookup* UserTransaction methods JNDI access to java:comp/env Resource manager access Enterprise bean access EntityManagerFactory access EntityManager access TimerService and Timer methods
business method from web service endpoint	SessionContext methods: *getBusinessObject, getEJBHome, getEJBLocalHome, getCallerPrincipal, isCallerInRole, getRollbackOnly, setRollbackOnly, getEJBObject, getEJBLocalObject, getTimerService, getMessageContext, lookup* Message context methods JNDI access to java:comp/env Resource manager access Enterprise bean access EntityManagerFactory access EntityManager access TimerService and Timer methods	SessionContext methods: *getBusinessObject, getEJBHome, getEJBLocalHome, getCallerPrincipal, isCallerInRole, getEJBObject, getEJBLocalObject, getUserTransaction, getTimerService, getMessageContext, lookup* UserTransaction methods Message context methods JNDI access to java:comp/env Resource manager access Enterprise bean access EntityManagerFactory access EntityManager access TimerService and Timer methods

Table 2 Operations Allowed in the Methods of a Stateless Session Bean

Bean method	**Bean method can perform the following operations**	
	Container-managed transaction demarcation	**Bean-managed transaction demarcation**
timeout callback method	SessionContext methods: *getBusinessObject, getEJBHome, getEJBLocalHome, getCallerPrincipal, isCallerInRole, getRollbackOnly, setRollbackOnly, getEJBObject, getEJBLocalObject, getTimerService, lookup* JNDI access to java:comp/env Resource manager access Enterprise bean access EntityManagerFactory access EntityManager access TimerService and Timer methods	SessionContext methods: *getBusinessObject, getEJBHome, getEJBLocalHome, getCallerPrincipal, isCallerInRole, getEJBObject, getEJBLocalObject, getUserTransaction, getTimerService, lookup* UserTransaction methods JNDI access to java:comp/env Resource manager access Enterprise bean access EntityManagerFactory access EntityManager access TimerService and Timer methods

Additional restrictions:

- The `getRollbackOnly` and `setRollbackOnly` methods of the `SessionContext` interface should be used only in the session bean methods that execute in the context of a transaction. The container must throw the `java.lang.IllegalStateException` if the methods are invoked while the instance is not associated with a transaction.

The reasons for disallowing operations in Table 2:

- Invoking the `getBusinessObject` method is disallowed if the session bean does not define an EJB 3.0 business interface.
- Invoking the `getInvokedBusinessInterface` method is disallowed if the session bean does not define an EJB 3.0 business interface.
- Invoking the `getEJBObject` and `getEJBHome` methods is disallowed if the session bean does not define a remote client view.
- Invoking the `getEJBLocalObject` and `getEJBLocalHome` methods is disallowed if the session bean does not define a local client view.
- Invoking the `getRollbackOnly` and `setRollbackOnly` methods is disallowed in the session bean methods for which the container does not have a meaningful transaction context, and for all session beans with bean-managed transaction demarcation.
- Invoking the `getMessageContext` method is disallowed in session bean methods that were not invoked by the container through the session bean's web service endpoint. The `getMessageContext` method returns the `javax.xml.rpc.handler.MessageContext` interface of a stateless session bean that implements a JAX-RPC web service endpoint.
- Accessing resource managers, enterprise beans, and the EntityManager is disallowed in the session bean methods for which the container does not have a meaningful transaction context and/or client security context.
- The `UserTransaction` interface is unavailable to session beans with container-managed transaction demarcation.

4.5.3 Dealing with Exceptions

A `RuntimeException` thrown from any method of the enterprise bean class (including the business methods and the lifecycle callback interceptor methods invoked by the container) results in the transition to the "does not exist" state. Exception handling is described in detail in Chapter 13. See section 11.4.2 for the rules pertaining to lifecycle callback interceptor methods when more than one such method applies to the bean class.

From the client perspective, the session object continues to exist. The client can continue accessing the session object because the container can delegate the client's requests to another instance.

4.6 The Responsibilities of the Bean Provider

This section describes the responsibilities of the session Bean Provider to ensure that a session bean can be deployed in any EJB container.

4.6.1 Classes and Interfaces

The session Bean Provider is responsible for providing the following class files[18]:

- Session bean class.
- Session bean's business interface(s), if the session bean provides an EJB 3.0 local or remote client view.
- Session bean's remote interface and remote home interface, if the session bean provides an EJB 2.1 remote client view.
- Session bean's local interface and local home interface, if the session bean provides an EJB 2.1 local client view.
- Session bean's web service endpoint interface, if any.
- Interceptor classes, if any.

The Bean Provider for a stateless session bean that provides a web service client view may also define JAX-WS or JAX-RPC message handlers for the bean. The requirements for such message handlers are defined in [31] and [32].

4.6.2 Session Bean Class

The following are the requirements for the session bean class:

- The class must be defined as `public`, must not be `final`, and must not be `abstract`. The class must be a top level class.
- The class must have a `public` constructor that takes no parameters. The container uses this constructor to create instances of the session bean class.
- The class must not define the `finalize()` method.
- The class must implement the bean's business interface(s) or the methods of the bean's business interface(s), if any.

[18] Note that the interfaces provided by the Bean Provider may have been generated by tools.

- The class must implement the business methods of the bean's EJB 2.1 client view interfaces, if any.

Optionally:

- The class may implement, directly or indirectly, the `javax.ejb.SessionBean` interface.
- If the class is a stateful session bean, it may implement the `javax.ejb.SessionSynchronization` interface.
- The class may implement the session bean's web service endpoint or component interface.
- If the class is a stateless session bean, it may implement the `javax.ejb.TimedObject` interface. See Chapter 17, "Timer Service".
- The class may implement the `ejbCreate` method(s).
- The session bean class may have superclasses and/or superinterfaces. If the session bean has superclasses, the business methods, lifecycle callback interceptor methods, the timeout callback method, the methods of the optional `SessionSynchronization` interface, the `Init` or `ejbCreate<METHOD>` methods, and the methods of the `SessionBean` interface, may be defined in the session bean class, or in any of its superclasses. A session bean class must not have a superclass that is itself a session bean class.
- The session bean class is allowed to implement other methods (for example helper methods invoked internally by the business methods) in addition to the methods required by the EJB specification.

4.6.3 Lifecycle Callback Interceptor Methods

`PostConstruct`, `PreDestroy`, `PrePassivate`, and `PostActivate` lifecycle callback interceptor methods may be defined for stateful session beans.

`PostConstruct` and `PreDestroy` lifecycle callback interceptor methods may be defined for stateless session beans. If `PrePassivate` or `PostActivate` lifecycle callbacks are defined for stateless session beans, they are ignored.[19]

[19] This might result from the use of default interceptor classes, for example.

Compatibility Note: If the `PostConstruct` *lifecycle callback interceptor method is the* `ejbCreate` *method, if the* `PreDestroy` *lifecycle callback interceptor method is the* `ejbRemove` *method, if the* `PostActivate` *lifecycle callback interceptor method is the* `ejbActivate` *method, or if the* `PrePassivate` *lifecycle callback interceptor method is the* `ejbPassivate` *method, these callback methods must be implemented on the bean class itself (or on its superclasses). Except for these cases, the method names can be arbitrary, but must not start with "ejb" to avoid conflicts with the callback methods defined by the javax.ejb.EnterpriseBean interfaces.*

Lifecycle callback interceptor methods may be defined on the bean class and/or on an interceptor class of the bean. Rules applying to the definition of lifecycle callback interceptor methods are defined in Section 11.4, "Interceptors for LifeCycle Event Callbacks" .

4.6.4 ejbCreate<METHOD> Methods

The session bean class of a session bean that has a home interface may define one or more `ejbCreate<METHOD>` methods. These `ejbCreate` methods are intended for use only with the EJB 2.1 components. The signatures of the `ejbCreate` methods must follow these rules:

- The method name must have `ejbCreate` as its prefix.
- The method must be declared as `public`.
- The method must not be declared as `final` or `static`.
- The return type must be `void`.
- The method arguments must be legal types for RMI/IIOP if there is a `create<METHOD>` corresponding to the `ejbCreate<METHOD>` method on the session bean's remote home interface.
- A stateless session bean may define only a single `ejbCreate` method, with no arguments.
- The `throws` clause may define arbitrary application exceptions, possibly including the `javax.ejb.CreateException`.

Compatibility Note: EJB 1.0 allowed the `ejbCreate` *method to throw the* `java.rmi.RemoteException` *to indicate a non-application exception. This practice was deprecated in EJB 1.1—an EJB 1.1 or EJB 2.0 or later compliant enterprise bean should throw the* `javax.ejb.EJBException` *or another* `RuntimeException` *to indicate non-application exceptions to the container (see Section 13.2.2). An EJB 2.0 and later compliant enterprise bean should not throw the* `java.rmi.RemoteException` *from the* `ejbCreate` *method .*

4.6.5 Business Methods

The session bean class may define zero or more business methods whose signatures must follow these rules:

- The method names can be arbitrary, but they must not start with “ejb” to avoid conflicts with the callback methods used by the EJB architecture.
- The business method must be declared as `public`.
- The method must not be declared as `final` or `static`.
- The argument and return value types for a method must be legal types for RMI/IIOP if the method corresponds to a business method on the session bean’s remote business interface or remote interface.
- The argument and return value types for a method must be legal types for JAX-WS / JAX-RPC if the method is a web service method or corresponds to a method on the session bean’s web service endpoint.
- The `throws` clause may define arbitrary application exceptions.

Compatibility Note: EJB 1.0 allowed the business methods to throw the `java.rmi.RemoteException` to indicate a non-application exception. This practice was deprecated in EJB 1.1—an EJB 1.1 or EJB 2.0 or later compliant enterprise bean should throw the `javax.ejb.EJBException` or another `RuntimeException` to indicate non-application exceptions to the container (see Section 13.2.2). An EJB 2.0 or later compliant enterprise bean should not throw the `java.rmi.RemoteException` from a business method.

4.6.6 Session Bean’s Business Interface

The following are the requirements for the session bean’s business interface:

- The interface must not extend the `javax.ejb.EJBObject` or `javax.ejb.EJBLocalObject` interface.
- If the business interface is a remote business interface, the argument and return values must be of valid types for RMI/IIOP. The remote business interface is not required or expected to be a `java.rmi.Remote` interface. The `throws` clause should not include the `java.rmi.RemoteException`. The methods of the business interface may only throw the `java.rmi.RemoteException` if the interface extends `java.rmi.Remote`.
- The interface is allowed to have superinterfaces.
- If the interface is a remote business interface, its methods must not expose local interface types, timers or timer handles, or the managed collection classes that are used for EJB 2.1 entity beans with container-managed persistence as arguments or results.
- The bean class must implement the interface or the interface must be designated as a local or remote business interface of the bean by means of the `Local` or `Remote` annotation or in the deployment descriptor. The following rules apply:
    - If bean class implements a single interface, that interface is assumed to be the business interface of the bean. This business interface will be a local interface unless the

interface is designated as a remote business interface by use of the `Remote` annotation on the bean class or interface or by means of the deployment descriptor.

- A bean class is permitted to have more than one interface. If a bean class has more than one interface—excluding the interfaces listed below—any business interface of the bean class must be explicitly designated as a business interface of the bean by means of the `Local` or `Remote` annotation on the bean class or interface or in the deployment descriptor.
- The following interfaces are excluded when determining whether the bean class has more than one interface: `java.io.Serializable`; `java.io.Externalizable`; any of the interfaces defined by the `javax.ejb` package.
- While it is expected that the bean class will typically implement its business interface(s), if the bean class uses annotations or the deployment descriptor to designate its business interface(s), it is not required that the bean class also be specified as implementing the interface(s).

4.6.7 Session Bean's Remote Interface

The following are the requirements for the session bean's remote interface:

- The interface must extend the `javax.ejb.EJBObject` interface.
- The methods defined in this interface must follow the rules for RMI/IIOP. This means that their argument and return values must be of valid types for RMI/IIOP, and their `throws` clauses must include the `java.rmi.RemoteException`.
- The remote interface is allowed to have superinterfaces. Use of interface inheritance is subject to the RMI/IIOP rules for the definition of remote interfaces.
- For each method defined in the remote interface, there must be a matching method in the session bean's class. The matching method must have:
  - The same name.
  - The same number and types of arguments, and the same return type.
  - All the exceptions defined in the `throws` clause of the matching method of the session bean class must be defined in the `throws` clause of the method of the remote interface.
- The remote interface methods must not expose local interface types, local home interface types, timers or timer handles, or the managed collection classes that are used for entity beans with container-managed persistence as arguments or results.

4.6.8 Session Bean's Remote Home Interface

The following are the requirements for the session bean's remote home interface:

- The interface must extend the `javax.ejb.EJBHome` interface.

- The methods defined in this interface must follow the rules for RMI/IIOP. This means that their argument and return values must be of valid types for RMI/IIOP, and that their `throws` clauses must include the `java.rmi.RemoteException`.
- The remote home interface is allowed to have superinterfaces. Use of interface inheritance is subject to the RMI/IIOP rules for the definition of remote interfaces.
- A session bean's remote home interface must define one or more `create<METHOD>` methods. A stateless session bean must define exactly one `create` method with no arguments.
- Each `create` method of a stateful session bean must be named `create<METHOD>`, and it must match one of the `Init` methods or `ejbCreate<METHOD>` methods defined in the session bean class. The matching `Init` method or `ejbCreate<METHOD>` method must have the same number and types of arguments. (Note that the return type is different.) The `create` method for a stateless session bean must be named "`create`" but need not have a matching "`ejbCreate`" method.
- The return type for a `create<METHOD>` method must be the session bean's remote interface type.
- All the exceptions defined in the `throws` clause of an `ejbCreate<METHOD>` method of the session bean class must be defined in the `throws` clause of the matching `create<METHOD>` method of the remote home interface.
- The `throws` clause must include `javax.ejb.CreateException`.

4.6.9 Session Bean's Local Interface

The following are the requirements for the session bean's local interface:

- The interface must extend the `javax.ejb.EJBLocalObject` interface.
- The `throws` clause of a method defined in the local interface must not include the `java.rmi.RemoteException`.
- The local interface is allowed to have superinterfaces.
- For each method defined in the local interface, there must be a matching method in the session bean's class. The matching method must have:
  - The same name.
  - The same number and types of arguments, and the same return type.
  - All the exceptions defined in the `throws` clause of the matching method of the session bean class must be defined in the `throws` clause of the method of the local interface.

4.6.10 Session Bean's Local Home Interface

The following are the requirements for the session bean's local home interface:

- The interface must extend the `javax.ejb.EJBLocalHome` interface.
- The `throws` clause of a method in the local home interface must not include the `java.rmi.RemoteException`.
- The local home interface is allowed to have superinterfaces.
- A session bean's local home interface must define one or more `create<METHOD>` methods. A stateless session bean must define exactly one `create` method with no arguments.
- Each `create` method of a stateful session bean must be named `create<METHOD>`, and it must match one of the `Init` methods or `ejbCreate<METHOD>` methods defined in the session bean class. The matching `Init` method or `ejbCreate<METHOD>` method must have the same number and types of arguments. (Note that the return type is different.) The `create` method for a stateless session bean must be named "`create`" but need not have a matching "`ejbCreate`" method.
- The return type for a `create<METHOD>` method must be the session bean's local interface type.
- All the exceptions defined in the `throws` clause of an `ejbCreate<METHOD>` method of the session bean class must be defined in the `throws` clause of the matching `create<METHOD>` method of the local home interface.
- The `throws` clause must include `javax.ejb.CreateException`.

4.6.11 Session Bean's Web Service Endpoint Interface

EJB 3.0 does not require the definition of a web service endpoint interface for session beans that implement a web service endpoint.

The following are requirements for session beans with JAX-RPC web service endpoint interfaces. The JAX-WS and Web Services for Java EE specifications do not require that a separate interface be defined for a web service endpoint. The requirements for web service endpoints under JAX-WS and Web Services for Java EE are given in [32] and [31].

The following are the requirements for a stateless session bean's web service endpoint interface. The web service endpoint interface must follow the rules for JAX-RPC service endpoint interfaces [25].

- The web service endpoint interface must extend the `java.rmi.Remote` interface.
- The methods defined in the interface must follow the rules for JAX-RPC service endpoint interfaces. This means that their argument and return values must be of valid types for

JAX-RPC, and their `throws` clauses must include the `java.rmi.RemoteException`. The `throws` clause may additionally include application exceptions.

Note that JAX-RPC Holder classes may be used as method parameters. The JAX-RPC specification requires support for Holder classes as part of the standard Java mapping of WSDL operations in order to handle out and inout parameters. Holder classes implement the `javax.xml.rpc.holders.Holder` interface. See the JAX-RPC specification [25] for further details.

- For each method defined in the web service endpoint interface, there must be a matching method in the session bean's class. The matching method must have:
  - The same name.
  - The same number and types of arguments, and the same return type.
  - All the exceptions defined in the `throws` clause of the matching method of the session bean class must be defined in the `throws` clause of the method of the web service endpoint interface.

- The web service endpoint interface must not include an EJBObject or EJBLocalObject as either a parameter or return type. An array or JAX-RPC value type must not include an EJBObject or EJBLocalObject as a contained element. The web service endpoint interface methods must not expose business interface types, local or remote interface types, local or remote home interface types, timers or timer handles, or the managed collection classes that are used for entity beans with container-managed persistence as arguments or results or as fields of value types.

- JAX-RPC serialization rules apply for any value types that are used by the web service endpoint interface. If it is important that Java serialization semantics apply, the Bean Provider should use the restricted set of JAX-RPC value types for which the semantics of Java serialization apply under JAX-RPC serialization. See the JAX-RPC specification [25] for details.

- The web service endpoint interface must not include constant (as `public final static`) declarations.

- The Bean Provider must designate the web service endpoint interface in the deployment descriptor by means of the `service-endpoint` element. The service endpoint itself is only exposed within a web service if it is referenced by a web service deployment descriptor as defined by [31].

4.7 The Responsibilities of the Container Provider

This section describes the responsibilities of the Container Provider to support a session bean. The Container Provider is responsible for providing the deployment tools and for managing the session bean instances at runtime.

Because the EJB specification does not define the API between deployment tools and the container, we assume that the deployment tools are provided by the Container Provider. Alternatively, the deployment tools may be provided by a different vendor who uses the container vendor's specific API.

4.7.1 Generation of Implementation Classes

The deployment tools provided by the container are responsible for the generation of additional classes when the session bean is deployed. The tools obtain the information that they need for generation of the additional classes by introspecting the classes and interfaces provided by the Bean Provider and by examining the session bean's deployment descriptor.

The deployment tools must generate the following classes:

- A class that implements the session bean's business interface.
- A class that implements the session bean's remote home interface (session EJBHome class).
- A class that implements the session bean's remote interface (session EJBObject class).
- A class that implements the session bean's local home interface (session EJBLocalHome class).
- A class that implements the session bean's local interface (session EJBLocalObject class).
- A class that implements the session bean's web service endpoint.

The deployment tools may also generate a class that mixes some container-specific code with the session bean class. This code may, for example, help the container to manage the bean instances at runtime. The tools can use subclassing, delegation, and code generation.

The deployment tools may also allow the generation of additional code that wraps the business methods and is used to customize the business logic to an existing operational environment. For example, a wrapper for a `debit` function on the `AccountManager` bean may check that the debited amount does not exceed a certain limit.

4.7.2 Generation of WSDL

Reference [31] describes the generation of a WSDL document for a web service endpoint. The Java to WSDL mapping must adhere to the requirements of JAX-RPC or JAX-WS [32].

4.7.3 Session Business Interface Implementation Class

The container's implementation of the session business interface, which is generated by the deployment tools, implements the business methods specific to the session bean.

The implementation of each business method must activate the instance (if the instance is in the passive state), invoke any business method interceptor methods, and invoke the matching business method on the instance.

The container provider is responsible for providing the implementation of the `equals` and `hashCode` methods for the business interface, in conformance with the requirements of section 3.6.5.

4.7.4 Session EJBHome Class

The session EJBHome class, which is generated by the deployment tools, implements the session bean's remote home interface. This class implements the methods of the `javax.ejb.EJBHome` interface and the `create<METHOD>` methods specific to the session bean.

The implementation of each `create<METHOD>` method invokes a matching `ejbCreate<METHOD>` method.

4.7.5 Session EJBObject Class

The session EJBObject class, which is generated by the deployment tools, implements the session bean's remote interface. It implements the methods of the `javax.ejb.EJBObject` interface and the business methods specific to the session bean.

The implementation of each business method must activate the instance (if the instance is in the passive state), invoke any business method interceptor methods, and invoke the matching business method on the instance.

4.7.6 Session EJBLocalHome Class

The session EJBLocalHome class, which is generated by the deployment tools, implements the session bean's local home interface. This class implements the methods of the `javax.ejb.EJBLocalHome` interface and the `create<METHOD>` methods specific to the session bean.

The implementation of each `create<METHOD>` method invokes a matching `ejbCreate<METHOD>` method.

4.7.7 Session EJBLocalObject Class

The session EJBLocalObject class, which is generated by the deployment tools, implements the session bean's local interface. It implements the methods of the `javax.ejb.EJBLocalObject` interface and the business methods specific to the session bean.

The implementation of each business method must activate the instance (if the instance is in the passive state), invoke any business method interceptor methods, and invoke the matching business method on the instance.

4.7.8 Web Service Endpoint Implementation Class

The implementation class for a stateless session bean's web service endpoint is generated by the container's deployment tools. This class must handle requests to the web service endpoint, unmarshall the SOAP request, and invoke the stateless session bean method that matches the web service endpoint method that corresponds to the request.

4.7.9 Handle Classes

The deployment tools are responsible for implementing the handle classes for the session bean's remote home and remote interfaces.

4.7.10 EJBMetaData Class

The deployment tools are responsible for implementing the class that provides metadata to the remote client view contract. The class must be a valid RMI Value class and must implement the `javax.ejb.EJBMetaData` interface.

4.7.11 Non-reentrant Instances

The container must ensure that only one thread can be executing an instance at any time. If a client request arrives for an instance while the instance is executing another request, the container may throw the `javax.ejb.ConcurrentAccessException` to the second client[20]. If the EJB 2.1 client view is used, the container may throw the `java.rmi.RemoteException` to the second request if the client is a remote client, or the `javax.ejb.EJBException` if the client is a local client.[21]

> *Note that a session object is intended to support only a single client. Therefore, it would be an application error if two clients attempted to invoke the same session object.*

One implication of this rule is that an application cannot make loopback calls to a session bean instance.

4.7.12 Transaction Scoping, Security, Exceptions

The container must follow the rules with respect to transaction scoping, security checking, and exception handling, as described in Chapters 12, 16, and 13, respectively.

4.7.13 JAX-WS and JAX-RPC Message Handlers for Web Service Endpoints

The container must support the use of JAX-WS and JAX-RPC message handlers for web service endpoints. Container requirements for support of message handlers are specified in [32] and [31].

If message handlers are present, they must be invoked before any business method interceptor methods.

---

[20] If the business interface is a remote business interface that extends `java.rmi.Remote`, the `java.rmi.RemoteException` is thrown to the client instead.

[21] In certain special circumstances (e.g., to handle clustered web container architectures), the container may instead queue or serialize such concurrent requests. Clients, however, cannot rely on this behavior.

4.7.14 SessionContext

The container must implement the `SessionContext.getEJBObject` method such that the bean instance can use the Java language cast to convert the returned value to the session bean's remote interface type. Specifically, the bean instance does not have to use the `PortableRemoteObject.narrow` method for the type conversion.

The container must implement the `EJBContext.lookup` method such that when the `lookup` method is used to look up a bean's remote home interface, a bean instance can use the Java language cast to convert the returned value to a session bean's remote home interface type. Specifically, the bean instance does not have to use the `PortableRemoteObject.narrow` method for the type conversion.

Chapter 5

Message-Driven Bean Component Contract

This chapter specifies the contract between a message-driven bean and its container. It defines the life cycle of the message-driven bean instances.

This chapter defines the developer's view of message-driven bean state management and the container's responsibility for managing message-driven bean state.

5.1 Overview

A message-driven bean is an asynchronous message consumer. A message-driven bean is invoked by the container as a result of the arrival of a message at the destination or endpoint that is serviced by the message-driven bean. A message-driven bean instance is an instance of a message-driven bean class. A message-driven bean is defined for a single messaging type, in accordance with the message listener interface it employs.

To a client, a message-driven bean is a message consumer that implements some business logic running on the server. A client accesses a message-driven bean by sending messages to the destination or endpoint for which the message-driven bean class is the message listener.

Message-driven beans are anonymous. They have no client-visible identity.

Message-driven bean instances have no conversational state. This means that all bean instances are equivalent when they are not involved in servicing a client message.

A message-driven bean instance is created by the container to handle the processing of the messages for which the message-driven bean is the consumer. Its lifetime is controlled by the container.

A message-driven bean instance has no state for a specific client. However, the instance variables of the message-driven bean instance can contain state across the handling of client messages. Examples of such state include an open database connection and a reference to an enterprise bean.

5.2 Goals

The goal of the message-driven bean model is to make developing an enterprise bean that is asynchronously invoked to handle the processing of incoming messages as simple as developing the same functionality in any other message listener.

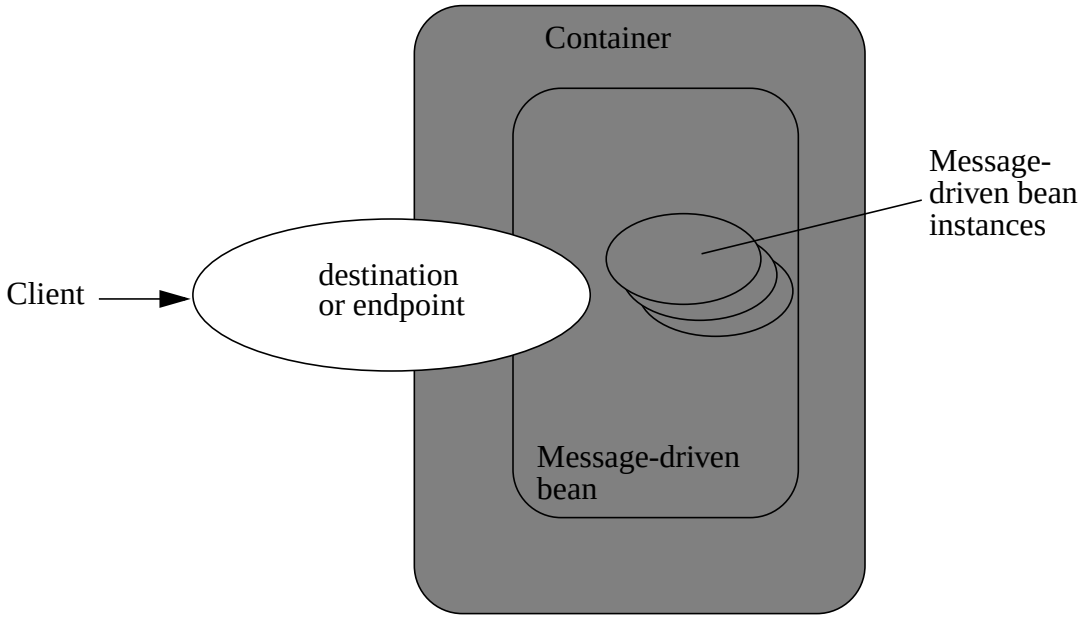
A further goal of the message-driven bean model is to allow for the concurrent processing of a stream of messages by means of container-provided pooling of message-driven bean instances.

5.3 Client View of a Message-Driven Bean

To a client, a message-driven bean is simply a message consumer. The client sends messages to the destination or endpoint for which the message-driven bean is the message listener just as it would to any other destination or endpoint. The message-driven bean, as a message consumer, handles the processing of the messages.

From the perspective of the client, the existence of a message-driven bean is completely hidden behind the destination or endpoint for which the message-driven bean is the message listener. The following diagram illustrates the view that is provided to a message-driven bean’s clients.

Figure 7 Client view of Message-Driven Beans Deployed in a Container



A client's JNDI name space may be configured to include the destinations or endpoints of message-driven beans installed in multiple EJB containers located on multiple machines on a network. The actual locations of an enterprise bean and EJB container are, in general, transparent to the client using the enterprise bean.

References to message destinations can be injected, or they can be looked up in the client's JNDI namespace.

For example, the reference to the queue for a JMS message-driven bean might be injected as follows.

```
@Resource Queue stockInfoQueue;
```

Alternatively, the queue for the `StockInfo` JMS message-driven bean might be located using the following code segment:

```
Context initialContext = new InitialContext();
Queue stockInfoQueue = (javax.jms.Queue)initialContext.lookup
    (“java:comp/env/jms/stockInfoQueue”);
```

The remainder of this section describes the message-driven bean life cycle in detail and the protocol between the message-driven bean and its container.

5.4 Protocol Between a Message-Driven Bean Instance and its Container

From its creation until destruction, a message-driven bean instance lives in a container. The container provides security, concurrency, transactions, and other services for the message-driven bean. The container manages the life cycle of the message-driven bean instances, notifying the instances when bean action may be necessary, and providing a full range of services to ensure that the message-driven bean implementation is scalable and can support the concurrent processing of a large number of messages.

From the Bean Provider’s point of view, a message-driven bean exists as long as its container does. It is the container’s responsibility to ensure that the message-driven bean comes into existence when the container is started up and that instances of the bean are ready to receive an asynchronous message delivery before the delivery of messages is started.

Containers themselves make no actual service demands on the message-driven bean instances. The calls a container makes on a bean instance provide it with access to container services and deliver notifications issued by the container.

Since all instances of a message-driven bean are equivalent, a client message can be delivered to any available instance.

5.4.1 Required MessageDrivenBean Metadata

A message-driven bean must be annotated with the `MessageDriven` annotation or denoted in the deployment descriptor as a message-driven bean.

5.4.2 The Required Message Listener Interface

The message-driven bean class must implement the appropriate message listener interface for the messaging type that the message-driven bean supports or specify the message listener interface using the `MessageDriven` metadata annotation or the `messaging-type` deployment descriptor element. The specific message listener interface that is implemented by a message-driven bean class distinguishes the messaging type that the message-driven bean supports.

The message-driven bean class's implementation of the `javax.jms.MessageListener` *interface distinguishes the message-driven bean as a JMS message-driven bean.*

The bean's message listener method (e.g., `onMessage` in the case of `javax.jms.MessageListener`) is called by the container when a message has arrived for the bean to service. The message listener method contains the business logic that handles the processing of the message.

A bean's message listener interface may define more than one message listener method. If the message listener interface contains more than one method, it is the resource adapter that determines which method is invoked. See [15].

If the message-driven bean class implements more than one interface other than `java.io.Serializable`, `java.io.Externalizable`, or any of the interfaces defined by the `javax.ejb` package, the message listener interface must be specified by the `messageListenerInterface` element of the `MessageDriven` annotation or the `messaging-type` element of the `message-driven` deployment descriptor element.

5.4.3 Dependency Injection

A message-driven bean may use dependency injection mechanisms to acquire references to resources or other objects in its environment (see Chapter 15, "Enterprise Bean Environment"). If a message-driven bean makes use of dependency injection, the container injects these references after the bean instance is created, and before any message-listener methods are invoked on the bean instance. If a dependency on the `MessageDrivenContext` is declared, or if the bean class implements the optional `MessageDrivenBean` interface (see Section 5.4.6), the `MessageDrivenContext` is also injected at this time. If dependency injection fails, the bean instance is discarded.

Under the EJB 3.0 API, the bean class may acquire the `MessageDrivenContext` *interface through dependency injection without having to implement the* `MessageDrivenBean` *interface. In this case, the* `Resource` *annotation (or* `resource-ref` *deployment descriptor element) is used to denote the bean's dependency on the* `MessageDrivenContext`. *See Chapter 15, "Enterprise Bean Environment".*

5.4.4 The MessageDrivenContext Interface

If the bean specifies a dependency on the `MessageDrivenContext` interface (or if the bean class implements the `MessageDrivenBean` interface), the container must provide the message-driven bean instance with a `MessageDrivenContext`. This gives the message-driven bean instance access to the instance's context maintained by the container. The `MessageDrivenContext` interface has the following methods:

- The `setRollbackOnly` method allows the instance to mark the current transaction such that the only outcome of the transaction is a rollback. Only instances of a message-driven bean with container-managed transaction demarcation can use this method.
- The `getRollbackOnly` method allows the instance to test if the current transaction has been marked for rollback. Only instances of a message-driven bean with container-managed transaction demarcation can use this method.

- The `getUserTransaction` method returns the `javax.transaction.UserTransaction` interface that the instance can use to demarcate transactions, and to obtain transaction status. Only instances of a message-driven bean with bean-managed transaction demarcation can use this method.
- The `getTimerService` method returns the `javax.ejb.TimerService` interface.
- The `getCallerPrincipal` method returns the `java.security.Principal` that is associated with the invocation.
- The `isCallerInRole` method is inherited from the `EJBContext` interface. Message-driven bean instances must not call this method.
- The `getEJBHome` and `getEJBLocalHome` methods are inherited from the `EJBContext` interface. Message-driven bean instances must not call these methods.
- The `lookup` method enables the message-driven bean to look up its environment entries in the JNDI naming context.

5.4.5 Message-Driven Bean Lifecycle Callback Interceptor Methods

The following lifecycle event callbacks are supported for message-driven beans. Callback methods may be defined directly on the bean class or on a separate interceptor class. See Section 5.6.4.

- `PostConstruct`
- `PreDestroy`

The `PostConstruct` callback occurs before the first message listener method invocation on the bean. This is at a point after which any dependency injection has been performed by the container.

The `PostConstruct` lifecycle callback interceptor method executes in an unspecified transaction and security context.

The `PreDestroy` callback occurs at the time the bean is removed from the pool or destroyed.

The `PreDestroy` lifecycle callback interceptor method executes in an unspecified transaction and security context.

5.4.6 The Optional MessageDrivenBean Interface

The message-driven bean class is not required to implement the `javax.ejb.MessageDrivenBean` interface.

Compatibility Note: The MessageDrivenBean interface was required by earlier versions of the Enterprise JavaBeans specification. In EJB 3.0, the functionality previously provided by the MessageDrivenBean interface is available to the bean class through selective use of dependency injection (of the MessageDrivenContext) and optional lifecycle callback methods.

The `MessageDrivenBean` interface defines two methods, `setMessageDrivenContext` and `ejbRemove`.

The `setMessageDrivenContext` method is called by the bean's container to associate a message-driven bean instance with its context maintained by the container. Typically a message-driven bean instance retains its message-driven context as part of its state.

The `ejbRemove` notification signals that the instance is in the process of being removed by the container. In the `ejbRemove` method, the instance releases the resources that it is holding.

Under the EJB 3.0 API, the bean class may optionally define a `PreDestroy` *callback method for notification of the container's removal of the bean instance.*

This specification requires that the `ejbRemove` and the `ejbCreate` methods of a message-driven bean be treated as the `PreDestroy` and `PostConstruct` lifecycle callback methods, respectively. If the message-driven bean implements the `MessageDrivenBean` interface, the `PreDestroy` annotation can only be applied to the `ejbRemove` method. Similar requirements apply to use of deployment descriptor metadata as an alternative to the use of annotations.

5.4.7 Timeout Callbacks

A message driven bean can be registered with the EJB timer service for time-based event notifications if it provides a timeout callback method. The container invokes the bean instance's timeout callback method when a timer for the bean has expired. See Chapter 17, "Timer Service".

5.4.8 Message-Driven Bean Creation

The container creates an instance of a message-driven bean in three steps. First, the container calls the bean class' `newInstance` method to create a new message-driven bean instance. Second, the container injects the bean's `MessageDrivenContext`, if applicable, and performs any other dependency injection as specified by metadata annotations on the bean class or by the deployment descriptor. Third, the container calls the instance's `PostConstruct` lifecycle callback methods, if any. See Section 5.6.4.

Compatibility Note: EJB 2.1 required the message-driven bean class to implement the `ejbCreate` *method. This requirement has been removed in EJB 3.0. If the message-driven bean class implements the* `ejbCreate` *method, the* `ejbCreate` *method is treated as the bean's* `PostConstruct` *method, and the* `PostConstruct` annotation can only be applied to the `ejbCreate` method.

5.4.9 Message Listener Interceptor Methods for Message-Driven Beans

The `AroundInvoke` business method interceptor methods are supported for message-driven beans. These interceptor methods may be defined on the bean class or on a interceptor class and apply to the handling of the invocation of the bean's message listener method(s).

Interceptors are described in Chapter 11, "Interceptors".

5.4.10 Serializing Message-Driven Bean Methods

The container serializes calls to each message-driven bean instance. Most containers will support many instances of a message-driven bean executing concurrently; however, each instance sees only a serialized sequence of method calls. Therefore, a message-driven bean does not have to be coded as reentrant.

The container must serialize all the container-invoked callbacks (e.g., lifecycle callback interceptor methods and timeout callback methods), and it must serialize these callbacks with the message listener method calls.

5.4.11 Concurrency of Message Processing

A container allows many instances of a message-driven bean class to be executing concurrently, thus allowing for the concurrent processing of a stream of messages. No guarantees are made as to the exact order in which messages are delivered to the instances of the message-driven bean class, although the container should attempt to deliver messages in order when it does not impair the concurrency of message processing. Message-driven beans should therefore be prepared to handle messages that are out of sequence: for example, the message to cancel a reservation may be delivered before the message to make the reservation.

5.4.12 Transaction Context of Message-Driven Bean Methods

A bean's message listener and timeout callback methods are invoked in the scope of a transaction determined by the transaction attribute specified in the bean's metadata annotations or deployment descriptor. If the bean is specified as using container-managed transaction demarcation, either the `REQUIRED` or the `NOT_SUPPORTED` transaction attribute must be used. See Chapter 12, "Support for Transactions"

When a message-driven bean using bean-managed transaction demarcation uses the `javax.transaction.UserTransaction` interface to demarcate transactions, the message receipt that causes the bean to be invoked is not part of the transaction. If the message receipt is to be part of the transaction, container-managed transaction demarcation with the `REQUIRED` transaction attribute must be used.

The `newInstance` method, `setMessageDrivenContext`, the message-driven bean's dependency injection methods, and lifecycle callback methods are called with an unspecified transaction context. Refer to Subsection 12.6.5 for how the container executes methods with an unspecified transaction context.

5.4.13 Activation Configuration Properties

The Bean Provider may provide information to the Deployer about the configuration of the message-driven bean in its operational environment. This may include information about message acknowledgement modes, message selectors, expected destination or endpoint types, etc.

Activation configuration properties for JMS message-driven beans are described in Sections 5.4.14 through 5.4.16.

5.4.14 Message Acknowledgment for JMS Message-Driven Beans

JMS message-driven beans should not attempt to use the JMS API for message acknowledgment. Message acknowledgment is automatically handled by the container. If the message-driven bean uses container-managed transaction demarcation, message acknowledgment is handled automatically as a part of the transaction commit. If bean-managed transaction demarcation is used, the message receipt cannot be part of the bean-managed transaction, and, in this case, the receipt is acknowledged by the container. If bean-managed transaction demarcation is used, the Bean Provider can indicate whether JMS `AUTO_ACKNOWLEDGE` semantics or `DUPS_OK_ACKNOWLEDGE` semantics should apply by using the `activationConfig` element of the `MessageDriven` annotation or by using the `activation-config-property` deployment descriptor element. The property name used to specify the acknowledgment mode is `acknowledgeMode`. If the `acknowledgeMode` property is not specified, JMS `AUTO_ACKNOWLEDGE` semantics are assumed. The value of the `acknowledgeMode` property must be either `Auto-acknowledge` or `Dups-ok-acknowledge` for a JMS message-driven bean.

5.4.15 Message Selectors for JMS Message-Driven Beans

The Bean Provider may declare the JMS message selector to be used in determining which messages a JMS message-driven bean is to receive. If the Bean Provider wishes to restrict the messages that a JMS message-driven bean receives, the Bean Provider can specify the value of the message selector by using the `activationConfig` element of the `MessageDriven` annotation or by using the `activation-config-property` deployment descriptor element. The property name used to specify the message selector is `messageSelector`.

For example:

```
@MessageDriven(activationConfig={
     @ActivationConfigProperty(
          propertyName=”messageSelector”,
          propertyValue=”JMSType = ‘car’ AND color = ‘blue’ and weight
> 2500”)})
```

```
<activation-config>
<activation-config-property>
<activation-config-property-name>messageSelector</activation-con-
fig-property-name>
<activation-config-property-value>JMSType = ‘car’ AND color = ‘blue’
AND weight &gt; 2500</activation-config-property-value>
</activation-config-property>
</activation-config>
```

The Application Assembler may further restrict, but not replace, the value of the `messageSelector` property of a JMS message-driven bean.

5.4.16 Association of a Message-Driven Bean with a Destination or Endpoint

A message-driven bean is associated with a destination or endpoint when the bean is deployed in the container. It is the responsibility of the Deployer to associate the message-driven bean with a destination or endpoint.

5.4.16.1 JMS Message-Driven Beans

A JMS message-driven bean is associated with a JMS Destination (Queue or Topic) when the bean is deployed in the container. It is the responsibility of the Deployer to associate the message-driven bean with a Queue or Topic.

The Bean Provider may provide advice to the Deployer as to whether a message-driven bean is intended to be associated with a queue or a topic by using the `activationConfig` element of the `MessageDriven` annotation or by using the `activation-config-property` deployment descriptor element. The property name used to specify the destination type associated with the bean is `destinationType`. The value for this property must be either `javax.jms.Queue` or `javax.jms.Topic` for a JMS message-driven bean.

If the message-driven bean is intended to be used with a topic, the Bean Provider may further indicate whether a durable or non-durable subscription should be used by using the `activationConfig` element of the `MessageDriven` annotation or by using the `activation-config-property` deployment descriptor element. The property name used to specify whether a durable or non-durable subscription should be used is `subscriptionDurability`. The value for this property must be either `Durable` or `NonDurable` for a JMS message-driven bean. If a topic subscription is specified and `subscriptionDurability` is not specified, a non-durable subscription is assumed.

- Durable topic subscriptions, as well as queues, ensure that messages are not missed even if the EJB server is not running. Reliable applications will typically make use of queues or durable topic subscriptions rather than non-durable topic subscriptions.

- If a non-durable topic subscription is used, it is the container's responsibility to make sure that the message driven bean subscription is active (i.e., that there is a message driven bean available to service the message) in order to ensure that messages are not missed as long as the EJB server is running. Messages may be missed, however, when a bean is not available to service them. This will occur, for example, if the EJB server goes down for any period of time.

The Deployer should avoid associating more than one message-driven bean with the same JMS Queue. If there are multiple JMS consumers for a queue, JMS does not define how messages are distribued between the queue receivers.

5.4.17 Dealing with Exceptions

A message-driven bean's message listener method must not throw the `java.rmi.RemoteException`.

Message-driven beans should not, in general, throw `RuntimeExceptions`.

A `RuntimeException` thrown from any method of the message-driven bean class (including a message listener method and the callbacks invoked by the container) results in the transition to the "does not exist" state. If a message-driven bean uses bean-managed transaction demarcation and throws a `RuntimeException`, the container should not acknowledge the message. Exception handling is described in detail in Chapter 13. See Section 11.4.2 for the rules pertaining to lifecycle callback interceptor methods when more than one such method applies to the bean class.

From the client perspective, the message consumer continues to exist. If the client continues sending messages to the destination or endpoint associated with the bean, the container can delegate the client's messages to another instance.

The message listener methods of some messaging types may throw application exceptions. An application exception is propagated by the container to the resource adapter.

5.4.18 Missed PreDestroy Callbacks

The Bean Provider cannot assume that the container will always invoke the `PreDestroy` callback method (or `ejbRemove` method) for a message-driven bean instance. The following scenarios result in the `PreDestroy` callback method not being called on an instance:

- A crash of the EJB container.
- A system exception thrown from the instance's method to the container.

If the message-driven bean instance allocates resources in the `PostConstruct` lifecycle callback method and/or in the message listener method, and releases normally the resources in the `PreDestroy` method, these resources will not be automatically released in the above scenarios. The application using the message-driven bean should provide some clean up mechanism to periodically clean up the unreleased resources.

5.4.19 Replying to a JMS Message

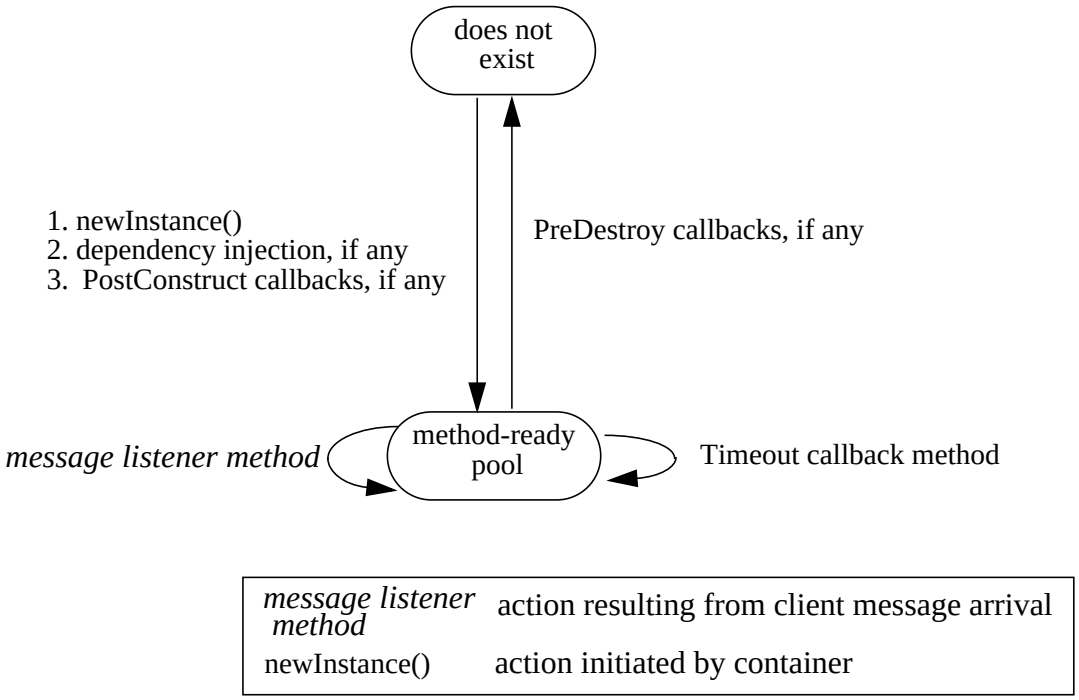
In standard JMS usage scenarios, the messaging mode of a message's `JMSReplyTo` destination (Queue or Topic) is the same as the mode of the destination to which the message has been sent. Although a message-driven bean is not directly dependent on the mode of the JMS destination from which it is consuming messages, it may contain code that depends on the mode of its message's `JMSReplyTo` destination. In particular, if a message-driven bean replies to a message, the mode of the reply's message producer and the mode of the `JMSReplyTo` destination must be the same. In order to implement a message-driven bean that is independent of `JMSReplyTo` mode, the Bean Provider should use `instanceOf` to test whether a `JMSReplyTo` destination is a Queue or Topic, and then use a matching message producer for the reply.

5.5 Message-Driven Bean State Diagram

When a client sends a message to a Destination for which a message-driven bean is the consumer, the container selects one of its method-ready instances and invokes the instance's message listener method.

The following figure illustrates the life cycle of a message-driven bean instance.

Figure 8 Life Cycle of a Message-Driven Bean.



The following steps describe the life cycle of a message-driven bean instance:

- A message-driven bean instance's life starts when the container invokes `newInstance` on the message-driven bean class to create a new instance. Next, the container injects the bean's MessageDrivenContext, if applicable, and performs any other dependency injection as specified by metadata annotations on the bean class or by the deployment descriptor. The container then calls the bean's `PostConstruct` lifecycle callback methods, if any.

- The message-driven bean instance is now ready to be delivered a message sent to its associated destination or endpoint by any client or a call from the container to the timeout callback method.

- When the container no longer needs the instance (which usually happens when the container wants to reduce the number of instances in the method-ready pool), the container invokes the `PreDestroy` lifecycle callback methods for it, if any. This ends the life of the message-driven bean instance.

5.5.1 Operations Allowed in the Methods of a Message-Driven Bean Class

Table 3 defines the methods of a message-driven bean class in which the message-driven bean instances can access the methods of the `javax.ejb.MessageDrivenContext` interface, the `java:comp/env` environment naming context, resource managers, `TimerService` and `Timer` methods, the EntityManager, and other enterprise beans.

If a message-driven bean instance attempts to invoke a method of the `MessageDrivenContext` interface, and the access is not allowed in Table 3, the container must throw and log the `java.lang.IllegalStateException.`

If a message-driven bean instance attempts to invoke a method of the `TimerService` or `Timer` interface and the access is not allowed in Table 3, the container must throw the `java.lang.IllegalStateException`.

If a bean instance attempts to access a resource manager, an enterprise bean, or the EntityManager, and the access is not allowed in Table 3, the behavior is undefined by the EJB architecture.

Table 3 Operations Allowed in the Methods of a Message-Driven Bean

Bean method	**Bean method can perform the following operations**	
	Container-managed transaction demarcation	**Bean-managed transaction demarcation**
constructor	-	-
dependency injection methods (e.g., setMessageDrivenContext)	MessageDrivenContext methods: *lookup* JNDI access to java:comp/env	MessageDrivenContext methods: *lookup* JNDI access to java:comp/env
PostConstruct, PreDestroy lifecycle callback methods	MessageDrivenContext methods: *getTimerService, lookup* JNDI access to java:comp/env	MessageDrivenContext methods: *getUserTransaction, getTimerService, lookup* JNDI access to java:comp/env
message listener method, business method interceptor method	MessageDrivenContext methods: *getRollbackOnly, setRollbackOnly, getCallerPrincipal, getTimerService, lookup* JNDI access to java:comp/env Resource manager access Enterprise bean access EntityManagerFactory access EntityManager access Timer service or Timer methods	MessageDrivenContext methods: *getUserTransaction, getCallerPrincipal, getTimerService, lookup* UserTransaction methods JNDI access to java:comp/env Resource manager access Enterprise bean access EntityManagerFactory access EntityManager access Timer service or Timer methods
timeout callback method	MessageDrivenContext methods: *getRollbackOnly, setRollbackOnly, getCallerPrincipal, getTimerService, lookup* JNDI access to java:comp/env Resource manager access Enterprise bean access EntityManagerFactory access EntityManager access Timer service or Timer methods	MessageDrivenContext methods: *getUserTransaction, getCallerPrincipal, getTimerService, lookup* UserTransaction methods JNDI access to java:comp/env Resource manager access Enterprise bean access EntityManagerFactory access EntityManager access Timer service or Timer methods

Additional restrictions:

- The `getRollbackOnly` and `setRollbackOnly` methods of the `MessageDrivenContext` interface should be used only in the message-driven bean methods that execute in the context of a transaction. The container must throw the `java.lang.IllegalStateException` if the methods are invoked while the instance is not associated with a transaction.

The reasons for disallowing operations in Table 3:

- Invoking the `getRollbackOnly` and `setRollbackOnly` methods is disallowed in the message-driven bean methods for which the container does not have a meaningful transaction context, and for all message-driven beans with bean-managed transaction demarcation.
- The `UserTransaction` interface is unavailable to message-driven beans with container-managed transaction demarcation.
- Invoking `getEJBHome` or `getEJBLocalHome` is disallowed in message-driven bean methods because there are no EJBHome or EJBLocalHome objects for message-driven beans. The container must throw and log the `java.lang.IllegalStateException` if these methods are invoked.

5.6 The Responsibilities of the Bean Provider

This section describes the responsibilities of the message-driven Bean Provider to ensure that a message-driven bean can be deployed in any EJB container.

5.6.1 Classes and Interfaces

The message-driven Bean Provider is responsible for providing the following class files:

- Message-driven bean class.
- Interceptor classes, if any.

5.6.2 Message-Driven Bean Class

The following are the requirements for the message-driven bean class:

- The class must implement, directly or indirectly, the message listener interface required by the messaging type that it supports or the methods of the message listener interface. In the case of JMS, this is the `javax.jms.MessageListener` interface.
- The class must be defined as `public`, must not be `final`, and must not be `abstract`. The class must be a top level class.
- The class must have a `public` constructor that takes no arguments. The container uses this constructor to create instances of the message-driven bean class.
- The class must not define the `finalize` method.

Optionally:

- The class may implement, directly or indirectly, the `javax.ejb.MessageDrivenBean` interface.
- The class may implement, directly or indirectly, the `javax.ejb.TimedObject` interface.
- The class may implement the `ejbCreate` method.

The message-driven bean class may have superclasses and/or superinterfaces. If the message-driven bean has superclasses, the methods of the message listener interface, lifecycle callback interceptor methods, the timeout method, the `ejbCreate` method, and the methods of the `MessageDrivenBean` interface may be defined in the message-driven bean class or in any of its superclasses. A message-driven bean class must not have a superclass that is itself a message-driven bean class

The message-driven bean class is allowed to implement other methods (for example, helper methods invoked internally by the message listener method) in addition to the methods required by the EJB specification.

5.6.3 Message Listener Method

The message-driven bean class must define the message listener methods. The signature of a message listener method must follow these rules:

The method must be declared as `public`.

The method must not be declared as `final` or `static`.

5.6.4 Lifecycle Callback Interceptor Methods

`PostConstruct` and `PreDestroy` lifecycle callback interceptor methods may be defined for message-driven beans. If `PrePassivate` or `PostActivate` lifecycle callbacks are defined, they are ignored.[22]

Compatibility Note: If the `PostConstruct` *lifecycle callback interceptor method is the* `ejbCreate` *method, or if the* `PreDestroy` *lifecycle callback interceptor method is the* `ejbRemove` *method, these callback methods must be implemented on the bean class itself (or on its superclasses). Except for these cases, the method names can be arbitrary, but must not start with "ejb" to avoid conflicts with the callback methods defined by the javax.ejb.EnterpriseBean interfaces.*

Lifecycle callback interceptor methods may be defined on the bean class and/or on an interceptor class of the bean. Rules applying to the definition of lifecycle callback interceptor methods are defined in Section 11.4, "Interceptors for LifeCycle Event Callbacks" .

[22] This might result from the use of default interceptor classes, for example.

5.7 The Responsibilities of the Container Provider

This section describes the responsibilities of the Container Provider to support a message-driven bean. The Container Provider is responsible for providing the deployment tools, and for managing the message-driven bean instances at runtime.

Because the EJB specification does not define the API between deployment tools and the container, we assume that the deployment tools are provided by the Container Provider. Alternatively, the deployment tools may be provided by a different vendor who uses the container vendor's specific API.

5.7.1 Generation of Implementation Classes

The deployment tools provided by the container are responsible for the generation of additional classes when the message-driven bean is deployed. The tools obtain the information that they need for generation of the additional classes by introspecting the classes and interfaces provided by the enterprise Bean Provider and by examining the message-driven bean's deployment descriptor.

The deployment tools may generate a class that mixes some container-specific code with the message-driven bean class. This code may, for example, help the container to manage the bean instances at runtime. Subclassing, delegation, and code generation can be used by the tools.

5.7.2 Deployment of JMS Message-Driven Beans

The Container Provider must support the deployment of a JMS message-driven bean as the consumer of a JMS queue or a durable subscription.

5.7.3 Request/Response Messaging Types

If the message listener supports a request/response messaging type, it is the container's responsibility to deliver the message response.

5.7.4 Non-reentrant Instances

The container must ensure that only one thread can be executing an instance at any time.

5.7.5 Transaction Scoping, Security, Exceptions

The container must follow the rules with respect to transaction scoping, security checking, and exception handling, as described in Chapters 12, 16, and 13.

Chapter 6 Entity Beans and Persistence

The model for Entity Beans, persistence, and object/relational mapping has been considerably revised and enhanced in the Enterprise JavaBeans 3.0 release.

The contracts and requirements for entities defined by Enterprise JavaBeans 3.0 are specified in the document "*Java Persistence*" [2], which also contains the full specification of the Enterprise JavaBeans Query Language (EJB QL), and the metadata for object/relational mapping.

A compliant implementation of this specification is required to implement the contracts for persistence, query language, and object/relational mapping specified in the document "*Java Persistence*" [2].

Chapters 7, 8, and 9 of this specification document discuss the client view of entity beans under the earlier EJB 2.1 programming model, the contracts for EJB 2.1 Entity Beans with Container-Managed Persistence, and the contracts for EJB 2.1 Entity Beans with Bean-Managed Persistence respectively. Use of these earlier APIs is required to be supported by EJB 3.0 implementations.

Chapter 7 Client View of an EJB 2.1 Entity Bean

This chapter describes the client view of an EJB 2.1 entity bean. It is actually a contract fulfilled by the container in which the entity bean is deployed. Only the business methods are supplied by the enterprise bean itself.

Although the client view of the deployed entity beans is provided by classes implemented by the container, the container itself is transparent to the client.

The contents of this chapter apply only to entities as defined in the Enterprise JavaBeans 2.1 specification[3]. The client view of a persistent entity as defined by Enterprise JavaBeans 3.0 is described in the document "*Java Persistence*" of this specification [2].

7.1 Overview

For a client, an entity bean is a component that represents an object-oriented view of some entities stored in a persistent storage, such as a database, or entities that are implemented by an existing enterprise application.

The client of an entity bean may be a local client or the client may be a remote client.

This section provides an overview of the entity bean client view that is independent of whether the client is a remote client or a local client. The differences between remote clients and local clients are discussed in the following sections.

From its creation until its destruction, an entity object lives in a container. Transparently to the client, the container provides security, concurrency, transactions, persistence, and other services for the entity objects that live in the container. The container is transparent to the client—there is no API that a client can use to manipulate the container.

Multiple clients can access an entity object concurrently. The container in which the entity bean is deployed properly synchronizes access to the entity object's state using transactions.

Each entity object has an identity which, in general, survives a crash and restart of the container in which the entity object has been created. The object identity is implemented by the container with the cooperation of the enterprise bean class.

Multiple enterprise beans can be deployed in a container. For each entity bean deployed in a container, the container provides a class that implements a home interface for the entity bean. This interface allows the client to create, find, and remove entity objects within the enterprise bean's home as well as to execute home business methods, which are not specific to a particular entity bean object. A client can obtain the entity bean's home interface through dependency injection, or the client can look up the entity bean's home interface through JNDI. It is the responsibility of the container to make the entity bean's home interface available in the JNDI name space.

A client view of an entity bean is independent of the implementation of the entity bean and its container. This ensures that a client application is portable across all container implementations in which the entity bean might be deployed.

7.2 Remote Clients

A remote client accesses an entity bean through the entity bean's remote and remote home interfaces. The remote and remote home interfaces of the entity bean provide the remote client view.

The remote client view of an entity bean is location independent. A client running in the same JVM as an entity bean instance uses the same API to access the entity bean as a client running in a different JVM on the same or different machine.

The container provides classes that implement the entity bean's remote and remote home interfaces. The objects that implement the remote home and remote objects are remote Java objects, and are accessible from a client through the standard Java™ APIs for remote object invocation [6].

A remote client of an entity object can be another enterprise bean deployed in the same or different container or can be an arbitrary Java program, such as an application, applet, or servlet. The remote client view of an entity bean can also be mapped to non-Java client environments, such as CORBA clients not written in the Java programming language.

7.3 Local Clients

Entity beans may also have local clients. A local client is a client that is collocated with the entity bean and which may be tightly coupled to the bean.

Unlike the remote client view, the local client view of an entity bean is not location independent. The local client view requires the collocation in the same JVM of both the local client and the entity bean that provides the local client view. The local client view therefore does not provide the location transparency provided by the remote client view.

A local client accesses an entity bean through the entity bean's local home and local component interfaces. The container provides classes that implement the entity bean's local home and local component interfaces. The objects that implement the local home and local component interfaces are local Java objects.

The arguments of the methods of the local component interface and local home interface are passed by reference[23]. Such entity beans and their clients must be coded to assume that the state of any Java object that is passed as an argument or result is potentially shared by caller and callee.

A local client of an entity bean may be a session bean, a message-driven bean, another entity bean, or a web-tier component.

The choice between the use of a local or remote programming model is a design decision that the Bean Provider makes when developing the entity bean application. In general, however, entity beans are intended to be used with local clients. While it is possible to provide both a client view and a local client view for an entity bean with container-managed persistence, it is more likely that the entity bean will be designed with the local view in mind.

Entity beans that have container-managed relationships with other entity beans, as described in Chapter 8, "EJB 2.1 Entity Bean Component Contract for Container-Managed Persistence", must be accessed in the same local scope as those related beans, and therefore typically provide a local client view. In order to be the target of a container-managed relationship, an entity bean with container-managed persistence must provide a local component interface.

7.4 EJB Container

An EJB container (container for short) is a system that functions as a runtime container for enterprise beans.

[23] More literally, references are passed by value in the JVM: an argument variable of primitive type holds a value of that primitive type; an argument variable of a reference type hold a reference to the object. See [28].

Multiple enterprise beans can be deployed in a single container. For each entity bean deployed in a container, the container provides a home interface that allows the client to create, find, and remove entity objects that belong to the entity bean. The home interface may also provide home business methods, which are not specific to a particular entity bean object. The container makes the entity bean's home interface (defined by the Bean Provider and implemented by the Container Provider) available in the JNDI name space for clients.

An EJB server may host one or multiple EJB containers. The containers are transparent to the client: there is no client-level API to manipulate the container.

7.4.1 Locating an Entity Bean's Home Interface

A client obtains an entity bean's home interface through dependency injection, or the client locates an entity bean's home interface using JNDI. A client's JNDI name space may be configured to include the home interfaces of enterprise beans deployed in multiple EJB containers located on multiple machines on a network. The actual location of an EJB container is, in general, transparent to the client.

For example, the local home interface for the `Account` entity bean can be located using the following code segment:

```
Context initialContext = new InitialContext();
AccountHome accountHome = (AccountHome)
             initialContext.lookup("java:comp/env/ejb/accounts");
```

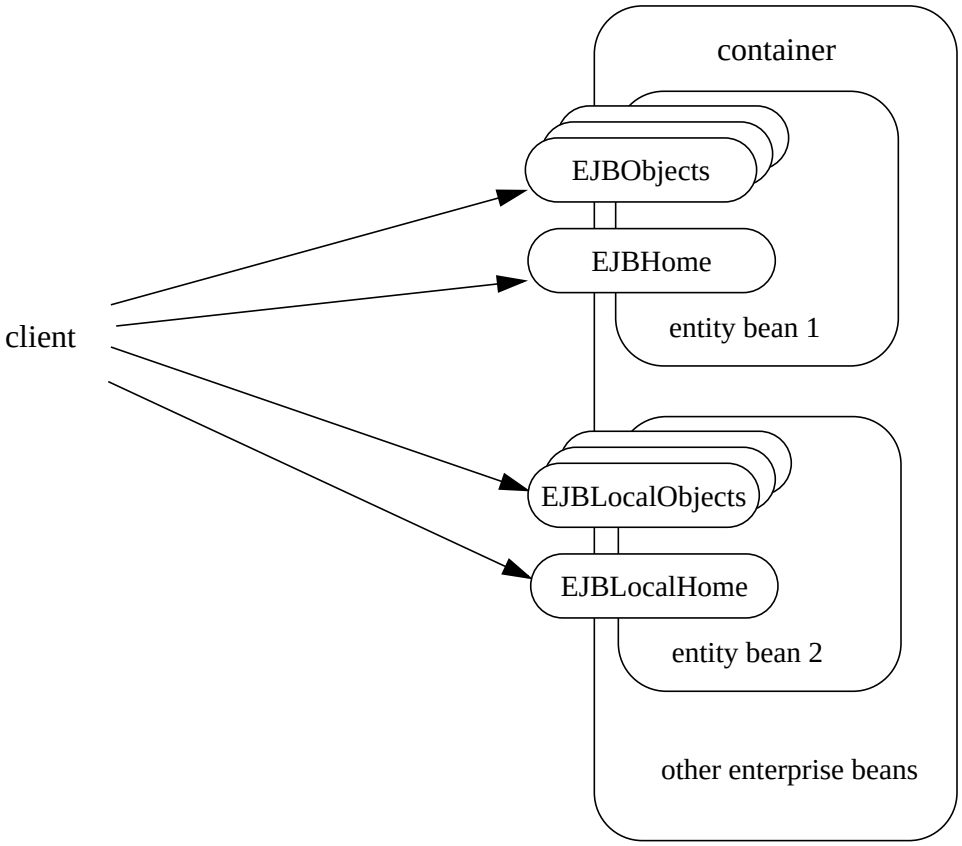
If dependency injection were used, the home interface could be obtained as follows:

```
@EJB AccountHome accountHome;
```

7.4.2 What a Container Provides

The following diagram illustrates the view that a container provides to the client of the entity beans deployed in the container. Note that a client may be a local client of some entity beans and a remote client of others.

Figure 9 Client View of Entity Beans Deployed in a Container



7.5 Entity Bean's Remote Home Interface

This section is specific to entity beans that provide a remote client view. Local home interfaces are described in Section 7.6.

The container provides the implementation of the remote home interface for each entity bean deployed in the container that defines a remote home interface. An object that implements an entity bean's remote home interface is called an **EJBHome** object.

The entity bean's remote home interface allows a client to do the following:

- Create new entity objects within the home.
- Find existing entity objects within the home.

- Remove an entity object from the home.
- Execute a home business method.
- Get the `javax.ejb.EJBMetaData` interface for the entity bean. The `javax.ejb.EJBMetaData` interface is intended to allow application assembly tools to discover the metadata information about the entity bean. The metadata information allows loose client/server binding and scripting.
- Obtain a handle for the home interface. The home handle can be serialized and written to stable storage. Later, possibly in a different JVM, the handle can be deserialized from stable storage and used to obtain a reference to the home interface.

An entity bean's remote home interface must extend the `javax.ejb.EJBHome` interface and follow the standard rules for Java programming language remote interfaces.

7.5.1 Create Methods

An entity bean's remote home interface can define zero or more `create<METHOD>` methods, one for each way to create an entity object. The arguments of the `create` methods are typically used to initialize the state of the created entity object. The name of each create method starts with the prefix "`create`".

The return type of a `create<METHOD>` method on the remote home interface is the entity bean's remote interface.

The `throws` clause of every `create<METHOD>` method on the remote home interface includes the `java.rmi.RemoteException` and the `javax.ejb.CreateException`. It may include additional application-level exceptions.

The following home interface illustrates three possible `create` methods:

```
public interface AccountHome extends javax.ejb.EJBHome {
     public Account create(String firstName, String lastName,
          double initialBalance)
           throws RemoteException, CreateException;
     public Account create(String accountNumber,
          double initialBalance)
           throws RemoteException, CreateException,
              LowInitialBalanceException;
     public Account createLargeAccount(String firstname,
          String lastname, double initialBalance)
           throws RemoteException, CreateException;
      ...
}
```

The following example illustrates how a client creates a new entity object:

```
AccountHome accountHome = ...;
Account account = accountHome.create("John", "Smith", 500.00);
```

7.5.2 Finder Methods

An entity bean's remote home interface defines one or more finder methods[24], one for each way to find an entity object or collection of entity objects within the home. The name of each finder method starts with the prefix "`find`", such as `findLargeAccounts`. The arguments of a finder method are used by the entity bean implementation to locate the requested entity objects. The return type of a finder method on the remote home interface must be the entity bean's remote interface, or a type representing a collection of objects that implement the entity bean's remote interface (see Subsections 8.5.7 and 9.1.9).

The `throws` clause of every finder method on the remote home interface includes the `java.rmi.RemoteException` and the `javax.ejb.FinderException` exceptions.

The remote home interface includes the `findByPrimaryKey(primaryKey)` method, which allows a client to locate an entity object using a primary key. The name of the method is always `findByPrimaryKey`; it has a single argument that is the same type as the entity bean's primary key type, and its return type is the entity bean's remote interface. There is a unique `findByPrimaryKey(primaryKey)` method for an entity bean on its remote home interface; this method must not be overloaded. The implementation of the `findByPrimaryKey(primaryKey)` method must ensure that the entity object exists.

The following example shows the `findByPrimaryKey` method:

```
public interface AccountHome extends javax.ejb.EJBHome {
      ...
     public Account findByPrimaryKey(String AccountNumber)
          throws RemoteException, FinderException;
}
```

The following example illustrates how a client uses the `findByPrimaryKey` method:

```
AccountHome = ...;
Account account = accountHome.findByPrimaryKey("100-3450-3333");
```

[24] The *`findByPrimaryKey`* method is mandatory for the remote home interface of all entity beans.

7.5.3 Remove Methods

The `javax.ejb.EJBHome` interface defines several methods that allow the client to remove an entity object.

```
public interface EJBHome extends Remote {
     void remove(Handle handle) throws RemoteException,
          RemoveException;
     void remove(Object primaryKey) throws RemoteException,
          RemoveException;
}
```

After an entity object has been removed, subsequent attempts to access the entity object by a remote client result in the `java.rmi.NoSuchObjectException`.

7.5.4 Home Methods

An entity bean's remote home interface may define one or more home methods. Home methods are methods that the Bean Provider supplies for business logic that is not specific to an entity bean instance.

Home methods on the remote home interface can have arbitrary method names, but they must not start with "`create`", "`find`", or "`remove`". The arguments of a home method are used by the entity bean implementation in computations that do not depend on a specific entity bean instance. The method arguments and return value types of a home method on the remote home interface must be legal types for RMI-IIOP.

The `throws` clause of every home method on the remote home interface includes the `java.rmi.RemoteException`. It may also include additional application-level exceptions.

The following example shows two home methods:

```
public interface EmployeeHome extends javax.ejb.EJBHome {
     ...
     // this method returns a living index depending on
     // the state and the base salary of an employee:
     // the method is not specific to an instance
     public float livingIndex(String state, float Salary)
          throws RemoteException;

     // this method adds a bonus to all of the employees
     // based on a company profit-sharing index
     public void addBonus(float company_share_index)
          throws RemoteException, ShareIndexOutOfRangeException;

     ...
}
```

7.6 Entity Bean's Local Home Interface

The container provides the implementation of the local home interface for each entity bean deployed in the container that defines a local home interface. An object that implements an entity bean's local home interface is called an **EJBLocalHome** object.

The entity bean's local home interface allows a local client to do the following:

- Create new entity objects within the home.
- Find existing entity objects within the home.
- Remove an entity object from the home.
- Execute a home business method.

An entity bean's local home interface must extend the `javax.ejb.EJBLocalHome` interface.

7.6.1 Create Methods

An entity bean's local home interface can define zero or more `create<METHOD>` methods, one for each way to create an entity object. The arguments of the `create` methods are typically used to initialize the state of the created entity object. The name of each create method starts with the prefix "`create`".

The return type of a `create<METHOD>` method on the local home interface is the entity bean's local interface.

The `throws` clause of every `create<METHOD>` method on the local home interface includes the `javax.ejb.CreateException`. It may include additional application-level exceptions. It must not include the `java.rmi.RemoteException`.

The following local home interface illustrates three possible `create` methods:

```
public interface AccountHome extends javax.ejb.EJBLocalHome {
     public Account create(String firstName, String lastName,
          double initialBalance)
           throws CreateException;
     public Account create(String accountNumber,
          double initialBalance)
           throws CreateException, LowInitialBalanceException;
     public Account createLargeAccount(String firstname,
          String lastname, double initialBalance)
           throws CreateException;
       ...
}
```

The following example illustrates how a client creates a new entity object:

```
AccountHome accountHome = ...;
Account account = accountHome.create("John", "Smith", 500.00);
```

7.6.2 Finder Methods

An entity bean's local home interface defines one or more finder methods[25], one for each way to find an entity object or collection of entity objects within the home. The name of each finder method starts with the prefix "`find`", such as `findLargeAccounts`. The arguments of a finder method are used by the entity bean implementation to locate the requested entity objects. The return type of a finder method on the local home interface must be the entity bean's local interface, or a type representing a collection of objects that implement the entity bean's local interface (see Subsections 8.5.7 and 9.1.9).

The `throws` clause of every finder method on the local home interface includes the `javax.ejb.FinderException`. The `throws` clause must not include the `java.rmi.RemoteException.`

The local home interface includes the `findByPrimaryKey(primaryKey)` method, which allows a client to locate an entity object using a primary key. The name of the method is always `findByPrimaryKey`; it has a single argument that is the same type as the entity bean's primary key type, and its return type is the entity bean's local interface. There is a unique `findByPrimaryKey(primaryKey)` method for an entity bean on its local home interface; this method must not be overloaded. The implementation of the `findByPrimaryKey` method must ensure that the entity object exists.

The following example shows the `findByPrimaryKey` method:

```
public interface AccountHome extends javax.ejb.EJBLocalHome {
    ...
    public Account findByPrimaryKey(String AccountNumber)
        throws FinderException;
}
```

The following example illustrates how a client uses the `findByPrimaryKey` method:

```
AccountHome = ...;
Account account = accountHome.findByPrimaryKey("100-3450-3333");
```

7.6.3 Remove Methods

The `javax.ejb.EJBLocalHome` interface defines the `remove` method to allow the client to remove an entity object.

```
public interface EJBLocalHome {
    void remove(Object primaryKey) throws RemoveException,
        EJBException;
}
```

[25] The *`findByPrimaryKey`* method is mandatory for the local home interface of all Entity Beans.

After an entity object has been removed, subsequent attempts to access the local entity object by the local client result in the `javax.ejb.NoSuchObjectLocalException`.

7.6.4 Home Methods

An entity bean's local home interface may define one or more home methods. Home methods are methods that the Bean Provider supplies for business logic that is not specific to an entity bean instance.

Home methods can have arbitrary method names, but they must not start with "`create`", "`find`", or "`remove`". The arguments of a home method are used by the entity bean implementation in computations that do not depend on a specific entity bean instance.

The `throws` clause of a home method on the local home interface may include additional application-level exceptions. It must not include the `java.rmi.RemoteException.`

The following example shows two home methods:

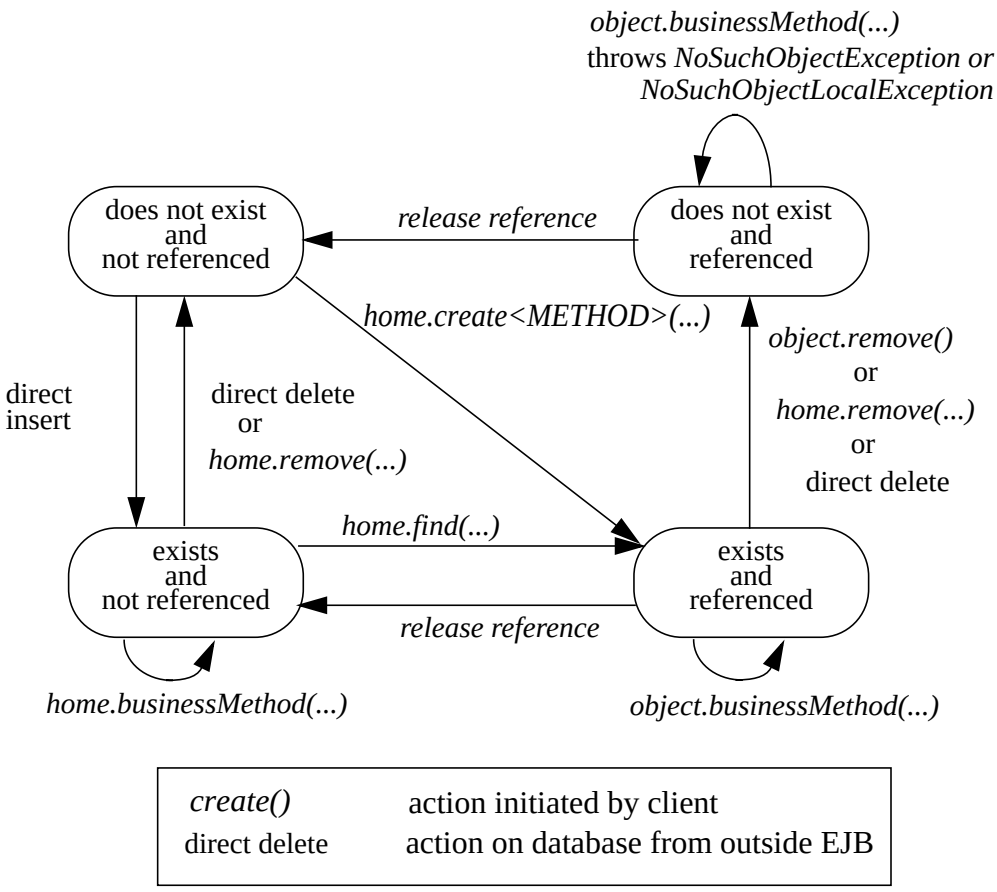
```
public interface EmployeeHome extends javax.ejb.EJBLocalHome {
     ...
     // this method returns a living index depending on
     // the state and the base salary of an employee:
     // the method is not specific to an instance
     public float livingIndex(String state, float Salary);

     // this method adds a bonus to all of the employees
     // based on a company profit sharing index
     public void addBonus(float company_share_index)
          throws ShareIndexOutOfRangeException;

     ...
}
```

7.7 Entity Object's Life Cycle

This section describes the life cycle of an entity object from the perspective of a client.

The following diagram illustrates a client's point of view of an entity object life cycle. (The term "referenced" in the diagram means that the client program has a reference to the entity object's remote or local interface.)

Figure 10 Client View of Entity Object Life Cycle



An entity object does not exist until it is created. Until it is created, it has no identity. After it is created, it has identity. A client creates an entity object using the entity bean's home interface, whose class is implemented by the container. When a client creates an entity object, the client obtains a reference to the newly created entity object.

In an environment with legacy data, entity objects may "exist" before the container and entity bean are deployed. In addition, an entity object may be "created" in the environment via a mechanism other than by invoking a `create<METHOD>` method of the home interface (e.g. by inserting a database record), but still may be accessible via the finder methods. Also, an entity object may be deleted directly using other means than the `remove` operation (e.g. by deletion of a database record). The "direct insert" and "direct delete" transitions in the diagram represent such direct database manipulation.

All entity objects are considered persistent objects. The lifetime of an entity object is not limited by the lifetime of the Java Virtual Machine process in which the entity bean instance executes. While a crash of the Java Virtual Machine may result in a rollback of current transactions, it does not destroy previously created entity objects nor invalidate the references to the home and component interfaces held by clients.

Multiple clients can access the same entity object concurrently. Transactions are used to isolate the clients' work from each other.

7.7.1 References to Entity Object Remote Interfaces

A client can get a reference to an existing entity object's remote interface in any of the following ways:

- Receive the reference as a parameter in a method call (input parameter or result).
- Find the entity object using a finder method defined in the entity bean's remote home interface.
- Obtain the reference from the entity object's handle. (See Section 7.11).

A client that has a reference to an entity object's remote interface can do any of the following:

- Invoke business methods on the entity object through the remote interface.
- Obtain a reference to the enterprise bean's remote home interface.
- Pass the reference as a parameter or return value of a method call.
- Obtain the entity object's primary key.
- Obtain the entity object's handle.
- Remove the entity object.

All references to an entity object that does not exist are invalid. All attempted invocations on an entity object that does not exist result in an `java.rmi.NoSuchObjectException` being thrown.

7.7.2 References to Entity Object Local Interfaces

A local client can get a reference to an existing entity object's local interface in any of the following ways:

- Receive the reference as a result of a method call.
- Find the entity object using a finder method defined in the entity bean's local home interface.

A local client that has a reference to an entity object's local interface can do any of the following:

- Invoke business methods on the entity object through the local interface.

- Obtain a reference to the enterprise bean's local home interface.
- Pass the reference as a parameter or return value of a local method call.
- Obtain the entity object's primary key.
- Remove the entity object.

All local references to an entity object that does not exist are invalid. All attempted invocations on an entity object that does not exist result in a `javax.ejb.NoSuchObjectLocalException` being thrown.

A local interface type must not be passed as an argument or result of a remote interface method.

7.8 Primary Key and Object Identity

Every entity object has a unique identity within its home. If two entity objects have the same home and the same primary key, they are considered identical.

The Enterprise JavaBeans architecture allows a primary key class to be any class that is a legal Value Type in RMI-IIOP, subject to the restrictions defined in Subsections 8.6.13 and 9.2.12. The primary key class may be specific to an entity bean class (i.e., each entity bean class may define a different class for its primary key, but it is possible that multiple entity beans use the same primary key class).

A client that holds a reference to an entity object's component interface can determine the entity object's identity within its home by invoking the `getPrimaryKey` method on the reference.

The object identity associated with a reference does not change over the lifetime of the reference. (That is, `getPrimaryKey` always returns the same value when called on the same entity object reference). If an entity object has both a remote home interface and a local home interface, the result of invoking the `getPrimaryKey` method on a reference to the entity object's remote interface and on a reference to the entity object's local interface is the same.

A client can test whether two entity object references refer to the same entity object by using the `isIdentical` method. Alternatively, if a client obtains two entity object references from the same home, it can determine if they refer to the same entity by comparing their primary keys using the `equals` method.

The following code illustrates using the `isIdentical` method to test if two object references refer to the same entity object:

```
Account acc1 = ...;
Account acc2 = ...;

if (acc1.isIdentical(acc2)) {
     // acc1 and acc2 are the same entity object
} else {
     // acc2 and acc2 are different entity objects
}
```

A client that knows the primary key of an entity object can obtain a reference to the entity object by invoking the `findByPrimaryKey(key)` method on the entity bean's home interface.

Note that the Enterprise JavaBeans architecture does not specify "object equality" (i.e. use of the `==` operator) for entity object references. The result of comparing two object references using the Java programming language `Object.equals(Object obj)` method is unspecified. Performing the `Object.hashCode()` method on two object references that represent the entity object is not guaranteed to yield the same result. Therefore, a client should always use the `isIdentical` method to determine if two entity object references refer to the same entity object.

> *Note that the use of* `isIdentical` *for the comparison of object references applies to the implementation of the methods of the* `java.util.Collection` *API as well.*

7.9 Entity Bean's Remote Interface

A client can access an entity object through the entity bean's remote interface. An entity bean's remote interface must extend the `javax.ejb.EJBObject` interface. A remote interface defines the business methods that are callable by remote clients.

The following example illustrates the definition of an entity bean's remote interface:

```
public interface Account extends javax.ejb.EJBObject {
     void debit(double amount)
          throws java.rmi.RemoteException,
               InsufficientBalanceException;
     void credit(double amount)
          throws java.rmi.RemoteException;
     double getBalance()
          throws java.rmi.RemoteException;
}
```

The `javax.ejb.EJBObject` interface defines the methods that allow the client to perform the following operations on an entity object's reference:

- Obtain the remote home interface for the entity object.
- Remove the entity object.

- Obtain the entity object's handle.
- Obtain the entity object's primary key.

The container provides the implementation of the methods defined in the `javax.ejb.EJBObject` interface. Only the business methods are delegated to the instances of the enterprise bean class.

Note that the entity object does not expose the methods of the `javax.ejb.EnterpriseBean` interface to the client. These methods are not intended for the client—they are used by the container to manage the enterprise bean instances.

7.10 Entity Bean's Local Interface

A local client can access an entity object through the entity bean's local interface. An entity bean's local interface must extend the `javax.ejb.EJBLocalObject` interface. A local interface defines the business methods that are callable by local clients.

The following example illustrates the definition of an entity bean's local interface:

```
public interface Account extends javax.ejb.EJBLocalObject {
     void debit(double amount)
          throws InsufficientBalanceException;
     void credit(double amount);
     double getBalance();
}
```

Note that the methods of the entity bean's local interface must not throw the `java.rmi.RemoteException`.

The `javax.ejb.EJBLocalObject` interface defines the methods that allow the local client to perform the following operations on an entity object's local reference:

- Obtain the local home interface for the entity object.
- Remove the entity object.
- Obtain the entity object's primary key.

The container provides the implementation of the methods defined in the `javax.ejb.EJBLocalObject` interface. Only the business methods are delegated to the instances of the enterprise bean class.

Note that the entity object does not expose the methods of the `javax.ejb.EntityBean` or the optional `javax.ejb.TimedObject` interface to the local client. These methods are not intended for the local client—they are used by the container to manage the enterprise bean instances.

7.11 Entity Bean's Handle

An entity object's handle is an object that identifies the entity object on a network. A client that has a reference to an entity object's remote interface can obtain the entity object's handle by invoking the `getHandle` method on the remote interface. The `getHandle` method is only available on the remote interface.

Since a handle class extends `java.io.Serializable`, a client may serialize the handle. The client may use the serialized handle later, possibly in a different process or even system, to re-obtain a reference to the entity object identified by the handle.

The client code must use the `javax.rmi.PortableRemoteObject.narrow` method to convert the result of the `getEJBObject` method invoked on a handle to the entity bean's remote interface type.

The lifetime and scope of a handle is specific to the handle implementation. At the minimum, a program running in one JVM must be able to obtain and serialize the handle, and another program running in a different JVM must be able to deserialize it and re-create an object reference. An entity handle is typically implemented to be usable over a long period of time—it must be usable at least across a server restart.

Containers that store long-lived entities will typically provide handle implementations that allow clients to store a handle for a long time (possibly many years). Such a handle will be usable even if parts of the technology used by the container (e.g. ORB, DBMS, server) have been upgraded or replaced while the client has stored the handle. Support for this "quality of service" is not required by the EJB specification.

An EJB container is not required to accept a handle that was generated by another vendor's EJB container.

The use of a handle is illustrated by the following example:

```
// A client obtains a handle of an account entity object and
// stores the handle in stable storage.
//
ObjectOutputStream stream = ...;
Account account = ...;
Handle handle = account.getHandle();
stream.writeObject(handle);

// A client can read the handle from stable storage, and use the
// handle to resurrect an object reference to the
// account entity object.
//
ObjectInputStream stream = ...;
Handle handle = (Handle) stream.readObject(handle);
Account account = (Account)javax.rmi.PortableRemoteObject.narrow(
         handle.getEJBObject(), Account.class);
account.debit(100.00);
```

A handle is not a capability, in the security sense, that would automatically grant its holder the right to invoke methods on the object. When a reference to an object is obtained from a handle, and then a method on the object is invoked, the container performs the usual access checks based on the caller's principal.

7.12 Entity Home Handles

The EJB specification allows a client to obtain a handle for the remote home interface. The client can use the home handle to store a reference to an entity bean's remote home interface in stable storage, and re-create the reference later. This handle functionality may be useful to a client that needs to use the remote home interface in the future, but does not know the JNDI name of the remote home interface.

A handle to a remote home interface must implement the `javax.ejb.HomeHandle` interface.

The client code must use the `javax.rmi.PortableRemoteObject.narrow` method to convert the result of the `getEJBHome` method invoked on a handle to the home interface type.

The lifetime and scope of a handle is specific to the handle implementation. At a minimum, a program running in one JVM must be able to serialize the handle, and another program running in a different JVM must be able to deserialize it and re-create an object reference. An entity handle is typically implemented to be usable over a long period of time—it must be usable at least across a server restart.

7.13 Type Narrowing of Object References

A client program that is intended to be interoperable with all compliant EJB container implementations must use the `javax.rmi.PortableRemoteObject.narrow` method to perform type-narrowing of the client-side representations of the remote home and remote interfaces.

Note: Programs that use the cast operator to narrow the remote and remote home interfaces are likely to fail if the container implementation uses RMI-IIOP as the underlying communication transport.

Chapter 8

EJB 2.1 Entity Bean Component Contract for Container-Managed Persistence

The EJB 2.1 entity bean component contract for container-managed persistence is the contract between an entity bean and its container. It defines the life cycle of the entity bean instances, the model for method delegation of the business methods invoked by the client, and the model for the management of the entity bean's persistent state and relationships. The main goal of this contract is to ensure that an entity bean component using container-managed persistence is portable across all compliant EJB containers.

This chapter defines the Enterprise Bean Provider's view of this contract and responsibilities of the Container Provider for managing the life cycle of the enterprise bean instances and their persistent state and relationships.

The contents of this chapter apply only to entity bean components with container-managed persistence as defined in the Enterprise JavaBeans 2.1 specification [3]. The contracts for persistent entities, as defined by Enterprise JavaBeans 3.0, are described in the document "*Java Persistence*" of this specification [2].

Note that use of dependency injection, interceptors, and any Java language metadata annotations is not supported for EJB 2.1 entity beans.

8.1 Overview

In accordance with the architecture for container-managed persistence, the Bean Provider develops a set of entity beans for an application, and determines the relationships among them. The Bean Provider designs an abstract persistence schema for each entity bean, which defines its container-managed fields and relationships, and determines the methods for accessing them. The entity bean instance accesses its container-managed fields and relationships at runtime by means of the methods defined for its abstract persistence schema.

The abstract persistence schema is specified in the deployment descriptor that is produced by the Bean Provider. The Deployer, using the Container Provider's tools, determines how the persistent fields and relationships defined by the abstract persistence schema are mapped to a database or other persistent store, and generates the necessary additional classes and interfaces that enable the container to manage the persistent fields and relationships of the entity bean instances at runtime.

This chapter describes the component contract for an EJB 2.1 entity bean with container-managed persistence, and how data independence is maintained between the entity bean instance and its representation in the persistent store. It describes this contract from the viewpoints of both the Bean Provider and the container.

8.2 Container-Managed Entity Persistence and Data Independence

The EJB component model provides a separation between the client view of a bean (as presented by its home and component interfaces) and the entity bean class (which provides the implementation of the client view). The EJB architecture for container-managed persistence adds to this a separation between the entity bean class (as defined by the Bean Provider) and its persistent representation. The container-managed persistence architecture thus provides not only a layer of data independence between the client view of a bean as an *entity object* and the Bean Provider's internal view of the bean in terms of the entity bean instance, but also between the entity bean instance and its persistent representation. This allows an entity bean to be evolved independently from its clients, without requiring the redefinition or recompilation of those clients, and it allows an entity bean to be redeployed across different containers and different persistent data stores, without requiring the redefinition or recompilation of the entity bean class.

In container-managed persistence, unlike in bean-managed persistence, the Bean Provider does not write database access calls in the methods of the entity bean class. Instead, persistence is handled by the container at runtime. The entity Bean Provider must specify in the deployment descriptor those persistent fields and relationships for which the container must handle data access. The Bean Provider codes all persistent data access by using the accessor methods that are defined for the abstract persistence schema. The implementation of the persistent fields and relationships, as well as all data access, is deferred to the container.

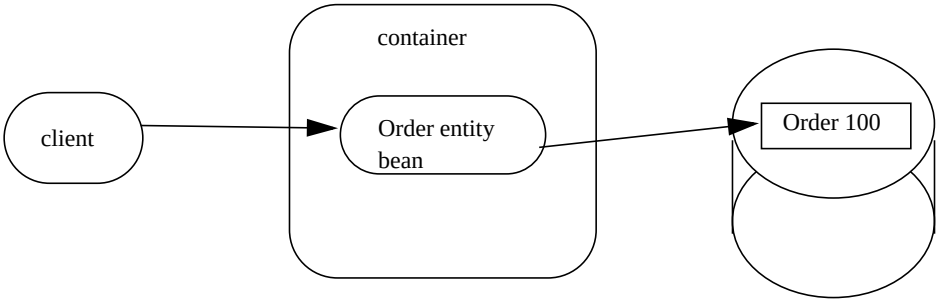
It is the responsibility of the Deployer to map the abstract persistence schema of a set of interrelated entity bean classes into the physical schema used by the underlying data store (e.g., into a relational schema) by using the Container Provider's tools. The Deployer uses the deployment descriptor as input to the Container Provider's tools to perform this mapping. The Container Provider's tools are also used to generate the concrete implementation of the entity bean classes, including the code that delegates calls to the accessor methods of the entity bean class to the runtime persistent data access layer of the container.

The EJB deployment descriptor for EJB 2.1 entity beans describes *logical* relationships among entity beans. It does not provide a mechanism for specifying how the abstract persistence schema of an entity bean or of a set of interrelated entity beans is to be mapped to an underlying database. This is the responsibility of the Deployer, who, using the Container Provider's tools, uses the logical relationships that are specified in the deployment descriptor to map to the physical relationships that are specific to the underlying resource. It is the responsibility of the container to manage the mapping between the logical and physical relationships at runtime and to manage the referential integrity of the relationships.

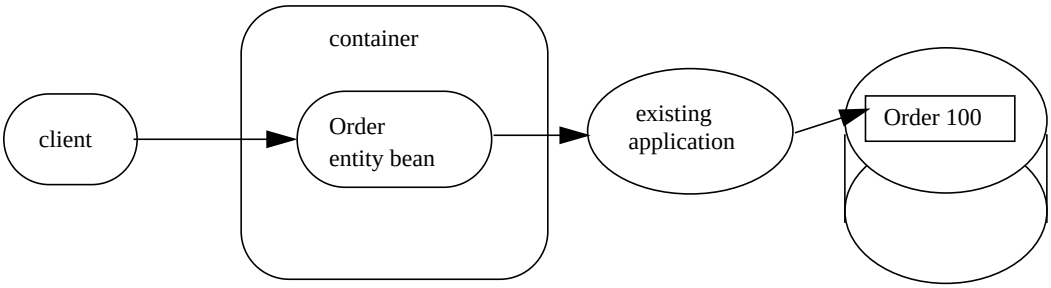
The advantage of using container-managed persistence is that the entity bean can be logically independent of the data source in which the entity is stored. The Container Provider's tools can, for example, generate classes that use JDBC or SQLJ to access the entity state in a relational database; classes that implement access to a non-relational data source, such as an IMS database; or classes that implement function calls to existing enterprise applications. These tools are typically specific to each data source.

Figure 11 View of Underlying Data Sources Accessed Through Entity Bean

(a) Entity bean is an object view of a collection of related records in the database



(b) Entity bean is an object view of an existing application



8.3 The Entity Bean Provider's View of Container-Managed Persistence

An entity bean implements an object view of a business entity or set of business entities stored in an underlying database or implemented by an existing enterprise application (for example, by a mainframe program or by an ERP application).

An entity bean with container-managed persistence typically consists of its entity bean class; a component interface which defines its client view business methods; a home interface which defines the create, remove, home, and finder methods of its client view; and its abstract persistence schema as specified in the deployment descriptor.

A client of an entity bean can control the life cycle of a bean by using the bean's home interface and can manipulate the bean as a business entity by using the methods defined by its component interface. The home and component interfaces of a bean define its client view.

An entity bean with container-managed persistence typically has container-managed relationships with other container-managed persistence entity beans, as defined by the `relationships` element of the deployment descriptor. The architecture for container-managed persistence thus allows the Bean Provider to implement complex applications by defining a complex abstract persistence schema encompassing multiple entity bean classes related by means of container-managed relationships.

An entity bean accesses related entity beans by means of the accessor methods for its container-managed relationship fields, which are specified by the `cmr-field` elements of its abstract persistence schema defined in the deployment descriptor. Entity bean relationships are defined in terms of the local interfaces of the related beans, and the view an entity bean presents to its related beans is defined by its local home and local interfaces. Thus, an entity bean can be the target of a relationship from another entity bean only if it has a local interface.

The Bean Provider programming an application that uses container-managed persistence typically avoids calls to the methods of the remote home and remote interfaces in favor of invoking related beans by means of the methods of their local interfaces. Unlike remote method calls, such internal method invocations are made using call-by-reference and commonly do not involve the checking of method permissions.

The Enterprise JavaBeans architecture for container-managed persistence provides great flexibility to the Bean Provider in designing an application.

> *For example, a group of related entity beans—`Order`, `LineItem`, and `Customer`—might all be defined as having only local interfaces, with a remotable session bean containing the business logic that drives their invocation. The individual entity beans form a coordinated whole that provides an interrelated set of services that are exposed by their several home and component interfaces. The services provided by the local network of entity beans is exposed to the remote client view through the home and remote interfaces of the session bean, which offers a coarser grained remote service.*
>
> *Alternatively, a single entity bean might represent an independent, remotable business object that forms a unit of distribution that is designed to be referenced remotely by multiple enterprise beans and/or other remote clients. Such a remotable entity bean might make use of other entity beans within its local scope to further model its complex internal state. For example, an `Order` entity bean might make use of a `LineItem` entity bean internally, not exposing it to remote clients. In this case, the `Order` entity bean might define both a remote and a local component interface, where the local interface is presented only to the related entity beans, such as `LineItem`, and the remote interface is presented to session beans and/or web-tier clients.*

8.3.1 The Entity Bean Provider's Programming Contract

The Bean Provider must observe the following programming contract when defining an entity bean class that uses container-managed persistence:

- The Bean Provider must define the entity bean class as an abstract class. The container provides the implementation class that is used at runtime.
- The container-managed persistent fields and container-managed relationship fields must *not* be defined in the entity bean class. From the perspective of the Bean Provider, the container-man-

aged persistent fields and container-managed relationship fields are virtual fields only, and are accessed through get and set accessor methods. The implementation of the container-managed persistent fields and container-managed relationship fields is supplied by the container.

- The container-managed persistent fields and container-managed relationship fields must be specified in the deployment descriptor using the `cmp-field` and `cmr-field` elements respectively. The names of these fields must be valid Java identifiers and must begin with a lowercase letter, as determined by `java.lang.Character.isLowerCase`.

- The Bean Provider must define the accessor methods for the container-managed persistent fields and container-managed relationship fields as get and set methods, using the JavaBeans conventions. The implementation of the accessor methods is supplied by the container.

- The accessor methods must be public, must be abstract, and must bear the name of the container-managed persistent field (`cmp-field`) or container-managed relationship field (`cmr-field`) that is specified in the deployment descriptor, and in which the first letter of the name of the cmp-field or cmr-field has been uppercased and prefixed by "`get`" or "`set`".

- The accessor methods for a container-managed relationship field must be defined in terms of the local interface of the related entity bean, as described in Section 8.3.2.

- The accessor methods for container-managed relationship fields for one-to-many or many-to-many relationships must utilize one of the following Collection interfaces: `java.util.Collection` or `java.util.Set`. The Collection interfaces used in relationships are specified in the deployment descriptor. The implementation of the collection classes used for the container-managed relationship fields is supplied by the container.

- An entity bean local interface type (or a collection of such) can be the type of a cmr-field. An entity bean local interface type (or a collection of such) cannot be the type of a cmp-field.

- The accessor methods for the container-managed relationship fields must not be exposed in the remote interface of an entity bean.

- The local interface types of the entity bean and of related entity beans must not be exposed through the remote interface of the entity bean.

- The collection classes that are used for container-managed relationships must not be exposed through the remote interface of the entity bean.

- Once the primary key for an entity bean has been set, the Bean Provider must not attempt to change it by use of set accessor methods on the primary key cmp-fields. The Bean Provider should therefore not expose the set accessor methods for the primary key cmp-fields in the component interface of the entity bean.

- The Bean Provider must ensure that the Java types assigned to the cmp-fields are restricted to the following: Java primitive types and Java serializable types.

8.3.2 The Entity Bean Provider's View of Persistent Relationships

An entity bean may have relationships with other entity beans with container-managed persistence.

Relationships may be one-to-one, one-to-many, or many-to-many relationships.

Container-managed relationships can exist only among entity beans within the same local relationship scope, as defined by the `relationships` element in the deployment descriptor. Container-managed relationships are defined in terms of the local interfaces of the related beans.

Relationships may be either bidirectional or unidirectional. If a relationship is bidirectional, it can be navigated in both directions, whereas a unidirectional relationship can be navigated in one direction only.

A unidirectional relationship is implemented with a cmr-field on the entity bean instance from which navigation can take place, and no related cmr-field on the entity bean instance that is the target of the relationship. Unidirectional relationships are typically used when the Bean Provider wishes to restrict the visibility of a relationship.

An entity bean that does not have a local interface can have only unidirectional relationships from itself to other entity beans. The lack of a local interface prevents other entity beans from having a relationship to it.

The bean developer navigates or manipulates relationships by using the get and set accessor methods for the container-managed relationship fields and the `java.util.Collection` API for collection-valued container-managed relationship fields.

The Bean Provider must consider the type and cardinality of relationships when the entity bean classes are programmed. The get method for a cmr-field must return either the local interface of the entity bean or a collection (either `java.util.Collection` or `java.util.Set`) of the same. The set method for the relationship must take as an argument the entity bean's local interface or a collection of the same.

8.3.3 Dependent Value Classes

A dependent value class is a concrete class that is the value of a cmp-field. A dependent value class may be a class that the Bean Provider wishes to use internally within an entity bean with container-managed persistence, and/or it may be a class that the Bean Provider chooses to expose through the remote (or local) interface of the entity bean.

A dependent value class can be the value of a cmp-field; it cannot be the value of a cmr-field.

The get accessor method for a cmp-field that corresponds to a dependent value class returns a *copy* of the dependent value class instance. The assignment of a dependent value class value to a cmp-field using the set accessor method causes the value to be copied to the target cmp-field.

A dependent value class must be serializable. The internal structure of a dependent value class is not described in the EJB deployment descriptor.

8.3.4 Remove Protocols

The Bean Provider can specify the removal of an entity object in two ways:

- By the use of a `remove` method on the entity bean's component interface or home interface.
- By the use of a `cascade-delete` specification in the deployment descriptor.

8.3.4.1 Remove Methods

When the `remove` method is invoked on an entity object, the container must invoke the entity Bean Provider's `ejbRemove` method as described in Section 8.5.3. After the Bean Provider's `ejbRemove` method returns (and prior to returning to the client), the container must remove the entity object from all relationships in which it participates, and then remove its persistent representation. [26]

- Once an entity has been removed from a relationship, the accessor methods for any relationships to the entity will reflect this removal. An accessor method for a one-to-one or many-to-one relationship to the entity will return null; and an accessor method for a many-to-many relationship to the entity will return a collection from which the entity object has been removed.
- The container must detect any subsequent attempt to invoke an accessor method on the removed entity object and throw the `java.rmi.NoSuchObjectException` if the client is a remote client or the `javax.ejb.NoSuchObjectLocalException` if the client is a local client. The container must detect an attempt to assign a removed entity object as the value of a cmr-field of another object (whether as an argument to a set accessor method or as an argument to a method of the `java.util.Collection` API) and throw the `java.lang.IllegalArgumentException`.

After removing the entity object from all relationships and removing its persistent representation, the container must then cascade the removal to all entity beans with which the entity had been previously in container-managed relationships for which the `cascade-delete` option was specified.

More than one relationship may be affected by the removal of an entity object, as in the following example. Once the shipping address object used by the `Order` bean has been removed, the billing address accessor method will also return null.

```
public void changeAddress()
     Address a = createAddress();
     setShippingAddress(a);
     setBillingAddress(a);
     //both relationships now reference the same entity object
     getShippingAddress().remove();
     if (getBillingAddress() == null) // it must be
          ...
     else ...
          // this is impossible....
```

[26] At this point it must appear to the application that the entity has been removed from the persistent store. If the container employs an optimistic caching strategy and defers the removal of the entity from the database (e.g., to the end of transaction), this must be invisible to the application.

The `remove` method, alone, causes only the entity on which it is invoked to be removed. It does not cause the deletion to be cascaded to other entity objects. In order for the deletion of one entity object to be automatically cascaded to another, the `cascade-delete` mechanism should be used.

8.3.4.2 Cascade-delete

The `cascade-delete` deployment descriptor element is used within a particular relationship to specify that the lifetime of one or more entity objects is dependent upon the lifetime of another entity object.

The `cascade-delete` deployment descriptor element is contained within the `ejb-relationship-role` element. The `cascade-delete` element can only be specified for an `ejb-relationship-role` element contained in an `ejb-relation` element if the *other* `ejb-relationship-role` element in the same `ejb-relation` element specifies a `multiplicity` of `One`. The `cascade-delete` option cannot be specified for a many-to-many relationship. The deletion of one entity object can only be cascaded to cause the deletion of other entity objects if the first entity object is in a one-to-one or one-to-many relationship with those other entity objects.

If an entity is deleted, and the `cascade-delete` deployment descriptor element is specified for a related entity bean, then the removal is cascaded to cause the removal of the related entity object or objects. As with the `remove` operation, the removal triggered by the `cascade-delete` option causes the container to invoke the `ejbRemove` method on the entity bean instance that is to be removed before the persistent representation of that entity object is removed. Once an entity has been removed from a relationship because of a cascaded delete, the accessor methods for any relationships to the entity will reflect this removal. An accessor method for a one-to-one or many-to-one relationship to the entity will return null; and an accessor method for a many-to-many relationship to the entity will return a collection from which the entity object has been removed. After removing the entity object from all relationships and removing its persistent representation, the container must then cascade the removal to all entity beans with which the entity had been previously been in container-managed relationships for which the `cascade-delete` option was specified.

The use of `cascade-delete` causes only the entity object or objects in the relationship for which it is specified to be deleted. It does not cause the deletion to be further cascaded to other entity objects, unless they are participants in relationship roles for which `cascade-delete` has also been specified.

8.3.5 Identity of Entity Objects

From the viewpoint of the Bean Provider, entity objects have a runtime object identity that is maintained by the container.

The container maintains the persistent identity of an entity object on the basis of its primary key.

The primary key of an entity bean may or may not be visible as one or more cmp-fields of the instance, depending on the way in which it is specified. The Bean Provider specifies the primary key as described in Section 8.8. Once it has been set, the Bean Provider must not attempt to change the value of a primary key field by means of a set method on its cmp-fields.

When a new instance of an entity bean whose primary key fields are visible in the entity bean class is created, the Bean Provider must use the `ejbCreate<METHOD>` method to set all the primary key fields of the entity bean instance before the instance can participate in a relationship, e.g. be used in a set accessor method for a cmr-field. The Bean Provider must not reset a primary key value by means of a set method on any of its cmp-fields after it has been set in the `ejbCreate<METHOD>` method. If the Bean Provider attempts to reset a primary key value, the container must throw the `java.lang.IllegalStateException`.

Note that the container's implementation of the referential integrity semantics for container-managed relationships must not cause the value of the primary key to change.

The Bean Provider should not use untrimmed or blank-padded string-valued primary key fields. Use of untrimmed primary key fields may cause comparison operations based on primary keys to fail, and may result in non-portable behavior. If untrimmed strings are used in primary key fields or other cmp-fields, the container or database system may trim them.

8.3.6 Semantics of Assignment for Relationships

The assignment operations for container-managed relationships have a special semantics that is determined by the referential integrity semantics for the relationship multiplicity.

In the case of a one-to-one relationship, when the Bean Provider uses a set accessor method to assign an object from a cmr-field in one instance to a cmr-field of the *same relationship type* (i.e., as defined by the `ejb-relation` and `ejb-relationship-role` deployment descriptor elements) in another instance, the object is effectively *moved* and the value of the source cmr-field is set to null in the same transaction context. If the argument to the set accessor method is not of the same type as the cmr-field, the container must throw the `java.lang.IllegalArgumentException`.

In the case of a one-to-many or many-to-many relationship, either the `java.util.Collection` API or a set accessor method may be used to manipulate the contents of a collection-valued cmr-field. These two approaches are discussed below.

8.3.6.1 Use of the java.util.Collection API to Update Relationships

The methods of the `java.util.Collection` API for the container-managed collections used for collection-valued cmr-fields have the usual semantics, with the following exception: the `add` and `addAll` methods applied to container-managed collections in one-to-many relationships have a special semantics that is determined by the referential integrity of one-to-many relationships.

- If the argument to the `add` method is already an element of a collection-valued relationship field of the *same relationship type* as the target collection (as defined by the `ejb-relation` and `ejb-relationship-role` deployment descriptor elements), it is removed from this first relationship and added, in the same transaction context, to the target relationship (i.e., it is effectively moved from one collection of the relationship type to the other). For example, if there is a one-to-many relationship between field offices and sales representatives, adding a sales representative to a new field office will have the effect of removing him or her from his or her current field office. If the argument to the `add` method is not an element of a collection-valued relationship of the *same relationship type*, it is simply added to the target collection and not removed from its current collection, if any.

- The `addAll` method, when applied to a target collection in a one-to-many relationship, has similar semantics, applied to the members of its collection argument individually.

Note that in the case of many-to-many relationships, adding an element or elements to the contents of a collection-valued cmr-field has no effect on the source collection, if any. For example, if there is a many-to-many relationship between customers and sales representatives, a customer can be added to the set of customers handled by a particular sales representative without affecting the set of customers handled by any other sales representative.

When the `java.util.Collection` API is used to manipulate the contents of container-managed relationship fields, the argument to any Collection method defined with a single Object parameter must be of the element type of the collection defined for the target cmr-field. The argument for any collection-valued parameter must be a `java.util.Collection` (or `java.util.Set`), all of whose elements are of the element type of the collection defined for the target cmr-field. If an argument is not of the correct type for the relationship, the container must throw the `java.lang.IllegalArgumentException`.

The Bean Provider should exercise caution when using an Iterator over a collection in a container-managed relationship. In particular, the Bean Provider should not modify the container-managed collection while the iteration is in progress in any way that causes elements to be added or removed, other than by the `java.util.Iterator.remove()` method. If elements are added or removed from the underlying container-managed collection used by an iterator other than by the `java.util.Iterator.remove()` method, the container should throw the `java.lang.IllegalStateException` on the next operation on the iterator.

The following example illustrates how operations on container-managed relationships that affect the contents of a collection-valued cmr-field viewed through an iterator can be avoided. Because there is a one-to-many relationship between field offices and sales representatives, adding a sales representative to a new field office causes the sales representative to be removed from the current field office.

```
Collection nySalesreps = nyOffice.getSalesreps();
Collection sfSalesreps = sfOffice.getSalesreps();

Iterator i = nySalesreps.iterator();
Salesrep salesrep;

// a wrong way to transfer the salesrep
while (i.hasNext()) {
     salesrep = (Salesrep)i.next();
     sfSalesreps.add(salesrep); // removes salesrep from nyOffice
}

// this is a correct and safe way to transfer the salesrep
while (i.hasNext()) {
     salesrep = (Salesrep)i.next();
     i.remove();
     sfSalesreps.add(salesrep);
}
```

8.3.6.2 Use of Set Accessor Methods to Update Relationships

The semantics of a set accessor method, when applied to a collection-valued cmr-field, is also determined by the referential integrity semantics associated with the multiplicity of the relationship. The identity of the collection object referenced by a cmr-field does not change when a set accessor method is executed.

In the case of a one-to-many relationship, if a collection of entity objects is assigned from a cmr-field of in one instance to a cmr-field of the same relationship type in another instance, the objects in the collection are effectively moved. The contents of the collection of the target instance are replaced with the contents of the collection of the source instance, but the *identity* of the collection object containing the instances in the relationship does not change. The source cmr-field references the same collection object as before (i.e., the identity of the collection object is preserved), but the collection is empty.

The Bean Provider can thus use the set method to move objects between the collections referenced by cmr-fields of the same relationship type in different instances. The set accessor method, when applied to a cmr-field in a one-to-many relationship thus has the semantics of the `java.util.Collection` methods `clear`, followed by `addAll`, applied to the target collection; and `clear`, applied to the source collection. It is the responsibility of the container to transfer the contents of the collection instances in the same transaction context.

> *Note that if the collection that is passed to the cmr setter method is an unmanaged collection (i.e., not itself the value of a collection-valued cmr-field), the same requirements apply in the case that the collection contains entity objects that already participate in a one-to-many relationship of the same relationship type as the target cmr-field.*

In the following example, the telephone numbers associated with the billing address of an `Order` bean instance are transferred to the shipping address. Billing address and shipping address are different instances of the same local interface type, `Address`. `Address` is related to `TelephoneNumber` in a one-to-many relationship. The example illustrates how a Bean Provider uses the set method to move a set of instances.

```
public void changeTelephoneNumber() {
     Address a = getShippingAddress();
     Address b = getBillingAddress();
     Collection c = b.getTelephoneNumbers();
     a.setTelephoneNumbers(b.getTelephoneNumbers());
     if (c.isEmpty()) { // must be true...
     ...
}
```

In the case of a many-to-many relationship, if the value of a cmr-field is assigned to a cmr-field of the same relationship type in another instance, the objects in the collection of the first instance are assigned as the contents of the cmr-field of the second instance. The identities of the collection objects referenced by the cmr-fields do not change. The contents of the collections are shared, but not the collections themselves. The set accessor method, when applied to a cmr-field in a many-to-many relationship thus has the semantics of the `java.util.Collection` methods `clear`, followed by `addAll`, applied to the target collection.

For example, if there is a many-to-many relationship between customers and sales representatives, assigning the set of customers of one sales representative to the another sales representative will result in both sales representatives handling the same customers. If the second sales representative originally handled a different group of customers, those customers will no longer be handled by that sales representative.

```
public void shareCustomers(SalesRep rep) {
     setCustomers(rep.getCustomers());
     // the customers are shared among the sales reps
}
```

The following section, 8.3.7, "Assignment Rules for Relationships", defines the semantics of assignment for relationships in further detail.

8.3.7 Assignment Rules for Relationships

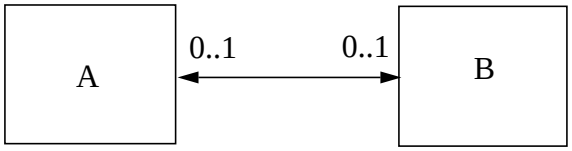
This section defines the semantics of assignment and collection manipulation in one-to-one, one-to-many, and many-to-many container-managed relationships.

The figures make use of two entity beans, with local interface types A and B. Instances with local interface type `A` are typically designated as `a1`,...,`an`; instances with local interface type `B` are typically designated as `b1`,...,`bm`. Interface `A` exposes accessor methods `getB` and `setB` for navigable relationships with `B`: `getB` returns an instance of `B` or a collection of instances of `B`, depending on the multiplicity of the relationship. Similarly, `B` exposes accessor methods `getA` and `setA` for navigable relationships with `A`.

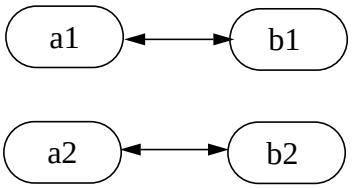
All changes in each subsection are assumed to be applied to the figure labeled "Before change" at the beginning of the subsection (i.e., changes are not cumulative). The results of changes are designated graphically as well as in conditional expressions expressed in the Java™ programming language.

8.3.7.1 One-to-one Bidirectional Relationships

A and B are in a one-to-one bidirectional relationship:



Before change:



Before change:

```
B b1 = a1.getB();
B b2 = a2.getB();
```

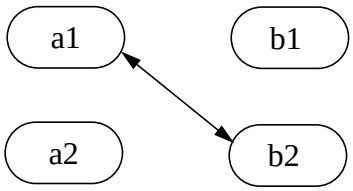
Change:

```
a1.setB(a2.getB());
```

Expected result:

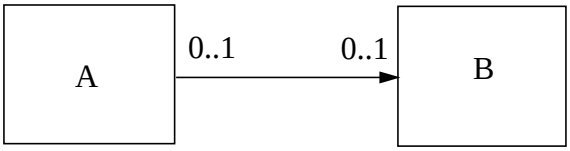
```
(b2.isIdentical(a1.getB())) &&
(a2.getB() == null) &&
(b1.getA() == null) &&
(a1.isIdentical(b2.getA()))
```

After change:

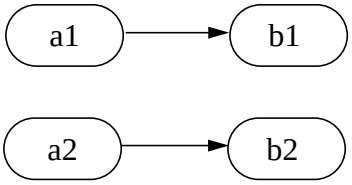


8.3.7.2 One-to-one Unidirectional Relationships

A and B are in a one-to-one unidirectional relationship:



Before change:



Before change:

```
B b1 = a1.getB();
B b2 = a2.getB();
```

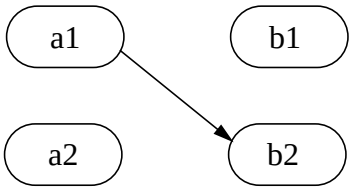
Change:

```
a1.setB(a2.getB());
```

Expected result:

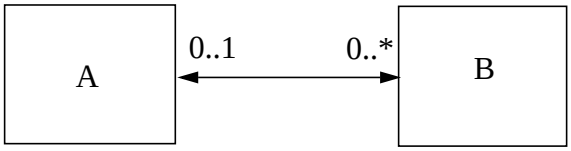
```
(b2.isIdentical(a1.getB())) && (a2.getB() == null)
```

After change:

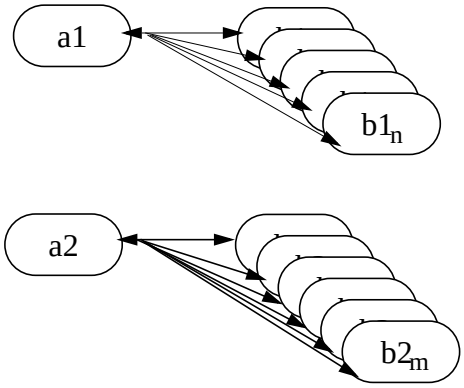


8.3.7.3 One-to-many Bidirectional Relationships

A and B are in a one-to-many bidirectional relationship:



Before change:



Before change:

```
Collection b1 = a1.getB();
Collection b2 = a2.getB();
B b11, b12, ... , b1n; // members of b1
B b21, b22, ... , b2m; // members of b2
```

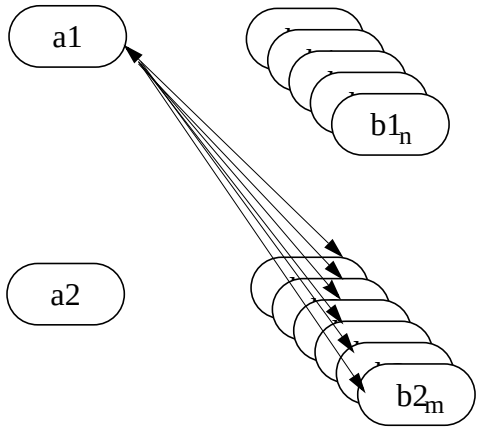
Change:

```
a1.setB(a2.getB());
```

Expected result:

```
(a2.getB().isEmpty()) &&
(b2.isEmpty()) &&
(b1 == a1.getB()) &&
(b2 == a2.getB()) &&
(a1.getB().contains(b21)) &&
(a1.getB().contains(b22)) &&  ... &&
(a1.getB().contains(b2m)) &&
(b11.getA() == null) &&
(b12.getA() == null) &&  ... &&
(b1n.getA() == null) &&
(a1.isIdentical(b21.getA())) &&
(a1.isIdentical(b22.getA())) && ...&&
(a1.isIdentical(b2m.getA()))
```

After change:



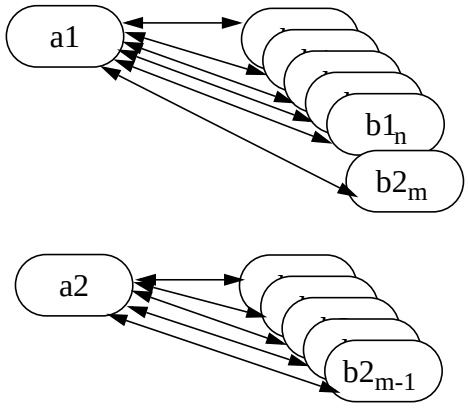
Change:

```
b2m.setA(b1n.getA());
```

Expected result:

```
(b1.contains(b11)) &&
(b1.contains(b12)) &&  ... &&
(b1.contains(b1n)) &&
(b1.contains(b2m)) &&
(b2.contains(b21)) &&
(b2.contains(b22)) &&  ... &&
(b2.contains(b2m_1)) &&
(a1.isIdentical(b11.getA())) &&
(a1.isIdentical(b12.getA())) &&  ... &&
(a1.isIdentical(b1n.getA())) &&
(a2.isIdentical(b21.getA())) &&
(a2.isIdentical(b22.getA())) &&  ... &&
(a2.isIdentical(b2m_1.getA())) &&
(a1.isIdentical(b2m.getA()))
```

After change:



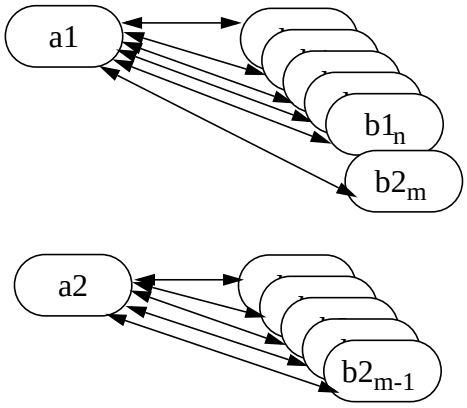
Change:

```
a1.getB().add(b2m);
```

Expected result:

```
(b1.contains(b11)) &&
(b1.contains(b12)) &&  ... &&
(b1.contains(b1n)) &&
(b1.contains(b2m)) &&
(b2.contains(b21)) &&
(b2.contains(b22)) &&  ... &&
(b2.contains(b2m_1)) &&
(a1.isIdentical(b11.getA())) &&
(a1.isIdentical(b12.getA())) &&  ... &&
(a1.isIdentical(b1n.getA())) &&
(a2.isIdentical(b21.getA())) &&
(a2.isIdentical(b22.getA())) &&  ... &&
(a2.isIdentical(b2m_1.getA())) &&
(a1.isIdentical(b2m.getA()))
```

After change:



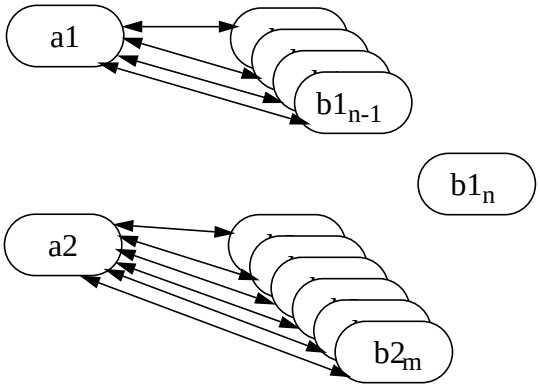
Change:

```
a1.getB().remove(b1n);
```

Expected result:

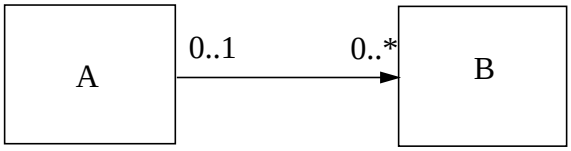
```
(b1n.getA() == null) &&
(b1 == a1.getB()) &&
(b1.contains(b11)) &&
(b1.contains(b12)) &&  ... &&
(b1.contains(b1n_1)) &&
!(b1.contains(b1n))
```

After change:

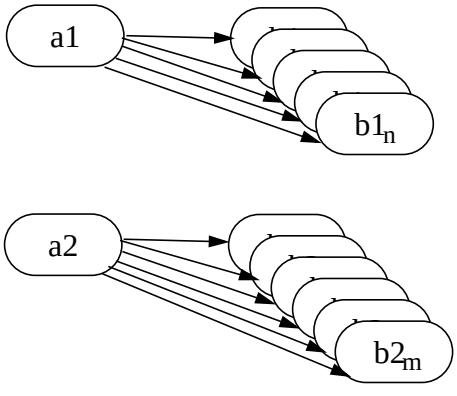


8.3.7.4 One-to-many Unidirectional Relationships

A and B are in a one-to-many unidirectional relationship:



Before change:



Before change:

```
Collection b1 = a1.getB();
Collection b2 = a2.getB();
B b11, b12, ... , b1n; // members of b1
B b21, b22, ... , b2m; // members of b2
```

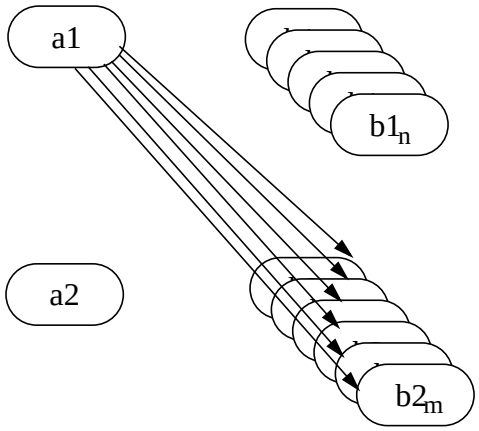
Change:

```
a1.setB(a2.getB());
```

Expected result:

```
(a2.getB().isEmpty()) &&
(b2.isEmpty()) &&
(b1 == a1.getB()) &&
(b2 == a2.getB()) &&
(a1.getB().contains(b21)) &&
(a1.getB().contains(b22)) &&  ... &&
(a1.getB().contains(b2m))
```

After change:



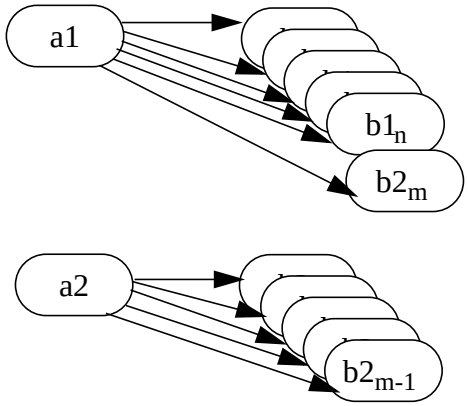
Change:

```
a1.getB().add(b2m);
```

Expected result:

```
(b1 == a1.getB()) &&
(b1.contains(b2m))
```

After change:



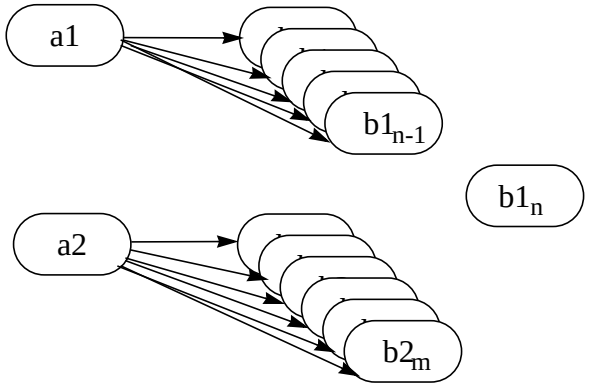
Change:

```
a1.getB().remove(b1n);
```

Expected result:

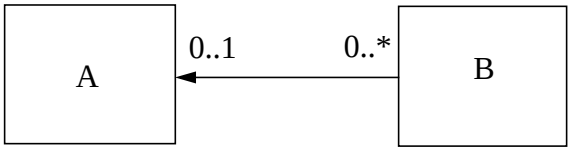
```
(a1.getB().contains(b11)) &&
(a1.getB().contains(b12)) &&  ... &&
(a1.getB().contains(b1n_1)) &&
!(a1.getB().contains(b1n)) &&
```

After change:

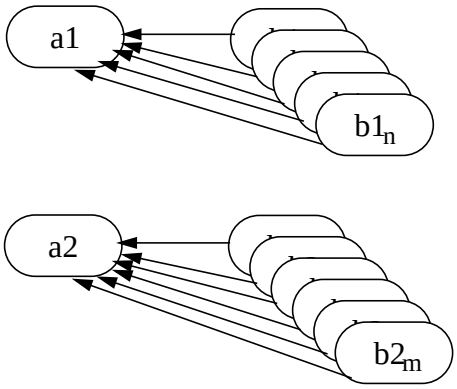


8.3.7.5 Many-to-one Unidirectional Relationships

A and B are in a many-to-one unidirectional relationship:



Before change:



Before change:

```
B b11, b12, ... , b1n;
B b21, b22, ... , b2m;
// the following is true
// (a1.isIdentical(b11.getA())) && ... && (a1.isIdentical(b1n.getA()
)) &&
// (a2.isIdentical(b21.getA())) && ... && (a2.isIdentical(b2m.getA()
))
```

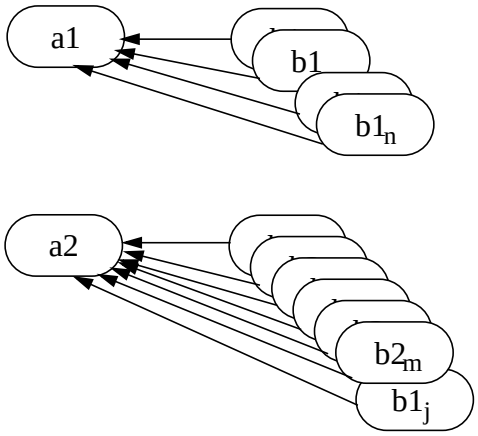
Change:

```
b1j.setA(b2k.getA());
```

Expected result:

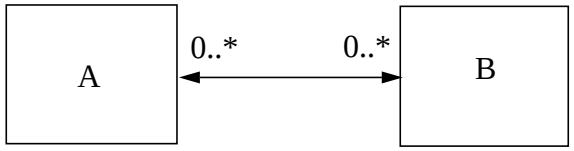
```
(a1.isIdentical(b11.getA())) &&
(a1.isIdentical(b12.getA())) &&
...
(a2.isIdentical(b1j.getA())) &&
...
(a1.isIdentical(b1n.getA())) &&
(a2.isIdentical(b21.getA())) &&
(a2.isIdentical(b22.getA())) &&
...
(a2.isIdentical(b2k.getA())) &&
...
(a2.isIdentical(b2m.getA()))
```

After change:

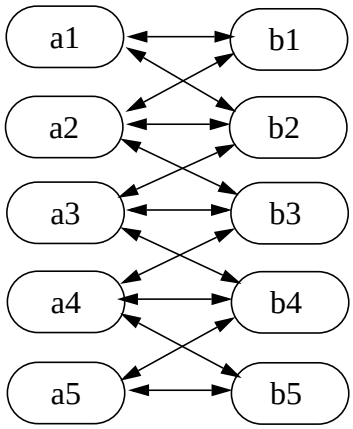


8.3.7.6 Many-to-many Bidirectional Relationships

A and B are in a many-to-many bidirectional relationship:



Before change:



Before change the following holds:

```
(a1.getB().contains(b1)) &&
(a1.getB().contains(b2)) &&
(a2.getB().contains(b1)) &&
(a2.getB().contains(b2)) &&
(a2.getB().contains(b3)) &&
(a3.getB().contains(b2)) &&
(a3.getB().contains(b3)) &&
(a3.getB().contains(b4)) &&
(a4.getB().contains(b3)) &&
(a4.getB().contains(b4)) &&
(a4.getB().contains(b5)) &&
(a5.getB().contains(b4)) &&
(a5.getB().contains(b5)) &&
(b1.getA().contains(a1)) &&
(b1.getA().contains(a2)) &&
(b2.getA().contains(a1)) &&
(b2.getA().contains(a2)) &&
(b2.getA().contains(a3)) &&
(b3.getA().contains(a2)) &&
(b3.getA().contains(a3)) &&
(b3.getA().contains(a4)) &&
(b4.getA().contains(a3)) &&
(b4.getA().contains(a4)) &&
(b4.getA().contains(a5)) &&
(b5.getA().contains(a4)) &&
(b5.getA().contains(a5)) &&
```

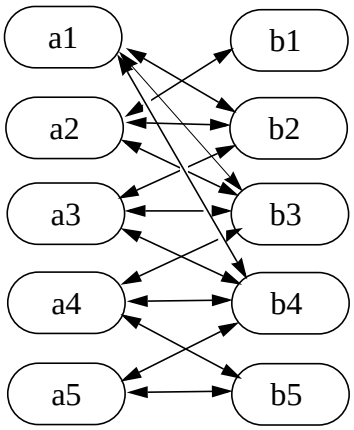
Change:

```
a1.setB(a3.getB());
```

Expected result:

```
(a1.getB().contains(b2)) &&
(a1.getB().contains(b3)) &&
(a1.getB().contains(b4)) &&
(a3.getB().contains(b2)) &&
(a3.getB().contains(b3)) &&
(a3.getB().contains(b4)) &&
(b1.getA().contains(a2)) &&
(b2.getA().contains(a1)) &&
(b2.getA().contains(a2)) &&
(b2.getA().contains(a3)) &&
(b3.getA().contains(a1)) &&
(b3.getA().contains(a2)) &&
(b3.getA().contains(a3)) &&
(b3.getA().contains(a4)) &&
(b4.getA().contains(a1)) &&
(b4.getA().contains(a3)) &&
(b4.getA().contains(a4)) &&
(b4.getA().contains(a5))
```

After change:



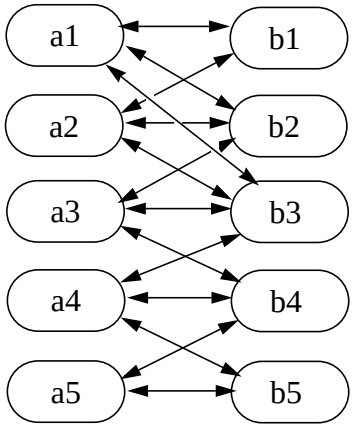
Change:

```
a1.getB().add(b3);
```

Expected result:

```
(a1.getB().contains(b1)) &&
(a1.getB().contains(b2)) &&
(a1.getB().contains(b3)) &&
(b3.getA().contains(a1)) &&
(b3.getA().contains(a2)) &&
(b3.getA().contains(a3)) &&
(b3.getA().contains(a4)) &&
```

After change:



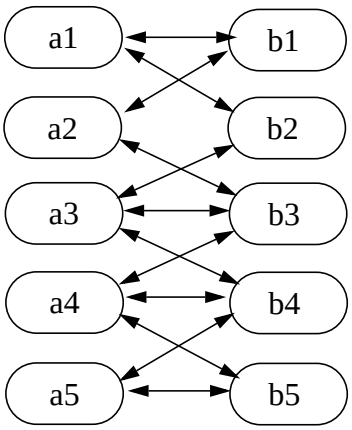
Change:

```
a2.getB().remove(b2);
```

Expected result:

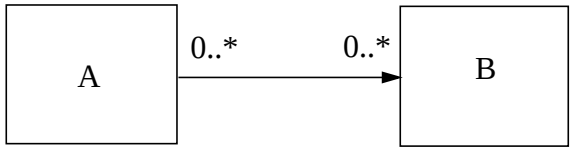
```
(a2.getB().contains(b1)) &&
(a2.getB().contains(b3)) &&
(b2.getA().contains(a1)) &&
(b2.getA().contains(a3))
```

After change:

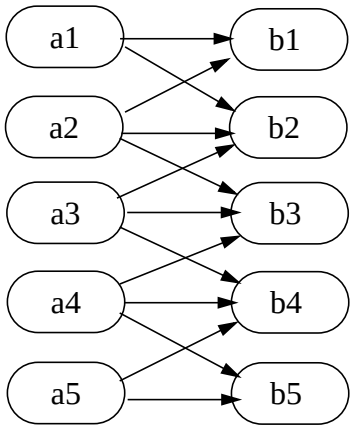


8.3.7.7 Many-to-many Unidirectional Relationships

A and B are in a many-to-many unidirectional relationship:



Before change:



Before change the following holds:

```
(a1.getB().contains(b1)) &&
(a1.getB().contains(b2)) &&
(a2.getB().contains(b1)) &&
(a2.getB().contains(b2)) &&
(a2.getB().contains(b3)) &&
(a3.getB().contains(b2)) &&
(a3.getB().contains(b3)) &&
(a3.getB().contains(b4)) &&
(a4.getB().contains(b3)) &&
(a4.getB().contains(b4)) &&
(a4.getB().contains(b5)) &&
(a5.getB().contains(b4)) &&
(a5.getB().contains(b5)) &&
```

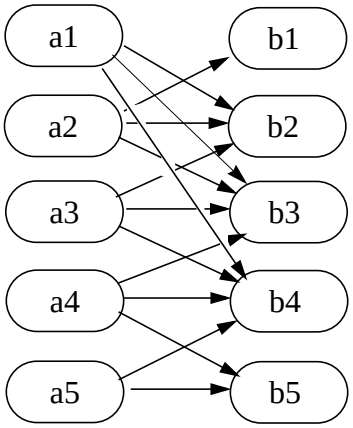
Change:

```
a1.setB(a3.getB());
```

Expected Result:

```
(a1.getB().contains(b2)) &&
(a1.getB().contains(b3)) &&
(a1.getB().contains(b4)) &&
(a3.getB().contains(b2)) &&
(a3.getB().contains(b3)) &&
(a3.getB().contains(b4)) &&
```

After change:



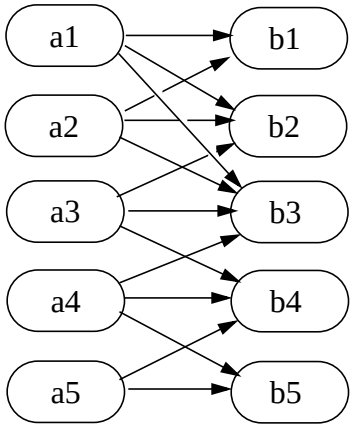
Change:

```
a1.getB().add(b3);
```

Expected result:

```
(a1.getB().contains(b1)) &&
(a1.getB().contains(b2)) &&
(a1.getB().contains(b3))
```

After change:



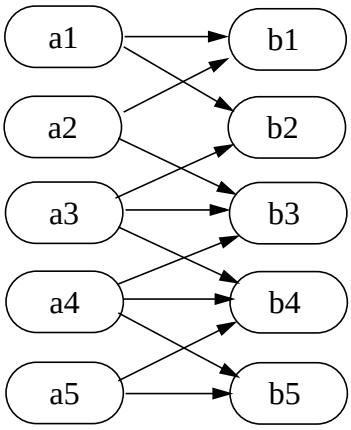
Change:

```
a2.getB().remove(b2);
```

Expected result:

```
(a2.getB().contains(b1)) &&
(a2.getB().contains(b3))
```

After change:



8.3.8 Collections Managed by the Container

The collections that are used in the representation of one-to-many and many-to-many container-managed relationships are implemented and managed by the container. The following semantics apply to these collections:

- It is the responsibility of the container to preserve the runtime identity of the collection objects used in container-managed relationships.
- There is no constructor available to the Bean Provider for the container-managed collections.
- If there are no related values for a given container-managed relationship, the get accessor method for that cmr-field returns an empty collection (and not `null`).
- It is the responsibility of the container to raise the `java.lang.IllegalArgumentException` if the Bean Provider attempts to assign `null` as the value of a collection-valued cmr-field by means of the set accessor method.
- It is the responsibility of the container to ensure that when the `java.util.Collection` API is used to manipulate the contents of container-managed relationship fields, the argument to any Collection method defined with a single Object parameter must be of the element type of the collection defined for the target cmr-field. The argument for any collection-valued parameter must be a `java.util.Collection` (or `java.util.Set`), all of whose elements are of the element type of the collection defined for the target cmr-field. If an argument is not of the correct type for the relationship, the container must throw the `java.lang.IllegalArgumentException`.
- It is the responsibility of the container to throw the `java.lang.IllegalStateException` if an attempt is made to modify a container-managed collection corresponding to a multivalued cmr-field using the `java.util.Collection` API outside of the transaction context in which the collection object was initially materialized.
- It is the responsibility of the container to throw the `java.lang.IllegalStateException` if an attempt is made to use a `java.util.Iterator` for a container-managed collection in a transaction context other than that in which the iterator was obtained.

8.3.9 Non-persistent State

The Bean Provider may use instance variables in the entity bean instance to maintain non-persistent state, e.g. a JMS connection.

The Bean Provider can use instance variables to store values that depend on the persistent state of the entity bean instance, although this use is not encouraged. The Bean Provider should use the `ejbLoad` method to resynchronize the values of any instance variables that depend on the entity bean's persistent state. In general, any non-persistent state that depends on the persistent state of an entity bean should be recomputed during the `ejbLoad` method.

The Bean Provider should exercise care in passing the contents of instance variables as the arguments or results of method invocations when local interfaces are used. In general, the Bean Provider should avoid passing state that is maintained in instance variables as the argument or result of a local method invocation.

8.3.10 The Relationship Between the Internal View and the Client View

In designing the entity bean, the Bean Provider should keep in mind the following:

- The classes that are exposed by the remote interface are decoupled from the persistence layer. Instances of these classes are passed to and from the client by value.
- The classes that are exposed by the local interface of the bean may be tightly coupled to the bean's internal state. Instances of these classes are passed to and from the client by reference and may therefore be modified by the client. The Bean Provider should exercise care in determining what is exposed through the local interface of the bean.

8.3.10.1 Restrictions on Remote Interfaces

The following restrictions apply to the remote interface of an entity bean with container-managed persistence.

- The Bean Provider must not expose the get and set methods for container-managed relationship fields or the persistent `Collection` classes that are used in container-managed relationships through the remote interface of the bean.
- The Bean Provider must not expose local interface types or local home interface types through the remote interface or remote home interface of the bean.
- The Bean Provider must not expose the container-managed collection classes that are used for relationships through the remote interface of the bean.
- The Bean Provider must not expose timers or timer handles through the remote interface of the bean.

Dependent value classes can be exposed in the remote interface or remote home interface and can be included in the client ejb-jar file.

The Bean Provider is free to expose get and set methods that correspond to cmp-fields of the entity bean through the bean's remote interface.

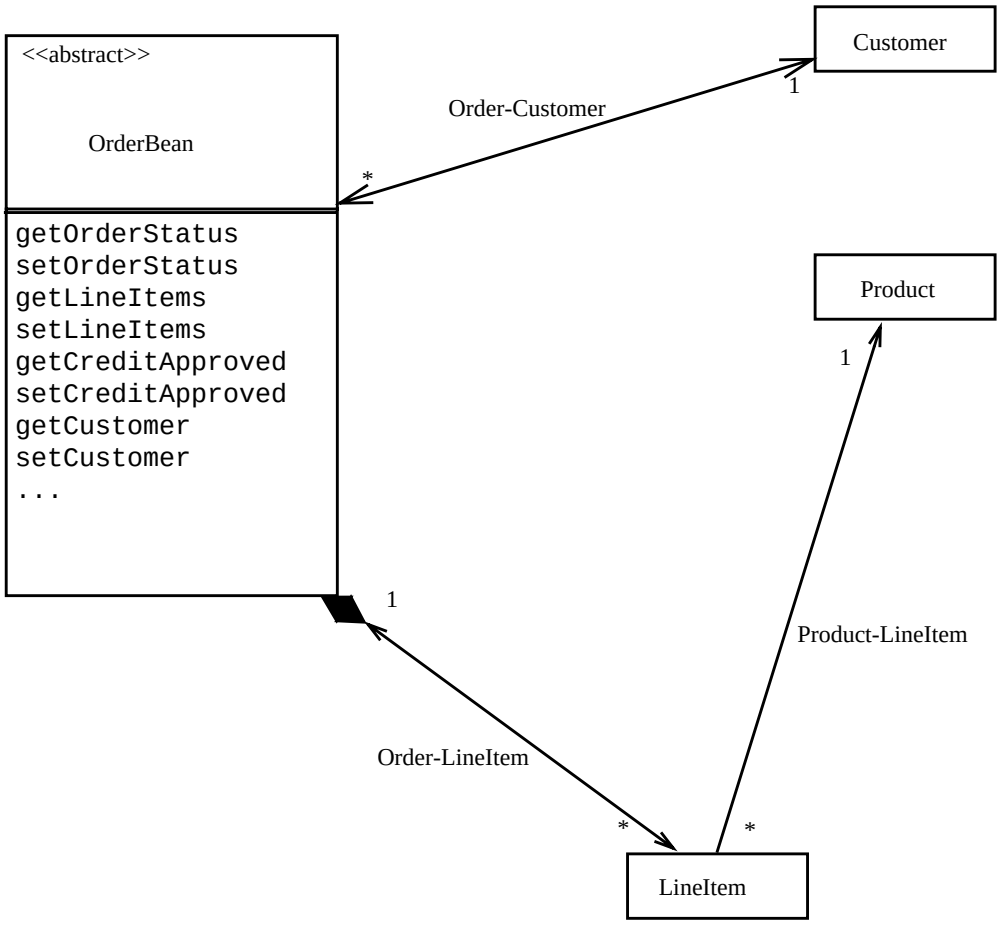
8.3.11 Mapping Data to a Persistent Store

This specification does not prescribe how the abstract persistence schema of an entity bean should be mapped to a relational (or other) schema of a persistent store, or define how such a mapping is described.

8.3.12 Example

Figure 12 illustrates an `Order` entity bean with relationships to line items and customers, which are other entity beans within the same local scope. `Product` is indirectly related to `Order` by means of the relationship between `LineItem` and `Product`. Sample code for the `OrderBean` class follows the figure.

Figure 12 Relationship Example



```
package com.acme.order;

// This example shows the implementation of OrderBean, the
// entity bean class for the OrderEJB entity bean. OrderEJB has
// container-managed relationships with the entity beans
// CustomerEJB and LineItemEJB.
// This example illustrates the use of local interfaces.

import java.util.Collection;
import java.util.Vector;
import java.util.Date;

import javax.naming.*;

public abstract class OrderBean implements javax.ejb.EntityBean {

    private javax.ejb.EntityContext context;

    // define status codes for processing

    static final int BACKORDER = 1;
    static final int SHIPPED = 2;
    static final int UNSHIPPED = 3;

    // get and set methods for the cmp fields

    public abstract int getOrderStatus();
    public abstract void setOrderStatus(int orderStatus);

    public abstract boolean getCreditApproved();
    public abstract void setCreditApproved(boolean creditapproved);

    public abstract Date getOrderDate();
    public abstract void setOrderDate(Date orderDate);

    // get and set methods for the relationship fields

    public abstract Collection getLineItems();
    public abstract void setLineItems(Collection lineitems);

    public abstract Customer getCustomer();
    public abstract void setCustomer(Customer customer);

    // business methods.


    // addLineItem:
    // This method is used to add a line item.
    // It creates the lineitem object and adds it to the
    // persistent managed relationship.

    public void addLineItem(Product product,
                 int quantity,
                 Address address)
     throws InsufficientInfoException
    {
     // create a new line item
```

```
if (validAddress(address)) {
    // Address is a legacy class. It is a dependent value
    // class that is available both in the client and in
    // the entity bean, and is serializable.
    // We will use the address as the value of a cmp field
    // of lineItem.

        try {
            Context ic = new InitialContext();
            LineItemLocalHome litemLocalHome =
                (LineItemLocalHome)ic.lookup("LineItemEJB");
            LineItem litem = litemLocalHome.create();

            litem.setProduct(product);
            litem.setQuantity(quantity);
            litem.setTax(calculateTax(product.getPrice(),
                                      quantity,
                                      address));
            litem.setStatus(UNSHIPPED);
            // set the address for the line item to be shipped
            litem.setAddress(address);
            // The lineItem entity bean uses a dependent value
            // class to represent the dates for the order status.
            // This class holds shipment date, expected shipment
            // date, credit approval date, and inventory
            // dates which are internal to the order fullfillment
           // process. Not all this information will be available
            // to the client.

            Dates dates = new Dates();
            litem.setDates(dates);
            getLineItems().add(litem);
        } catch (Exception someexception) {}
} else {
    throw new InsufficientInfoException();
}
}


// getOrderLineItems:
// This method makes a view of the lineitems that are in this
// order available in the client. It makes only the relevant
// information visible to the client and hides the internal
// details of the representation of the lineitem
public Collection getOrderLineItems() {
Vector clientlineitems = new Vector();
Collection lineitems = getLineItems();
java.util.Iterator iterator = lineitems.iterator();
// ClientLineItem is a value class that is used in
// the client view.
// The entity bean provider abstracts from the persistent
// representation of the line item to construct the client
// view.
ClientLineItem clitem;
while (iterator.hasNext()) {
    LineItem litem = (LineItem)iterator.next();
    clitem = new ClientLineItem();
    // only the name of the product is available in the
    // client view
```

```
            clitem.setProductName(litem.getProduct().getName());
            clitem.setQuantity(litem.getQuantity());
            // the client view gets a specific descriptive message
            // depending on the line item status.
            clitem.setCurrentStatus(
            statusCodeToString(litem.getStatus()));
            // address is not copied to the client view.
            // as this class includes other information with
            // respect to the order handing that should not be
            // available to the client. Only the relevant info
            // is copied.
            int lineitemStatus = litem.getStatus();
            if ( lineitemStatus == BACKORDER) {
             clitem.setShipDate(
             litem.getDates().getExpectedShipDate());
            } else if (lineitemStatus == SHIPPED) {
             clitem.setShipDate(
             litem.getDates().getShippedDate());
            }
            //add the new line item
            clientlineitems.add(clitem);
        }
        // return the value objects to the client
        return clientlineitems;
       }

        // other methods internal to the entity bean class
        ...

       // other javax.ejb.EntityBean methods
        ...
}
```

8.3.13 The Bean Provider's View of the Deployment Descriptor

The persistent fields (cmp-fields) and relationships (cmr-fields) of an entity bean must be declared in the deployment descriptor.

The deployment descriptor provides the following information about the abstract persistence schemas of entity beans and their container-managed relationships:

- An `ejb-name` element for each entity bean. The `ejb-name` must be a valid Java identifier and must be unique within the `ejb-name` elements of the ejb-jar file.
- An `abstract-schema-name` element for each entity bean. The `abstract-schema-name` must be a valid Java identifier and must be unique within the `abstract-schema-name` elements of the ejb-jar file. The `abstract-schema-name` element is used in the specification of EJB QL queries.
- A set of `ejb-relation` elements, each of which contains a pair of `ejb-relation-ship-role` elements to describe the two roles in the relationship.[27]

- Each `ejb-relationship-role` element describes a relationship role: its name, its multiplicity within a relation, and its navigability. It specifies the name of the `cmr-field` that is used from the perspective of the relationship participant. The `cmr-field-type` element must be specified if the type of the cmr-field is `java.util.Collection` or `java.util.Set`. Each relationship role refers to an entity bean by means of an `ejb-name` element contained in the `relationship-role-source` element.

[27] The relation names and the relationship role names are not used in the code provided by the Bean Provider.

The following example shows a deployment descriptor segment that defines the abstract persistence schema for a set of related entity beans. The deployment descriptor elements for container-managed persistence and relationships are described further in Chapter 18.

```
<ejb-jar>

...

<enterprise-beans>
...
</enterprise-beans>

<relationships>

<!--
ONE-TO-MANY: Order LineItem
-->

     <ejb-relation>
          <ejb-relation-name>Order-LineItem</ejb-relation-name>
          <ejb-relationship-role>
               <ejb-relationship-role-name>
               order-has-lineitems
               </ejb-relationship-role-name>
               <multiplicity>One</multiplicity>
               <relationship-role-source>
                    <ejb-name>OrderEJB</ejb-name>
               </relationship-role-source>
               <cmr-field>
                    <cmr-field-name>lineItems</cmr-field-name>
                    <cmr-field-type>java.util.Collection
                    </cmr-field-type>
               </cmr-field>
          </ejb-relationship-role>

          <ejb-relationship-role>
               <ejb-relationship-role-name>lineitem-belongsto-order
               </ejb-relationship-role-name>
               <multiplicity>Many</multiplicity>
               <cascade-delete/>
               <relationship-role-source>
                    <ejb-name>LineItemEJB</ejb-name>
               </relationship-role-source>
               <cmr-field>
                    <cmr-field-name>order</cmr-field-name>
               </cmr-field>
          </ejb-relationship-role>
     </ejb-relation>

<!--
ONE-TO-MANY unidirectional relationship:
Product is not aware of its relationship with LineItem
-->

     <ejb-relation>
          <ejb-relation-name>Product-LineItem</ejb-relation-name>

          <ejb-relationship-role>
               <ejb-relationship-role-name>
```

```
                product-has-lineitems
                </ejb-relationship-role-name>
                <multiplicity>One</multiplicity>
                <relationship-role-source>
                    <ejb-name>ProductEJB</ejb-name>
                </relationship-role-source>
                  <!-- since Product does not know about LineItem
                       there is no cmr field in Product for accessing
                         Lineitem
                  -->
            </ejb-relationship-role>

            <ejb-relationship-role>
                <ejb-relationship-role-name>
                lineitem-for-product
                </ejb-relationship-role-name>
                <multiplicity>Many</multiplicity>
                <relationship-role-source>
                    <ejb-name>LineItemEJB</ejb-name>
                </relationship-role-source>
                <cmr-field>
                    <cmr-field-name>product</cmr-field-name>
                </cmr-field>
            </ejb-relationship-role>

        </ejb-relation>


    <!--
    ONE-TO-MANY: Order Customer:
    -->

        <ejb-relation>
            <ejb-relation-name>Order-Customer</ejb-relation-name>

            <ejb-relationship-role>
                <ejb-relationship-role-name>
                customer-has-orders
                </ejb-relationship-role-name>
                <multiplicity>One</multiplicity>
                <relationship-role-source>
                    <ejb-name>CustomerEJB</ejb-name>
                </relationship-role-source>
                <cmr-field>
                    <cmr-field-name>orders</cmr-field-name>
                    <cmr-field-type>java.util.Collection
                    </cmr-field-type>
                </cmr-field>
            </ejb-relationship-role>

            <ejb-relationship-role>
                <ejb-relationship-role-name>
                order-belongsto-customer
                </ejb-relationship-role-name>
                <multiplicity>Many</multiplicity>
                <relationship-role-source>
                    <ejb-name>OrderEJB</ejb-name>
                </relationship-role-source>
                <cmr-field>
```

```
                    <cmr-field-name>customer</cmr-field-name>
                </cmr-field>
            </ejb-relationship-role>

        </ejb-relation>


    </relationships>

    ...

    </ejb-jar>
```

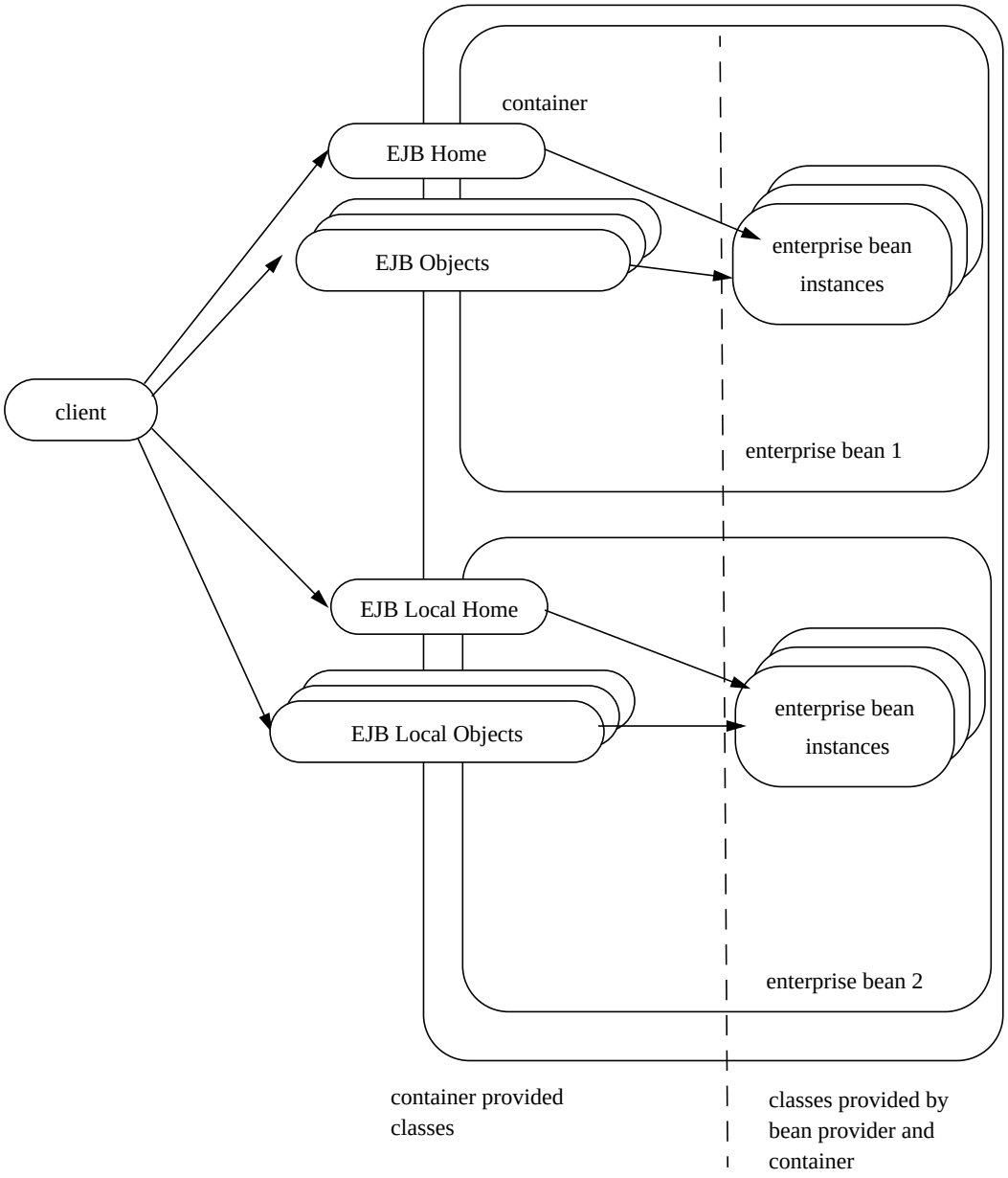
8.4 The Entity Bean Component Contract

This section specifies the container-managed persistence contract between an entity bean and its container.

8.4.1 Runtime Execution Model of Entity Beans

This subsection describes the runtime model and the classes used in the description of the contract between an entity bean and its container. Figure 13 shows an overview of the runtime model. The client of an entity bean may be a local client or it may be a remote client.

Figure 13 Overview of the Entity Bean Runtime Execution Model



An enterprise bean is an object whose class is provided by the Bean Provider. The class of an entity bean with container-managed persistence is abstract. The concrete bean class is generated by the Container Provider's tools at deployment time. The container is also responsible for providing the implementation of the `java.util.Collection` classes that are used in maintaining the container-managed relationships of the entity bean.

An entity **EJBObject** or **EJBLocalObject** is an object whose class was generated at deployment time by the Container Provider's tools. A client never references an entity bean instance directly—a client always references an entity EJBObject or EJBLocalObject whose class is generated by the Container Provider's tools. The entity EJBObject class implements an entity bean's remote interface. The entity EJBLocalObject class implements an entity bean's local interface. A related entity bean never references another entity bean instance directly—a related entity bean, like any other local client of an entity bean, always references an entity EJBLocalObject whose class is generated by the Container Provider's tools.

An entity **EJBHome** or **EJBLocalHome** object provides life cycle operations (create, find, remove) for its entity objects as well as home business methods, which are business methods that are not specific to an entity bean instance. The class for the entity EJBHome or EJBLocalHome object is generated by the Container Provider's tools at deployment time. The entity EJBHome or EJBLocalHome object implements the entity bean's remote or local home interface that was defined by the Bean Provider.

8.4.2 Container Responsibilities

The following are the container responsibilities for the management of persistent state.

8.4.2.1 Container-Managed Fields

An entity bean with container-managed persistence relies on the container to perform persistent data access on behalf of the entity bean instances. The container transfers data between an entity bean instance and the underlying resource manager. The container also implements the creation, removal, and lookup of the entity object in the underlying database.

The container transfers data between the entity bean and the underlying data source as a result of the execution of the entity bean's methods. Because of the requirement that all data access occur through the accessor methods, the container can implement both eager and lazy loading and storing schemes.

The container is responsible for implementing the entity bean class by providing the implementation of the get and set accessor methods for its abstract persistence schema. The container is allowed to use Java serialization to store the container-managed persistent fields (cmp-fields).

The container must also manage the mapping between primary keys and EJBLocalObjects or EJBObjects. If both a remote and a local interface are specified for the entity bean, the container must manage the mapping between EJBObjects and EJBLocalObjects.

Because the container is free to optimize the delivery of persistent data to the bean instance (for example, by the use of lazy loading strategies), the contents of the entity bean instance and the contents of container-managed collections may not be fully materialized.

8.4.2.2 Container-Managed Relationships

The container maintains the relationships among entity beans.

- It is the responsibility of the container to maintain the referential integrity of the container-managed relationships, as described in Section 8.3.6, in accordance with the semantics of the relationship type as specified in the deployment descriptor. For example, if an entity

bean is added to a collection corresponding to the container-managed relationship field of another entity bean, the container-managed relationship field of the first entity bean must also be updated by the container in the same transaction context.

- It is the responsibility of the container to throw the `java.lang.IllegalArgumentException` when the argument to a set method in a relationship is an instance of the wrong relationship type or a collection containing instances of the wrong type, or when an argument to a method of the `java.util.Collection` API used to manipulate a collection-valued container-managed relationship field is an instance of the wrong type or a collection that contains instances of the wrong type (see Section 8.3.6).

- It is the responsibility of the container to throw the `java.lang.IllegalStateException` when a method of the `java.util.Collection` API is used to access a collection-valued cmr-field within a transaction context other than the transaction context in which the cmr-field was initially materialized. For example, if the container-managed collection is returned as the result of a local interface method with transaction attribute `RequiresNew`, and the client attempts to access the collection, the container must throw the `IllegalStateException`.

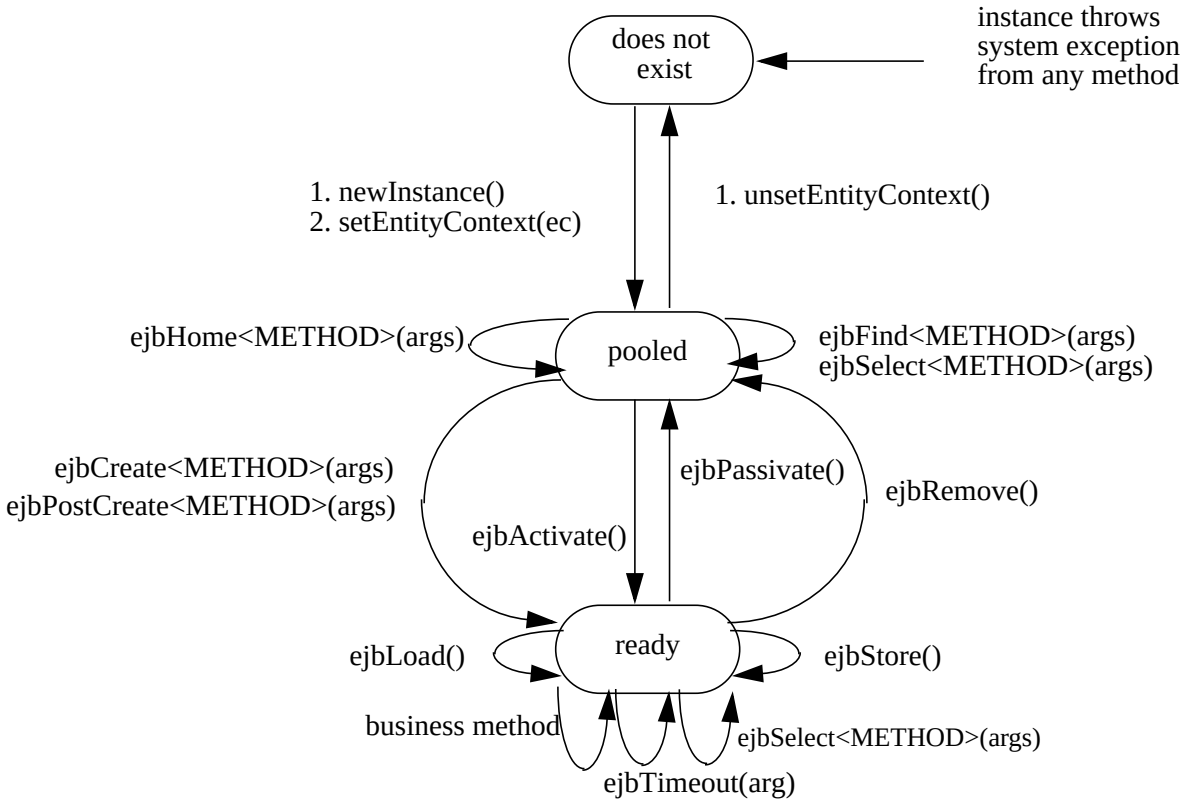
- It is the responsibility of the container to throw the `java.lang.IllegalStateException` when a `java.util.Iterator` is used to access a collection-valued cmr-field within a transaction context other than the transaction context in which the iterator was initially obtained.

8.5 Instance Life Cycle Contract Between the Bean and the Container

This section describes the part of the component contract between the entity bean and the container that relates to the management of the entity bean instance's life cycle.

8.5.1 Instance Life Cycle

Figure 14 Life Cycle of an Entity Bean Instance.



An entity bean instance is in one of the following three states:

- It does not exist.
- Pooled state. An instance in the pooled state is not associated with any particular entity object identity.
- Ready state. An instance in the ready state is assigned an entity object identity.

The following steps describe the life cycle of an entity bean instance:

- An entity bean instance's life starts when the container creates the instance using `newInstance`. The container then invokes the `setEntityContext` method to pass the instance a reference to the `EntityContext` interface. The `EntityContext` interface allows the

instance to invoke services provided by the container and to obtain the information about the caller of a client-invoked method.

- The instance enters the pool of available instances. Each entity bean has its own pool. While the instance is in the available pool, the instance is not associated with any particular entity object identity. All instances in the pool are considered equivalent, and therefore any instance can be assigned by the container to any entity object identity at the transition to the ready state. While the instance is in the pooled state, the container may use the instance to execute any of the entity bean's finder methods (shown as `ejbFind<METHOD>` in the diagram) or any of the entity bean's home methods (shown `ejbHome<METHOD>` in the diagram). The instance does *not* move to the ready state during the execution of a finder or a home method. An `ejbSelect<METHOD>` method may be called by an entity bean's home method while the instance is in the pooled state.

- An instance transitions from the pooled state to the ready state when the container selects that instance to service a client call to an entity object or an `ejbTimeout` method. There are two possible transitions from the pooled to the ready state: through the `ejbCreate<METHOD>` and `ejbPostCreate<METHOD>` methods, or through the `ejbActivate` method. The container invokes the `ejbCreate<METHOD>` and `ejbPostCreate<METHOD>` methods when the instance is assigned to an entity object during entity object creation (i.e., when the client invokes a create method on the entity bean's home object). The container invokes the `ejbActivate` method on an instance when an instance needs to be activated to service an invocation on an existing entity object—this occurs because there is no suitable instance in the ready state to service the client's call or the `ejbTimeout` method.

- When an entity bean instance is in the ready state, the instance is associated with a specific entity object identity. While the instance is in the ready state, the container can synchronize the state of the instance with the state of the entity in the underlying data source whenever it determines the need to, in the process invoking the `ejbLoad` and `ejbStore` methods zero or more times. A business method can be invoked on the instance zero or more times. The `ejbTimeout` method can be invoked on the instance zero or more times. Invocations of the `ejbLoad` and `ejbStore` methods can be arbitrarily mixed with invocations of business methods and `ejbTimeout` method invocations. An `ejbSelect<METHOD>` method can be called by a business method (or `ejbLoad` or `ejbStore` method or `ejbTimeout` method) while the instance is in the ready state.

- The container can choose to passivate an entity bean instance within a transaction. To passivate an instance, the container first invokes the `ejbStore` method to allow the instance to prepare itself for the synchronization of the database state with the instance's state, and then the container invokes the `ejbPassivate` method to return the instance to the pooled state.

- Eventually, the container will transition the instance to the pooled state. There are three possible transitions from the ready to the pooled state: through the `ejbPassivate` method, through the `ejbRemove` method, and because of a transaction rollback for `ejbCreate`, `ejbPostCreate`, or `ejbRemove` (not shown in Figure 14). The container invokes the `ejbPassivate` method when the container wants to disassociate the instance from the entity object identity without removing the entity object. The container invokes the `ejbRemove` method when the container is removing the entity object (i.e., when the client invoked the `remove` method on the entity object's component interface or a `remove` method on the entity bean's home interface). If `ejbCreate`, `ejbPostCreate`, or `ejbRemove` is called

and the transaction rolls back, the container will transition the bean instance to the pooled state.

- When the instance is put back into the pool, it is no longer associated with an entity object identity. The container can assign the instance to any entity object within the same entity bean home.

- The container can remove an instance in the pool by calling the `unsetEntityContext` method on the instance.

Notes:

1. The `EntityContext` interface passed by the container to the instance in the `setEntityContext` method is an interface, not a class that contains static information. For example, the result of the `EntityContext.getPrimaryKey` method might be different each time an instance moves from the pooled state to the ready state, and the result of the `getCallerPrincipal` and `isCallerInRole` methods may be different in each business method.

2. A `RuntimeException` thrown from any method of an entity bean class (including the business methods and the callbacks invoked by the container) results in the transition to the "does not exist" state. The container must not invoke any method on the instance after a `RuntimeException` has been caught. From the caller's perspective, the corresponding entity object continues to exist. The client can continue accessing the entity object through its component interface because the container can use a different entity bean instance to delegate the client's requests. Exception handling is described further in Chapter 13.

3. The container is not required to maintain a pool of instances in the pooled state. The pooling approach is an example of a possible implementation, but it is not the required implementation. Whether the container uses a pool or not has no bearing on the entity bean coding style.

8.5.2 Bean Provider's Entity Bean Instance's View

The following describes the entity bean instance's view of the contract as seen by the Bean Provider:

The entity Bean Provider is responsible for implementing the following methods in the abstract entity bean class:

- A public constructor that takes no arguments.

- `public void setEntityContext(EntityContext ic);`

  A container uses this method to pass a reference to the `EntityContext` interface to the entity bean instance. If the entity bean instance needs to use the `EntityContext` interface during its lifetime, it must remember the `EntityContext` interface in an instance variable.

  This method executes with an unspecified transaction context (Refer to Subsection 12.6.5 for how the container executes methods with an unspecified transaction context). An identity of an entity object is not available during this method. The entity bean must not attempt to access its persistent state and relationships using the accessor methods during this method.

The instance can take advantage of the `setEntityContext()` method to allocate any resources that are to be held by the instance for its lifetime. Such resources cannot be specific to an entity object identity because the instance might be reused during its lifetime to serve multiple entity object identities.

- `public void unsetEntityContext();`

  A container invokes this method before terminating the life of the instance.

  This method executes with an unspecified transaction context. An identity of an entity object is not available during this method. The entity bean must not attempt to access its persistent state and relationships using the accessor methods during this method.

  The instance can take advantage of the `unsetEntityContext` method to free any resources that are held by the instance. (These resources typically had been allocated by the `setEntityContext` method.)

- `public` *`PrimaryKeyClass`* `ejbCreate<METHOD>(...);`

  There are zero[28] or more `ejbCreate<METHOD>` methods, whose signatures match the signatures of the `create<METHOD>` methods of the entity bean's home interface. The container invokes an `ejbCreate<METHOD>` method on an entity bean instance when a client invokes a matching `create<METHOD>` method on the entity bean's home interface.

  The entity Bean Provider's responsibility is to initialize the instance in the `ejbCreate<METHOD>` methods from the input arguments, using the get and set accessor methods, such that when the `ejbCreate<METHOD>` method returns, the persistent representation of the instance can be created. The entity Bean Provider is guaranteed that the values that will be initially returned by the instance's get methods for container-managed fields will be the Java language defaults (e.g. 0 for integer, `null` for pointers), except for collection-valued cmr-fields, which will have the empty collection (or set) as their value. The entity Bean Provider must not attempt to modify the values of cmr-fields in an `ejbCreate<METHOD>` method. This should be done in the `ejbPostCreate<METHOD>` method instead.

  The entity object created by the `ejbCreate<METHOD>` method must have a unique primary key. This means that the primary key must be different from the primary keys of all the existing entity objects within the same home. However, it is legal to reuse the primary key of a previously removed entity object. The implementation of the Bean Provider's `ejbCreate<METHOD>` methods should be coded to return a null.[29]

  An `ejbCreate<METHOD>` method executes in the transaction context determined by the transaction attribute of the matching `create<METHOD>` method. The database insert operations are performed by the container within the same transaction context after the Bean Provider's `ejbCreate<METHOD>` method completes.

- `public void ejbPostCreate<METHOD>(...);`

  For each `ejbCreate<METHOD>` method, there is a matching `ejbPostCreate<METHOD>` method that has the same input parameters but whose return type is `void`. The container invokes the matching `ejbPostCreate<METHOD>` method on an instance

[28] An entity bean has no *`ejbCreate`*`<METHOD>` and *`ejbPostCreate`*`<METHOD>`methods if it does not define any create methods in its home interface. Such an entity bean does not allow its clients to create new EJB objects. The entity bean restricts the clients to accessing entities that were created through direct database inserts.

[29] The above requirement is to allow the creation of an entity bean with bean-managed persistence by subclassing an entity bean with container-managed persistence.

after it invokes the `ejbCreate<METHOD>` method with the same arguments. The instance can discover the primary key by calling `getPrimaryKey` on its entity context object.

The entity object identity is available during the `ejbPostCreate<METHOD>` method. The instance may, for example, obtain the component interface of the associated entity object and pass it to another enterprise bean as a method argument.

The entity Bean Provider may use the `ejbPostCreate<METHOD>` to set the values of cmr-fields to complete the initialization of the entity bean instance.

An `ejbPostCreate<METHOD>` method executes in the same transaction context as the previous `ejbCreate<METHOD>` method.

- `public void ejbActivate();`

  The container invokes this method on the instance when the container picks the instance from the pool and assigns it to a specific entity object identity. The `ejbActivate` method gives the entity bean instance the chance to acquire additional resources that it needs while it is in the ready state.

  This method executes with an unspecified transaction context. The entity bean must not attempt to access its persistent state or relationships using the accessor methods during this method.

  The instance can obtain the identity of the entity object via the `getPrimaryKey`, `getEJBLocalObject`, or `getEJBObject` method on the entity context. The instance can rely on the fact that the primary key and entity object identity will remain associated with the instance until the completion of `ejbPassivate` or `ejbRemove`.

- `public void ejbPassivate();`

  The container invokes this method on an instance when the container decides to disassociate the instance from an entity object identity, and to put the instance back into the pool of available instances. The `ejbPassivate` method gives the instance the chance to release any resources that should not be held while the instance is in the pool. (These resources typically had been allocated during the `ejbActivate` method.)

  This method executes with an unspecified transaction context. The entity bean must not attempt to access its persistent state or relationships using the accessor methods during this method.

  The instance can still obtain the identity of the entity object via the `getPrimaryKey`, `getEJBLocalObject`, or `getEJBObject` method of the `EntityContext` interface.

- `public void ejbRemove();`

  The container invokes the `ejbRemove` method on an entity bean instance in response to a client-invoked `remove` operation on the entity bean's home or component interface or as the result of a cascade-delete operation. The instance is in the ready state when `ejbRemove` is invoked and it will be entered into the pool when the method completes.

  The entity Bean Provider can use the `ejbRemove` method to implement any actions that must be done before the entity object's persistent representation is removed.

  The container synchronizes the instance's state before it invokes the `ejbRemove` method. This means that the state of the instance at the beginning of the `ejbRemove` method is the same as it would be at the beginning of a business method.

  This method and the database delete operation(s) execute in the transaction context determined by the transaction attribute of the `remove` method that triggered the `ejbRemove` method.

The instance can still obtain the identity of the entity object via the `getPrimaryKey`, `getEJBLocalObject`, or `getEJBObject` method of the `EntityContext` interface.

After the entity Bean Provider's `ejbRemove` returns, and in the same transaction context, the container removes the entity bean from all relationships in which it participates before removing the entity object's persistent representation.

Since the instance will be entered into the pool, the state of the instance at the end of this method must be equivalent to the state of a passivated instance. This means that the instance must release any resource that it would normally release in the `ejbPassivate` method.

- `public void ejbLoad();`

  When the container needs to synchronize the state of an enterprise bean instance with the entity object's persistent state, the container calls the `ejbLoad` method.

  The entity Bean Provider can assume that the instance's persistent state has been loaded just before the `ejbLoad` method is invoked. It is the responsibility of the Bean Provider to use the `ejbLoad` method to recompute or initialize the values of any instance variables that depend on the entity bean's persistent state. In general, any transient state that depends on the persistent state of an entity bean should be recalculated using the `ejbLoad` method. The entity bean can use the `ejbLoad` method, for instance, to perform some computation on the values returned by the accessor methods (for example, uncompressing text fields).

  This method executes in the transaction context determined by the transaction attribute of the business method or `ejbTimeout` method that triggered the `ejbLoad` method.

- `public void ejbStore();`

  When the container needs to synchronize the state of the entity object's persistent state with the state of the enterprise bean instance, the container first calls the `ejbStore` method on the instance.

  The entity Bean Provider should use the `ejbStore` method to update the instance using the accessor methods before its persistent state is synchronized. For example, the `ejbStore` method may perform compression of text before the text is stored in the database.

  The Bean Provider can assume that after the `ejbStore` method returns, the persistent state of the instance is synchronized.

  This method executes in the same transaction context as the previous `ejbLoad` or `ejbCreate` method invoked on the instance. All business methods or the `ejbTimeout` method invoked between the previous `ejbLoad` or `ejbCreate<METHOD>` method and this `ejbStore` method are also invoked in the same transaction context.

- `public` <primary key type or collection> `ejbFind<METHOD>(...);`

  The Bean Provider of an entity bean with container-managed persistence does not write the finder (`ejbFind<METHOD>`) methods.

  The finder methods are generated at the entity bean deployment time using the Container Provider's tools. The syntax for the Bean Provider's specification of finder methods is described in the document "*Java Persistence*" of this specification [2].

- `public` <type> `ejbHome<METHOD>(...);`

  The container invokes this method on the instance when the container selects the instance to execute a matching client-invoked `<METHOD>` home method. The instance is in the pooled state (i.e., it is not assigned to any particular entity object identity) when the container selects

the instance to execute the `ejbHome<METHOD>` method on it, and it is returned to the pooled state when the execution of the `ejbHome<METHOD>` method completes.

The `ejbHome<METHOD>` method executes in the transaction context determined by the transaction attribute of the matching `<METHOD>` home method, as described in Section 12.6.2.

The entity Bean Provider provides the implementation of the `ejbHome<METHOD>` method. The entity bean must not attempt to access its persistent state or relationships using the accessor methods during this method because a home method is not specific to a particular bean instance.

- `public abstract` <type> `ejbSelect<METHOD>(...);`

  The Bean Provider may provide zero or more select methods. A select method is a query method that is not directly exposed to the client in the home or component interface. The Bean Provider typically calls a select method within a business method.

  The Bean Provider defines the select methods as `abstract` methods.

  The select methods are generated at the entity bean deployment time using the Container Provider's tools.

  The syntax for the specification of select methods is described in the document "*Java Persistence*" of this specification [2].

  The `ejbSelect<METHOD>` method executes in the transaction context determined by the transaction attribute of the invoking business method.

- `public void ejbTimeout(...);`

  The container invokes the `ejbTimeout` method on an instance when a timer for the instance has expired. The `ejbTimeout` method notifies the instance of the time-based event and allows the instance to execute the business logic to handle it.

  The `ejbTimeout` method executes in the transaction context determined by its transaction attribute.

8.5.3 Container's View

This subsection describes the container's view of the state management contract. The container must call the following methods:

- `public void setEntityContext(ec);`

  The container invokes this method to pass a reference to the `EntityContext` interface to the entity bean instance. The container must invoke this method after it creates the instance, and before it puts the instance into the pool of available instances.

  The container invokes this method with an unspecified transaction context. At this point, the `EntityContext` is not associated with any entity object identity.

- `public void unsetEntityContext();`

  The container invokes this method when the container wants to reduce the number of instances in the pool. After this method completes, the container must not reuse this instance.

  The container invokes this method with an unspecified transaction context.

- `public` *`PrimaryKeyClass`* `ejbCreate<METHOD>(...);`
  `public void ejbPostCreate<METHOD>(...);`

  The container invokes these two methods during the creation of an entity object as a result of a client invoking a `create<METHOD>` method on the entity bean's home interface.

  The container invokes the `ejbCreate<METHOD>` method whose signature matches the `create<METHOD>` method invoked by the client.

  Prior to invoking the `ejbCreate<METHOD>` method provided by the Bean Provider, the container must ensure that the values that will be initially returned by the instance's get methods for container-managed fields will be the Java language defaults (e.g. 0 for integer, `null` for pointers), except for collection-valued cmr-fields, which must have the empty collection (or set) as their value.

  The container is responsible for calling the `ejbCreate<METHOD>` method, for obtaining the primary key fields of the newly created entity object persistent representation, and for creating an entity EJBObject reference and/or EJBLocalObject reference for the newly created entity object. The container must establish the primary key before it invokes the `ejbPostCreate<METHOD>` method.

  The entity object created by the `ejbCreate<METHOD>` method must have a unique primary key. This means that the primary key must be different from the primary keys of all the existing entity objects within the same home. However, it is legal to reuse the primary key of a previously removed entity object. The container may, but is not required to, throw the `DuplicateKeyException` on the Bean Provider's attempt to create an entity object with a duplicate primary key[30].

  The container may create the representation of the entity in the database immediately, or it can defer it to a later time (for example to the time after the matching `ejbPostCreate<METHOD>` has been called, or to the end of the transaction), depending on the caching strategy that it uses.

  The container then invokes the matching `ejbPostCreate<METHOD>` method with the same arguments on the instance to allow the instance to fully initialize itself. The instance can discover the primary key by calling the `getPrimaryKey` method on its entity context object.

  Finally, the container returns the entity object's remote interface (i.e., a reference to the entity EJBObject) to the client if the client is a remote client or the entity object's local interface (i.e., a reference to the entity EJBLocalObject) if the client is a local client.

  The container must invoke the `ejbCreate<METHOD>` and ej`bPostCreate<METHOD>` methods and create the representation of the persistent instance in the database in the transaction context determined by the transaction attribute of the matching `create<METHOD>` method, as described in subsection 12.6.2.

- `public void ejbActivate();`

  The container invokes this method on an entity bean instance at activation time (i.e., when the instance is taken from the pool and assigned to an entity object identity). The container must ensure that the primary key of the associated entity object is available to the instance if the instance invokes the `getPrimaryKey`, `getEJBLocalObject`, or `getEJBObject` method on its `EntityContext` interface.

  The container invokes this method with an unspecified transaction context.

[30] Containers using optimistic caching strategies, for example, may rollback the transaction at a later point.

Note that instance is not yet ready for the delivery of a business method. The container must still invoke the `ejbLoad` method prior to a business method.

- `public void ejbPassivate();`

  The container invokes this method on an entity bean instance at passivation time (i.e., when the instance is being disassociated from an entity object identity and moved into the pool). The container must ensure that the identity of the associated entity object is still available to the instance if the instance invokes the `getPrimaryKey`, `getEJBLocalObject`, or `getEJBObject` method on its entity context.

  The container invokes this method with an unspecified transaction context.

  Note that if the instance state has been updated by a transaction, the container must first invoke the `ejbStore` method on the instance before it invokes `ejbPassivate` on it.

- `public void ejbRemove();`

  The container invokes the `ejbRemove` method in response to a client-invoked `remove` operation on the entity bean's home or component interface or as the result of a cascade-delete operation. The instance is in the ready state when `ejbRemove` is invoked and it will be entered into the pool when the method completes.

  The container synchronizes the instance's state before it invokes the `ejbRemove` method. This means that the persistent state of the instance at the beginning of the `ejbRemove` method is the same as it would be at the beginning of a business method (i.e., if the instance is not already synchronized from the state in the database, the container must invoke `ejbLoad` before it invokes `ejbRemove`).

  The container must ensure that the identity of the associated entity object is still available to the instance in the `ejbRemove` method (i.e., the instance can invoke the `getPrimaryKey`, `getEJBLocalObject`, or `getEJBObject` method on its `EntityContext` in the `ejbRemove` method).

  After the entity Bean Provider's `ejbRemove` method returns, and in the same transaction context, the container removes the entity bean instance from all relationships in which it participates and then removes the entity object's persistent representation.

  The container may delete the representation of the entity in the database immediately, or it can defer it to a later time (for example to the end of the transaction), depending on the caching strategy that it uses.

  The container must ensure that the `ejbRemove` method and database delete operations are performed in the transaction context determined by the transaction attribute of the invoked `remove` method, as described in subsection 12.6.2.

- `public void ejbLoad();`

  When the container needs to synchronize the state of an enterprise bean instance with the entity object's state in the database, the container calls the `ejbLoad` method. Depending on its caching strategy, the container may first read the entity object's state from the database, before invoking the `ejbLoad` method, or it may use a lazy loading strategy in making this state visible to the instance.

  The exact times that the container invokes `ejbLoad` depend on the configuration of the component and the container, and are not defined by the EJB architecture. Typically, the container will call `ejbLoad` before the first business method within a transaction or before invoking the `ejbTimeout` method on an instance.

The container must invoke this method in the transaction context determined by the transaction attribute of the business method or `ejbTimeout` method that triggered the `ejbLoad` method.

- `public void ejbStore();`

  When the container needs to synchronize the state of the entity object in the database with the state of the enterprise bean instance, the container calls the `ejbStore` method on the instance. This synchronization always happens at the end of a transaction, unless the bean is specified as read-only (see section 8.5.4). However, the container may also invoke this method when it passivates the instance in the middle of a transaction, or when it needs to transfer the most recent state of the entity object to another instance for the same entity object in the same transaction.

  The container must invoke this method in the same transaction context as the previous `ejbLoad`, `ejbCreate<METHOD>`, or `ejbTimeout` method invoked on the instance. All business methods or the `ejbTimeout` method invoked between the previous `ejbLoad` or `ejbCreate <METHOD>` method and this `ejbStore` method are also invoked in the same transaction context.

  After the `ejbStore` method returns, the container may store the persistent state of the instance to the database, depending on its caching strategy. If the container uses a lazy storing caching strategy, it is the container's responsibility to write the representation of the persistent object to the database in the same transaction context as that of the `ejbStore` method.

- `public` <primary key type or collection> `ejbFind<METHOD>(...);`

  The implementation of the `ejbFind<METHOD>` method is supplied by the container.

  The container invokes the `ejbFind<METHOD>` method on an instance when a client invokes a matching `find<METHOD>` method on the entity bean's home interface. The container must pick an instance that is in the pooled state (i.e., the instance is not associated with any entity object identity) for the execution of the `ejbFind<METHOD>` method. If there is no instance in the pooled state, the container creates one and calls the `setEntityContext` method on the instance before dispatching the finder method.

  The container must invoke the `ejbFind<METHOD>` method in the transaction context determined by the transaction attribute of the matching `find` method, as described in subsection 12.6.2.

  The container is responsible for ensuring that updates to the states of all entity beans in the same transaction context as the `ejbFind<METHOD>` method and whose abstract schema types are accessed in the method's EJB QL query are visible in the results of the `ejbFind<METHOD>` method. Before invoking the `ejbFind<METHOD>` method, the container must first synchronize the state of those entity bean instances by invoking the `ejbStore` method on them. This requirement does not apply to the `ejbFindByPrimaryKey` method. The results of the `ejbFindByPrimaryKey` method, however, must reflect the entities that have been created or removed within the same transaction context.

  After the `ejbFind<METHOD>` method completes, the instance remains in the pooled state. The container may, but is not required to, immediately activate the objects that were located by the finder using the transition through the `ejbActivate` method.

  If the `ejbFind<METHOD>` method is declared to return a single primary key, the container creates an entity EJBObject (EJBLocalObject) reference for the primary key and returns it to the client (local client). If the `ejbFind<METHOD>` method is declared to return a collection

of primary keys, the container creates a collection of entity EJBObject (EJBLocalObject) references for the primary keys returned from the `ejbFind<METHOD>` method, and returns the collection to the client (local client).

The implementations of the finder methods are generated at the entity bean deployment time using the Container Provider's tools.

- `public` <type> `ejbSelect<METHOD>(...);`

  A select method is a query method that is not directly exposed to the client in the home or component interface. The Bean Provider typically calls a select method within a business method or home method.

  A select method executes in the transaction context determined by the transaction attribute of the invoking business method.

  The container is responsible for ensuring that all updates to the states of all entity beans in the same transaction context as the `ejbSelect<METHOD>` method and whose abstract schema types are accessed in the EJB QL query for the `ejbSelect<METHOD>` method are visible in the results of the `ejbSelect<METHOD>` method by invoking the `ejbStore` method on those entity bean instances.

  The implementations of the select methods are generated at the entity bean deployment time using the Container Provider's tools.

- `public` <type> `ejbHome<METHOD>(...);`

  The container invokes the `ejbHome<METHOD>` method on an instance when a client invokes a matching `<METHOD>` home method on the entity bean's home interface. The container must pick an instance that is in the pooled state (i.e., the instance is not associated with any entity object identity) for the execution of the `ejbHome<METHOD>` method. If there is no instance in the pooled state, the container creates one and calls the `setEntityContext` method on the instance before dispatching the home method.

  After the `ejbHome<METHOD>` method completes, the instance remains in the pooled state.

  The container must invoke the `ejbHome<METHOD>` method in the transaction context determined by the transaction attribute of the matching `<METHOD>` home method, as described in subsection 12.6.2.

- `public void ejbTimeout(...);`

  The container invokes the `ejbTimeout` method on the instance when a timer with which the entity has been registered expires. If there is no suitable instance in the ready state, the container must activate an instance, invoking the `ejbActivate` method and transitioning it to the ready state.

  The container invokes the `ejbTimeout` method in the context of a transaction determined by its transaction attribute.

8.5.4 Read-only Entity Beans

Compliant implementations of this specification may optionally support read-only entity beans. A read-only entity bean is an entity bean whose instances are not intended to be updated and/or created by the application. Read-only beans are best suited for situations where the underlying data never changes or changes infrequently.

Containers that support read-only beans do not call the `ejbStore` method on them. The `ejbLoad` method should typically be called by the container when the state of the bean instance is initially loaded from the database, or at designated refresh intervals.[31]

If a read-only bean is used, the state of such a bean should not be updated by the application, and the behavior is unspecified if this occurs.[32]

Read-only beans are designated by vendor-specific means that are outside the scope of this specification, and their use is therefore not portable.

8.5.5 The EntityContext Interface

A container provides the entity bean instances with an `EntityContext`, which gives the entity bean instance access to the instance's context maintained by the container. The `EntityContext` interface has the following methods:

- The `getEJBObject` method returns the entity bean's remote interface.
- The `getEJBHome` method returns the entity bean's remote home interface.
- The `getEJBLocalObject` method returns the entity bean's local interface.
- The `getEJBLocalHome` method returns the entity bean's local home interface.
- The `getCallerPrincipal` method returns the `java.security.Principal` that identifies the invoker.
- The `isCallerInRole` method tests if the entity bean instance's caller has a particular role.
- The `setRollbackOnly` method allows the instance to mark the current transaction such that the only outcome of the transaction is a rollback.
- The `getRollbackOnly` method allows the instance to test if the current transaction has been marked for rollback.
- The `getPrimaryKey` method returns the entity bean's primary key.
- The `getTimerService` method returns the `javax.ejb.TimerService` interface.
- The `getUserTransaction` method returns the `javax.transaction.UserTransaction` interface. Entity bean instances must not call this method.

[31] The ability to refresh the state of a read-only bean and the intervals at which such refresh occurs are vendor-specific.

[32] For example, an implementation might choose to ignore such updates or to disallow them.

- The `lookup` method enables the entity bean to look up its environment entries in the JNDI naming context.

8.5.6 Operations Allowed in the Methods of the Entity Bean Class

Table 4 defines the methods of an entity bean class in which the enterprise bean instances can access the methods of the `javax.ejb.EntityContext` interface, the `java:comp/env` environment naming context, resource managers, `TimerService` and `Timer` methods, and other enterprise beans.

If an entity bean instance attempts to invoke a method of the `EntityContext` interface, and the access is not allowed in Table 4, the container must throw the `java.lang.IllegalStateException.`

If a entity bean instance attempts to invoke a method of the `TimerService` or `Timer` interface and the access is not allowed in Table 4, the container must throw the `java.lang.IllegalStateException.`

If an entity bean instance attempts to access a resource manager or an enterprise bean, and the access is not allowed in Table 4, the behavior is undefined by the EJB architecture.

Table 4 Operations Allowed in the Methods of an Entity Bean

Bean method	Bean method can perform the following operations
constructor	-
setEntityContext **unsetEntityContext**	**EntityContext methods:** ***getEJBHome, getEJBLocalHome, lookup*** **JNDI access to java:comp/env**
ejbCreate	**EntityContext methods:** ***getEJBHome, getEJBLocalHome, getCallerPrincipal, getRollbackOnly, isCallerInRole, setRollbackOnly, getTimerService, lookup*** **JNDI access to java:comp/env** **Resource manager access** **Enterprise bean access**
ejbPostCreate	**EntityContext methods:** ***getEJBHome, getEJBLocalHome, getCallerPrincipal, getRollbackOnly, isCallerInRole, setRollbackOnly, getEJBObject, getEJBLocalObject, getPrimaryKey, getTimerService, lookup*** **JNDI access to java:comp/env** **Resource manager access** **Enterprise bean access** **Timer service or Timer methods**

Table 4 Operations Allowed in the Methods of an Entity Bean

Bean method	Bean method can perform the following operations
ejbRemove	**EntityContext methods: *getEJBHome, getEJBLocalHome, getCallerPrincipal, getRollbackOnly, isCallerInRole, setRollbackOnly, getEJBObject, getEJBLocalObject, getPrimaryKey, getTimerService, lookup*** **JNDI access to java:comp/env** **Resource manager access** **Enterprise bean access** **Timer service or Timer methods**
ejbHome	**EntityContext methods: *getEJBHome, getEJBLocalHome, getCallerPrincipal, getRollbackOnly, isCallerInRole, setRollbackOnly, getTimerService, lookup*** **JNDI access to java:comp/env** **Resource manager access** **Enterprise bean access**
ejbActivate **ejbPassivate**	**EntityContext methods: *getEJBHome, getEJBLocalHome, getEJBObject, getEJBLocalObject, getPrimaryKey, getTimerService, lookup*** **JNDI access to java:comp/env**
ejbLoad **ejbStore**	**EntityContext methods: *getEJBHome, getEJBLocalHome, getCallerPrincipal, getRollbackOnly, isCallerInRole, setRollbackOnly, getEJBObject, getEJBLocalObject, getPrimaryKey, getTimerService, lookup*** **JNDI access to java:comp/env** **Resource manager access** **Enterprise bean access** **Timer service or Timer methods**
business method from component interface	**EntityContext methods: *getEJBHome, getEJBLocalHome, getCallerPrincipal, getRollbackOnly, isCallerInRole, setRollbackOnly, getEJBObject, getEJBLocalObject, getPrimaryKey, getTimerService, lookup*** **JNDI access to java:comp/env** **Resource manager access** **Enterprise bean access** **Timer service or Timer methods**
ejbTimeout	**EntityContext methods: *getEJBHome, getEJBLocalHome, getRollbackOnly, setRollbackOnly, getCallerPrincipal, isCallerInRole, getEJBObject, getEJBLocalObject, getPrimaryKey, getTimerService, lookup*** **JNDI access to java:comp/env** **Resource manager access** **Enterprise bean access** **Timer service or Timer methods**

Additional restrictions:

- The `getRollbackOnly` and `setRollbackOnly` methods of the `EntityContext` interface should be used only in the enterprise bean methods that execute in the context of a transaction. The container must throw the `java.lang.IllegalStateException` if the methods are invoked while the instance is not associated with a transaction.

Reasons for disallowing operations:

- Invoking the `getEJBObject`, `getEJBLocalObject`, and `getPrimaryKey` methods is disallowed in the entity bean methods in which there is no entity object identity associated with the instance.
- Invoking the `getEJBObject` and `getEJBHome` methods is disallowed if the entity bean does not define a remote client view.
- Invoking the `getEJBLocalObject` and `getEJBLocalHome` methods is disallowed if the entity bean does not define a local client view.
- Invoking the `getRollbackOnly` and `setRollbackOnly` methods is disallowed in the entity bean methods for which the container does not have a meaningful transaction context.
- Accessing resource managers and enterprise beans, including accessing the persistent state of an entity bean instance, is disallowed in the entity bean methods for which the container does not have a meaningful transaction context or client security context.

8.5.7 Finder Methods

An entity bean's home interface defines one or more `finder` methods[33], one for each way to find an entity object or collection of entity objects within the home. The name of each finder method starts with the prefix "`find`", such as `findLargeAccounts`. The arguments of a finder method are used in the implementation of the query for the finder method to locate the requested entity objects.

Every finder method except `findByPrimaryKey(key)` must be associated with a `query` element in the deployment descriptor. The entity Bean Provider declaratively specifies the EJB QL finder query and associates it with the finder method in the deployment descriptor. A finder method is normally characterized by an EJB QL query string specified in the `query` element. EJB QL is described in the document "*Java Persistence*" of this specification [2]. A compliant implementation of this specification is required to support only that subset of EJB QL defined in the Enterprise JavaBeans 2.1 specification [3] for use with finder methods.

In the case that both the remote home interface and local home interface define a finder method with the same name and argument types, the EJB QL query string specified by the `query` element defines the semantics of both methods.

8.5.7.1 Single-Object Finder Methods

Some finder methods (such as `findByPrimaryKey`) are designed to return at most one entity object. For single-object finders, the result type of a `find<METHOD>`method defined in the entity bean's remote home interface is the entity bean's remote interface, and the result type of the `find<METHOD>`method defined in the entity bean's local home interface is the entity bean's local interface.

[33] The *`findByPrimaryKey`* method is mandatory for all entity beans.

The following code illustrates the definition of a single-object finder defined on the remote home interface.

```
// Entity's home interface
public interface AccountHome extends javax.ejb.EJBHome {
     ...
     Account findByPrimaryKey(AccountPrimaryKey primkey)
          throws FinderException, RemoteException;
     ...
}
```

Note that a finder method defined on the local home interface must not throw the `RemoteException`.

In general, when defining a single-object finder method other than `findByPrimaryKey`, the entity Bean Provider should be sure that the finder method will always return only a single entity object. This may occur, for example, if the EJB QL query string that is used to specify the finder query includes an equality test on the entity bean's primary key fields. If the entity Bean Provider uses an unknown primary key class (see Section 8.8.3), the Bean Provider will typically define the finder method as a multi-object finder.

Note that a single-object finder method may return a null value. If the result set of the query consists of a single null value, the container must return the null value as the result of the method. If the result set of a query for a single-object finder method contains more than one value (whether non-null, null, or a combination), the container must throw the `FinderException` from the finder method. If the result set of the query contains no values, the container must throw the `ObjectNotFoundException`.

8.5.7.2 Multi-Object Finder Methods

Some finder methods are designed to return multiple entity objects. For multi-object finders defined on the entity bean's local home interface, the result type of the `find<METHOD>`method is a collection of objects implementing the entity bean's local interface. For multi-object finders defined on the entity bean's remote home interface, the result type of the `find<METHOD>`method is a collection of objects implementing the entity bean's remote interface.

The Bean Provider uses the Java™ 2 `java.util.Collection` interface to define a collection type for the result type of a finder method for an entity bean with container-managed persistence.

The collection of values returned by the container may contain duplicates if DISTINCT is not specified in the SELECT clause of the query for the finder method.

The collection of values returned by the container may contain null values if the finder method returns the values of a cmr-field and null values are not eliminated by the query.

A portable client program must use the `PortableRemoteObject.narrow` method to convert the objects contained in the collections returned by a finder method on the entity bean's remote home interface to the entity bean's remote interface type.

The following is an example of a multi-object finder method defined on the remote home interface:

```
// Entity's home interface
public interface AccountHome extends javax.ejb.EJBHome {
     ...
     java.util.Collection findLargeAccounts(double limit)
          throws FinderException, RemoteException;
     ...
}
```

Note that if this finder method were defined on the local home interface, it would not throw the `RemoteException`.

8.5.8 Select Methods

Select methods are query methods for use by the Bean Provider within an entity bean instance. Unlike finder methods, select methods are not specified in the entity bean's home interface. A select method is an abstract method defined by the Bean Provider on an entity bean class. A select method must not be exposed in the home or component interface of an entity bean.

The semantics of a select method, like those of a finder method, are defined by an EJB QL query string. A select method is similar to a finder method, but unlike a finder method, but it can return values that correspond to any cmp- or cmr-field type.

Every select method must be associated with a `query` element in the deployment descriptor. The entity Bean Provider declaratively specifies the EJB QL query and associates it with the select method in the deployment descriptor. A select method is normally characterized by an EJB QL query string specified in the `query` element. EJB QL is described in the document "*Java Persistence*" of this specification [2]. A compliant implementation of this specification is required to support only that subset of EJB QL defined in the Enterprise JavaBeans 2.1 specification [3] for use with select methods.

Typically an `ejbSelect<METHOD>`method that returns entity objects returns these as EJBLocalObjects. If the `ejbSelect<METHOD>`method returns an EJBObject or collection of EJBObjects, the Bean Provider must specify the value of the `result-type-mapping` element in the `query` deployment descriptor element for the select method as `Remote`.

An `ejbSelect<METHOD>` is not based on the identity of the entity bean instance on which it is invoked. However, the Bean Provider can use the primary key of an entity bean as an argument to an `ejbSelect<METHOD>` to define a query that is logically scoped to a particular entity bean instance.

The following table illustrates the semantics of finder and select methods.

Table 5 Comparison of Finder and Select Methods

	Finder methods	**Select methods**
method	`find<METHOD>`	`ejbSelect<METHOD>`
visibility	exposed to client	internal to entity bean class
instance	arbitrary bean instance in pooled state	instance: current instance (could be bean instance in pooled state or ready state)
return value	EJBObjects or EJBLocalObjects of the same type as the entity bean	EJBObjects, EJBLocalObjects, or cmp-field types

8.5.8.1 Single-Object Select Methods

Some select methods are designed to return at most one value. In general, when defining a single-object select method, the entity Bean Provider must be sure that the select method will always return only a single object or value. If the query specified by the select method returns multiple values of the designated type, the container must throw the `FinderException`.

Note that a single-object select method may return a null value. If the result set of the query consists of a single null value, the container must return the null value as the result of the method. If the result set of a query for a single-object select method contains more than one value (whether non-null, null, or a combination), the container must throw the `FinderException` from the select method. If the result set of the query contains no values, the contain must throw the `ObjectNotFoundException`.

The Bean Provider will typically define a select method as a multi-object select method.

8.5.8.2 Multi-Object Select Methods

Some select methods are designed to return multiple values. For these multi-object select methods, the result type of the `ejbSelect<METHOD>`method is a collection of objects.

The Bean Provider uses the Java™ 2 `java.util.Collection` interface or `java.util.Set` interface to define a collection type for the result type of a select method. The type of the elements of the collection is determined by the type of the SELECT clause of the corresponding EJB QL query. If the Bean Provider uses the `java.util.Collection` interface, the collection of values returned by the container may contain duplicates if DISTINCT is not specified in the SELECT clause of the query. If a query for a select method whose result type is `java.util.Set` does not specify DISTINCT in its SELECT clause, the container must interpret the query as if SELECT DISTINCT had been specified.

The collection of values returned by the container may contain null values if the select method returns the values of a cmr-field or cmp-field and null values are not eliminated by the query.

The following is an example of a multi-object select method definition in the `OrderBean` class:

```
// OrderBean implementation class
public abstract class OrderBean implements javax.ejb.EntityBean{
    ...
    public abstract java.util.Collection
                ejbSelectAllOrderedProducts(Customer customer)
        throws FinderException;
        // internal finder method to find all products ordered
```

8.5.9 Timer Notifications

An entity bean can be registered with the EJB timer service for time-based event notifications if it implements the `javax.ejb.TimedObject` interface. The container invokes the bean instance's `ejbTimeout` method when a timer for the bean has expired. See Chapter 17, "Timer Service".

8.5.10 Standard Application Exceptions for Entities

The EJB specification defines the following standard application exceptions:

- `javax.ejb.CreateException`
- `javax.ejb.DuplicateKeyException`
- `javax.ejb.FinderException`
- `javax.ejb.ObjectNotFoundException`
- `javax.ejb.RemoveException`

This section describes the use of these exceptions by entity beans with container-managed persistence.

8.5.10.1 CreateException

From the client's perspective, a `CreateException` (or a subclass of `CreateException`) indicates that an application level error occurred during a `create<METHOD>` operation. If a client receives this exception, the client does not know, in general, whether the entity object was created but not fully initialized, or not created at all. Also, the client does not know whether or not the transaction has been marked for rollback. (However, the client may determine the transaction status using the `UserTransaction` interface or the `setRollbackOnly` method of the `EJBContext` interface.)

Both the container and the Bean Provider may throw the `CreateException` (or subclass of `CreateException`) from the `create<METHOD>`, `ejbCreate<METHOD>` and `ejbPostCreate<METHOD>` methods to indicate an application-level error from the create or initialization operation. Optionally, the container or Bean Provider may mark the transaction for rollback before throwing this exception.

The container or Bean Provider is encouraged to mark the transaction for rollback only if data integrity would be lost if the transaction were committed by the client. Typically, when a `CreateException` *is thrown, it leaves the database in a consistent state, allowing the client to recover. For example, the* `ejbCreate<METHOD>` *method may throw the* `CreateException` *to indicate that the some of the arguments to the* `create<METHOD>` *method are invalid.*

The container treats the `CreateException` as any other application exception. See Section 13.3.

8.5.10.2 DuplicateKeyException

The `DuplicateKeyException` is a subclass of `CreateException`. It may be thrown by the container to indicate to the client or local client that the entity object cannot be created because an entity object with the same key already exists. The unique key causing the violation may be the primary key, or another key defined in the underlying database.

Normally, the container should not mark the transaction for rollback before throwing the exception.

When the client or local client receives a `DuplicateKeyException`, the client knows that the entity was not created, and that the transaction has not typically been marked for rollback.

8.5.10.3 FinderException

From the client's perspective, a `FinderException` (or a subclass of `FinderException`) indicates that an application level error occurred during the `find` operation. Typically, the transaction has not been marked for rollback because of the `FinderException`.

The container throws the `FinderException` (or subclass of `FinderException`) from the implementation of a finder or select method to indicate an application-level error in the finder or select method. The container should not, typically, mark the transaction for rollback before throwing the `FinderException`.

The container treats the `FinderException` as any other application exception. See Section 13.3.

8.5.10.4 ObjectNotFoundException

The `ObjectNotFoundException` is a subclass of `FinderException`. The container throws the `ObjectNotFoundException` from the implementation of a finder or select method to indicate that the requested object does not exist.

Only single-object finder or select methods (see Subsections 8.5.7 and 8.5.8) should throw this exception. Multi-object finder or select methods must not throw this exception. Multi-object finder or select methods should return an empty collection as an indication that no matching objects were found.

8.5.10.5 RemoveException

From the client's perspective, a `RemoveException` (or a subclass of `RemoveException`) indicates that an application level error occurred during a `remove` operation. If a client receives this exception, the client does not know, in general, whether the entity object was removed or not. The client also does not know if the transaction has been marked for rollback. (However, the client may determine the transaction status using the `UserTransaction` interface.)

The container or Bean Provider throws the `RemoveException` (or subclass of `RemoveException`) from a remove method to indicate an application-level error from the entity object removal operation. Optionally, the container or Bean Provider may mark the transaction for rollback before throwing this exception.

The container or Bean Provider is encouraged to mark the transaction for rollback only if data integrity would be lost if the transaction were committed by the client. Typically, when a `RemoveException` is thrown, it leaves the database in a consistent state, allowing the client to recover.

The container treats the `RemoveException` as any other application exception. See Section 13.3.

8.5.11 Commit Options

The Entity Bean protocol is designed to give the container the flexibility to select the disposition of the instance state at transaction commit time. This flexibility allows the container to optimally manage the association of an entity object identity with the enterprise bean instances.

The container can select from the following commit-time options:

- **Option A**: The container caches a "ready" instance between transactions. The container knows that the bean instance has exclusive access to the state of the object in the persistent storage. Therefore, the container does not have to synchronize the instance's state from the persistent storage at the beginning of the next transaction or have to verify that the instance's state is in sync with the persistent storage at the beginning of the next transaction.

- **Option B**: The container caches a "ready" instance between transactions. In contrast to Option A, in this option the instance may not have exclusive access to the state of the object in the persistent storage. Therefore, the container must synchronize the instance's state from the persistent storage at the beginning of the next transaction if the instance's state in the persistent storage has changed. Containers using optimistic concurrency control strategies may instead choose to rollback the transaction if this invariant has not been met: The container must ensure that in order for a transaction to be successfully committed, the transaction must only operate on instance data that is in sync with the persistent storage at the beginning of the transaction.

- **Option C**: The container does not cache a "ready" instance between transactions. The container returns the instance to the pool of available instances after a transaction has completed.

Variants of these strategies that capture the same semantics from the Bean Provider's viewpoint may be employed, e.g., to optimize data access.

The following illustrative lazy loading strategies are consistent with the intent of these requirements:

- If `ejbLoad` is called at the beginning of the transaction without the instance's persistent state having been loaded from the persistent storage, the persistent state must be faulted in when `ejbLoad` causes the bean's getter accessor methods to be invoked. If the `ejbLoad` method is empty, data may be faulted in as needed in the course of executing the businesss methods of the bean.

- If the instance's persistent state is cached between transactions, `ejbLoad` need not be called and persistent data need not be faulted in from the persistent storage (unless it has not previously been accessed). In this case, because `ejbLoad` has been previously called when the instance was entered into the ready state for the first time, and because the bean instance's state is consistent with its persistent state, there is no need to call `ejbLoad` unless the instance's state in the persistent storage has changed. In this case, the container must ensure that in order for the transaction to be successfully committed, the instance's persistent state was in sync with the persistent storage at the beginning of the transaction.

The following table provides a summary of the commit-time options.

Table 6 Summary of Commit-Time Options

	Write instance state to database	**Instance stays ready**	**Instance state remains valid**
Option A	Yes	Yes	Yes
Option B	Yes	Yes	No
Option C	Yes	No	No

Note that the container synchronizes the instance's state with the persistent storage at transaction commit for all three options.

The selection of the commit option is transparent to the entity bean implementation—the entity bean will work correctly regardless of the commit-time option chosen by the container. The Bean Provider writes the entity bean in the same way.

Note: The Bean Provider relies on the `ejbLoad` method to be invoked in order to resynchronize the bean's transient state with its persistent state. It is the responsibility of the container to call the `ejbLoad` method at the beginning of a new transaction if the bean instance's persistent data has changed.[34]

[34] It is consistent with this specification to provide options for this refresh to be deferred or avoided in the case of read-only beans.

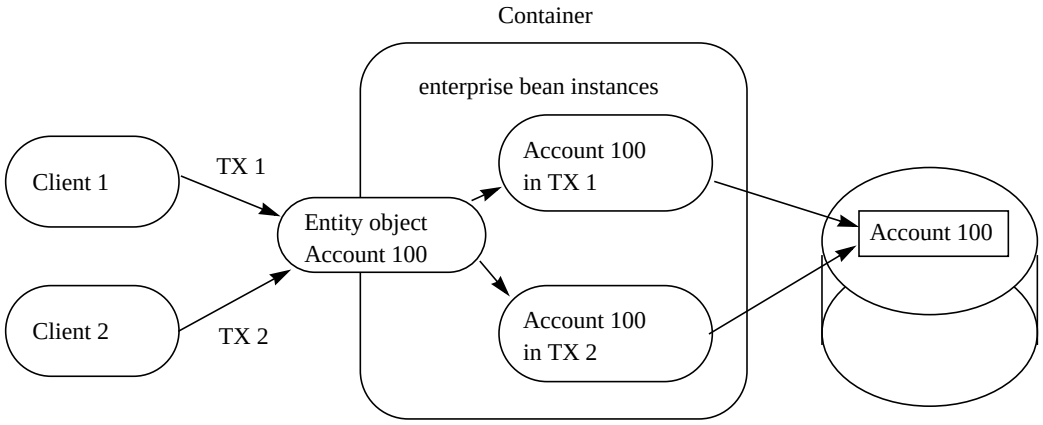
8.5.12 Concurrent Access from Multiple Transactions

When writing the entity bean business methods, the Bean Provider does not have to worry about concurrent access from multiple transactions. The Bean Provider may assume that the container will ensure appropriate synchronization for entity objects that are accessed concurrently from multiple transactions.

The container typically uses one of the following implementation strategies to achieve proper synchronization. (These strategies are illustrative, not prescriptive.)

- The container activates multiple instances of the entity bean, one for each transaction in which the entity object is being accessed. The transaction synchronization is performed by the underlying database during the accessor method calls performed by the business methods, the `ejbTimeout` method, and by the `ejbLoad`, `ejbCreate<METHOD>`, `ejbStore`, and `ejbRemove` methods. The commit-time options B and C in Subsection 8.5.11 apply to this type of container.

Figure 15 Multiple Clients Can Access the Same Entity Object Using Multiple Instances

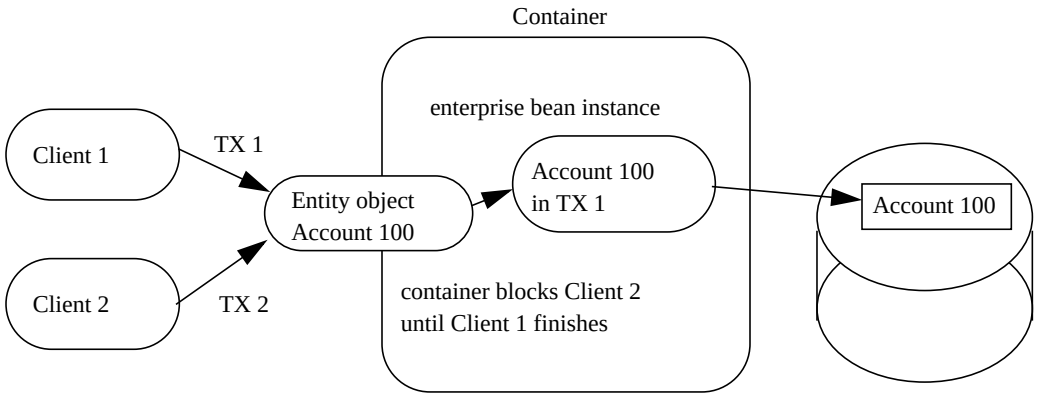


With this strategy, the type of lock acquired by `ejbLoad` or get accessor method (if a lazy loading cache management strategy is used) leads to a trade-off. If `ejbLoad` or the accessor method acquires an exclusive lock on the instance's state in the database, the throughput of read-only transactions could be impacted. If `ejbLoad` or the accessor method acquires a shared lock and the instance is updated, then either `ejbStore` or a set accessor method will need to promote the lock to an exclusive lock (which may cause a deadlock if it happens concurrently under multiple transactions), or, if the container uses an optimistic cache concurrency control strategy, the container will need to validate the state of the cache against the database at transaction commit (which may result in a rollback of the transaction).

It is expected that containers will provide deployment-time configuration options that will allow control to be exercised over the logical transaction isolation levels that their caching strategies provide.

- The container acquires exclusive access to the entity object's state in the database. The container activates a single instance and serializes the access from multiple transactions to this instance. The commit-time option A in Subsection 8.5.11 applies to this type of container.

Figure 16 Multiple Clients Can Access the Same Entity Object Using Single Instance



8.5.13 Non-reentrant and Re-entrant Instances

An entity Bean Provider can specify that an entity bean is non-reentrant. If an instance of a non-reentrant entity bean executes a client request in a given transaction context, and another request with the same transaction context arrives for the same entity object, the container will throw an exception to the second request. This rule allows the Bean Provider to program the entity bean as single-threaded, non-reentrant code.

The functionality of entity beans with container-managed persistence may require loopbacks in the same transaction context. An example of a loopback is when the client calls entity object A, A calls entity object B, and B calls back A in the same transaction context. The entity bean's method invoked by the loopback shares the current execution context (which includes the transaction and security contexts) with the Bean's method invoked by the client.

If the entity bean is specified as non-reentrant in the deployment descriptor, the container must reject an attempt to re-enter the instance via the entity bean's component interface while the instance is executing a business method. (This can happen, for example, if the instance has invoked another enterprise bean, and the other enterprise bean tries to make a loopback call.) If the attempt is made to reenter the instance through the remote interface, the container must throw the `java.rmi.RemoteException` to the caller. If the attempt is made to reenter the instance through the local interface, the container must throw the `javax.ejb.EJBException` to the caller. The container must allow the call if the Bean's deployment descriptor specifies that the entity bean is re-entrant.

Re-entrant entity beans must be programmed and used with caution. First, the Bean Provider must code the entity 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 a concurrent call in the same transaction context.

Concurrent calls in the same transaction context targeted at the same entity 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 should be marked as non-reentrant in the deployment descriptor, allowing the container to detect and prevent illegal concurrent calls from clients.

8.6 Responsibilities of the Enterprise Bean Provider

This section describes the responsibilities of an entity Bean Provider to ensure that an entity bean with container-managed persistence can be deployed in any EJB container.

8.6.1 Classes and Interfaces

The entity Bean Provider is responsible for providing the following class files:

- Entity bean class and any dependent classes
- Primary key class
- Entity bean's remote interface and entity bean's remote home interface, if the entity bean provides a remote client view
- Entity bean's local interface and local home interface, if the entity bean provides a local client view

The Bean Provider must provide a remote interface and a remote home interface or a local interface and a local home interface for the bean. The Bean Provider may provide a remote interface, remote home interface, local interface, and local home interface for the bean. Other combinations are not allowed.

8.6.2 Enterprise Bean Class

The following are the requirements for an entity bean class:

The class must implement, directly or indirectly, the `javax.ejb.EntityBean` interface.

The class may implement, directly or indirectly, the `javax.ejb.TimedObject` interface.

The class must be defined as `public` and must be `abstract`. The class must be a top level class.

The class must define a public constructor that takes no arguments.

The class must not define the `finalize()` method.

The class may, but is not required to, implement the entity bean's component interface[35]. If the class implements the entity bean's component interface, the class must provide no-op implementations of the methods defined by that interface. The container will never invoke these methods on the bean instances at runtime.

The entity bean class must implement the business methods, and the `ejbCreate<METHOD>` and `ejbPostCreate<METHOD>` methods as described later in this section.

The entity bean class must implement the `ejbHome<METHOD>` methods that correspond to the home business methods specified in the bean's home interface. These methods are executed on an instance in the pooled state; hence they must not access state that is particular to a specific bean instance (e.g., the accessor methods for the bean's abstract persistence schema must not be used by these methods).

The entity bean class must implement the get and set accessor methods of the bean's abstract persistence schema as `abstract` methods.

The entity bean class may have superclasses and/or superinterfaces. If the entity bean has superclasses, the business methods, the `ejbCreate<METHOD>` and `ejbPostCreate<METHOD>` methods, and the methods of the `EntityBean` interface and/or the `TimedObject` interface may be implemented in the enterprise bean class or in any of its superclasses.

The entity bean class is allowed to implement other methods (for example helper methods invoked internally by the business methods) in addition to the methods required by the EJB specification.

The entity bean class does not implement the finder methods. The implementations of the finder methods are provided by the container.

The entity bean class must implement any `ejbSelect<METHOD>` methods as abstract methods.

8.6.3 Dependent Value Classes

The following are the requirements for a dependent value class:

The class must be defined as `public` and must not be `abstract`.

The class must be serializable.

8.6.4 ejbCreate<METHOD> Methods

The entity bean class must implement the `ejbCreate<METHOD>` methods that correspond to the `create<METHOD>` methods specified in the entity bean's home interface or local home interface.

[35] If the entity bean class does implement the component interface, care must be taken to avoid passing of `this` as a method argument or result. This potential error can be avoided by choosing not to implement the component interface in the entity bean class.

The entity bean class may define zero or more `ejbCreate<METHOD>` methods whose signatures must follow these rules:

The method name must have `ejbCreate` as its prefix.

The method must be declared as `public`.

The method must not be declared as `final` or `static`.

The return type must be the entity bean's primary key type.

If the `ejbCreate<METHOD>` method corresponds to a `create<METHOD>` on the entity bean's remote home interface, the method arguments and return value types must be legal types for RMI-IIOP.

The `throws` clause must define the `javax.ejb.CreateException`. The `throws` clause may define arbitrary application specific exceptions.

Compatibility Note: EJB 1.0 allowed the `ejbCreate` method to throw the `java.rmi.RemoteException` to indicate a non-application exception. This practice was deprecated in EJB 1.1—an EJB 1.1 or EJB 2.0 or later compliant enterprise bean should throw the `javax.ejb.EJBException` or another `java.lang.RuntimeException` to indicate non-application exceptions to the container (see Section 13.2.2). The `ejbCreate` method of an entity bean with cmp-version 2.x must not throw the `java.rmi.RemoteException`.

8.6.5 ejbPostCreate<METHOD> Methods

For each `ejbCreate<METHOD>` method, the entity bean class must define a matching `ejbPostCreate<METHOD>` method, using the following rules:

The method name must have `ejbPostCreate` as its prefix.

The method must be declared as `public`.

The method must not be declared as `final` or `static`.

The return type must be `void`.

The method arguments must be the same as the arguments of the matching `ejbCreate<METHOD>` method.

The `throws` clause may define arbitrary application specific exceptions, including the `javax.ejb.CreateException`.

Compatibility Note: EJB 1.0 allowed the `ejbPostCreate` method to throw the `java.rmi.RemoteException` to indicate a non-application exception. This practice was deprecated in EJB 1.1—an EJB 1.1 or EJB 2.0 or later compliant enterprise bean should throw the `javax.ejb.EJBException` or another `java.lang.RuntimeException` to indicate non-application exceptions to the container (see Section 13.2.2). The `ejbPostCreate` method of an entity bean with cmp-version 2.x must not throw the `java.rmi.RemoteException`.

8.6.6 ejbHome<METHOD> Methods

The entity bean class may define zero or more home methods whose signatures must follow the following rules:

An `ejbHome<METHOD>` method must exist for every home `<METHOD>` method on the entity bean's remote home or local home interface. The method name must have `ejbHome` as its prefix followed by the name of the `<METHOD>` method in which the first character has been uppercased.

The method must be declared as `public`.

The method must not be declared as `static`.

If the `ejbHome<METHOD>` method corresponds to a home `<METHOD>` on the entity bean's remote home interface, the method argument and return value types must be legal types for RMI-IIOP.

The `throws` clause may define arbitrary application specific exceptions. The `throws` clause must not throw the `java.rmi.RemoteException`.

8.6.7 ejbSelect<METHOD> Methods

The entity bean class may define one or more select methods whose signatures must follow the following rules:

The method name must have `ejbSelect` as its prefix.

The method must be declared as `public`.

The method must be declared as `abstract`.

The `throws` clause must define the `javax.ejb.FinderException`. The `throws` clause may define arbitrary application specific exceptions.

8.6.8 Business Methods

The entity bean class may define zero or more business methods whose signatures must follow these rules:

The method names can be arbitrary, but they must not start with '`ejb`' to avoid conflicts with the callback methods used by the EJB architecture.

The business method must be declared as `public`.

The method must not be declared as `final` or `static`.

If the business method corresponds to a method of the entity bean's remote interface, the method argument and return value types must be legal types for RMI-IIOP.

The `throws` clause may define arbitrary application specific exceptions.

Compatibility Note: EJB 1.0 allowed the business methods to throw the `java.rmi.RemoteException` to indicate a non-application exception. This practice was deprecated in EJB 1.1—an EJB 1.1 or EJB 2.0 or later compliant enterprise bean should throw the `javax.ejb.EJBException` or another `java.lang.RuntimeException` to indicate non-application exceptions to the container (see Section 13.2.2).The business methods of an entity bean with cmp-version 2.x must not throw the `java.rmi.RemoteException`.

8.6.9 Entity Bean's Remote Interface

The following are the requirements for the entity bean's remote interface:

The interface must extend the `javax.ejb.EJBObject` interface.

The methods defined in the remote interface must follow the rules for RMI-IIOP. This means that their argument and return value types must be valid types for RMI-IIOP, and their `throws` clauses must include the `java.rmi.RemoteException`.

The remote interface is allowed to have superinterfaces. Use of interface inheritance is subject to the RMI-IIOP rules for the definition of remote interfaces.

For each method defined in the remote interface, there must be a matching method in the entity bean's class. The matching method must have:

- The same name.
- The same number and types of its arguments, and the same return type.
- All the exceptions defined in the `throws` clause of the matching method of the enterprise Bean class must be defined in the `throws` clause of the method of the remote interface.

The remote interface methods must not expose local interface types, local home interface types, timer handles, or the managed collection classes that are used for entity beans with container-managed persistence as arguments or results.

8.6.10 Entity Bean's Remote Home Interface

The following are the requirements for the entity bean's home interface:

The interface must extend the `javax.ejb.EJBHome` interface.

The methods defined in this interface must follow the rules for RMI-IIOP. This means that their argument and return types must be of valid types for RMI-IIOP, and their `throws` clauses must include the `java.rmi.RemoteException`.

The remote home interface is allowed to have superinterfaces. Use of interface inheritance is subject to the RMI-IIOP rules for the definition of remote interfaces.

Each method defined in the remote home interface must be one of the following:

- A create method.
- A finder method.
- A home method.

Each `create` method must be named "`create<METHOD>`", e.g. `createLargeAccounts`. Each create method name must match one of the `ejbCreate<METHOD>` methods defined in the enterprise bean class. The matching `ejbCreate<METHOD>` method must have the same number and types of its arguments. (Note that the return type is different.)

The return type for a `create<METHOD>` method must be the entity bean's remote interface type.

All the exceptions defined in the `throws` clause of the matching `ejbCreate<METHOD>` and `ejbPostCreate<METHOD>` methods of the enterprise bean class must be included in the `throws` clause of the matching `create` method of the home interface (i.e., the set of exceptions defined for the `create` method must be a superset of the union of exceptions defined for the `ejbCreate<METHOD>` and `ejbPostCreate<METHOD>` methods).

The `throws` clause of a `create<METHOD>` method must include the `javax.ejb.CreateException`.

Each `finder` method must be named "`find<METHOD>`" (e.g. `findLargeAccounts`).

The return type for a `find<METHOD>` method must be the entity bean's remote interface type (for a single-object finder), or a collection thereof (for a multi-object finder).

The remote home interface must always include the `findByPrimaryKey` method, which is always a single-object finder. The method must declare the primary key class as the method argument.

The `throws` clause of a `finder` method must include the `javax.ejb.FinderException`.

Home methods can have arbitrary names, but they must not start with "`create`", "`find`", or "`remove`". Their argument and return types must be of valid types for RMI-IIOP, and their `throws` clauses must include the `java.rmi.RemoteException`. The matching `ejbHome` method specified in the entity bean class must have the same number and types of arguments and must return the same type as the home method as specified in the remote home interface of the bean.

The remote home interface methods must not expose local interface types, local home interface types, timers or timer handles, or the managed collection classes that are used for entity beans with container-managed persistence as arguments or results.

8.6.11 Entity Bean's Local Interface

The following are the requirements for the entity bean's local interface:

The interface must extend the `javax.ejb.EJBLocalObject` interface.

For each method defined in the local interface, there must be a matching method in the entity bean's class. The matching method must have:

- The same name.
- The same number and types of its arguments, and the same return type.
- All the exceptions defined in the `throws` clause of the matching method of the enterprise Bean class must be defined in the `throws` clause of the method of the local interface.

8.6.12 Entity Bean's Local Home Interface

The following are the requirements for the entity bean's local home interface:

The interface must extend the `javax.ejb.EJBLocalHome` interface.

Each method defined in the home interface must be one of the following:

- A create method.
- A finder method.
- A home method.

Each `create` method must be named "`create<METHOD>`", e.g. `createLargeAccounts`. Each create method name must match one of the `ejbCreate<METHOD>` methods defined in the enterprise bean class. The matching `ejbCreate<METHOD>` method must have the same number and types of its arguments. (Note that the return type is different.)

The return type for a `create<METHOD>` method on the local home interface must be the entity bean's local interface type.

All the exceptions defined in the `throws` clause of the matching `ejbCreate<METHOD>` and `ejbPostCreate<METHOD>` methods of the enterprise bean class must be included in the `throws` clause of the matching `create` method of the local home interface (i.e., the set of exceptions defined for the `create` method must be a superset of the union of exceptions defined for the `ejbCreate<METHOD>` and `ejbPostCreate<METHOD>` methods).

The `throws` clause of a `create<METHOD>` method must include the `javax.ejb.CreateException`.

Each `finder` method must be named "`find<METHOD>`" (e.g. `findLargeAccounts`).

The return type for a `find<METHOD>` method defined on the local home interface must be the entity bean's local interface type (for a single-object finder), or a collection thereof (for a multi-object finder).

The local home interface must always include the `findByPrimaryKey` method, which is always a single-object finder. The method must declare the primary key class as the method argument.

The `throws` clause of a `finder` method must include the `javax.ejb.FinderException`.

Home methods can have arbitrary names, but they must not start with "`create`", "`find`", or "`remove`". The matching `ejbHome` method specified in the entity bean class must have the same number and types of arguments and must return the same type as the home method as specified in the home interface of the bean. The `throws` clause of a home method defined on the local home interface must not include the `java.rmi.RemoteException`.

8.6.13 Entity Bean's Primary Key Class

The Bean Provider must specify a primary key class in the deployment descriptor.

The primary key type must be a legal Value Type in RMI-IIOP.

The class must provide suitable implementation of the `hashCode()` and `equals(Object other)` methods to simplify the management of the primary keys by the container.

8.6.14 Entity Bean's Deployment Descriptor

The Bean Provider must specify the relationships in which the entity beans participate in the `relationships` element.

The Bean Provider must provide unique names to designate entity beans as follows, and as described in Section 8.3.13.

- The Bean Provider must specify unique names for entity beans which are defined in the ejb-jar file by using the `ejb-name` element.
- The Bean Provider must specify a unique abstract schema name for an entity bean using the `abstract-schema-name` deployment descriptor element.

The Bean Provider must define a query for each finder or select method except `findByPrimaryKey(key)`. Typically this will be provided as the content of the `ejb-ql` element contained in the `query` element for the entity bean. The syntax of EJB QL is defined in the document "*Java Persistence*" of this specification [2].

Since EJB QL query strings are embedded in the deployment descriptor, which is an XML document, it may be necessary to encode the following characters in the query string: "`>`", "`<`".

8.7 The Responsibilities of the Container Provider

This section describes the responsibilities of the Container Provider to support entity beans. The Container Provider is responsible for providing the deployment tools, and for managing the entity beans at runtime, including their persistent state and relationships.

Because the EJB specification does not define the API between deployment tools and the container, we assume that the deployment tools described in this section are provided by the Container Provider. Alternatively, the deployment tools may be provided by a different vendor who uses the container vendor's specific API.

8.7.1 Generation of Implementation Classes

The deployment tools provided by the Container Provider are responsible for the generation of additional classes when the entity bean is deployed. The tools obtain the information that they need for generation of the additional classes by introspecting the classes and interfaces provided by the Bean Provider and by examining the entity bean's deployment descriptor.

The deployment tools must generate the following classes:

- A class that implements the entity bean's remote home interface (i.e., the entity EJBHome class).
- A class that implements the entity bean's remote interface (i.e., the entity EJBObject class).
- A class that implements the entity bean's local home interface (i.e., the entity EJBLocalHome class).
- A class that implements the entity bean's local interface (i.e., the EJBLocalObject class).
- A class that implements the entity bean class (i.e., a concrete class corresponding to the abstract entity bean class that was provided by the Bean Provider).

The deployment tools may also generate a class that mixes some container-specific code with the entity bean class. The code may, for example, help the container to manage the entity bean instances at runtime. Tools can use subclassing, delegation, and code generation.

The deployment tools may also allow generation of additional code that wraps the business methods and that is used to customize the business logic for an existing operational environment. For example, a wrapper for a `debit` function on the `Account` bean may check that the debited amount does not exceed a certain limit, or perform security checking that is specific to the operational environment.

8.7.2 Enterprise Bean Class

The following are the requirements for a concrete entity bean class:

The class must extend the abstract entity bean class provided by the Bean Provider.

The class must be defined as `public` and must not be `abstract`.

The class must define a public constructor that takes no arguments.

The class must implement the get and set accessor methods of the bean's abstract persistence schema.

The class must not define the `finalize` method.

The entity bean class must implement the `ejbFind<METHOD>` methods.

The entity bean class must implement the `ejbSelect<METHOD>` methods.

The entity bean class is allowed to implement other methods in addition to the methods required by the EJB specification.

8.7.3 ejbFind<METHOD> Methods

For each `find<METHOD>` method in the remote home interface or local home interface of the entity bean, there must be a corresponding `ejbFind<METHOD>` method with the same argument types in the concrete entity bean class.

The method name must have `ejbFind` as its prefix.

The method must be declared as `public`.

If the `ejbFind<METHOD>` method corresponds to a `find<METHOD>` on the entity bean's remote home interface, the method argument and return value types must be legal types for RMI-IIOP.

The return type of an `ejbFind<METHOD>` method must be the entity bean's primary key type, or a collection of primary keys.

The `throws` clause must define the `javax.ejb.FinderException`. The `throws` clause may define arbitrary application specific exceptions.

Every finder method except `ejbFindByPrimaryKey(key)` is specified in the `query` deployment descriptor element for the entity. The container must use the EJB QL query string that is the content of the `ejb-ql` element or the descriptive query specification contained in the `description` element as the definition of the query of the corresponding `ejbFind<METHOD>` method.

8.7.4 ejbSelect<METHOD> Methods

For each `ejbSelect<METHOD>` method in the abstract entity bean class, there must be a method with the same argument and result types in the concrete entity bean class.

Every select method is specified in a `query` deployment descriptor element for the entity. The container must use the EJB QL query string that is the content of the `ejb-ql` element or the descriptive query specification that is contained in the `description` element as the definition of the query of the corresponding `ejbSelect<METHOD>`method.

The container must use the contents of the `query` element, the corresponding EJB QL string and the type of the values selected as specified by the SELECT clause to determine the type of the values returned by a select method.

The container must ensure that there are no duplicates returned by a select method if the return type is `java.util.Set`.

8.7.5 Entity EJBHome Class

The entity EJBHome class, which is generated by deployment tools, implements the entity bean's remote home interface. This class implements the methods of the `javax.ejb.EJBHome` interface, and the type-specific `create` and `finder` methods specific to the entity bean.

The implementation of each `create<METHOD>` method invokes a matching `ejbCreate<METHOD>` method, followed by the matching `ejbPostCreate<METHOD>` method, passing the `create<METHOD>` parameters to these matching methods.

The implementation of the `remove` methods defined in the `javax.ejb.EJBHome` interface must activate an instance (if an instance is not already in the ready state) and invoke the `ejbRemove` method on the instance.

The implementation of each `find<METHOD>` method invokes a matching `ejbFind<METHOD>` method. The implementation of the `find<METHOD>` method must create an entity object reference for the primary key returned from the `ejbFind<METHOD>` and return the entity object reference to the client. If the `ejbFind<METHOD>` method returns a collection of primary keys, the implementation of the `find<METHOD>` method must create a collection of entity object references for the primary keys and return the collection to the client.

The implementation of each `<METHOD>` home method invokes a matching `ejbHome<METHOD>` method (in which the first character of `<METHOD>` is uppercased in the name of the `ejbHome<METHOD>` method), passing the parameters of the `<METHOD>` method to the matching `ejbHome<METHOD>`method.

8.7.6 Entity EJBObject Class

The entity EJBObject class, which is generated by deployment tools, implements the entity bean's remote interface. It implements the methods of the `javax.ejb.EJBObject` interface and the remote business methods specific to the entity bean.

The implementation of the `remove` method (defined in the `javax.ejb.EJBObject` interface) must activate an instance (if an instance is not already in the ready state) and invoke the `ejbRemove` method on the instance.

The implementation of each remote business method must activate an instance (if an instance is not already in the ready state) and invoke the matching business method on the instance.

8.7.7 Entity EJBLocalHome Class

The entity EJBLocalHome class, which is generated by deployment tools, implements the entity bean's local home interface. This class implements the methods of the `javax.ejb.EJBLocalHome` interface, and the type-specific `create` and `finder` methods specific to the entity bean.

The implementation of each `create<METHOD>` method invokes a matching `ejbCreate<METHOD>` method, followed by the matching `ejbPostCreate<METHOD>` method, passing the `create<METHOD>` parameters to these matching methods.

The implementation of the `remove` method defined in the `javax.ejb.EJBLocalHome` interface must activate an instance (if an instance is not already in the ready state) and invoke the `ejbRemove` method on the instance.

The implementation of each `find<METHOD>` method invokes a matching `ejbFind<METHOD>` method. The implementation of the `find<METHOD>` method must create a local entity object reference for the primary key returned from the `ejbFind<METHOD>` and return the local entity object reference to the local client. If the `ejbFind<METHOD>` method returns a collection of primary keys, the implementation of the `find<METHOD>` method must create a collection of local entity object references for the primary keys and return the collection to the local client.

The implementation of each `<METHOD>` home method invokes a matching `ejbHome<METHOD>` method (in which the first character of `<METHOD>` is uppercased in the name of the `ejbHome<METHOD>` method), passing the parameters of the `<METHOD>` method to the matching `ejbHome<METHOD>` method.

8.7.8 Entity EJBLocalObject Class

The entity EJBLocalObject class, which is generated by deployment tools, implements the entity bean's local interface. It implements the methods of the `javax.ejb.EJBLocalObject` interface and the local business methods specific to the entity bean.

The implementation of the `remove` method (defined in the `javax.ejb.EJBLocalObject` interface) must activate an instance (if an instance is not already in the ready state) and invoke the `ejbRemove` method on the instance.

The implementation of each local business method must activate an instance (if an instance is not already in the ready state) and invoke the matching business method on the instance.

8.7.9 Handle Class

The deployment tools are responsible for implementing the handle class for the entity bean. The handle class must be serializable by the Java Serialization protocol.

As the handle class is not entity bean specific, the container may, but is not required to, use a single class for all deployed entity beans.

8.7.10 Home Handle Class

The deployment tools responsible for implementing the home handle class for the entity bean. The handle class must be serializable by the Java Serialization protocol.

Because the home handle class is not entity bean specific, the container may, but is not required to, use a single class for the home handles of all deployed entity beans.

8.7.11 Metadata Class

The deployment tools are responsible for implementing the class that provides metadata information to the remote client view contract. The class must be a valid RMI-IIOP Value Type, and must implement the `javax.ejb.EJBMetaData` interface.

Because the metadata class is not entity bean specific, the container may, but is not required to, use a single class for all deployed enterprise beans.

8.7.12 Instance's Re-entrance

The container runtime must enforce the rules defined in Section 8.5.13.

8.7.13 Transaction Scoping, Security, Exceptions

The container runtime must follow the rules on transaction scoping, security checking, and exception handling described in Chapters 12, 16, and 13.

8.7.14 Implementation of Object References

The container should implement the distribution protocol between the remote client and the container such that the object references of the remote home and remote interfaces used by entity bean clients are usable for a long period of time. Ideally, a remote client should be able to use an object reference across a server crash and restart. An object reference should become invalid only when the entity object has been removed, or after a reconfiguration of the server environment (for example, when the entity bean is moved to a different EJB server or container).

The motivation for this is to simplify the programming model for the entity bean client. While the client code needs to have a recovery handler for the system exceptions thrown from the individual method invocations on the remote home and remote interface, the client should not be forced to re-obtain the object references.

8.7.15 EntityContext

The container must implement the `EntityContext.getEJBObject` method such that the bean instance can use the Java language cast to convert the returned value to the entity bean's remote interface type. Specifically, the bean instance does not have to use the `PortableRemoteObject.narrow` method for the type conversion.

8.8 Primary Keys

The container must be able to manipulate the primary key type of an entity bean. Therefore, the primary key type for an entity bean with container-managed persistence must follow the rules in this subsection, in addition to those specified in Subsection 8.6.13.

There are two ways to specify a primary key class for an entity bean with container-managed persistence:

- Primary key that maps to a single field in the entity bean class.
- Primary key that maps to multiple fields in the entity bean class.

The second method is necessary for implementing compound keys, and the first method is convenient for single-field keys. Without the first method, simple types such as String would have to be wrapped in a user-defined class.

8.8.1 Primary Key That Maps to a Single Field in the Entity Bean Class

The Bean Provider uses the `primkey-field` element of the deployment descriptor to specify the container-managed field of the entity bean class that contains the primary key. The field's type must be the primary key type.

8.8.2 Primary Key That Maps to Multiple Fields in the Entity Bean Class

The primary key class must be `public`, and must have a `public` constructor with no parameters.

All fields in the primary key class must be declared as public.

The names of the fields in the primary key class must be a subset of the names of the container-managed fields. (This allows the container to extract the primary key fields from an instance's container-managed fields, and vice versa.)

8.8.3 Special Case: Unknown Primary Key Class

In special situations, the entity Bean Provider may choose not to specify the primary key class or the primary key fields for an entity bean with container-managed persistence. This case usually happens when the entity bean does not have a natural primary key, and/or the Bean Provider wants to allow the Deployer using the Container Provider's tools to select the primary key fields at deployment time. The entity bean's primary key type will usually be derived from the primary key type used by the underlying database system that stores the entity objects. The primary key used by the database system may not be known to the Bean Provider.

In this special case, the type of the argument of the `findByPrimaryKey` method must be declared as `java.lang.Object`. The Bean Provider must specify the primary key class in the deployment descriptor as of the type `java.lang.Object`.

When defining the primary key for the enterprise bean, the Deployer using the Container Provider's tools will typically add additional container-managed fields to the concrete subclass of the entity bean class (this typically happens for entity beans that do not have a natural primary key, and the primary keys are system-generated by the underlying database system that stores the entity objects). In this case, the container must generate the primary key value when the entity bean instance is created (and before `ejbPostCreate` is invoked on the instance.)

The primary key class is specified at deployment time in the situations when the Bean Provider develops an entity bean that is intended to be used with multiple back-ends that provide persistence, and when these multiple back-ends require different primary key structures.

Use of entity beans with a deferred primary key type specification limits the client application programming model, because the clients written prior to deployment of the entity bean may not use, in general, the methods that rely on the knowledge of the primary key type.

The implementation of the enterprise bean class methods must be done carefully. For example, the methods should not depend on the type of the object returned from `EntityContext.getPrimaryKey`, because the return type is determined by the Deployer after the EJB class has been written.

Chapter 9

EJB 2.1 Entity Bean Component Contract for Bean-Managed Persistence

The entity bean component contract for bean-managed persistence is the contract between an entity bean and its container. It defines the life cycle of the entity bean instances and the model for method delegation of the client-invoked business methods. The main goal of this contract is to ensure that a component using bean-managed persistence is portable across all compliant EJB containers.

This chapter defines the Enterprise Bean Provider's view of this contract and the Container Provider's responsibility for managing the life cycle of the enterprise bean instances. It also describes the Bean Provider's responsibilities when persistence is provided by the Bean Provider.

The contents of this chapter apply only to bean-managed persistence entities as defined in the Enterprise JavaBeans 2.1 specification [3]. The contracts for persistent entities, as defined by Enterprise JavaBeans 3.0, are described in the document "*Java Persistence*" of this specification [2].

Note that use of dependency injection, interceptors, and any Java language metadata annotations is not supported for EJB 2.1 entity beans.

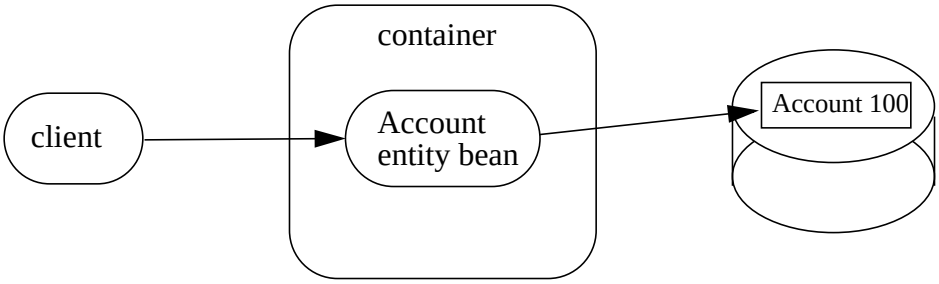
9.1 Overview of Bean-Managed Entity Persistence

An entity bean implements an object view of an entity stored in an underlying database, or an entity implemented by an existing enterprise application (for example, by a mainframe program or by an ERP application). The data access protocol for transferring the state of the entity between the entity bean instances and the underlying database is referred to as object persistence.

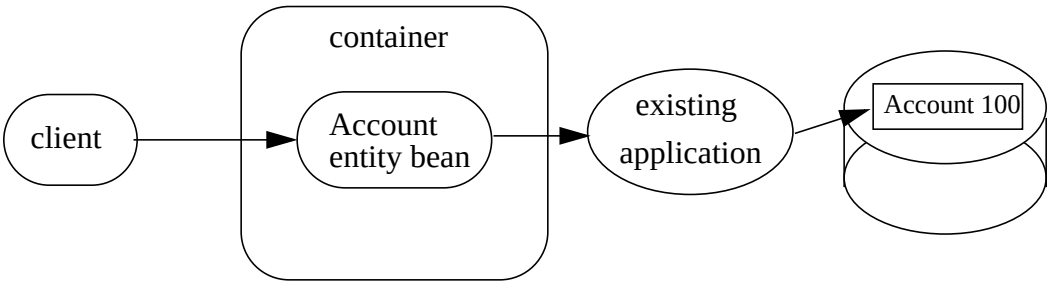
The entity bean component protocol for bean-managed persistence allows the entity Bean Provider to implement the entity bean's persistence directly in the entity bean class or in one or more helper classes provided with the entity bean class. This chapter describes the contracts for bean-managed persistence.

Figure 17 Client View of Underlying Data Sources Accessed Through Entity Bean

(a) Entity bean is an object view of a record in the database



(b) Entity bean is an object view of an existing application



9.1.1 Entity Bean Provider's View of Persistence

Using bean-managed persistence, the entity Bean Provider writes database access calls (e.g. using JDBC™ or SQLJ) directly in the entity bean component. The data access calls are performed in the `ejbCreate<METHOD>`, `ejbRemove`, `ejbFind<METHOD>`, `ejbLoad`, and `ejbStore` methods, and/or in the business methods.

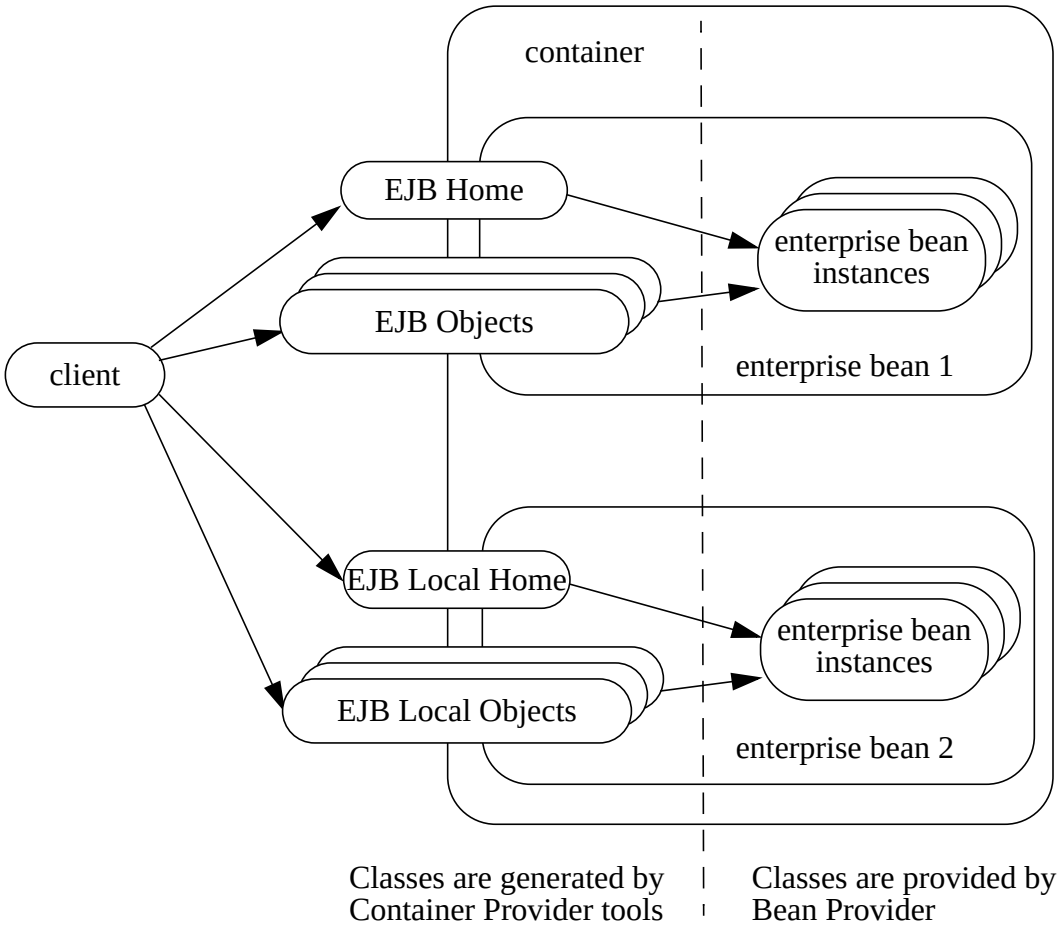
The data access calls can be coded directly into the entity bean class, or they can be encapsulated in a data access component that is part of the entity bean. Directly coding data access calls in the entity bean class may make it more difficult to adapt the entity bean to work with a database that has a different schema, or with a different type of database.

We expect that most enterprise beans with bean-managed persistence will be created by application development tools which will encapsulate data access in components. These data access components will probably not be the same for all tools. Further, if the data access calls are encapsulated in data access components, the data access components may require deployment interfaces to allow adapting data access to different schemas or even to a different database type. This EJB specification does not define the architecture for data access objects, strategies for tailoring and deploying data access components or ensuring portability of these components for bean-managed persistence.

9.1.2 Runtime Execution Model

This section describes the runtime model and the classes used in the description of the contract between an entity bean with bean-managed persistence and its container.

Figure 18 Overview of the Entity Bean Runtime Execution Model



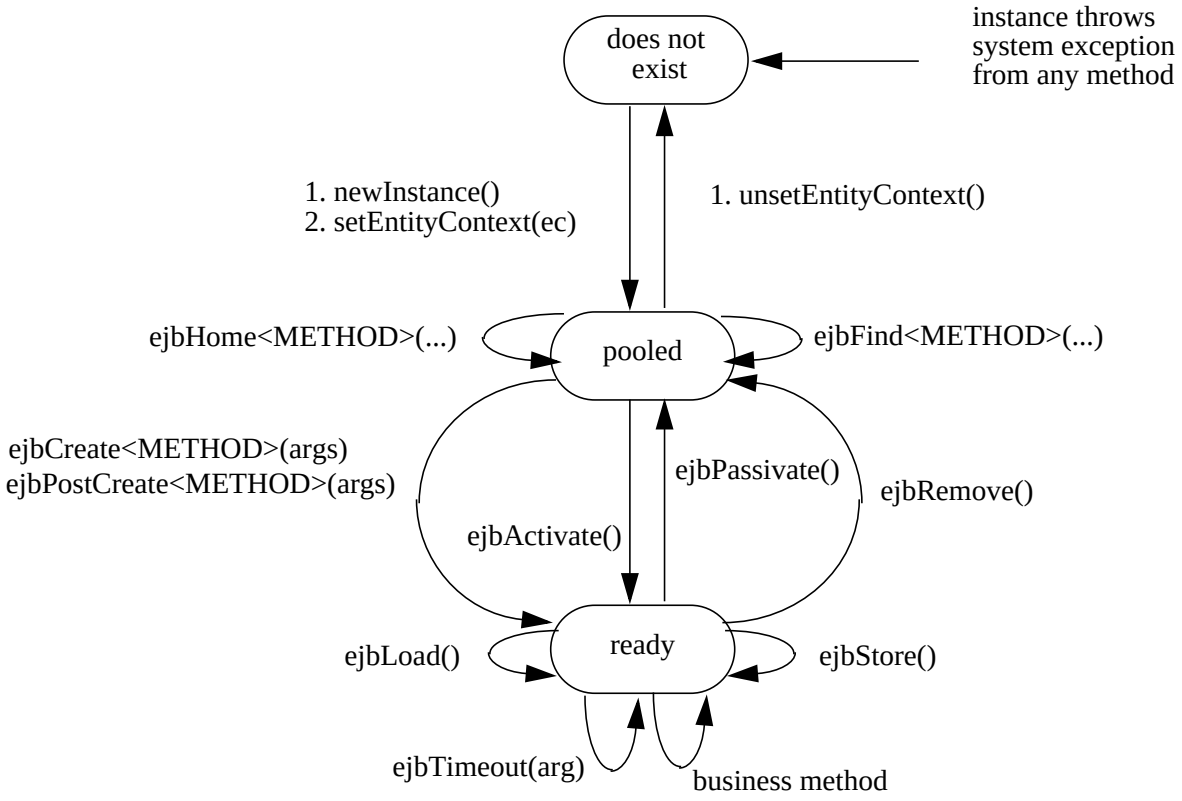
An enterprise bean instance is an object whose class is provided by the Bean Provider.

An entity **EJBObject** or **EJBLocalObject** is an object whose class is generated at deployment time by the Container Provider's tools. The entity EJBObject class implements the entity bean's remote interface. The entity EJBLocalObject class implements the entity bean's local interface. A client never references an entity bean instance directly—a client always references an entity EJBObject or entity EJBLocalObject whose class is generated by the Container Provider's tools.

An entity **EJBHome** or **EJBLocalHome** object provides the life cycle operations (create, remove, find) for its entity objects as well as home business methods, which are not specific to an entity bean instance. The class for the entity EJBHome or EJBLocalHome object is generated by the Container Provider's tools at deployment time. The entity EJBHome or EJBLocalHome object implements the entity bean's home interface that was defined by the Bean Provider.

9.1.3 Instance Life Cycle

Figure 19 Life Cycle of an Entity Bean Instance.



An entity bean instance is in one of the following three states:

- It does not exist.
- Pooled state. An instance in the pooled state is not associated with any particular entity object identity.
- Ready state. An instance in the ready state is assigned an entity object identity.

The following steps describe the life cycle of an entity bean instance:

- An entity bean instance's life starts when the container creates the instance using `newInstance`. The container then invokes the `setEntityContext` method to pass the instance a reference to the `EntityContext` interface. The `EntityContext` interface allows the

instance to invoke services provided by the container and to obtain the information about the caller of a client-invoked method.

- The instance enters the pool of available instances. Each entity bean has its own pool. While the instance is in the available pool, the instance is not associated with any particular entity object identity. All instances in the pool are considered equivalent, and therefore any instance can be assigned by the container to any entity object identity at the transition to the ready state. While the instance is in the pooled state, the container may use the instance to execute any of the entity bean's finder methods (shown as `ejbFind<METHOD>` in the diagram) or home methods (shown as `ejbHome<METHOD>` in the diagram). The instance does not move to the ready state during the execution of a finder or a home method.

- An instance transitions from the pooled state to the ready state when the container selects that instance to service a client call to an entity object or an `ejbTimeout` method. There are two possible transitions from the pooled to the ready state: through the `ejbCreate<METHOD>` and `ejbPostCreate<METHOD>` methods, or through the `ejbActivate` method. The container invokes the `ejbCreate<METHOD>` and `ejbPostCreate<METHOD>` methods when the instance is assigned to an entity object during entity object creation (i.e., when the client invokes a `create<METHOD>` method on the entity bean's home object). The container invokes the `ejbActivate` method on an instance when an instance needs to be activated to service an invocation on an existing entity object—this occurs because there is no suitable instance in the ready state to service the client's call or the `ejbTimeout` method.

- When an entity bean instance is in the ready state, the instance is associated with a specific entity object identity. While the instance is in the ready state, the container can invoke the `ejbLoad` and `ejbStore` methods zero or more times. A business method can be invoked on the instance zero or more times. The `ejbTimeout` method can be invoked on the instance zero or more times. Invocations of the `ejbLoad` and `ejbStore` methods can be arbitrarily mixed with invocations of business methods or the `ejbTimeout` method. The purpose of the `ejbLoad` and `ejbStore` methods is to synchronize the state of the instance with the state of the entity in the underlying data source—the container can invoke these methods whenever it determines a need to synchronize the instance's state.

- The container can choose to passivate an entity bean instance within a transaction. To passivate an instance, the container first invokes the `ejbStore` method to allow the instance to synchronize the database state with the instance's state, and then the container invokes the `ejbPassivate` method to return the instance to the pooled state.

- Eventually, the container will transition the instance to the pooled state. There are three possible transitions from the ready to the pooled state: through the `ejbPassivate` method, through the `ejbRemove` method, and because of a transaction rollback for `ejbCreate`, `ejbPostCreate`, or `ejbRemove` (not shown in Figure 19). The container invokes the `ejbPassivate` method when the container wants to disassociate the instance from the entity object identity without removing the entity object. The container invokes the `ejbRemove` method when the container is removing the entity object (i.e., when the client invoked the `remove` method on the entity object's component interface, or a `remove` method on the entity bean's home interface). If `ejbCreate`, `ejbPostCreate`, or `ejbRemove` is called and the transaction rolls back, the container will transition the bean instance to the pooled state.

- When the instance is put back into the pool, it is no longer associated with an entity object identity. The container can assign the instance to any entity object within the same entity bean home.

- An instance in the pool can be removed by calling the `unsetEntityContext` method on the instance.

Notes:

1. The `EntityContext` interface passed by the container to the instance in the `setEntityContext` method is an interface, not a class that contains static information. For example, the result of the `EntityContext.getPrimaryKey` method might be different each time an instance moves from the pooled state to the ready state, and the result of the `getCallerPrincipal` and `isCallerInRole` methods may be different in each business method.

2. A `RuntimeException` thrown from any method of the entity bean class (including the business methods and the callbacks invoked by the container) results in the transition to the "does not exist" state. The container must not invoke any method on the instance after a `RuntimeException` has been caught. From the client perspective, the corresponding entity object continues to exist. The client can continue accessing the entity object through its component interface because the container can use a different entity bean instance to delegate the client's requests. Exception handling is described further in Chapter 13.

3. The container is not required to maintain a pool of instances in the pooled state. The pooling approach is an example of a possible implementation, but it is not the required implementation. Whether the container uses a pool or not has no bearing on the entity bean coding style.

9.1.4 The Entity Bean Component Contract

This section specifies the contract between an entity bean with bean-managed persistence and its container.

9.1.4.1 Entity Bean Instance's View

The following describes the entity bean instance's view of the contract:

The Bean Provider is responsible for implementing the following methods in the entity bean class:

- A public constructor that takes no arguments. The container uses this constructor to create instances of the entity bean class.

- `public void setEntityContext(EntityContext ic);`

  A container uses this method to pass a reference to the `EntityContext` interface to the entity bean instance. If the entity bean instance needs to use the `EntityContext` interface during its lifetime, it must remember the `EntityContext` interface in an instance variable.

  This method executes with an unspecified transaction context (Refer to Subsection 12.6.5 for how the container executes methods with an unspecified transaction context). An identity of an entity object is not available during this method.

The instance can take advantage of the `setEntityContext` method to allocate any resources that are to be held by the instance for its lifetime. Such resources cannot be specific to an entity object identity because the instance might be reused during its lifetime to serve multiple entity object identities.

- `public void unsetEntityContext();`

  A container invokes this method before terminating the life of the instance.

  This method executes with an unspecified transaction context. An identity of an entity object is not available during this method.

  The instance can take advantage of the `unsetEntityContext` method to free any resources that are held by the instance. (These resources typically had been allocated by the `setEntityContext` method.)

- `public` *`PrimaryKeyClass`* `ejbCreate<METHOD>(...);`

  There are zero[36] or more `ejbCreate<METHOD>` methods, whose signatures match the signatures of the `create<METHOD>` methods of the entity bean home interface. The container invokes an `ejbCreate<METHOD>` method on an entity bean instance when a client invokes a matching `create<METHOD>` method to create an entity object.

  The implementation of the `ejbCreate<METHOD>` method typically validates the client-supplied arguments, and inserts a record representing the entity object into the database. The method also initializes the instance's variables. The `ejbCreate<METHOD>` method must return the primary key for the created entity object.

  An `ejbCreate<METHOD>` method executes in the transaction context determined by the transaction attribute of the matching `create<METHOD>` method, as described in subsection 12.6.2.

- `public void ejbPostCreate<METHOD>(...);`

  For each `ejbCreate<METHOD>` method, there is a matching `ejbPostCreate<METHOD>` method that has the same input parameters but whose return value is `void`. The container invokes the matching `ejbPostCreate<METHOD>` method on an instance after it invokes the `ejbCreate<METHOD>` method with the same arguments. The entity object identity is available during the `ejbPostCreate<METHOD>` method. The instance may, for example, obtain the component interface of the associated entity object and pass it to another enterprise bean as a method argument.

  An `ejbPostCreate<METHOD>` method executes in the same transaction context as the previous `ejbCreate<METHOD>` method.

- `public void ejbActivate();`

  The container invokes this method on the instance when the container picks the instance from the pool and assigns it to a specific entity object identity. The `ejbActivate` method gives the entity bean instance the chance to acquire additional resources that it needs while it is in the ready state.

---

[36] An entity Bean has no *`ejbCreate<METHOD>`* and *`ejbPostCreate<METHOD>`* methods if it does not define any create methods in its home interface. Such an entity bean does not allow the clients to create new entity objects. The entity bean restricts the clients to accessing entities that were created through direct database inserts.

This method executes with an unspecified transaction context. The instance can obtain the identity of the entity object via the `getPrimaryKey`, `getEJBLocalObject`, or `getEJBObject` method on the entity context. The instance can rely on the fact that the primary key and entity object identity will remain associated with the instance until the completion of `ejbPassivate` or `ejbRemove`.

Note that the instance should not use the `ejbActivate` method to read the state of the entity from the database; the instance should load its state only in the `ejbLoad` method.

- `public void ejbPassivate();`
  The container invokes this method on an instance when the container decides to disassociate the instance from an entity object identity, and to put the instance back into the pool of available instances. The `ejbPassivate` method gives the instance the chance to release any resources that should not be held while the instance is in the pool. (These resources typically had been allocated during the `ejbActivate` method.)

  This method executes with an unspecified transaction context. The instance can still obtain the identity of the entity object via the `getPrimaryKey`, `getEJBLocalObject`, or `getEJBObject` method of the `EntityContext` interface.

  Note that an instance should not use the `ejbPassivate` method to write its state to the database; an instance should store its state only in the `ejbStore` method.

- `public void ejbRemove();`

  The container invokes this method on an instance as a result of a client's invoking a `remove` method. The instance is in the ready state when `ejbRemove` is invoked and it will be entered into the pool when the method completes.

  This method executes in the transaction context determined by the transaction attribute of the `remove` method that triggered the `ejbRemove` method. The instance can still obtain the identity of the entity object via the `getPrimaryKey`, `getEJBLocalObject`, or `getEJBObject` method of the `EntityContext` interface.

  The container synchronizes the instance's state before it invokes the `ejbRemove` method. This means that the state of the instance variables at the beginning of the `ejbRemove` method is the same as it would be at the beginning of a business method.

  An entity bean instance should use this method to remove the entity object's representation from the database.

  Since the instance will be entered into the pool, the state of the instance at the end of this method must be equivalent to the state of a passivated instance. This means that the instance must release any resource that it would normally release in the `ejbPassivate` method.

- `public void ejbLoad();`

  The container invokes this method on an instance in the ready state to inform the instance that it should synchronize the entity state cached in its instance variables from the entity state in the database. The instance should be prepared for the container to invoke this method at any time that the instance is in the ready state.

  If the instance is caching the entity state (or parts of the entity state), the instance should not use the previously cached state in the subsequent business method. The instance may take advantage of the `ejbLoad` method, for example, to refresh the cached state by reading it from the database.

This method executes in the transaction context determined by the transaction attribute of the business method or `ejbTimeout` method that triggered the `ejbLoad` method.

- `public void ejbStore();`

  The container invokes this method on an instance to inform the instance that the instance should synchronize the entity state in the database with the entity state cached in its instance variables. The instance should be prepared for the container to invoke this method at any time that the instance is in the ready state.

  An instance should write any updates cached in the instance variables to the database in the `ejbStore` method.

  This method executes in the same transaction context as the previous `ejbLoad` or `ejbCreate<METHOD>` method invoked on the instance. All business methods or the `ejbTimeout` method invoked between the previous `ejbLoad` or `ejbCreate<METHOD>` method and this `ejbStore` method are also invoked in the same transaction context.

- `public` <primary key type or collection> `ejbFind<METHOD>(...);`

  The container invokes this method on the instance when the container selects the instance to execute a matching client-invoked `find<METHOD>` method. The instance is in the pooled state (i.e., it is not assigned to any particular entity object identity) when the container selects the instance to execute the `ejbFind<METHOD>` method on it, and it is returned to the pooled state when the execution of the `ejbFind<METHOD>` method completes.

  The `ejbFind<METHOD>` method executes in the transaction context determined by the transaction attribute of the matching `find` method, as described in subsection 12.6.2.

  The implementation of an `ejbFind<METHOD>` method typically uses the method's arguments to locate the requested entity object or a collection of entity objects in the database. The method must return a primary key or a collection of primary keys to the container (see Subsection 9.1.9).

- `public` <type> `ejbHome<METHOD>(...);`

  The container invokes this method on any instance when the container selects the instance to execute a matching client-invoked `<METHOD>` home method. The instance is in the pooled state (i.e., it is not assigned to any particular entity object identity) when the container selects the instance to execute the `ejbHome<METHOD>` method on it, and it is returned to the pooled state when the execution of the `ejbHome<METHOD>` method completes.

  The `ejbHome<METHOD>` method executes in the transaction context determined by the transaction attribute of the matching `<METHOD>` home method, as described in subsection 12.6.2.

- `public void ejbTimeout(...);`

  The container invokes the `ejbTimeout` method on an instance when a timer for the instance has expired. The `ejbTimeout` method notifies the instance of the time-based event and allows the instance to execute the business logic to handle it.

  The `ejbTimeout` method executes in the transaction context determined by its transaction attribute.

9.1.4.2 Container's View

This subsection describes the container's view of the state management contract. The container must call the following methods:

- `public void setEntityContext(ec);`

  The container invokes this method to pass a reference to the `EntityContext` interface to the entity bean instance. The container must invoke this method after it creates the instance, and before it puts the instance into the pool of available instances.

  The container invokes this method with an unspecified transaction context. At this point, the `EntityContext` is not associated with any entity object identity.

- `public void unsetEntityContext();`

  The container invokes this method when the container wants to reduce the number of instances in the pool. After this method completes, the container must not reuse this instance.

  The container invokes this method with an unspecified transaction context.

- `public` *`PrimaryKeyClass`* `ejbCreate<METHOD>(...);`
  `public void ejbPostCreate<METHOD>(...);`

  The container invokes these two methods during the creation of an entity object as a result of a client invoking a `create<METHOD>` method on the entity bean's home interface.

  The container first invokes the `ejbCreate<METHOD>` method whose signature matches the `create<METHOD>` method invoked by the client. The `ejbCreate<METHOD>` method returns a primary key for the created entity object. The container creates an entity EJBObject reference and/or EJBLocalObject reference for the primary key. The container then invokes a matching `ejbPostCreate<METHOD>` method to allow the instance to fully initialize itself. Finally, the container returns the entity object's remote interface (i.e., a reference to the entity EJBObject) to the client if the client is a remote client, or the entity object's local interface (i.e., a reference to the entity EJBLocalObject) to the client if the client is a local client.

  The container must invoke the `ejbCreate<METHOD>` and `ejbPostCreate<METHOD>` methods in the transaction context determined by the transaction attribute of the matching `create<METHOD>` method, as described in subsection 12.6.2.

- `public void ejbActivate();`

  The container invokes this method on an entity bean instance at activation time (i.e., when the instance is taken from the pool and assigned to an entity object identity). The container must ensure that the primary key of the associated entity object is available to the instance if the instance invokes the `getPrimaryKey`, `getEJBLocalObject`, or `getEJBObject` method on its `EntityContext` interface.

  The container invokes this method with an unspecified transaction context.

  Note that instance is not yet ready for the delivery of a business method. The container must still invoke the `ejbLoad` method prior to a business method or `ejbTimeout` method invocation.

- `public void ejbPassivate();`

  The container invokes this method on an entity bean instance at passivation time (i.e., when the instance is being disassociated from an entity object identity and moved into the pool). The

container must ensure that the identity of the associated entity object is still available to the instance if the instance invokes the `getPrimaryKey`, `getEJBLocalObject`, or `getEJBObject` method on its entity context.

The container invokes this method with an unspecified transaction context.

Note that if the instance state has been updated by a transaction, the container must first invoke the `ejbStore` method on the instance before it invokes `ejbPassivate` on it.

- `public void ejbRemove();`

  The container invokes this method before it ends the life of an entity object as a result of a client invoking a `remove` operation.

  The container invokes this method in the transaction context determined by the transaction attribute of the invoked `remove` method. The container must ensure that the identity of the associated entity object is still available to the instance in the `ejbRemove` method (i.e., the instance can invoke the `getPrimaryKey`, `getEJBLocalObject`, or `getEJBObject` method on its `EntityContext` in the `ejbRemove` method).

  The container must ensure that the instance's state is synchronized from the state in the database before invoking the `ejbRemove` method (i.e., if the instance is not already synchronized from the state in the database, the container must invoke `ejbLoad` before it invokes `ejbRemove`).

- `public void ejbLoad();`

  The container must invoke this method on the instance whenever it becomes necessary for the instance to synchronize its instance state from its state in the database. The exact times that the container invokes `ejbLoad` depend on the configuration of the component and the container, and are not defined by the EJB architecture. Typically, the container will call `ejbLoad` before the first business method within a transaction or before invoking the `ejbTimeout` method to ensure that the instance can refresh its cached state of the entity object from the database. After the first `ejbLoad` within a transaction, the container is not required to recognize that the state of the entity object in the database has been changed by another transaction, and it is not required to notify the instance of this change via another `ejbLoad` call.

  The container must invoke this method in the transaction context determined by the transaction attribute of the business method or `ejbTimeout` method that triggered the `ejbLoad` method.

- `public void ejbStore();`

  The container must invoke this method on the instance whenever it becomes necessary for the instance to synchronize its state in the database with the state of the instance's fields. This synchronization always happens at the end of a transaction, unless the bean is specified as read-only (see section 9.1.5). However, the container may also invoke this method when it passivates the instance in the middle of a transaction, or when it needs to transfer the most recent state of the entity object to another instance for the same entity object in the same transaction (see Subsection 12.7).

  The container must invoke this method in the same transaction context as the previously invoked `ejbLoad`, `ejbCreate<METHOD>`, or `ejbTimeout` method.

- `public` <primary key type or collection> `ejbFind<METHOD>(...);`

The container invokes the `ejbFind<METHOD>` method on an instance when a client invokes a matching `find<METHOD>` method on the entity bean's home interface. The container must pick an instance that is in the pooled state (i.e., the instance is not associated with any entity object identity) for the execution of the `ejbFind<METHOD>` method. If there is no instance in the pooled state, the container creates one and calls the `setEntityContext` method on the instance before dispatching the finder method.

Before invoking the `ejbFind<METHOD>` method, the container must first synchronize the state of any non-read-only entity bean instances that are participating in the same transaction context as is used to execute the `ejbFind<METHOD>` by invoking the `ejbStore` method on those entity bean instances. [37]

After the `ejbFind<METHOD>` method completes, the instance remains in the pooled state. The container may, but is not required to, immediately activate the objects that were located by the finder using the transition through the `ejbActivate` method.

The container must invoke the `ejbFind<METHOD>` method in the transaction context determined by the transaction attribute of the matching `find` method, as described in subsection 12.6.2.

If the `ejbFind<METHOD>` method is declared to return a single primary key, the container creates an entity EJBObject reference for the primary key and returns it to the client if the client is a remote client. If the client is a local client, the container creates and returns an entity EJBLocalObject reference for the primary key. If the `ejbFind<METHOD>` method is declared to return a collection of primary keys, the container creates a collection of entity EJBObject or EJBLocalObject references for the primary keys returned from `ejbFind<METHOD>`, and returns the collection to the client. (See Subsection 9.1.9 for information on collections.)

- `public` <type> `ejbHome<METHOD>(...);`

  The container invokes the `ejbHome<METHOD>` method on an instance when a client invokes a matching `<METHOD>` home method on the entity bean's home interface. The container must pick an instance that is in the pooled state (i.e., the instance is not associated with any entity object identity) for the execution of the `ejbHome<METHOD>` method. If there is no instance in the pooled state, the container creates one and calls the `setEntityContext` method on the instance before dispatching the home method.

  After the `ejbHome<METHOD>` method completes, the instance remains in the pooled state.

  The container must invoke the `ejbHome<METHOD>` method in the transaction context determined by the transaction attribute of the matching `<METHOD>` home method, as described in subsection 12.6.2.

- `public void ejbTimeout(...);`

  The container invokes the `ejbTimeout` method on the instance when a timer with which the entity has been registered expires. If there is no suitable instance in the ready state, the container must activate an instance, invoking the `ejbActivate` method and transitioning it to the ready state.

  The container invokes the `ejbTimeout` method in the context of a transaction determined by its transaction attribute.

[37] The EJB specification does not require the distributed flushing of state. The container in which the `ejbFind<METHOD>` method executes is not required to propagate the flush to a different container.

9.1.5 Read-only Entity Beans

Compliant implementations of this specification may optionally support read-only entity beans. A read-only entity bean is an entity bean whose instances are not intended to be updated and/or created by the application. Read-only beans are best suited for situations where the underlying data never changes or changes infrequently.

Containers that support read-only beans do not call the `ejbStore` method on them. The `ejbLoad` method should typically be called by the container when the state of the bean instance is initially loaded from the database, or at designated refresh intervals.[38]

If a read-only bean is used, the state of such a bean should not be updated by the application, and the behavior is unspecified if this occurs.[39]

Read-only beans are designated by vendor-specific means that are outside the scope of this specification, and their use is therefore not portable.

9.1.6 The EntityContext Interface

A container provides the entity bean instances with an `EntityContext`, which gives the entity bean instance access to the instance's context maintained by the container. The `EntityContext` interface has the following methods:

- The `getEJBObject` method returns the entity bean's remote interface.
- The `getEJBHome` method returns the entity bean's remote home interface.
- The `getEJBLocalObject` method returns the entity bean's local interface.
- The `getEJBLocalHome` method returns the entity bean's local home interface.
- The `getCallerPrincipal` method returns the `java.security.Principal` that identifies the invoker.
- The `isCallerInRole` method tests if the entity bean instance's caller has a particular role.
- The `setRollbackOnly` method allows the instance to mark the current transaction such that the only outcome of the transaction is a rollback.
- The `getRollbackOnly` method allows the instance to test if the current transaction has been marked for rollback.
- The `getPrimaryKey` method returns the entity bean's primary key.
- The `getTimerService` method returns the `javax.ejb.TimerService` interface.

---

[38] The ability to refresh the state of a read-only bean and the intervals at which such refresh occurs are vendor-specific.]

[39] For example, an implementation might choose to ignore such updates or to disallow them.

- The `getUserTransaction` method returns the `javax.transaction.UserTransaction` interface. Entity bean instances must not call this method.
- The `lookup` method enables the entity bean to look up its environment entries in the JNDI naming context.

9.1.7 Operations Allowed in the Methods of the Entity Bean Class

Table 7 defines the methods of an entity bean class in which the enterprise bean instances can access the methods of the `javax.ejb.EntityContext` interface, the `java:comp/env` environment naming context, resource managers, the `TimerService` and `Timer` methods, and other enterprise beans.

If an entity bean instance attempts to invoke a method of the `EntityContext` interface, and the access is not allowed in Table 7, the container must throw the `java.lang.IllegalStateException.`

If an entity bean instance attempts to invoke a method of the `TimerService` or `Timer` interface and the access is not allowed in Table 7, the container must throw the `java.lang.IllegalStateException.`

If an entity bean instance attempts to access a resource manager or an enterprise bean, and the access is not allowed in Table 7, the behavior is undefined by the EJB architecture.

Table 7 Operations Allowed in the Methods of an Entity Bean

Bean method	Bean method can perform the following operations
constructor	-
setEntityContext unsetEntityContext	EntityContext methods: *getEJBHome*, *getEJBLocalHome*, *lookup* JNDI access to java:comp/env
ejbCreate	EntityContext methods: *getEJBHome*, *getEJBLocalHome*, *getCallerPrincipal*, *getRollbackOnly*, *isCallerInRole*, *setRollbackOnly*, *getTimerService*, *lookup* JNDI access to java:comp/env Resource manager access Enterprise bean access
ejbPostCreate	EntityContext methods: *getEJBHome*, *getEJBLocalHome*, *getCallerPrincipal*, *getRollbackOnly*, *isCallerInRole*, *setRollbackOnly*, *getEJBObject*, *getEJBLocalObject*, *getPrimaryKey*, *getTimerService*, *lookup* JNDI access to java:comp/env Resource manager access Enterprise bean access Timer service or Timer methods

Table 7 Operations Allowed in the Methods of an Entity Bean

Bean method	Bean method can perform the following operations
ejbRemove	EntityContext methods: *getEJBHome*, *getEJBLocalHome*, *getCallerPrincipal*, *getRollbackOnly*, *isCallerInRole*, *setRollbackOnly*, *getEJBObject*, *getEJBLocalObject*, *getPrimaryKey*, *getTimerService*, *lookup* JNDI access to java:comp/env Resource manager access Enterprise bean access Timer service or Timer methods
ejbFind	EntityContext methods: *getEJBHome*, *getEJBLocalHome*, *getCallerPrincipal*, *getRollbackOnly*, *isCallerInRole*, *setRollbackOnly*, *lookup* JNDI access to java:comp/env Resource manager access Enterprise bean access
ejbHome	EntityContext methods: *getEJBHome*, *getEJBLocalHome*, *getCallerPrincipal*, *getRollbackOnly*, *isCallerInRole*, *setRollbackOnly*, *getTimerService*, *lookup* JNDI access to java:comp/env Resource manager access Enterprise bean access
ejbActivate ejbPassivate	EntityContext methods: *getEJBHome*, *getEJBLocalHome*, *getEJBObject*, *getEJBLocalObject*, *getPrimaryKey*, *getTimerService*, *lookup* JNDI access to java:comp/env
ejbLoad ejbStore	EntityContext methods: *getEJBHome*, *getEJBLocalHome*, *getCallerPrincipal*, *getRollbackOnly*, *isCallerInRole*, *setRollbackOnly*, *getEJBObject*, *getEJBLocalObject*, *getPrimaryKey*, *getTimerService*, *lookup* JNDI access to java:comp/env Resource manager access Enterprise bean access Timer service or Timer methods
business method from component interface	EntityContext methods: *getEJBHome*, *getEJBLocalHome*, *getCallerPrincipal*, *getRollbackOnly*, *isCallerInRole*, *setRollbackOnly*, *getEJBObject*, *getEJBLocalObject*, *getPrimaryKey*, *getTimerService*, *lookup* JNDI access to java:comp/env Resource manager access Enterprise bean access Timer service or Timer methods

Table 7 Operations Allowed in the Methods of an Entity Bean

Bean method	Bean method can perform the following operations
ejbTimeout	EntityContext methods: *getEJBHome, getEJBLocalHome, getCallerPrincipal, isCallerInRole, getRollbackOnly, setRollbackOnly, getEJBObject, getEJBLocalObject, getPrimaryKey, getTimerService, lookup* JNDI access to java:comp/env Resource manager access Enterprise bean access Timer service or Timer methods

Additional restrictions:

- The `getRollbackOnly` and `setRollbackOnly` methods of the `EntityContext` interface should be used only in the enterprise bean methods that execute in the context of a transaction. The container must throw the `java.lang.IllegalStateException` if the methods are invoked while the instance is not associated with a transaction.

Reasons for disallowing operations:

- Invoking the `getEJBObject`, `getEJBLocalObject`, and `getPrimaryKey` methods is disallowed in the entity bean methods in which there is no entity object identity associated with the instance.
- Invoking the `getEJBObject` and `getEJBHome` methods is disallowed if the entity bean does not define a remote client view.
- Invoking the `getEJBLocalObject` and `getEJBLocalHome` methods is disallowed if the entity bean does not define a local client view.
- Invoking the `getRollbackOnly` and `setRollbackOnly` methods is disallowed in the entity bean methods for which the container does not have a meaningful transaction context. These are the methods that have the `NotSupported`, `Never`, or `Supports` transaction attribute.
- Accessing resource managers and enterprise beans is disallowed in the entity bean methods for which the container does not have a meaningful transaction context or client security context.

9.1.8 Caching of Entity State and the ejbLoad and ejbStore Methods

An instance of an entity bean with bean-managed persistence can cache the entity object's state between business method invocations. An instance may choose to cache the entire entity object's state, part of the state, or no state at all.

The container-invoked `ejbLoad` and `ejbStore` methods assist the instance with the management of the cached entity object's state. The instance should handle the `ejbLoad` and `ejbStore` methods as follows:

- When the container invokes the `ejbStore` method on the instance, the instance should push all cached updates of the entity object's state to the underlying database. The container invokes the `ejbStore` method at the end of a transaction[40], and may also invoke it at other times when the instance is in the ready state. (For example the container may invoke `ejbStore` when passivating an instance in the middle of a transaction, or when transferring the instance's state to another instance to support distributed transactions in a multi-process server.)

- When the container invokes the `ejbLoad` method on the instance, the instance should discard any cached entity object's state. The instance may, but is not required to, refresh the cached state by reloading it from the underlying database.

The following examples, which are illustrative but not prescriptive, show how an instance may cache the entity object's state:

- An instance loads the entire entity object's state in the `ejbLoad` method and caches it until the container invokes the `ejbStore` method. The business methods read and write the cached entity state. The `ejbStore` method writes the updated parts of the entity object's state to the database.

- An instance loads the most frequently used part of the entity object's state in the `ejbLoad` method and caches it until the container invokes the `ejbStore` method. Additional parts of the entity object's state are loaded as needed by the business methods. The `ejbStore` method writes the updated parts of the entity object's state to the database.

- An instance does not cache any entity object's state between business methods. The business methods access and modify the entity object's state directly in the database. The `ejbLoad` and `ejbStore` methods have an empty implementation.

We expect that most entity developers will not manually code the cache management and data access calls in the entity bean class. We expect that they will rely on application development tools to provide various data access components that encapsulate data access and provide state caching.

9.1.8.1 ejbLoad and ejbStore with the NotSupported Transaction Attribute

The use of the `ejbLoad` and `ejbStore` methods for caching an entity object's state in the instance works well only if the container can use transaction boundaries to drive the `ejbLoad` and `ejbStore` methods. When the `NotSupported`[41] transaction attribute is assigned to a component interface method, the corresponding enterprise bean class method executes with an unspecified transaction context (See Subsection 12.6.5). This means that the container does not have any well-defined transaction boundaries to drive the `ejbLoad` and `ejbStore` methods on the instance.

---

[40] This call may be omitted if the bean has been specified as read-only.

[41] This applies also to the `Never` and `Supports` attribute.

Therefore, the `ejbLoad` and `ejbStore` methods are “unreliable” for the instances that the container uses to dispatch the methods with an unspecified transaction context. The following are the only guarantees that the container provides for the instances that execute the methods with an unspecified transaction context:

- The container invokes at least one `ejbLoad` between `ejbActivate` and the first business method in the instance.
- The container invokes at least one `ejbStore` between the last business method on the instance and the `ejbPassivate` method[42].

Because the entity object’s state accessed between the `ejbLoad` and `ejbStore` method pair is not protected by a transaction boundary for the methods that execute with an unspecified transaction context, the Bean Provider should not attempt to use the `ejbLoad` and `ejbStore` methods to control caching of the entity object’s state in the instance. Typically, the implementation of the `ejbLoad` and `ejbStore` methods should be a no-op (i.e., an empty method), and each business method should access the entity object’s state directly in the database.

9.1.9 Finder Method Return Type

9.1.9.1 Single-Object Finder

Some finder methods (such as `ejbFindByPrimaryKey`) are designed to return at most one entity object. For single-object finders, the result type of a `find<METHOD>`method defined in the entity bean’s remote home interface is the entity bean’s remote interface, and the result type of the `find<METHOD>`method defined in the entity bean’s local home interface is the entity bean’s local interface. The result type of the corresponding `ejbFind<METHOD>` method defined in the entity’s implementation class is the entity bean’s primary key type.

The following code illustrates the definition of a single-object finder on the remote home interface.

```
// Entity's home interface
public AccountHome extends javax.ejb.EJBHome {
     ...
     Account findByPrimaryKey(AccountPrimaryKey primkey)
          throws FinderException, RemoteException;
     ...
}
```

[42] This ejbStore call may be omitted if the bean has been specified as read-only.

Note that a finder method defined on the local home interface, however, must not throw the `RemoteException`.

```
// Entity's implementation class
public AccountBean implements javax.ejb.EntityBean {
    ...
    public AccountPrimaryKey ejbFindByPrimaryKey(
             AccountPrimaryKey primkey)
        throws FinderException
    {
        ...
    }
    ...
}
```

9.1.9.2 Multi-Object Finders

Some finder methods are designed to return multiple entity objects. For multi-object finders defined in the entity bean's remote home interface, the result type of the `find<METHOD>`method is a collection of objects implementing the entity bean's remote interface. For multi-object finders defined in the entity bean's local home interface, the result type is a collection of objects implementing the entity bean's local interface. In either case, the result type of the corresponding `ejbFind<METHOD>` implementation method defined in the entity bean's implementation class is a collection of objects of the entity bean's primary key type.

The Bean Provider can choose two types to define a collection type for a finder:

- the Java™ 2 `java.util.Collection` interface
- the JDK™ 1.1 `java.util.Enumeration` interface

A Bean Provider targeting containers and clients based on Java 2 should use the `java.util.Collection` interface for the finder's result type.

A Bean Provider who wants to ensure that the entity bean is compatible with containers and clients based on JDK 1.1 must use the `java.util.Enumeration` interface for the finder's result type[43].

The Bean Provider must ensure that the objects in the `java.util.Enumeration` or `java.util.Collection` returned from the `ejbFind<METHOD>` method are instances of the entity bean's primary key class.

A client program must use the `PortableRemoteObject.narrow` method to convert the objects contained in the collections returned by a finder method on the entity bean's remote home interface to the entity bean's remote interface type.

[43] The finder will be also compatible with Java 2-based containers and clients.

The following is an example of a multi-object finder method definition that is compatible with containers and clients based on Java 2:

```
// Entity's remote home interface
public AccountHome extends javax.ejb.EJBHome {
     ...
     java.util.Collection findLargeAccounts(double limit)
          throws FinderException, RemoteException;
     ...
}

// Entity's implementation class
public AccountBean implements javax.ejb.EntityBean {
     ...
     public java.util.Collection ejbFindLargeAccounts(
               double limit) throws FinderException
     {
          ...
     }
     ...
}
```

The following is an example of a multi-object finder method definition compatible with containers and clients that are based on both JDK 1.1 and Java 2:

```
// Entity's remote home interface
public AccountHome extends javax.ejb.EJBHome {
     ...
     java.util.Enumeration findLargeAccounts(double limit)
          throws FinderException, RemoteException;
     ...
}

// Entity's implementation class
public AccountBean implements javax.ejb.EntityBean {
     ...
     public java.util.Enumeration ejbFindLargeAccounts(
               double limit) throws FinderException
     {
          ...
     }
     ...
}
```

9.1.10 Timer Notifications

An entity bean can be registered with the EJB Timer Service for time-based event notifications if it implements the `javax.ejb.TimedObject` interface. The container invokes the bean instance's `ejbTimeout` method when a timer for the bean has expired. See Chapter 17, "Timer Service".

9.1.11 Standard Application Exceptions for Entities

The EJB specification defines the following standard application exceptions:

- `javax.ejb.CreateException`
- `javax.ejb.DuplicateKeyException`
- `javax.ejb.FinderException`
- `javax.ejb.ObjectNotFoundException`
- `javax.ejb.RemoveException`

9.1.11.1 CreateException

From the client's perspective, a `CreateException` (or a subclass of `CreateException`) indicates that an application level error occurred during the `create<METHOD>` operation. If a client receives this exception, the client does not know, in general, whether the entity object was created but not fully initialized, or not created at all. Also, the client does not know whether or not the transaction has been marked for rollback. (However, the client may determine the transaction status using the `UserTransaction` interface or the `setRollbackOnly` method of the `EJBContext` interface.)

The Bean Provider throws the `CreateException` (or subclass of `CreateException`) from the `ejbCreate<METHOD>` and `ejbPostCreate<METHOD>` methods to indicate an application-level error from the create or initialization operation. Optionally, the Bean Provider may mark the transaction for rollback before throwing this exception.

The Bean Provider is encouraged to mark the transaction for rollback only if data integrity would be lost if the transaction were committed by the client. Typically, when a `CreateException` is thrown, it leaves the database in a consistent state, allowing the client to recover. For example, `ejbCreate` may throw the `CreateException` to indicate that the some of the arguments to the `create<METHOD>` method are invalid.

The container treats the `CreateException` as any other application exception. See Section 13.3.

9.1.11.2 DuplicateKeyException

The `DuplicateKeyException` is a subclass of `CreateException`. It is thrown by the `ejbCreate<METHOD>` method to indicate to the client that the entity object cannot be created because an entity object with the same key already exists. The unique key causing the violation may be the primary key, or another key defined in the underlying database.

Normally, the Bean Provider should not mark the transaction for rollback before throwing the exception.

When the client receives the `DuplicateKeyException`, the client knows that the entity was not created, and that the client's transaction has not typically been marked for rollback.

9.1.11.3 FinderException

From the client's perspective, a `FinderException` (or a subclass of `FinderException`) indicates that an application level error occurred during the `find` operation. Typically, the client's transaction has not been marked for rollback because of the `FinderException`.

The Bean Provider throws the `FinderException` (or subclass of `FinderException`) from the `ejbFind<METHOD>` method to indicate an application-level error in the finder method. The Bean Provider should not, typically, mark the transaction for rollback before throwing the `FinderException`.

The container treats the `FinderException` as any other application exception. See Section 13.3.

9.1.11.4 ObjectNotFoundException

The `ObjectNotFoundException` is a subclass of `FinderException`. It is thrown by the `ejbFind<METHOD>` method to indicate that the requested entity object does not exist.

Only single-object finders (see Subsection 9.1.9) should throw this exception. Multi-object finders must not throw this exception. Multi-object finders should return an empty collection as an indication that no matching objects were found.

9.1.11.5 RemoveException

From the client's perspective, a `RemoveException` (or a subclass of `RemoveException`) indicates that an application level error occurred during a `remove` operation. If a client receives this exception, the client does not know, in general, whether the entity object was removed or not. The client also does not know if the transaction has been marked for rollback. (However, the client may determine the transaction status using the `UserTransaction` interface.)

The Bean Provider throws the `RemoveException` (or subclass of `RemoveException`) from the `ejbRemove` method to indicate an application-level error from the entity object removal operation. Optionally, the Bean Provider may mark the transaction for rollback before throwing this exception.

The Bean Provider is encouraged to mark the transaction for rollback only if data integrity would be lost if the transaction were committed by the client. Typically, when a `RemoveException` is thrown, it leaves the database in a consistent state, allowing the client to recover.

The container treats the `RemoveException` as any other application exception. See Section 13.3.

9.1.12 Commit Options

The Entity Bean protocol is designed to give the container the flexibility to select the disposition of the instance state at transaction commit time. This flexibility allows the container to optimally manage the caching of entity object's state and the association of an entity object identity with the enterprise bean instances.

The container can select from the following commit-time options:

- **Option A**: The container caches a "ready" instance between transactions. The container ensures that the instance has exclusive access to the state of the object in the persistent storage. Therefore, the container does not have to synchronize the instance's state from the persistent storage at the beginning of the next transaction.

- **Option B**: The container caches a "ready" instance between transactions. In contrast to Option A, in this option the container does not ensure that the instance has exclusive access to the state of the object in the persistent storage. Therefore, the container must synchronize the instance's state from the persistent storage at the beginning of the next transaction.

- **Option C**: The container does not cache a "ready" instance between transactions. The container returns the instance to the pool of available instances after a transaction has completed.

The following table provides a summary of the commit-time options.

Table 8 Summary of Commit-Time Options

	Write instance state to database	**Instance stays ready**	**Instance state remains valid**
Option A	Yes	Yes	Yes
Option B	Yes	Yes	No
Option C	Yes	No	No

Note that the container synchronizes the instance's state with the persistent storage at transaction commit for all three options.

The selection of the commit option is transparent to the entity bean implementation—the entity bean will work correctly regardless of the commit-time option chosen by the container. The Bean Provider writes the entity bean in the same way.

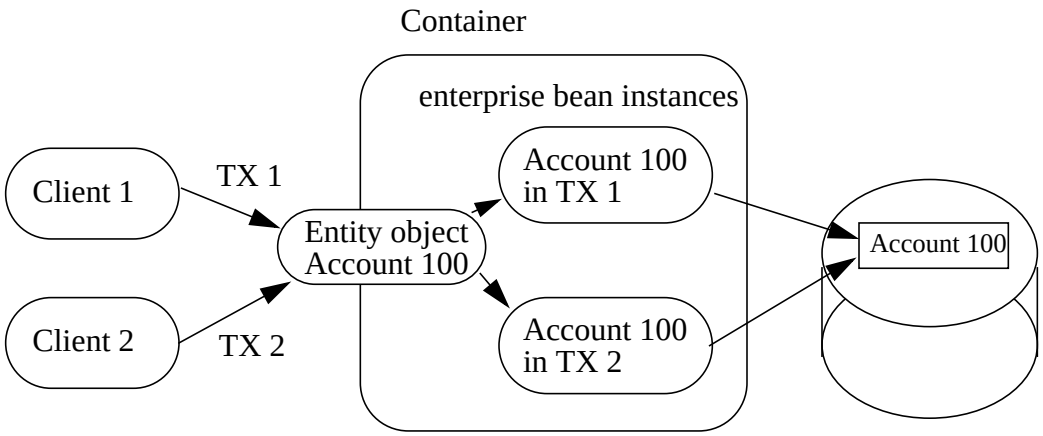
9.1.13 Concurrent Access from Multiple Transactions

When writing the entity bean business methods, the Bean Provider does not have to worry about concurrent access from multiple transactions. The Bean Provider may assume that the container will ensure appropriate synchronization for entity objects that are accessed concurrently from multiple transactions.

The container typically uses one of the following implementation strategies to achieve proper synchronization. (These strategies are illustrative, not prescriptive.)

- The container activates multiple instances of the entity bean, one for each transaction in which the entity object is being accessed. The transaction synchronization is performed automatically by the underlying database during the database access calls performed by the business methods and the `ejbTimeout` method; and by the `ejbLoad`, `ejbCreate<METHOD>`, `ejbStore`, and `ejbRemove` methods. The database system provides all the necessary transaction synchronization; the container does not have to perform any synchronization logic.

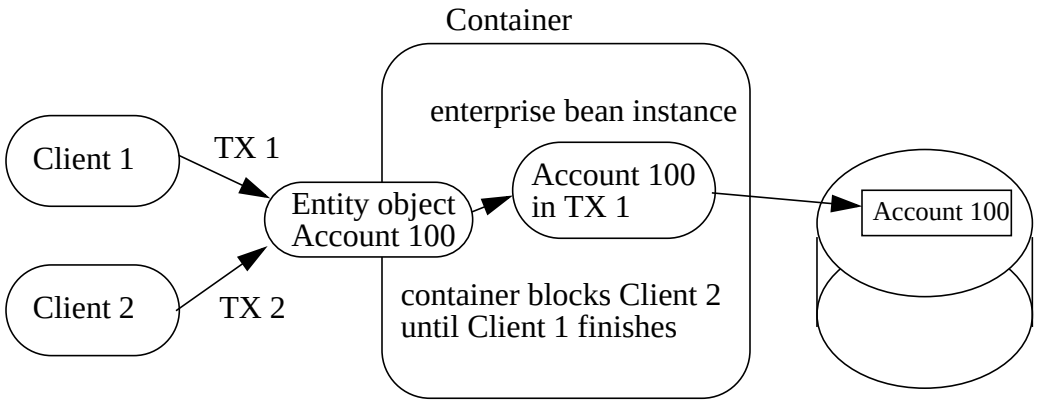
Figure 20 Multiple Clients Can Access the Same Entity Object Using Multiple Instances



With this strategy, the type of lock acquired by `ejbLoad` *leads to a trade-off. If* `ejbLoad` *acquires an exclusive lock on the instance's state in the database, then throughput of read-only transactions could be impacted. If* `ejbLoad` *acquires a shared lock and the instance is updated, then* `ejbStore` *will need to promote the lock to an exclusive lock. This may cause a deadlock if it happens concurrently under multiple transactions.*

- The container acquires exclusive access to the entity object's state in the database. The container activates a single instance and serializes the access from multiple transactions to this instance. The commit-time option A applies to this type of container.

Figure 21 Multiple Clients Can Access the Same Entity Object Using Single Instance



9.1.14 Non-reentrant and Re-entrant Instances

An entity Bean Provider can specify that an entity bean is non-reentrant. If an instance of a non-reentrant entity bean executes a client request in a given transaction context, and another request with the same transaction context arrives for the same entity object, the container will throw an exception to the second request. This rule allows the Bean Provider to program the entity bean as single-threaded, non-reentrant code.

The functionality of some entity beans may require loopbacks in the same transaction context. An example of a loopback is when the client calls entity object A, A calls entity object B, and B calls back A in the same transaction context. The entity bean's method invoked by the loopback shares the current execution context (which includes the transaction and security contexts) with the bean's method invoked by the client.

If the entity bean is specified as non-reentrant in the deployment descriptor, the container must reject an attempt to re-enter the instance via the entity bean's component interface while the instance is executing a business method. (This can happen, for example, if the instance has invoked another enterprise bean, and the other enterprise bean tries to make a loopback call.) If the attempt is made to reenter the instance through the remote interface, the container must throw the `java.rmi.RemoteException` to the caller. If the attempt is made to reenter the instance through the local interface, the container must throw the `javax.ejb.EJBException` to the caller. The container must allow the call if the bean's deployment descriptor specifies that the entity bean is re-entrant.

Re-entrant entity beans must be programmed and used with caution. First, the Bean Provider must code the entity 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 a concurrent call in the same transaction context.

Concurrent calls in the same transaction context targeted at the same entity 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 should be marked as non-reentrant in the deployment descriptor, allowing the container to detect and prevent illegal concurrent calls from clients.

9.2 Responsibilities of the Enterprise Bean Provider

This section describes the responsibilities of a bean-managed persistence entity Bean Provider to ensure that the entity bean can be deployed in any EJB container.

9.2.1 Classes and Interfaces

The Bean Provider is responsible for providing the following class files:

- Entity bean class and any dependent classes
- Primary key class
- Entity bean's remote interface and remote home interface, if the entity bean provides a remote client view
- Entity bean's local interface and local home interface, if the entity bean provides a local client view

The Bean Provider must provide a remote interface and a remote home interface or a local interface an local home interface for the bean. The Bean Provider may provide a remote interface, remote home interface, local interface, and local home interface for the bean. Other combinations are not allowed.

9.2.2 Enterprise Bean Class

The following are the requirements for an entity bean class:

The class must implement, directly or indirectly, the `javax.ejb.EntityBean` interface.

The class may implement, directly or indirectly, the `javax.ejb.TimedObject` interface.

The class must be defined as `public` and must not be `abstract`. The class must be a top level class.

The class must not be defined as `final`.

The class must define a public constructor that takes no arguments.

The class must not define the `finalize` method.

The class may, but is not required to, implement the entity bean's component interface[44]. If the class implements the entity bean's component interface, the class must provide no-op implementations of the methods defined in the `javax.ejb.EJBObject` or `javax.ejb.EJBLocalObject` interface. The container will never invoke these methods on the bean instances at runtime.

A no-op implementation of these methods is required to avoid defining the entity bean class as abstract.

The entity bean class must implement the business methods, and the `ejbCreate<METHOD>`, `ejbPostCreate<METHOD>`, `ejbFind<METHOD>`, and `ejbHome<METHOD>` methods as described later in this section.

The entity bean class may have superclasses and/or superinterfaces. If the entity bean has superclasses, the business methods, the `ejbCreate and ejbPostCreate` methods, the finder methods, and the methods of the `EntityBean` interface or the `TimedObject` interface may be implemented in the enterprise bean class or in any of its superclasses.

The entity bean class is allowed to implement other methods (for example helper methods invoked internally by the business methods) in addition to the methods required by the EJB specification.

9.2.3 ejbCreate<METHOD> Methods

The entity bean class must implement the `ejbCreate<METHOD>` methods that correspond to the `create<METHOD>` methods specified in the entity bean's home interface.

The entity bean class may define zero or more `ejbCreate<METHOD>` methods whose signatures must follow these rules:

The method name must have `ejbCreate` as its prefix.

The method must be declared as `public`.

The method must not be declared as `final` or `static`.

The return type must be the entity bean's primary key type.

The method argument and return value types must be legal types for RMI-IIOP if the `ejbCreate<METHOD>` corresponds to a `create<METHOD>` on the entity bean's remote home interface.

The `throws` clause may define arbitrary application specific exceptions, including the `javax.ejb.CreateException`.

[44] If the entity bean class does implement the component interface, care must be taken to avoid passing of `this` as a method argument or result. This potential error can be avoided by choosing not to implement the component interface in the entity bean class.

Compatibility Note: EJB 1.0 allowed the `ejbCreate` *method to throw the* `java.rmi.RemoteException` *to indicate a non-application exception. This practice was deprecated in EJB 1.1—an EJB 1.1 or EJB 2.0 or later compliant enterprise bean should throw the* `javax.ejb.EJBException` *or another* `java.lang.RuntimeException` *to indicate non-application exceptions to the container (see Section 13.2.2). An EJB 2.0 or later enterprise bean should not throw the* `java.rmi.RemoteException`*. from the* `ejbCreate` *method.*

The entity object created by the `ejbCreate<METHOD>` method must have a unique primary key. This means that the primary key must be different from the primary keys of all the existing entity objects within the same home. The `ejbCreate<METHOD>` method should throw the `DuplicateKeyException` on an attempt to create an entity object with a duplicate primary key. However, it is legal to reuse the primary key of a previously removed entity object.

9.2.4 ejbPostCreate<METHOD> Methods

For each `ejbCreate<METHOD>` method, the entity bean class must define a matching `ejbPostCreate<METHOD>` method, using the following rules:

The method name must have `ejbPostCreate` as its prefix.

The method must be declared as `public`.

The method must not be declared as `final` or `static`.

The return type must be `void`.

The method arguments must be the same as the arguments of the matching `ejbCreate<METHOD>` method.

The `throws` clause may define arbitrary application specific exceptions, including the `javax.ejb.CreateException`.

Compatibility Note: EJB 1.0 allowed the `ejbPostCreate` *method to throw the* `java.rmi.RemoteException` *to indicate a non-application exception. This practice was deprecated in EJB 1.1—an EJB 1.1 or EJB 2.0 or later compliant enterprise bean should throw the* `javax.ejb.EJBException` *or another* `java.lang.RuntimeException` *to indicate non-application exceptions to the container (see Section 13.2.2). An EJB 2.0 or later enterprise bean should not throw the* `java.rmi.RemoteException`*. from the* `ejbPostCreate` *method.*

9.2.5 ejbFind Methods

The entity bean class may also define additional `ejbFind<METHOD>` finder methods.

The signatures of the finder methods must follow the following rules:

A finder method name must start with the prefix "`ejbFind`" (e.g. `ejbFindByPrimaryKey`, `ejbFindLargeAccounts`, `ejbFindLateShipments`).

A finder method must be declared as `public`.

The method must not be declared as `final` or `static`.

The method argument types must be legal types for RMI-IIOP if the `ejbFind<METHOD>` method corresponds to a `find<METHOD>` method on the entity bean's remote home interface.

The return type of a finder method must be the entity bean's primary key type, or a collection of primary keys (see Subsection 9.1.9).

The `throws` clause may define arbitrary application specific exceptions, including the `javax.ejb.FinderException`.

Compatibility Note: EJB 1.0 allowed the finder methods to throw the `java.rmi.RemoteException` to indicate a non-application exception. This practice was deprecated in EJB 1.1—an EJB 1.1 or EJB 2.0 or later compliant enterprise bean should throw the `javax.ejb.EJBException` or another `java.lang.RuntimeException` to indicate non-application exceptions to the container (see Section 13.2.2). An EJB 2.0 or later enterprise bean should not throw the `java.rmi.RemoteException`. from the `ejbFind` method.

Every entity bean must define the `ejbFindByPrimaryKey` method. The result type for this method must be the primary key type (i.e., the `ejbFindByPrimaryKey` method must be a single-object finder).

9.2.6 ejbHome<METHOD> Methods

The entity bean class may define zero or more home methods whose signatures must follow the following rules:

An `ejbHome<METHOD>` method must exist for every home `<METHOD>` method on the entity bean's remote home or local home interface. The method name must have `ejbHome` as its prefix followed by the name of the `<METHOD>` method in which the first character has been uppercased.

The method must be declared as `public`.

The method must not be declared as `static`.

The method argument and return value types must be legal types for RMI-IIOP if the `ejbHome` method corresponds to a method on the entity bean's remote home interface.

The `throws` clause may define arbitrary application specific exceptions. The `throws` clause must not throw the `java.rmi.RemoteException`.

9.2.7 Business Methods

The entity bean class may define zero or more business methods whose signatures must follow these rules:

The method names can be arbitrary, but they must not start with '`ejb`' to avoid conflicts with the callback methods used by the EJB architecture.

The business method must be declared as `public`.

The method must not be declared as `final` or `static`.

The method argument and return value types must be legal types for RMI-IIOP if the method corresponds to a business method on the entity bean's remote interface.

The `throws` clause may define arbitrary application specific exceptions.

Compatibility Note: EJB 1.0 allowed the business methods to throw the `java.rmi.RemoteException` to indicate a non-application exception. This practice was deprecated in EJB 1.1—an EJB 1.1 or EJB 2.0 or later compliant enterprise bean should throw the `javax.ejb.EJBException` or another `java.lang.RuntimeException` to indicate non-application exceptions to the container (see Section 13.2.2). An EJB 2.0 or later enterprise bean should not throw the `java.rmi.RemoteException`. from a business method.

9.2.8 Entity Bean's Remote Interface

The following are the requirements for the entity bean's remote interface:

The interface must extend the `javax.ejb.EJBObject` interface.

The methods defined in the remote interface must follow the rules for RMI-IIOP. This means that their argument and return value types must be valid types for RMI-IIOP, and their `throws` clauses must include the `java.rmi.RemoteException`.

The remote interface is allowed to have superinterfaces. Use of interface inheritance is subject to the RMI-IIOP rules for the definition of remote interfaces.

For each method defined in the remote interface, there must be a matching method in the entity bean's class. The matching method must have:

- The same name.
- The same number and types of its arguments, and the same return type.
- All the exceptions defined in the `throws` clause of the matching method of the enterprise bean class must be defined in the `throws` clause of the method of the remote interface.

The remote interface methods must not expose local interface types, local home interface types, timers or timer handles, or the managed collection classes that are used for entity beans with container-managed persistence as arguments or results.

9.2.9 Entity Bean's Remote Home Interface

The following are the requirements for the entity bean's remote home interface:

The interface must extend the `javax.ejb.EJBHome` interface.

The methods defined in this interface must follow the rules for RMI-IIOP. This means that their argument and return types must be of valid types for RMI-IIOP, and that their `throws` clauses must include the `java.rmi.RemoteException`.

The remote home interface is allowed to have superinterfaces. Use of interface inheritance is subject to the RMI-IIOP rules for the definition of remote interfaces.

Each method defined in the remote home interface must be one of the following:

- A create method.
- A finder method.
- A home method.

Each `create` method must be the named "`create<METHOD>`", and it must match one of the `ejbCreate<METHOD>` methods defined in the enterprise bean class. The matching `ejbCreate<METHOD>` method must have the same number and types of its arguments. (Note that the return type is different.)

The return type for a `create<METHOD>` method must be the entity bean's remote interface type.

All the exceptions defined in the `throws` clause of the matching `ejbCreate<METHOD>` and `ejbPostCreate<METHOD>` methods of the enterprise bean class must be included in the `throws` clause of the matching `create<METHOD>` method of the remote home interface (i.e., the set of exceptions defined for the `create<METHOD>` method must be a superset of the union of exceptions defined for the `ejbCreate<METHOD>` and `ejbPostCreate<METHOD>` methods).

The `throws` clause of a `create<METHOD>` method must include the `javax.ejb.CreateException`.

Each `finder` method must be named "`find<METHOD>`" (e.g. `findLargeAccounts`), and it must match one of the `ejbFind<METHOD>` methods defined in the entity bean class (e.g. `ejbFindLargeAccounts`). The matching `ejbFind<METHOD>` method must have the same number and types of arguments. (Note that the return type may be different.)

The return type for a `find<METHOD>` method must be the entity bean's remote interface type (for a single-object finder), or a collection thereof (for a multi-object finder).

The remote home interface must always include the `findByPrimaryKey` method, which is always a single-object finder. The method must declare the primary key class as the method argument.

All the exceptions defined in the `throws` clause of an `ejbFind` method of the entity bean class must be included in the `throws` clause of the matching `find` method of the remote home interface.

The `throws` clause of a `finder` method must include the `javax.ejb.FinderException`.

Home methods can have arbitrary names, provided that they do not clash with `create`, `find`, and `remove` method names. The matching `ejbHome` method specified in the entity bean class must have the same number and types of arguments and must return the same type as the home method as specified in the remote home interface of the bean.

The remote home interface methods must not expose local interface types, local home interface types, timer handles, or the managed collection classes that are used for entity beans with container-managed persistence as arguments or results.

9.2.10 Entity Bean's Local Interface

The following are the requirements for the entity bean's local interface:

The interface must extend the `javax.ejb.EJBLocalObject` interface.

The `throws` clause of a method defined on the local interface must not include the `java.rmi.RemoteException`.

The local interface is allowed to have superinterfaces.

For each method defined in the local interface, there must be a matching method in the entity bean's class. The matching method must have:

- The same name.
- The same number and types of its arguments, and the same return type.
- All the exceptions defined in the `throws` clause of the matching method of the enterprise Bean class must be defined in the `throws` clause of the method of the local interface.

9.2.11 Entity Bean's Local Home Interface

The following are the requirements for the entity bean's local home interface:

The interface must extend the `javax.ejb.EJBLocalHome` interface.

The `throws` clause of a method on the local home interface must not include the `java.rmi.RemoteException`.

The local home interface is allowed to have superinterfaces.

Each method defined in the local home interface must be one of the following:

- A create method.
- A finder method.
- A home method.

Each `create` method must be the named "`create<METHOD>`", and it must match one of the `ejbCreate<METHOD>` methods defined in the enterprise bean class. The matching `ejbCreate<METHOD>` method must have the same number and types of its arguments. (Note that the return type is different.)

The return type for a `create<METHOD>` method must be the entity bean's local interface type.

All the exceptions defined in the `throws` clause of the matching `ejbCreate<METHOD>` and `ejbPostCreate<METHOD>` methods of the enterprise bean class must be included in the `throws` clause of the matching `create<METHOD>` method of the local home interface (i.e., the set of exceptions defined for the `create<METHOD>` method must be a superset of the union of exceptions defined for the `ejbCreate<METHOD>` and `ejbPostCreate<METHOD>` methods).

The `throws` clause of a `create<METHOD>` method must include the `javax.ejb.CreateException`.

Each `finder` method must be named "`find<METHOD>`" (e.g. `findLargeAccounts`), and it must match one of the `ejbFind<METHOD>` methods defined in the entity bean class (e.g. `ejbFindLargeAccounts`). The matching `ejbFind<METHOD>` method must have the same number and types of arguments. (Note that the return type may be different.)

The return type for a `find<METHOD>` method must be the entity bean's local interface type (for a single-object finder), or a collection thereof (for a multi-object finder).

The local home interface must always include the `findByPrimaryKey` method, which is always a single-object finder. The method must declare the primary key class as the method argument.

All the exceptions defined in the `throws` clause of an `ejbFind` method of the entity bean class must be included in the `throws` clause of the matching `find` method of the local home interface.

The `throws` clause of a `finder` method must include the `javax.ejb.FinderException`.

Home methods can have arbitrary names, provided that they do not clash with `create`, `find`, and `remove` method names. The matching `ejbHome` method specified in the entity bean class must have the same number and types of arguments and must return the same type as the home method as specified in the local home interface of the bean.

The `throws` clause of any method on the entity bean's local home interface must not include the `java.rmi.RemoteException`.

9.2.12 Entity Bean's Primary Key Class

The Bean Provider must specify a primary key class in the deployment descriptor.

The primary key type must be a legal Value Type in RMI-IIOP.

The class must provide suitable implementation of the `hashCode()` and `equals(Object other)` methods to simplify the management of the primary keys by client code.

9.3 The Responsibilities of the Container Provider

This section describes the responsibilities of the Container Provider to support bean-managed persistence entity beans. The Container Provider is responsible for providing the deployment tools, and for managing entity bean instances at runtime.

Because the EJB specification does not define the API between deployment tools and the container, we assume that the deployment tools are provided by the Container Provider. Alternatively, the deployment tools may be provided by a different vendor who uses the container vendor's specific API.

9.3.1 Generation of Implementation Classes

The deployment tools provided by the Container Provider are responsible for the generation of additional classes when the entity bean is deployed. The tools obtain the information that they need for generation of the additional classes by introspecting the classes and interfaces provided by the entity Bean Provider and by examining the entity bean's deployment descriptor.

The deployment tools must generate the following classes:

- A class that implements the entity bean's remote home interface (i.e., the entity EJBHome class).
- A class that implements the entity bean's remote interface (i.e., the entity EJBObject class).
- A class that implements the entity bean's local home interface (i.e., the entity EJBLocalHome class).
- A class that implements the entity bean's local interface (i.e., the entity EJBLocalObject class).

The deployment tools may also generate a class that mixes some container-specific code with the entity bean class. The code may, for example, help the container to manage the entity bean instances at runtime. Tools can use subclassing, delegation, and code generation.

The deployment tools may also allow generation of additional code that wraps the business methods and that is used to customize the business logic for an existing operational environment. For example, a wrapper for a `debit` function on the `Account` bean may check that the debited amount does not exceed a certain limit, or perform security checking that is specific to the operational environment.

9.3.2 Entity EJBHome Class

The entity EJBHome class, which is generated by deployment tools, implements the entity bean's remote home interface. This class implements the methods of the `javax.ejb.EJBHome` interface, and the type-specific create, finder, and home methods specific to the entity bean.

The implementation of each `create<METHOD>` method invokes a matching `ejbCreate<METHOD>` method, followed by the matching `ejbPostCreate<METHOD>` method, passing the `create<METHOD>` parameters to these matching methods.

The implementation of the `remove` methods defined in the `javax.ejb.EJBHome` interface must activate an instance (if an instance is not already in the ready state) and invoke the `ejbRemove` method on the instance.

The implementation of each `find<METHOD>` method invokes a matching `ejbFind<METHOD>` method. The implementation of the `find<METHOD>` method must create an entity object reference for the primary key returned from the `ejbFind<METHOD>` and return the entity object reference (i.e., EJBObject) to the client. If the `ejbFind<METHOD>` method returns a collection of primary keys, the implementation of the `find<METHOD>` method must create a collection of entity object references for the primary keys and return the collection to the client.

The implementation of each `<METHOD>` home method invokes a matching `ejbHome<METHOD>` method (in which the first character of `<METHOD>` is uppercased in the name of the `ejbHome<METHOD>` method), passing the `<METHOD>` parameters to the matching method.

9.3.3 Entity EJBObject Class

The entity EJBObject class, which is generated by deployment tools, implements the entity bean's remote interface. It implements the methods of the `javax.ejb.EJBObject` interface and the business methods specific to the entity bean.

The implementation of the `remove` method (defined in the `javax.ejb.EJBObject` interface) must activate an instance (if an instance is not already in the ready state) and invoke the `ejbRemove` method on the instance.

The implementation of each business method must activate an instance (if an instance is not already in the ready state) and invoke the matching business method on the instance.

9.3.4 Entity EJBLocalHome Class

The entity EJBLocalHome class, which is generated by deployment tools, implements the entity bean's local home interface. This class implements the methods of the `javax.ejb.EJBLocalHome` interface, and the type-specific create, finder, and home methods specific to the entity bean.

The implementation of each `create<METHOD>` method invokes a matching `ejbCreate<METHOD>` method, followed by the matching `ejbPostCreate<METHOD>` method, passing the `create<METHOD>` parameters to these matching methods.

The implementation of the `remove` method defined in the `javax.ejb.EJBLocalHome` interface must activate an instance (if an instance is not already in the ready state) and invoke the `ejbRemove` method on the instance.

The implementation of each `find<METHOD>` method invokes a matching `ejbFind<METHOD>` method. The implementation of the `find<METHOD>` method must create an entity object reference for the primary key returned from the `ejbFind<METHOD>` and return the entity object reference (i.e., EJBLocalObject) to the client. If the `ejbFind<METHOD>` method returns a collection of primary keys, the implementation of the `find<METHOD>` method must create a collection of entity object references for the primary keys and return the collection to the client.

The implementation of each `<METHOD>` home method invokes a matching `ejbHome<METHOD>` method (in which the first character of `<METHOD>` is uppercased in the name of the `ejbHome<METHOD>` method), passing the `<METHOD>` parameters to the matching method.

9.3.5 Entity EJBLocalObject Class

The entity EJBLocalObject class, which is generated by deployment tools, implements the entity bean's local interface. It implements the methods of the `javax.ejb.EJBLocalObject` interface and the business methods specific to the entity bean.

The implementation of the `remove` method (defined in the `javax.ejb.EJBLocalObject` interface) must activate an instance (if an instance is not already in the ready state) and invoke the `ejbRemove` method on the instance.

The implementation of each business method must activate an instance (if an instance is not already in the ready state) and invoke the matching business method on the instance.

9.3.6 Handle Class

The deployment tools are responsible for implementing the handle class for the entity bean. The handle class must be serializable by the Java Serialization protocol.

As the handle class is not entity-bean specific, the container may, but is not required to, use a single class for all deployed entity beans.

9.3.7 Home Handle Class

The deployment tools responsible for implementing the home handle class for the entity bean. The handle class must be serializable by the Java Serialization protocol.

Because the home handle class is not entity-bean specific, the container may, but is not required to, use a single class for the home handles of all deployed entity beans.

9.3.8 Metadata Class

The deployment tools are responsible for implementing the class that provides metadata information to the remote client view contract. The class must be a valid RMI-IIOP Value Type, and must implement the `javax.ejb.EJBMetaData` interface.

Because the metadata class is not entity-bean specific, the container may, but is not required to, use a single class for all deployed enterprise beans.

9.3.9 Instance's Re-entrance

The container runtime must enforce the rules defined in Section 9.1.14.

9.3.10 Transaction Scoping, Security, Exceptions

The container runtime must follow the rules on transaction scoping, security checking, and exception handling described in Chapters 12, 16, and 13.

9.3.11 Implementation of Object References

The container should implement the distribution protocol between the client and the container such that the object references of the remote home and remote interfaces used by entity bean clients are usable for a long period of time. Ideally, a client should be able to use an object reference across a server crash and restart. An object reference should become invalid only when the entity object has been removed, or after a reconfiguration of the server environment (for example, when the entity bean is moved to a different EJB server or container).

The motivation for this is to simplify the programming model for the entity bean client. While the client code needs to have a recovery handler for the system exceptions thrown from the individual method invocations on the home and remote interface, the client should not be forced to re-obtain the object references.

9.3.12 EntityContext

The container must implement the `EntityContext.getEJBObject` method such that the bean instance can use the Java language cast to convert the returned value to the entity bean's remote interface type. Specifically, the bean instance does not have to use the `PortableRemoteObject.narrow` method for the type conversion.

Chapter 10

EJB 1.1 Entity Bean Component Contract for Container-Managed Persistence

This chapter specifies the EJB 1.1 entity bean component contract for container-managed persistence.

EJB 1.1 entity beans are deprecated with this version of the Enterprise JavaBeans specification.

10.1 EJB 1.1 Entity Beans with Container-Managed Persistence

Chapter 9, "EJB 2.1 Entity Bean Component Contract for Bean-Managed Persistence" describes the component contract for entity beans with bean-managed persistence. The contract for an EJB 1.1 entity bean with container-managed persistence is the same as the contract for an entity bean with bean-managed persistence as described in Chapter 9, except for the differences described in this chapter.

An EJB 1.1 entity bean with container-managed persistence cannot have a local interface or local home interface. Use of the local interfaces of other enterprise beans is not supported for an EJB 1.1 entity bean with container-managed persistence.

Use of the EJB Timer Service is not supported for an EJB 1.1 entity bean with container-managed persistence. An EJB 1.1 entity bean with container-managed persistence should not implement the `javax.ejb.TimedObject` *interface. Use of dependency injection, interceptors, and any Java language metadata annotations is not supported for EJB 1.1 entity beans.*

10.1.1 Container-Managed Fields

An EJB 1.1 entity bean with container-managed persistence relies on the Container Provider's tools to generate methods that perform data access on behalf of the entity bean instances. The generated methods transfer data between the entity bean instance's variables and the underlying resource manager at the times defined by the EJB specification. The generated methods also implement the creation, removal, and lookup of the entity object in the underlying database.

An entity bean with container-manager persistence must not code explicit data access—all data access must be deferred to the container.

The EJB 1.1 entity Bean Provider is responsible for using the `cmp-field` elements of the deployment descriptor to declare the instance's fields that the container must load and store at the defined times. The fields must be defined in the entity bean class as `public`, and must not be defined as `transient`.

The container is responsible for transferring data between the entity bean's instance variables and the underlying data source before or after the execution of the `ejbCreate`, `ejbRemove`, `ejbLoad`, and `ejbStore` methods, as described in the following subsections. The container is also responsible for the implementation of the finder methods.

The EJB 2.0 or later deployment descriptor for an EJB 1.1 entity bean with container-managed persistence indicates that the entity bean uses container-managed persistence and that the value of its `cmp-version` element is `1.x`.

The EJB 1.1 component contract does not architect support for relationships for entity beans with container-managed persistence. The EJB 2.0 and later specifications do not support the use of the `cmr-field`, `ejb-relation`, or `query` deployment descriptor elements or their subelements for EJB 1.1 entity beans.

The following requirements ensure that an EJB 1.1 entity bean with container-managed persistence can be deployed in any compliant container.

- The Bean Provider must ensure that the Java types assigned to the container-managed fields are restricted to the following: Java primitive types, Java serializable types, and references of enterprise beans' remote or remote home interfaces.

- The Container Provider may, but is not required to, use Java Serialization to store the container-managed fields in the database. If the container chooses a different approach, the effect should be equivalent to that of Java Serialization. The container must also be capable of persisting references to enterprise beans' remote and remote home interfaces (for example, by storing their handle or primary key).

Although the above requirements allow the Bean Provider to specify almost any arbitrary type for the container-managed fields, we expect that in practice the Bean Provider of EJB 1.1 entity beans with container-managed persistence will use relatively simple Java types, and that most containers will be able to map these simple Java types to columns in a database schema to externalize the entity state in the database, rather than use Java serialization.

If the Bean Provider expects that the container-managed fields will be mapped to database fields, he or she should provide mapping instructions to the Deployer. The mapping between the instance's container-managed fields and the schema of the underlying database manager will be then realized by the data access classes generated by the Container Provider's tools. Because entity beans are typically coarse-grained objects, the content of the container-managed fields may be stored in multiple rows, possibly spread across multiple database tables. These mapping techniques are beyond the scope of the EJB specification, and do not have to be supported by an EJB compliant container. (The container may simply use the Java serialization protocol in all cases).

10.1.2 ejbCreate, ejbPostCreate

With bean-managed persistence, the entity Bean Provider is responsible for writing the code that inserts a record into the database in the `ejbCreate` methods. However, with container-managed persistence, the container performs the database insert after the `ejbCreate` method completes.

The container must ensure that the values of the container-managed fields are set to the Java language defaults (e.g. 0 for integer, `null` for pointers) prior to invoking an `ejbCreate` method on an instance.

The EJB 1.1 entity Bean Provider's responsibility is to initialize the container-managed fields in the `ejbCreate` methods from the input arguments such that when an `ejbCreate` method returns, the container can extract the container-managed fields from the instance and insert them into the database.

The `ejbCreate` methods must be defined to return the primary key class type. The implementation of the `ejbCreate` methods should be coded to return a `null`. The returned value is ignored by the container.

Note: The above requirement is to allow the creation of an entity bean with bean-managed persistence by subclassing an EJB 1.1 entity bean with container-managed persistence. The Java language rules for overriding methods in subclasses requires the signatures of the `ejbCreate` methods in the subclass and the superclass be the same.

The container is responsible for creating the entity object's representation in the underlying database, extracting the primary key fields of the newly created entity object representation in the database, and for creating an entity EJBObject reference for the newly created entity object. The container must establish the primary key before it invokes the `ejbPostCreate` method. The container may create the representation of the entity in the database immediately after `ejbCreate` returns, or it can defer it to a later time (for example to the time after the matching `ejbPostCreate` has been called, or to the end of the transaction).

The container then invokes the matching `ejbPostCreate` method on the instance. The instance can discover the primary key by calling the `getPrimaryKey` method on its entity context object.

The container must invoke `ejbCreate`, perform the database insert operation, and invoke `ejbPostCreate` in the transaction context determined by the transaction attribute of the matching `create` method, as described in subsection 12.6.2.

The container throws the `DuplicateKeyException` if the newly created entity object would have the same primary key as one of the existing entity objects within the same home.

10.1.3 ejbRemove

The container invokes the `ejbRemove` method on an entity bean instance with container-managed persistence in response to a client-invoked `remove` operation on the entity bean's remote home or remote interface.

The entity Bean Provider can use the `ejbRemove` method to implement any actions that must be done before the entity object's representation is removed from the database.

The container synchronizes the instance's state before it invokes the `ejbRemove` method. This means that the state of the instance variables at the beginning of the `ejbRemove` method is the same as it would be at the beginning of a business method.

After `ejbRemove` returns, the container removes the entity object's representation from the database.

The container must perform `ejbRemove` and the database delete operation in the transaction context determined by the transaction attribute of the invoked `remove` method, as described in subsection 12.6.2.

10.1.4 ejbLoad

When the container needs to synchronize the state of an enterprise bean instance with the entity object's state in the database, the container reads the entity object's state from the database into the container-managed fields and then it invokes the `ejbLoad` method on the instance.

The entity Bean Provider can rely on the container's having loaded the container-managed fields from the database just before the container invokes the `ejbLoad` method. The entity bean can use the `ejbLoad` method, for instance, to perform some computation on the values of the fields that were read by the container (for example, uncompressing text fields).

10.1.5 ejbStore

When the container needs to synchronize the state of the entity object in the database with the state of the enterprise bean instance, the container first calls the `ejbStore` method on the instance, and then it extracts the container-managed fields and writes them to the database.

The entity Bean Provider should use the `ejbStore` method to set up the values of the container-managed fields just before the container writes them to the database. For example, the `ejbStore` method may perform compression of text before the text is stored in the database.

10.1.6 Finder Hethods

The entity Bean Provider does not write the finder (`ejbFind<METHOD>`) methods.

The finder methods are generated at the entity bean deployment time using the Container Provider's tools. The tools can, for example, create a subclass of the entity bean class that implements the `ejbFind<METHOD>` methods, or the tools can generate the implementation of the finder methods directly in the class that implements the entity bean's remote home interface.

Note that the `ejbFind<METHOD>` names and parameter signatures of EJB 1.1 entity beans do not provide the container tools with sufficient information for automatically generating the implementation of the finder methods for methods other than `ejbFindByPrimaryKey`. Therefore, the Bean Provider is responsible for providing a description of each finder method. The entity bean Deployer uses container tools to generate the implementation of the finder methods based in the description supplied by the Bean Provider. The EJB1.1 component contract for container-managed persistence does not specify the format of the finder method description.

10.1.7 Home Methods

The EJB1.1 entity bean contract does not support `ejbHome` methods.

10.1.8 Create Methods

The EJB1.1 entity bean contract does not support `create<METHOD>` methods.

10.1.9 Primary Key Type

The container must be able to manipulate the primary key type. Therefore, the primary key type for an entity bean with container-managed persistence must follow the rules in this subsection, in addition to those specified in Subsection 9.2.12.

There are two ways to specify a primary key class for an entity bean with container-managed persistence:

- Primary key that maps to a single field in the entity bean class.
- Primary key that maps to multiple fields in the entity bean class.

The second method is necessary for implementing compound keys, and the first method is convenient for single-field keys. Without the first method, simple types such as `String` would have to be wrapped in a user-defined class.

10.1.9.1 Primary Key that Maps to a Single Field in the Entity Bean Class

The Bean Provider uses the `primkey-field` element of the deployment descriptor to specify the container-managed field of the entity bean class that contains the primary key. The field's type must be the primary key type.

10.1.9.2 Primary Key that Maps to Multiple Fields in the Entity Bean Class

The primary key class must be `public`, and must have a `public` constructor with no parameters.

All fields in the primary key class must be declared as public.

The names of the fields in the primary key class must be a subset of the names of the container-managed fields. (This allows the container to extract the primary key fields from an instance's container-managed fields, and vice versa.)

10.1.9.3 Special Case: Unknown Primary Key Class

In special situations, the entity Bean Provider may choose not to specify the primary key class for an entity bean with container-managed persistence. This case usually happens when the entity bean does not have a natural primary key, and the Bean Provider wants to allow the Deployer to select the primary key fields at deployment time. The entity bean's primary key type will usually be derived from the primary key type used by the underlying database system that stores the entity objects. The primary key used by the database system may not be known to the Bean Provider.

When defining the primary key for the enterprise bean, the Deployer may sometimes need to subclass the entity bean class to add additional container-managed fields (this typically happens for entity beans that do not have a natural primary key, and the primary keys are system-generated by the underlying database system that stores the entity objects).

In this special case, the type of the argument of the `findByPrimaryKey` method must be declared as `java.lang.Object`, and the return value of `ejbCreate` must be declared as `java.lang.Object`. The Bean Provider must specify the primary key class in the deployment descriptor as of the type `java.lang.Object`.

The primary key class is specified at deployment time in the situations when the Bean Provider develops an entity bean that is intended to be used with multiple back-ends that provide persistence, and when these multiple back-ends require different primary key structures.

Use of entity beans with a deferred primary key type specification limits the client application programming model, because the clients written prior to deployment of the entity bean may not use, in general, the methods that rely on the knowledge of the primary key type.

The implementation of the enterprise bean class methods must be done carefully. For example, the methods should not depend on the type of the object returned from `EntityContext.getPrimaryKey`, because the return type is determined by the Deployer after the EJB class has been written.

Chapter 11 Interceptors

Interceptors are used to interpose on business method invocations and lifecycle events that occur on an enterprise bean instance.

11.1 Overview

An interceptor method may be defined on an enterprise bean class or on an interceptor class associated with the bean. An interceptor class is a class—distinct from the bean class itself—whose methods are invoked in response to business method invocations and/or lifecycle events on the bean.

Any number of interceptor classes may be defined for a bean class.

It is possible to carry state across multiple interceptor method invocations for a single business method invocation or lifecycle callback event for a bean in the context data of the `InvocationContext` object.

An interceptor class must have a public no-arg constructor.

The programming restrictions that apply to enterprise bean components apply to interceptors as well. See Section 20.1.2.

Interceptor methods and interceptor classes are defined for a bean by means of metadata annotations or the deployment descriptor. When annotations are used, one or more interceptor classes are denoted using the `Interceptors` annotation on the bean class and/or on business methods of the bean class. If multiple interceptors are defined, the order in which they are invoked is determined by the order in which they are specified in the `Interceptors` annotation, as described in sections 11.3.1 and 11.4.1. The deployment descriptor may be used as an alternative to specify the invocation order of interceptors or to override the order specified in metadata annotations.

Default interceptors may be defined at the level of the ejb-jar file, to apply to all components defined within the ejb-jar. See Section 11.6.

11.2 Interceptor Life Cycle

The lifecycle of an interceptor instance is the same as that of the bean instance with which it is associated. When the bean instance is created, interceptor instances are created for each interceptor class defined for the bean. These interceptor instances are destroyed when the bean instance is removed. In the case of interceptors associated with stateful session beans, the interceptor instances are passivated upon bean instance passivation, and activated when the bean instance is activated. See sections 4.4, 4.5.1, and 5.5.

Both the interceptor instance and the bean instance are created or activated before any of the respective `PostConstruct` or `PostActivate` callbacks are invoked. Any `PreDestroy` and `PrePassivate` callbacks are invoked before the respective destruction or passivation of either the bean instance or interceptor instance.

An interceptor instance may hold state. An interceptor instance may be the target of dependency injection. Dependency injection is performed when the interceptor instance is created, using the naming context of the associated enterprise bean. The `PostConstruct` interceptor callback method is invoked after this dependency injection has taken place on both the interceptor instances and the bean instance.

Interceptors can invoke JNDI, JDBC, JMS, other enterprise beans, and the EntityManager. See Tables 1, 2, 3.

The interceptors for a bean share the enterprise naming context of the bean for whose methods and lifecycle events they are invoked. Annotations and/or XML deployment descriptor elements for dependency injection or for direct JNDI lookup refer to this shared naming context.

The EJBContext object may be injected into an interceptor class. The interceptor may use the `lookup` method of the EJBContext interface to access the bean's JNDI naming context.

The use of an extended persistence context is only allowed for interceptors that are associated with stateful session beans.

11.3 Business Method Interceptors

Interceptor methods may be defined for business methods of sessions beans and for the message listener methods of message-driven beans. Business method interceptor methods are denoted by the `AroundInvoke` annotation or `around-invoke-method` deployment descriptor element.

Only one `AroundInvoke` method may be present on a given class. An `AroundInvoke` method cannot be a business method of the bean class.

`AroundInvoke` methods have the following signature. `AroundInvoke` methods can have public, private, protected, or package level access. An interceptor method must not be declared as `final` or `static`.

```
Object <METHOD>(InvocationContext) throws Exception
```

An `AroundInvoke` method can invoke any component or resource that a business method can.

Business method interceptor method invocations occur within the same transaction and security context as the business method for which they are invoked.

Business method interceptor methods may be defined to apply to business methods individually, rather than to all the business methods of the bean class. See Section 11.7, “Method-level Interceptors”.

11.3.1 Multiple Business Method Interceptor Methods

If multiple interceptor methods are defined for a bean, the following rules governing their invocation order apply. The deployment descriptor may be used to override the interceptor invocation order specified in annotations, as described in section 11.8.

- Default interceptors, if any, are invoked first. Default interceptors can only be specified in the deployment descriptor. Default interceptors are invoked in the order of their specification in the deployment descriptor.
- If there are any interceptor classes defined on the bean class, the interceptor methods defined by those interceptor classes are invoked before any interceptor methods defined on the bean class itself.
- The `AroundInvoke` methods defined on those interceptor classes are invoked in the same order as the specification of the interceptor classes in the `Interceptors` annotation.
- If an interceptor class itself has superclasses, the interceptor methods defined by the interceptor class’s superclasses are invoked before the interceptor method defined by the interceptor class, most general superclass first.
- After the interceptor methods defined on interceptor classes have been invoked, then:
    - If a bean class has superclasses, any `AroundInvoke` methods defined on those superclasses are invoked, most general superclass first.
    - The `AroundInvoke` method, if any, on the bean class itself is invoked.

- If an `AroundInvoke` method is overridden by another method (regardless of whether that method is itself an `AroundInvoke` method), it will not be invoked.

The deployment descriptor may be used to override the interceptor invocation order specified in annotations. See Section 11.8.2.

The `InvocationContext` object provides metadata that enables interceptor methods to control the behavior of the invocation chain, including whether the next method in the chain is invoked and the values of its parameters and result. The use of the `InvocationContext` interface is described in Section 11.5.

11.3.2 Exceptions

Business method interceptor methods may throw runtime exceptions or application exceptions that are allowed in the `throws` clause of the business method.

`AroundInvoke` methods are allowed to catch and suppress exceptions and recover by calling `proceed()`. `AroundInvoke` methods are allowed to throw runtime exceptions or any checked exceptions that the business method allows within its throws clause.

`AroundInvoke` methods run in the same Java call stack as the bean business method. `InvocationContext.proceed()` will throw the same exception as any thrown by the business method unless an interceptor further down the Java call stack has caught it and thrown a different exception. Exceptions and initialization and/or cleanup operations should typically be handled in try/catch/finally blocks around the `proceed()` method.

`AroundInvoke` methods can mark the transaction for rollback by throwing a runtime exception or by calling the EJBContext `setRollbackOnly()` method. `AroundInvoke` methods may cause this rollback before or after `InvocationContext.proceed()` is called.

If a system exception escapes the interceptor chain the bean instance and any associated interceptor instances are discarded. The `PreDestroy` callbacks are not invoked in this case: the interceptor methods in the chain should perform any necessary clean-up operations as the interceptor chain unwinds.

11.4 Interceptors for LifeCycle Event Callbacks

Lifecycle callback interceptor methods may be defined for session beans and message driven beans.

Interceptor methods for lifecycle event callbacks can be defined on an interceptor class and/or directly on the bean class. The `PostConstruct`, `PreDestroy`, `PostActivate`, and `PrePassivate` annotations are used to define an interceptor method for a lifecycle callback event. If the deployment descriptor is used to define interceptors, the `post-construct-method`, `pre-destroy-method`, `post-activate-method`, and `pre-passivate-method` elements are used.

Lifecycle callback interceptor methods and business method interceptor methods may be defined on the same interceptor class.

Lifecycle callback interceptor methods are invoked in an unspecified transaction and security context.

A given class may not have more than one lifecycle callback interceptor method for the same lifecycle event. Any subset or combination of lifecycle callback annotations may be specified on a given class.

A single lifecycle callback interceptor method may be used to interpose on multiple callback events (e.g., `PostConstruct` and `PostActivate`).

Lifecycle callback interceptor methods defined on an interceptor class have the following signature:

```
void <METHOD> (InvocationContext)
```

Lifecycle callback interceptor methods defined on a bean class have the following signature:

```
void <METHOD>()
```

Lifecycle callback interceptor methods can have public, private, protected, or package level access. A lifecycle callback interceptor method must not be declared as `final` or `static`.

Examples:

```
@Stateful public class ShoppingCartBean implements ShoppingCart {
  private float total;
  private Vector productCodes;
  public int someShoppingMethod(){...};
  ...
  @PreDestroy void endShoppingCart() {...};
 }

public class MyInterceptor {
...
     @PostConstruct
     public void any-method-name (InvocationContext ctx) {
          ...
          ctx.proceed();
          ...
     }

     @PreDestroy
     public void any-other-method-name (InvocationContext ctx) {
          ...
          ctx.proceed();
          ...
     }
}
```

11.4.1 Multiple Callback Interceptor Methods for a Life Cycle Callback Event

If multiple callback interceptor methods are defined for a lifecycle event for a bean, the following rules governing their invocation order apply. The deployment descriptor may be used to override the interceptor invocation order specified in annotations, as described in section 11.8.

- Default interceptors, if any, are invoked first. Default interceptors can only be specified in the deployment descriptor. Default interceptors are invoked in the order of their specification in the deployment descriptor.

- If there are any interceptor classes defined on the bean class, the lifecycle callback interceptor methods defined by those interceptor classes are invoked before any lifecycle callback interceptor methods defined on the bean class itself.

- The lifecycle callback interceptor methods defined on those interceptor classes are invoked in the same order as the specification of the interceptor classes in the `Interceptors` annotation.

- If an interceptor class itself has superclasses, the lifecycle callback interceptor methods defined by the interceptor class's superclasses are invoked before the lifecycle callback interceptor method defined by the interceptor class, most general superclass first.

- After the lifecycle callback interceptor methods defined on interceptor classes have been invoked, then:
    - If a bean class has superclasses, any lifecycle callback interceptor methods defined on those superclasses are invoked, most general superclass first.
    - The lifecycle callback interceptor method, if any, on the bean class itself is invoked.

- If a lifecycle callback interceptor method is overridden by another method (regardless of whether that method is itself a lifecycle callback interceptor method (of the same or different type)), it will not be invoked.

The deployment descriptor may be used to override the interceptor invocation order specified in annotations, as described in section 11.8.2.

All lifecycle callback interceptor methods for a given lifecycle event run in the same Java call stack. If there is no corresponding callback method on the bean class (or any of its superclasses), the `InvocationContext.proceed` invocation on the last interceptor method defined on an interceptor class in the chain will be a no-op.

The `InvocationContext` object provides metadata that enables interceptor methods to control the invocation of further methods in the chain. See Section 11.5.

11.4.2 Exceptions

Lifecycle callback interceptor methods may throw system runtime exceptions, but not application exceptions.

A runtime exception thrown by any lifecycle interceptor callback method causes the bean instance and its interceptors to be discarded after the interceptor chain unwinds.

The lifecycle callback interceptor methods for a lifecycle event run in the same Java call stack as the lifecycle callback method on the bean class. `InvocationContext.proceed()` will throw the same exception as any thrown by another lifecycle callback interceptor method or lifecycle callback method on the bean class unless an interceptor further down the Java call stack has caught it and thrown a different exception. A lifecycle callback interceptor method (other than a method on the bean call or its superclasses) may catch an exception thrown by another lifecycle callback interceptor method in the invocation chain, and clean up before returning. Exceptions and initialization and/or cleanup operations should typically be handled in try/catch/finally blocks around the `proceed()` method.

The `PreDestroy` callbacks are not invoked when the bean and the interceptors are discarded as a result of such exceptions: the lifecycle callback interceptor methods in the chain should perform any necessary clean-up operations as the interceptor chain unwinds.

11.5 InvocationContext

The `InvocationContext` object provides metadata that enables interceptor methods to control the behavior of the invocation chain.

```
public interface InvocationContext {
     public Object getBean();
     public Method getMethod();
     public Object[] getParameters();
     public void setParameters(Object[]);
     public java.util.Map getContextData();
     public Object proceed() throws Exception;
}
```

The same `InvocationContext` instance will be passed to each interceptor method for a given business method or lifecycle event interception. This allows an interceptor to save information in the context data property of the `InvocationContext` that can be subsequently retrieved in other interceptors as a means to pass contextual data between interceptors. The contextual data is not sharable across separate business method invocations or lifecycle callback events. The lifecycle of the `InvocationContext` instance is otherwise unspecified.

The `getBean` method returns the bean instance. The `getMethod` method returns the method of the bean class for which the interceptor was invoked. For `AroundInvoke` methods, this is the business method on the bean class; for lifecycle callback interceptor methods, `getMethod` returns null.

The `proceed` method causes the invocation of the next interceptor method in the chain, or, when called from the last `AroundInvoke` interceptor method, the business method. Interceptor methods must always call `InvocationContext.proceed()` or no subsequent interceptor methods or bean business method or lifecycle callback methods will be invoked. The `proceed` method returns the result of the next method invoked. If a method returns `void`, `proceed` returns `null`. For lifecycle callback interceptor methods, if there is no callback method defined on the bean class, the invocation of `proceed` in the last interceptor method in the chain is a no-op, and `null` is returned. If there is more than one such interceptor method, the invocation of `proceed` causes the container to execute those methods in order.

11.6 Default Interceptors

Default interceptors may be defined to apply to all the components within the ejb-jar. The deployment descriptor is used to define default interceptors and their relative ordering. See Section 11.8.2.

Default interceptors are automatically applied to all components defined in the ejb-jar. The `ExcludeDefaultInterceptors` annotation or `exclude-default-interceptors` deployment descriptor element is used to exclude the invocation of default interceptors for a bean.

The default interceptors are invoked before any other interceptors for a bean. The `interceptor-order` deployment descriptor element may be used to specify alternative orderings. See Section 11.8.2.

11.7 Method-level Interceptors

A business method interceptor method may be defined to apply to a specific business method invocation, rather than to all of the business methods of the bean class.

Method-specific business method interceptors can be defined by applying the `Interceptors` annotation to the method for which the interceptors are to be invoked, or by means of the `interceptor-binding` deployment descriptor element. If more than one method-level interceptor is defined for a business method, the interceptors are invoked in the order specified. The deployment descriptor may be used to override this ordering.

The same interceptor may be applied to more than one method of the bean class:

```
@Stateless
public class MyBean ... {

     public void notIntercepted() {}

     @Interceptors(org.acme.MyInterceptor.class)
     public void someMethod() {
     }

     @Interceptors(org.acme.MyInterceptor.class)
     public void anotherMethod() {
     }
}
```

The applicability of a method-level interceptor to more than one business method of a bean does not affect the relationship between the interceptor instance and the bean class—only a single instance of the interceptor class is created per bean instance.

Method-level business method interceptors are invoked in addition to any default interceptors and interceptors defined for the bean class (and its superclasses). By default, the method-level interceptors are invoked after all other applicable interceptors. The `interceptor-binding` deployment descriptor element may be used to override this ordering.

The `ExcludeDefaultInterceptors` annotation or `exclude-default-interceptors` deployment descriptor element, when applied to a business method, is used to exclude the invocation of default interceptors for that method. The `ExcludeClassInterceptors` annotation or `exclude-class-interceptors` deployment descriptor element is used similarly to exclude the invocation of the class-level interceptors.

In the following example, if there are no default interceptors, only the interceptor `MyInterceptor` will be invoked when `someMethod` is called.

```
@Stateless
@Interceptors(org.acme.AnotherInterceptor.class)
public class MyBean ... {
     ...
     @Interceptors(org.acme.MyInterceptor.class)
     @ExcludeClassInterceptors
     public void someMethod() {
     }
}
```

If default interceptors have also been defined for the bean class, they can be excluded for the specific method by applying the `ExcludeDefaultInterceptors` annotation on the method.

```
@Stateless
@Interceptors(org.acme.AnotherInterceptor.class)
public class MyBean ... {
     ...
     @ExcludeDefaultInterceptors
     @ExcludeClassInterceptors
     @Interceptors(org.acme.MyInterceptor.class)
     public void someMethod() {
     }
}
```

11.8 Specification of Interceptors in the Deployment Descriptor

The deployment descriptor can be used as an alternative to metadata annotations to specify interceptors and their binding to enterprise beans or to override the invocation order of interceptors as specified in annotations.

11.8.1 Specification of Interceptors

The `interceptor` deployment descriptor element is used to specify the interceptor methods of an interceptor class. The interceptor methods are specified by using the `around-invoke-method`, `pre-construct-method`, `post-destroy-method`, `pre-passivate-method`, and `post-activate-method` elements.

At most one method of a given interceptor class can be designated as an around-invoke method, a pre-construct method, a post-destroy method, a pre-passivate method, or a post-activate method, regardless of whether the deployment descriptor is used to define interceptors or whether some combination of annotations and deployment descriptor elements is used.

11.8.2 Specification of the Binding of Interceptors to Beans

The `interceptor-binding` element is used to specify the binding of interceptor classes to enterprise beans and their business methods. The subelements of the `interceptor-binding` element are as follows:

- The `ejb-name` element must be the name of one of the enterprise beans contained in the ejb-jar or the wildcard value `"*"` (which is used to define interceptors that are bound to all beans in the ejb-jar).
- The `interceptor-class` element specifies the interceptor class. The `interceptor-order` element is used as an optional alternative to specify a total ordering over the interceptors defined for the given level and above.
- The `exclude-default-interceptors` and `exclude-class-interceptors` elements specify that default interceptors and class interceptors, respectively, are not to be applied to a bean class and/or business method.
- The `method-name` element specifies the method name for a method-level interceptor; and the optional `method-params` elements identify a single method among multiple methods with an overloaded method name.

Interceptors bound to all classes using the wildcard syntax "`*`" are default interceptors for the components in the ejb-jar. In addition, interceptors may be bound at the level of the bean class (class-level interceptors) or business methods of the class (method-level interceptors).

The binding of interceptors to classes is additive. If interceptors are bound at the class-level and/or default-level as well as at the method-level, both class-level and/or default-level as well as method-level interceptors will apply.

The `exclude-default-interceptors` element disables default interceptors for the level at which it is specified and lower. That is, `exclude-default-interceptors` when applied at the class-level disables the application of default-interceptors for all methods of the class. The `exclude-class-interceptors` element applied to a method, disables the application of class-level interceptors for the given method. Explicitly listing an excluded higher-level interceptor at a lower level causes it to be applied at that level and below.

By default, default interceptors will be invoked before class-level interceptors, and class-level interceptors will be invoked before method level interceptors. It is possible to override this ordering by using the `interceptor-order` element to specify a total ordering of interceptors at class-level and/or method-level. If the `interceptor-order` element is used, the ordering specified at the given level must be a total order over all interceptor classes that have been defined at that level and above (unless they have been explicitly excluded by means of one of the `exclude-` elements described above).[45]

There are four possible styles of the interceptor element syntax:

[45] The Deployer must insure that this condition is met if the interceptor-order is incomplete.

Style 1:

```
<interceptor-binding>
     <ejb-name>*</ejb-name>
     <interceptor-class>INTERCEPTOR</interceptor-class>
</interceptor-binding>
```

Specifying the ejb-name element as the wildcard value "*" designates default interceptors (interceptors that apply to all enterprise beans contained in the ejb-jar).

Style 2:

```
<interceptor-binding>
     <ejb-name>EJBNAME</ejb-name>
     <interceptor-class>INTERCEPTOR</interceptor-class>
</interceptor-binding>
```

This style is used to refer to interceptors associated with the specified enterprise bean class (class-level interceptors).

Style 3:

```
<interceptor-binding>
     <ejb-name>EJBNAME</ejb-name>
     <interceptor-class>INTERCEPTOR</interceptor-class>
     <method-name>METHOD</method-name>
</interceptor-binding>
```

This style is used to associate a method-level interceptor with the specified method of the specified enterprise bean. If there are multiple methods with the same overloaded name, the element of this style refers to all the methods with the overloaded name. Method-level interceptors can only be associated with business methods of the bean class. Note that the wildcard value "*" cannot be used to specify method-level interceptors.

Style 4:

```
<interceptor-binding>
     <ejb-name>EJBNAME</ejb-name>
     <interceptor-class>INTERCEPTOR</interceptor-class>
     <method-name>METHOD</method-name>
     <method-params>
          <method-param>PARAM-1</method-param>
          <method-param>PARAM-2</method-param>
          ...
          <method-param>PARAM-n</method-param>
     </method-params>
<interceptor-binding>
```

This style is used to associated a method-level interceptor with the specified method of the specified enterprise bean. This style is used to refer to a single method within a set of methods with an overloaded name. The values `PARAM-1` through `PARAM-n` are the fully-qualified Java types of the method's input parameters (if the method has no input arguments, the method-params element contains no method-param elements). Arrays are specified by the array element's type, followed by one or more pair of square brackets (e.g. `int[][]`).

11.8.2.1 Examples

Examples of the usage of the `interceptor-binding` syntax are given below.

Style 1: The following interceptors apply to all components in the ejb-jar as default interceptors. They will be invoked in the order specified.

```
<interceptor-binding>
   <ejb-name>*</ejb-name>
   <interceptor-class>org.acme.MyDefaultIC</interceptor-class>
   <interceptor-class>org.acme.MyDefaultIC2</interceptor-class>
</interceptor-binding>
```

Style 2: The following interceptors are the class-level interceptors of the `EmployeeService` bean. They will be invoked in the order specified after any default interceptors.

```
<interceptor-binding>
     <ejb-name>EmployeeService</ejb-name>
     <interceptor-class>org.acme.MyIC</interceptor-class>
     <interceptor-class>org.acme.MyIC2</interceptor-class>
</interceptor-binding>
```

Style 3: The following interceptors apply to all the `myMethod` methods of the `EmployeeService` bean. They will be invoked in the order specified after any default interceptors and class-level interceptors.

```
<interceptor-binding>
     <ejb-name>EmployeeService</ejb-name>
     <interceptor-class>org.acme.MyIC</interceptor-class>
     <interceptor-class>org.acme.MyIC2</interceptor-class>
     <method-name>myMethod</method-name>
</interceptor-binding>
```

Style 4: The following interceptor element refers to the `myMethod(String firstName, String LastName)` method of the `EmployeeService` bean.

```
<interceptor-binding>
     <ejb-name>EmployeeService</ejb-name>
     <interceptor-class>org.acme.MyIC</interceptor-class>
     <method-name>myMethod</method-name>
     <method-params>
          <method-param>java.lang.String</method-param>
          <method-param>java.lang.String</method-param>
     </method-params>
</interceptor-binding>
```

The following example illustrates a Style 3 element with more complex parameter types. The method `myMethod(char s, int i, int[] iar, mypackage.MyClass mycl, mypackage.MyClass[][] myclaar)` would be specified as:

```
    <interceptor-binding>
        <ejb-name>EmployeeService</ejb-name>
        <interceptor-class>org.acme.MyIC</interceptor-class>
        <method-name>myMethod</method-name>
        <method-params>
            <method-param>char</method-param>
            <method-param>int</method-param>
            <method-param>int[]</method-param>
            <method-param>mypackage.MyClass</method-param>
            <method-param>mypackage.MyClass[][]</method-param>
        </method-params>
    </interceptor-binding>
```

The following example illustrates the total ordering of interceptors using the `interceptor-order` element:

```
    <interceptor-binding>
        <ejb-name>EmployeeService</ejb-name>
        <interceptor-order>
            <interceptor-class>org.acme.MyIC</interceptor-class>
            <interceptor-class>org.acme.MyDefaultIC</intercep-
tor-class>
            <interceptor-class>org.acme.MyDefaultIC2</intercep-
tor-class>
            <interceptor-class>org.acme.MyIC2</interceptor-class>
        </interceptor-order>
    </interceptor-binding>
```

Chapter 12

Support for Transactions

One of the key features of the Enterprise JavaBeans™ architecture is support for distributed transactions. The Enterprise JavaBeans architecture allows an application developer to write an application that atomically updates data in multiple databases which may be distributed across multiple sites. The sites may use EJB servers from different vendors.

12.1 Overview

This section provides a brief overview of transactions and illustrates a number of transaction scenarios in EJB.

12.1.1 Transactions

Transactions are a proven technique for simplifying application programming. Transactions free the application programmer from dealing with the complex issues of failure recovery and multi-user programming. If the application programmer uses transactions, the programmer divides the application's work into units called transactions. The transactional system ensures that a unit of work either fully completes, or the work is fully rolled back. Furthermore, transactions make it possible for the programmer to design the application as if it ran in an environment that executes units of work serially.

Support for transactions is an essential element of the Enterprise JavaBeans architecture. The Enterprise Bean Provider and the client application programmer are not exposed to the complexity of distributed transactions. The Bean Provider can choose between using programmatic transaction demarcation in the enterprise bean code (this style is called *bean-managed transaction demarcation*) or declarative transaction demarcation performed automatically by the EJB container (this style is called *container-managed transaction demarcation*).

With bean-managed transaction demarcation, the enterprise bean code demarcates transactions using the `javax.transaction.UserTransaction` interface. All resource manager accesses between the `UserTransaction.begin` and `UserTransaction.commit` calls are part of a transaction.

> *The terms resource and resource manager used in this chapter refer to the resources declared using the Resource annotation in the enterprise bean class or using the resource-ref element in the enterprise bean's deployment descriptor. This includes not only database resources, but also other resources, such as JMS Connections. These resources are considered to be "managed" by the container.*[46]

With container-managed transaction demarcation, the container demarcates transactions per instructions provided by the developer in metadata annotations or in the deployment descriptor. These instructions, called *transaction attributes*, tell the container whether it should include the work performed by an enterprise bean method in a client's transaction, run the enterprise bean method in a new transaction started by the container, or run the method with "no transaction" (Refer to Subsection 12.6.5 for the description of the "no transaction" case).

Regardless of whether an enterprise bean uses bean-managed or container-managed transaction demarcation, the burden of implementing transaction management is on the EJB container and server provider. The EJB container and server implement the necessary low-level transaction protocols, such as the two-phase commit protocol between a transaction manager and a database system or messaging provider, transaction context propagation, and distributed two-phase commit.

Many applications will consist of one or several enterprise beans that all use a single resource manager (typically a relational database management system). The EJB container can make use of resource manager local transactions as an optimization technique for enterprise beans for which distributed transactions are not needed. A resource manager local transaction does not involve control or coordination by an external transaction manager. The container's use of local transactions as an optimization technique for enterprise beans with either container-managed transaction demarcation or bean-managed transaction demarcation is not visible to the enterprise beans. For a discussion of the use of resource manager local transactions as a container optimization strategy, refer to [12] and [15].

12.1.2 Transaction Model

The Enterprise JavaBeans architecture supports flat transactions. A flat transaction cannot have any child (nested) transactions.

[46] Note that environment entries other than resources are specified with the `Resource` annotation and/or `resource-ref` deployment descriptor element as well.

Note: The decision not to support nested transactions allows vendors of existing transaction processing and database management systems to incorporate support for Enterprise JavaBeans. If these vendors provide support for nested transactions in the future, Enterprise JavaBeans may be enhanced to take advantage of nested transactions.

12.1.3 Relationship to JTA and JTS

The Java™ Transaction API (JTA) [8] is a specification of the interfaces between a transaction manager and the other parties involved in a distributed transaction processing system: the application programs, the resource managers, and the application server.

The Java Transaction Service (JTS) [9] API is a Java binding of the CORBA Object Transaction Service (OTS) 1.1 specification. JTS provides transaction interoperability using the standard IIOP protocol for transaction propagation between servers. The JTS API is intended for vendors who implement transaction processing infrastructure for enterprise middleware. For example, an EJB server vendor may use a JTS implementation as the underlying transaction manager.

The EJB architecture does not require the EJB container to support the JTS interfaces. The EJB architecture requires that the EJB container support the JTA API defined in [8] and the Connector APIs defined in [15].

12.2 Sample Scenarios

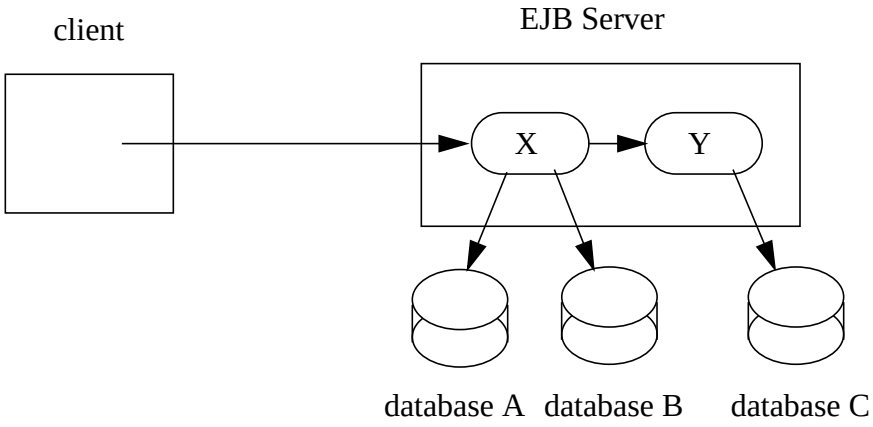
This section describes several scenarios that illustrate the distributed transaction capabilities of the Enterprise JavaBeans architecture.

12.2.1 Update of Multiple Databases

The Enterprise JavaBeans architecture makes it possible for an application program to update data in multiple databases in a single transaction.

In the following figure, a client invokes the enterprise bean X. Bean X updates data using two database connections that the Deployer configured to connect with two different databases, A and B. Then X calls another enterprise bean, Y. Bean Y updates data in database C. The EJB server ensures that the updates to databases A, B, and C are either all committed or all rolled back.

Figure 22 Updates to Simultaneous Databases



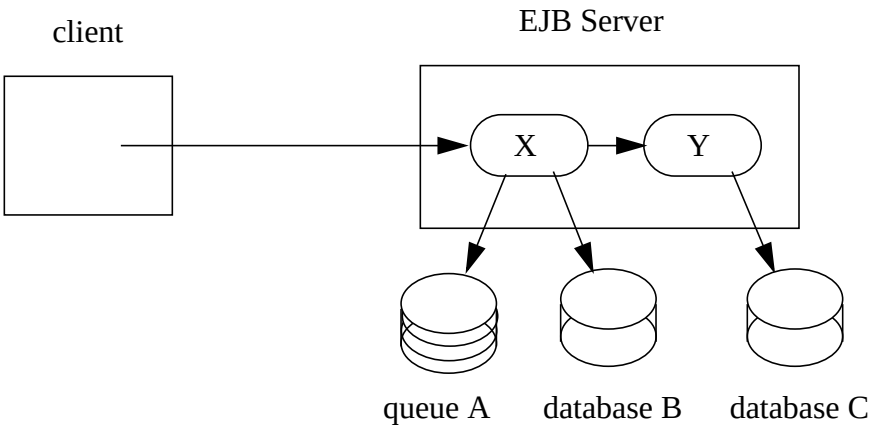
The application programmer does not have to do anything to ensure transactional semantics. Behind the scenes, the EJB server enlists the database connections as part of the transaction. When the transaction commits, the EJB server and the database systems perform a two-phase commit protocol to ensure atomic updates across all three databases.

12.2.2 Messages Sent or Received Over JMS Sessions and Update of Multiple Databases

The Enterprise JavaBeans architecture makes it possible for an application program to send messages to or receive messages from one or more JMS Destinations and/or to update data in one or more databases in a single transaction.

In the following figure, a client invokes the enterprise bean X. Bean X sends a message to a JMS queue A and updates data in a database B using connections that the Deployer configured to connect with a JMS provider and a database. Then X calls another enterprise bean, Y. Bean Y updates data in database C. The EJB server ensures that the operations on A, B, and C are either all committed, or all rolled back.

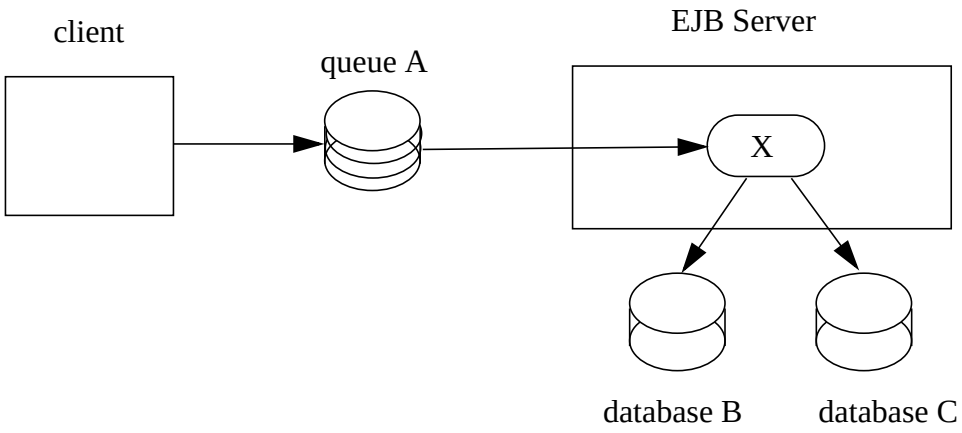
Figure 23 Message Sent to JMS Queue and Updates to Multiple Databases



The application programmer does not have to do anything to ensure transactional semantics. The enterprise beans X and Y perform the message send and database updates using the standard JMS and JDBC™ APIs. Behind the scenes, the EJB server enlists the session on the connection to the JMS provider and the database connections as part of the transaction. When the transaction commits, the EJB server and the messaging and database systems perform a two-phase commit protocol to ensure atomic updates across all the three resources.

In the following figure, a client sends a message to the JMS queue A serviced by the message-driven bean X. Bean X updates data using two database connections that the deployer configured to connect with two different databases, B and C. The EJB server ensures that the dequeuing of the JMS message, its receipt by bean X, and the updates to databases B and C are either all committed or all rolled back.

Figure 24 Message Sent to JMS Queue Serviced by Message-Driven Bean and Updates to Multiple Databases

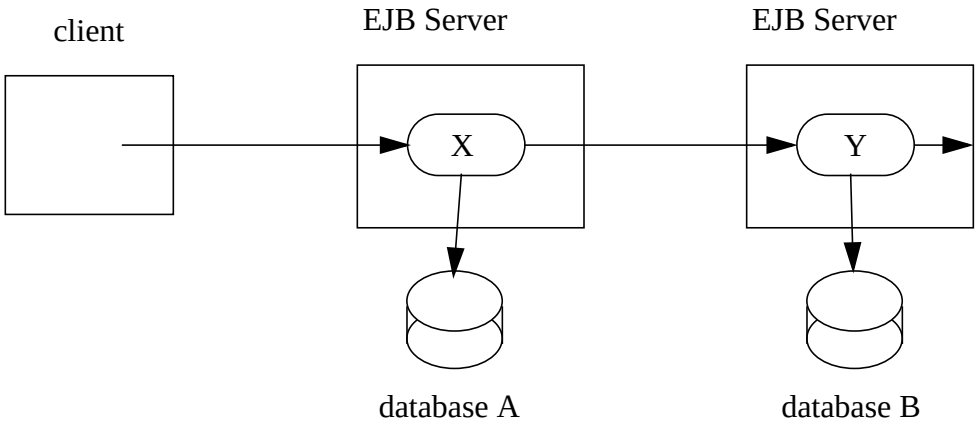


12.2.3 Update of Databases via Multiple EJB Servers

The Enterprise JavaBeans architecture allows updates of data at multiple sites to be performed in a single transaction.

In the following figure, a client invokes the enterprise bean X. Bean X updates data in database A, and then calls another enterprise bean Y that is installed in a remote EJB server. Bean Y updates data in database B. The Enterprise JavaBeans architecture makes it possible to perform the updates to databases A and B in a single transaction.

Figure 25 Updates to Multiple Databases in Same Transaction



When X invokes Y, the two EJB servers cooperate to propagate the transaction context from X to Y. This transaction context propagation is transparent to the application-level code.

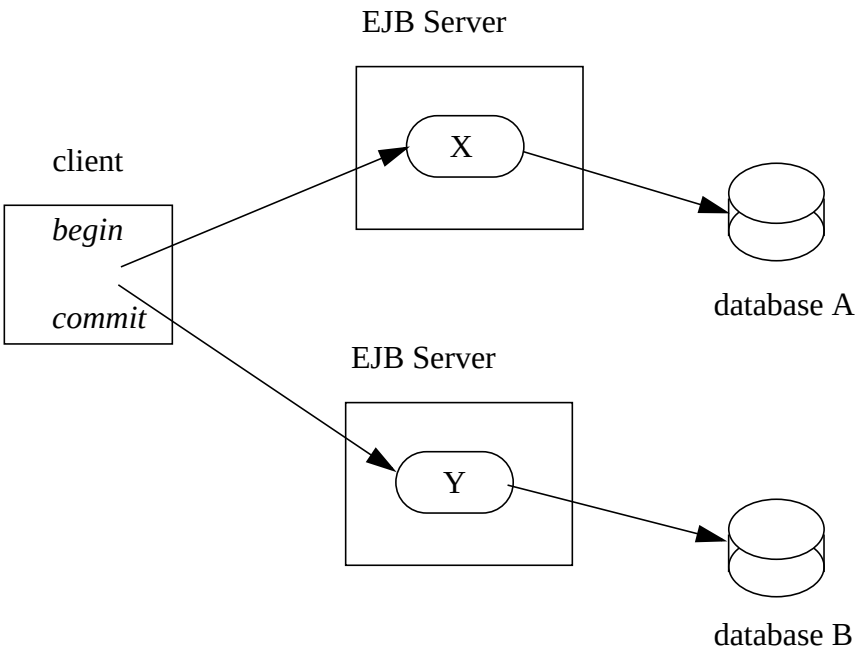
At transaction commit time, the two EJB servers use a distributed two-phase commit protocol (if the capability exists) to ensure the atomicity of the database updates.

12.2.4 Client-Managed Demarcation

A Java client can use the `javax.transaction.UserTransaction` interface to explicitly demarcate transaction boundaries. The client program obtains the `javax.transaction.UserTransaction` interface through dependency injection or lookup in the bean's EJBContext or in the JNDI name space.

A client program using explicit transaction demarcation may perform, via enterprise beans, atomic updates across multiple databases residing at multiple EJB servers, as illustrated in the following figure.

Figure 26 Updates on Multiple Databases on Multiple Servers



The application programmer demarcates the transaction with `begin` and `commit` calls. If the enterprise beans X and Y are configured to use a client transaction (i.e., their methods have transaction attributes that either require or support an existing transaction context), the EJB server ensures that the updates to databases A and B are made as part of the client's transaction.

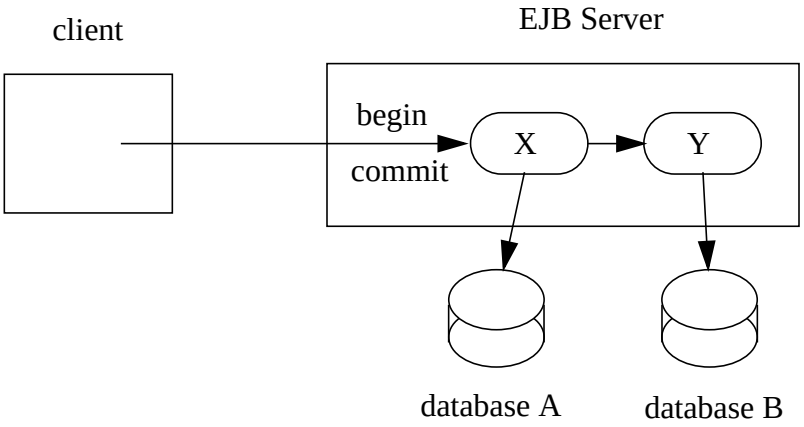
12.2.5 Container-Managed Demarcation

Whenever a client invokes a method on an enterprise bean's business interface (or on the home or component interface of an enterprise bean), the container interposes on the method invocation. The interposition allows the container to control transaction demarcation declaratively through the transaction attribute set by the developer. (See Section 12.3.7 for a description of transaction attributes.)

For example, if an enterprise bean method is configured with the `REQUIRED` transaction attribute, the container behaves as follows: If the client request is not associated with a transaction context, the container automatically initiates a transaction whenever a client invokes an enterprise bean method that requires a transaction context. If the client request contains a transaction context, the container includes the enterprise bean method in the client transaction.

The following figure illustrates such a scenario. A non-transactional client invokes the enterprise bean X, and the invoked method has the `REQUIRED`[47] *transaction attribute. Because the message from the client does not include a transaction context, the container starts a new transaction before dispatching the method on X. Bean X's work is performed in the context of the transaction. When X calls other enterprise beans (Y in our example), the work performed by the other enterprise beans is also automatically included in the transaction (subject to the transaction attribute of the other enterprise bean).*

Figure 27 Update of Multiple Databases from Non-Transactional Client



The container automatically commits the transaction at the time X returns a reply to the client.

If a message-driven bean's message listener method is configured with the `REQUIRED` transaction attribute, the container automatically starts a new transaction before the delivery of the message and, hence, before the invocation of the method.[48]

[47] In this chapter we use the `TransactionAttribute` annotation values to refer to transaction attributes. The deployment descriptor may be used as an overriding mechanism or an alternative to the use of annotations, and must be used for EJB 2.1 and 1.1 entity beans, for which the use of annotations is not supported.

JMS requires that the transaction be started before the dequeuing of the message. See [13].

The container automatically enlists the resource manager associated with the arriving message and all the resource managers accessed by the message listener method with the transaction.

12.3 Bean Provider's Responsibilities

This section describes the Bean Provider's view of transactions and defines the Bean Provider's responsibilities.

12.3.1 Bean-Managed Versus Container-Managed Transaction Demarcation

When designing an enterprise bean, the developer must decide whether the enterprise bean will demarcate transactions programmatically in the business methods (bean-managed transaction demarcation), or whether the transaction demarcation is to be performed by the container based on the transaction attributes specified in metadata annotations or in the deployment descriptor (container-managed transaction demarcation). Typically enterprise beans will be specified to have container-managed transaction demarcation. This is the default if no transaction management type is specified.

A session bean or a message-driven bean can be designed with bean-managed transaction demarcation or with container-managed transaction demarcation. (But it cannot be both at the same time.)

An EJB 2.1 or EJB 1.1 entity bean must always use container-managed transaction demarcation. An EJB 2.1 or EJB 1.1 entity bean must not be designated with bean-managed transaction demarcation.

A transaction management type cannot be specified for EJB 3.0 entities. EJB 3.0 entities execute within the transactional context of the caller. See the document *"Java Persistence"* [2] of this specification for a discussion of transactions involving EJB 3.0 entities.

An enterprise bean instance can access resource managers in a transaction only in the enterprise bean's methods in which there is a transaction context available.

12.3.1.1 Non-Transactional Execution

Some enterprise beans may need to access resource managers that do not support an external transaction coordinator. The container cannot manage the transactions for such enterprise beans in the same way that it can for the enterprise beans that access resource managers that support an external transaction coordinator.

If an enterprise bean needs to access a resource manager that does not support an external transaction coordinator, the Bean Provider should design the enterprise bean with container-managed transaction demarcation and assign the `NOT_SUPPORTED` transaction attribute to the bean class or to all the bean's methods. The EJB architecture does not specify the transactional semantics of the enterprise bean methods. See Subsection 12.6.5 for how the container implements this case.

[48] We use the term "container" here to encompass both the container and the messaging provider. When the contracts outlined in [15] are used, it may be the messaging provider that starts the transaction.

12.3.2 Isolation Levels

Transactions not only make completion of a unit of work atomic, but they also isolate the units of work from each other, provided that the system allows concurrent execution of multiple units of work.

The *isolation level* describes the degree to which the access to a resource manager by a transaction is isolated from the access to the resource manager by other concurrently executing transactions.

The following are guidelines for managing isolation levels in enterprise beans.

- The API for managing an isolation level is resource-manager-specific. (Therefore, the EJB architecture does not define an API for managing isolation levels.)
- If an enterprise bean uses multiple resource managers, the Bean Provider may specify the same or different isolation level for each resource manager. This means, for example, that if an enterprise bean accesses multiple resource managers in a transaction, access to each resource manager may be associated with a different isolation level.
- The Bean Provider must take care when setting an isolation level. Most resource managers require that all accesses to the resource manager within a transaction are done with the same isolation level. An attempt to change the isolation level in the middle of a transaction may cause undesirable behavior, such as an implicit sync point (a commit of the changes done so far).
- For session beans and message-driven beans with bean-managed transaction demarcation, the Bean Provider can specify the desirable isolation level programmatically in the enterprise bean's methods, using the resource-manager specific API. For example, the Bean Provider can use the `java.sql.Connection.setTransactionIsolation` method to set the appropriate isolation level for database access.
- The container provider should insure that suitable isolation levels are provided to guarantee data consistency for entity beans. Typically this means that an equivalent of a repeatable read or serializable isolation level should be available for applications that require a high degree of isolation.
- For entity beans with EJB 2.1 container-managed persistence and earlier, transaction isolation is managed by the data access classes that are generated by the container provider's tools. The tools must ensure that the management of the isolation levels performed by the data access classes will not result in conflicting isolation level requests for a resource manager within a transaction.
- Additional care must be taken if multiple enterprise beans access the same resource manager in the same transaction. Conflicts in the requested isolation levels must be avoided.

12.3.3 Enterprise Beans Using Bean-Managed Transaction Demarcation

This subsection describes the requirements for the Bean Provider of an enterprise bean with bean-managed transaction demarcation.

The enterprise bean with bean-managed transaction demarcation must be a session bean or a message-driven bean.

An instance that starts a transaction must complete the transaction before it starts a new transaction.

The Bean Provider uses the `UserTransaction` interface to demarcate transactions. All updates to the resource managers between the `UserTransaction.begin` and `UserTransaction.commit` methods are performed in a transaction. While an instance is in a transaction, the instance must not attempt to use the resource-manager specific transaction demarcation API (e.g. it must not invoke the `commit` or `rollback` method on the `java.sql.Connection` interface or on the `javax.jms.Session` interface).

A stateful session bean instance may, but is not required to, commit a started transaction before a business method returns. If a transaction has not been completed by the end of a business method, the container retains the association between the transaction and the instance across multiple client calls until the instance eventually completes the transaction.

A stateless session bean instance must commit a transaction before a business method or timeout callback method returns.

A message-driven bean instance must commit a transaction before a message listener method or timeout callback method returns.

The following example illustrates a business method that performs a transaction involving two database connections.

```
@Stateless
@TransactionManagement(BEAN)
public class MySessionBean implements MySession {
     @Resource javax.transaction.UserTransaction ut;
     @Resource javax.sql.DataSource database1;
     @Resource javax.sql.DataSource database2;

     public void someMethod(...) {
          java.sql.Connection con1;
          java.sql.Connection con2;
          java.sql.Statement stmt1;
          java.sql.Statement stmt2;

          // obtain con1 object and set it up for transactions
          con1 = database1.getConnection();

          stmt1 = con1.createStatement();

          // obtain con2 object and set it up for transactions
          con2 = database2.getConnection();

          stmt2 = con2.createStatement();

          //
          // Now do a transaction that involves con1 and con2.
          //
          // start the transaction
          ut.begin();

          // Do some updates to both con1 and con2. The container
          // automatically enlists con1 and con2 with the transaction.
          stmt1.executeQuery(...);
          stmt1.executeUpdate(...);
          stmt2.executeQuery(...);
          stmt2.executeUpdate(...);
          stmt1.executeUpdate(...);
          stmt2.executeUpdate(...);

          // commit the transaction
          ut.commit();

          // release connections
          stmt1.close();
          stmt2.close();
          con1.close();
          con2.close();
     }
     ...
}
```

The following example illustrates a business method that performs a transaction involving both a database connection and a JMS connection.

```
@Stateless
@TransactionManagement(BEAN)
public class MySessionBean implements MySession {
     @Resource javax.Transaction.UserTransaction ut;
     @Resource javax.sql.DataSource database1;
     @Resource javax.jms.QueueConnectionFactory qcf1;
     @Resource javax.jms.Queue queue1;

     public void someMethod(...) {
          java.sql.Connection dcon;
          java.sql.Statement stmt;
          javax.jms.QueueConnection qcon;
          javax.jms.QueueSession qsession;
          javax.jms.QueueSender qsender;
          javax.jms.Message message;


          // obtain db conn object and set it up for transactions

          dcon = database1.getConnection();

          stmt = dcon.createStatement();

          // obtain jms conn object and set up session for transactions
          qcon = qcf1.createQueueConnection();
          qsession = qcon.createQueueSession(true,0);
          qsender = qsession.createSender(queue1);
          message = qsession.createTextMessage();
          message.setText("some message");

          //
          // Now do a transaction that involves the two connections.
          //
          // start the transaction
          ut.begin();

          // Do database updates and send message. The container
          // automatically enlists dcon and qsession with the
          // transaction.
          stmt.executeQuery(...);
          stmt.executeUpdate(...);
          stmt.executeUpdate(...);
          qsender.send(message);

          // commit the transaction
          ut.commit();

          // release connections
          stmt.close();
          qsender.close();
          qsession.close();
          dcon.close();
          qcon.close();
     }
     ...
}
```

The following example illustrates a stateful session bean that retains a transaction across three client calls, invoked in the following order: `method1`, `method2`, and `method3`.[49]

```
@Stateful
@TransactionManagement(BEAN)
public class MySessionBean implements MySession {
     @Resource javax.Transaction.UserTransaction ut;
     @Resource javax.sql.DataSource database1;
     @Resource javax.sql.DataSource database2;
     java.sql.Connection con1;
     java.sql.Connection con2;

     public void method1(...) {
          java.sql.Statement stmt;

          // start a transaction
          ut.begin();

          // make some updates on con1
          con1 = database1.getConnection();
          stmt = con1.createStatement();
          stmt.executeUpdate(...);
          stmt.executeUpdate(...);

          //
          // The container retains the transaction associated with the
          // instance to the next client call (which is method2(...)).
     }

     public void method2(...) {
          java.sql.Statement stmt;

          con2 = database2.getConnection();
          stmt = con2.createStatement();
          stmt.executeUpdate(...);
          stmt.executeUpdate(...);

          // The container retains the transaction associated with the
          // instance to the next client call (which is method3(...)).
     }

     public void method3(...) {
          java.sql.Statement stmt;

          // make some more updates on con1 and con2
          stmt = con1.createStatement();
          stmt.executeUpdate(...);
          stmt = con2.createStatement();
          stmt.executeUpdate(...);

          // commit the transaction
          ut.commit();

          // release connections
          stmt.close();
```

[49] Note that the Bean Provider must use the pre-passivation callback method here to close the connections and set the instance variables for the connection to null.

```
            con1.close();
            con2.close();
        }
        ...
}
```

It is possible for an enterprise bean to open and close a database connection in each business method (rather than hold the connection open until the end of transaction). In the following example, if the client executes the sequence of methods (`method1`, `method2`, `method2`, `method2`, and `method3`), all the database updates done by the multiple invocations of `method2` are performed in the scope of the same transaction, which is the transaction started in `method1` and committed in `method3`.

```
@Stateful
@TransactionManagement(BEAN)
public class MySessionBean implements MySession {
    @Resource javax.Transaction.UserTransaction ut;
    @Resource javax.sql.DataSource database1;

    public void method1(...) {
        // start a transaction
        ut.begin();
    }

    public void method2(...) {
        java.sql.Connection con;
        java.sql.Statement stmt;

        // open connection
        con = database1.getConnection();

        // make some updates on con
        stmt = con.createStatement();
        stmt.executeUpdate(...);
        stmt.executeUpdate(...);

        // close the connection
        stmt.close();
        con.close();
    }

    public void method3(...) {

        // commit the transaction
        ut.commit();
    }
    ...
}
```

12.3.3.1 getRollbackOnly and setRollbackOnly Methods

An enterprise bean with bean-managed transaction demarcation must not use the `getRollbackOnly` and `setRollbackOnly` methods of the `EJBContext` interface.

An enterprise bean with bean-managed transaction demarcation has no need to use these methods, because of the following reasons:

- An enterprise bean with bean-managed transaction demarcation can obtain the status of a transaction by using the `getStatus` method of the `javax.transaction.UserTransaction` interface.

- An enterprise bean with bean-managed transaction demarcation can rollback a transaction using the `rollback` method of the `javax.transaction.UserTransaction` interface.

12.3.4 Enterprise Beans Using Container-Managed Transaction Demarcation

This subsection describes the requirements for the Bean Provider of an enterprise bean using container-managed transaction demarcation.

The enterprise bean's business methods, message listener methods, business method interceptor methods, lifecycle callback interceptor methods, or timeout callback method must not use any resource-manager specific transaction management methods that would interfere with the container's demarcation of transaction boundaries. For example, the enterprise bean methods must not use the following methods of the `java.sql.Connection` interface: `commit`, `setAutoCommit`, and `rollback`; or the following methods of the `javax.jms.Session` interface: `commit` and `rollback`.

The enterprise bean's business methods, message listener methods, business method interceptor methods, lifecycle callback interceptor methods, or timeout callback method must not attempt to obtain or use the `javax.transaction.UserTransaction` interface.

The following is an example of a business method in an enterprise bean with container-managed transaction demarcation. The business method updates two databases using JDBC™ connections. The container provides transaction demarcation as specified by the transaction attribute.[50]

```
@Stateless public class MySessionBean implements MySession {
     ...

     @TransactionAttribute(REQUIRED)
     public void someMethod(...) {
          java.sql.Connection con1;
          java.sql.Connection con2;
          java.sql.Statement stmt1;
          java.sql.Statement stmt2;

          // obtain con1 and con2 connection objects
          con1 = ...;
          con2 = ...;

          stmt1 = con1.createStatement();
          stmt2 = con2.createStatement();

          //
          // Perform some updates on con1 and con2. The container
          // automatically enlists con1 and con2 with the container-
          // managed transaction.
          //
          stmt1.executeQuery(...);
          stmt1.executeUpdate(...);

          stmt2.executeQuery(...);
          stmt2.executeUpdate(...);

          stmt1.executeUpdate(...);
          stmt2.executeUpdate(...);

          // release connections
          con1.close();
          con2.close();
     }
     ...
}
```

12.3.4.1 javax.ejb.SessionSynchronization Interface

A stateful session bean with container-managed transaction demarcation can optionally implement the `javax.ejb.SessionSynchronization` interface. The use of the `SessionSynchronization` interface is described in Subsection 4.3.7.

[50] REQUIRED is the default transaction attribute value for container managed transaction demarcation. The explicit specification of the transaction attribute is therefore not required in this example.

12.3.4.2 javax.ejb.EJBContext.setRollbackOnly Method

An enterprise bean with container-managed transaction demarcation can use the `setRollbackOnly` method of its `EJBContext` object to mark the transaction such that the transaction can never commit. Typically, an enterprise bean marks a transaction for rollback to protect data integrity before throwing an application exception, if the application exception class has not been specified to automatically cause the container to rollback the transaction.

For example, an `AccountTransfer` bean which debits one account and credits another account could mark a transaction for rollback if it successfully performs the debit operation, but encounters a failure during the credit operation.

12.3.4.3 javax.ejb.EJBContext.getRollbackOnly method

An enterprise bean with container-managed transaction demarcation can use the `getRollbackOnly` method of its `EJBContext` object to test if the current transaction has been marked for rollback. The transaction might have been marked for rollback by the enterprise bean itself, by other enterprise beans, or by other components (outside of the EJB specification scope) of the transaction processing infrastructure.

12.3.5 Use of JMS APIs in Transactions

The Bean Provider should not make use of the JMS request/reply paradigm (sending of a JMS message, followed by the synchronous receipt of a reply to that message) within a single transaction. Because a JMS message is typically not delivered to its final destination until the transaction commits, the receipt of the reply within the same transaction will not take place.

Because the container manages the transactional enlistment of JMS sessions on behalf of a bean, the parameters of the `createSession(boolean transacted, int acknowledgeMode)`, `createQueueSession(boolean transacted, int acknowledgeMode)` and `createTopicSession(boolean transacted, int acknowledgeMode)` methods are ignored. It is recommended that the Bean Provider specify that a session is transacted, but provide `0` for the value of the acknowledgment mode.

The Bean Provider should not use the JMS `acknowledge` method either within a transaction or within an unspecified transaction context. Message acknowledgment in an unspecified transaction context is handled by the container. Section 12.6.5 describes some of the techniques that the container can use for the implementation of a method invocation with an unspecified transaction context.

12.3.6 Specification of a Bean's Transaction Management Type

By default, a session bean or message-driven bean has container managed transaction demarcation if the transaction management type is not specified. The Bean Provider of a session bean or a message-driven bean can use the `TransactionManagement` annotation to declare whether the session bean or message-driven bean uses bean-managed or container-managed transaction demarcation. The value of the `TransactionManagement` annotation is either `CONTAINER` or `BEAN`. The `TransactionManagement` annotation is appled to the enterprise bean class.

Alternatively, the Bean Provider can use the `transaction-type` deployment descriptor element to specify the bean's transaction management type. If the deployment descriptor is used, it is only necessary to explicitly specify the bean's transaction management type if bean-managed transaction is used.

The transaction management type of a bean is determined by the Bean Provider. The application assembler is not permitted to use the deployment descriptor to override a bean's transaction management type. (See Chapter 18 for information about the deployment descriptor.)

12.3.7 Specification of the Transaction Attributes for a Bean's Methods

The Bean Provider of an enterprise bean with container-managed transaction demarcation may specify the transaction attributes for the enterprise bean's methods. By default, the value of the transaction attribute for a method of a bean with container-managed transaction demarcation is the `REQUIRED` transaction attribute, and the transaction attribute does not need to be explicitly specified in this case.

A transaction attribute is a value associated with each of the following methods

- a method of a bean's business interface
- a message listener method of a message-driven bean
- a timeout callback method
- a stateless session bean's web service endpoint method
- for beans written to the EJB 2.1 and earlier client view, a method of a session or entity bean's home or component interface

The transaction attribute specifies how the container must manage transactions for a method when a client invokes the method.

Transaction attributes are specified for the following methods:

- For a session bean written to the EJB 3.0 client view API, the transaction attributes are specified for those methods of the session bean class that correspond to the bean's business interface, the direct and indirect superinterfaces of the business interface, and for the timeout callback method, if any.
- For a stateless session bean that provides a web service client view, the transaction attributes are specified for the bean's web service endpoint methods, and for the timeout callback method, if any.
- For a message-driven bean, the transaction attributes are specified for those methods on the message-driven bean class that correspond to the bean's message listener interface and for the timeout callback method, if any.
- For a session bean written to the EJB 2.1 and earlier client view, the transaction attributes are specified for the methods of the component interface and all the direct and indirect superinterfaces of the component interface, excluding the methods of the `javax.ejb.EJBObject` or

`javax.ejb.EJBLocalObject` interface; and for the timeout callback method, if any. Transaction attributes must not be specified for the methods of a session bean's home interface.

- For a EJB 2.1 (and earlier) entity bean, the transaction attributes are specified for the methods defined in the bean's component interface and all the direct and indirect superinterfaces of the component interface, excluding the `getEJBHome`, `getEJBLocalHome`, `getHandle`, `getPrimaryKey`, and `isIdentical` methods; for the methods defined in the bean's home interface and all the direct and indirect superinterfaces of the home interface, excluding the `getEJBMetaData` and `getHomeHandle` methods specific to the remote home interface; and for the timeout callback method, if any.[51]

By default, if a `TransactionAttribute` annotation is not specified for a method of an enterprise bean with container-managed transaction demarcation, the value of the transaction attribute for the method is defined to be `REQUIRED`. The rules for the specification of transaction attributes are defined in Section 12.3.7.1.

The Bean Provider may use the deployment descriptor as an alternative to metadata annotations to specify the transaction attributes (or as a means to supplement or override metadata annotations for transaction attributes). Transaction attributes specified in the deployment descriptor are assumed to override or supplement transaction attributes specified in annotations. If a transaction attribute value is not specified in the deployment descriptor, it is assumed that the specified in annotations holds, or—in the case that no annotation has been specified—that the value is `Required`.

The application assembler is permitted to override the transaction attribute values using the bean's deployment descriptor. The deployer is also permitted to override the transaction attribute values at deployment time. Caution should be exercised when overriding the transaction attributes of an application, as the transactional structure of an application is typically intrinsic to the semantics of the application.

Enterprise JavaBeans defines the following values for the `TransactionAttribute` metadata annotation:

- `MANDATORY`
- `REQUIRED`
- `REQUIRES_NEW`
- `SUPPORTS`
- `NOT_SUPPORTED`
- `NEVER`

The deployment descriptor values that correspond to these annotation values are the following:

[51] Note that the deployment descriptor must be used to specify transaction attributes for EJB 2.1 and earlier entity bean methods if the transaction attribute is not `Required` (the default value).

- `Mandatory`
- `Required`
- `RequiresNew`
- `Supports`
- `NotSupported`
- `Never`

In this chapter, we use the `TransactionAttribute` annotation values to refer to transaction attributes. As noted, however, the deployment descriptor may be used.

Refer to Subsection 12.6.2 for the specification of how the value of the transaction attribute affects the transaction management performed by the container.

For a message-driven bean's message listener methods (or interface), only the `REQUIRED` and `NOT_SUPPORTED` `TransactionAttribute` values may be used.

For a stateless session bean's web service endpoint method (or interface), only the `REQUIRED`, `REQUIRES_NEW`, `SUPPORTS`, `NEVER`, and `NOT_SUPPORTED` transaction attributes may be used.

For an enterprise bean's timeout callback method only the `REQUIRES`, `REQUIRES_NEW` and `NOT_SUPPORTED` transaction attributes may be used.

If an enterprise bean implements the `javax.ejb.SessionSynchronization` interface, only the following values may be used for the transaction attributes of the bean's methods: `REQUIRED`, `REQUIRES_NEW`, `MANDATORY`.

The above restriction is necessary to ensure that the enterprise bean is invoked only in a transaction. If the bean were invoked without a transaction, the container would not be able to send the transaction synchronization calls.

For entity beans that use EJB 2.1 container-managed persistence, only the `Required`, `RequiresNew`, or `Mandatory` deployment descriptor transaction attribute values should be used for the methods defined in the bean's component interface and all the direct and indirect superinterfaces of the component interface, excluding the `getEJBHome`, `getEJBLocalHome`, `getHandle`, `getPrimaryKey`, and `isIdentical` methods; and for the methods defined in the bean's home interface and all the direct and indirect superinterfaces of the home interface, excluding the `getEJB-MetaData` and `getHomeHandle` methods specific to the remote home interface.

> *The Bean Provider and Application Assembler must exercise caution when using the `RequiresNew` transaction attributes with the navigation of container-managed relationships. If higher levels of isolation are used, navigating a container-managed relationship in a new transaction context may result in deadlock.*

Containers may *optionally* support the use of the `NotSupported`, `Supports`, and `Never` transaction attributes for the methods of EJB 2.1 entity beans with container-managed persistence. However, entity beans with container-managed persistence that use these transaction attributes will not be portable.

Containers may optionally support the use of the `NotSupported`, `Supports`, and `Never` transaction attributes for the methods of EJB 2.1 entity beans with container-managed persistence because the use of these transaction modes may be needed to make use of container-managed persistence with non-transactional data stores. In general, however, the Bean Provider and Application Assembler should avoid use of the `NotSupported`,`Supports`, and `Never` transaction attribute values for the methods of entity beans with container-managed persistence because it may lead to inconsistent results or to the inconsistent and/or to the partial updating of persistent state and relationships in the event of concurrent use.

12.3.7.1 Specification of Transaction Attributes with Metadata Annotations

The following is the description of the rules for the specification of transaction attributes using Java language metadata annotations.

The transaction attributes for the methods of a bean class may be specified on the class, the business methods of the class, or both.

The `TransactionAttribute` annotation is used to specify a transaction attribute. The value of the transaction attribute annotation is given by the enum `TransactionAttributeType:`

```
public enum TransactionAttributeType {
     MANDATORY,
     REQUIRED,
     REQUIRES_NEW,
     SUPPORTS,
     NOT_SUPPORTED,
     NEVER
}
```

Specifying the `TransactionAttribute` annotation on the bean class means that it applies to all applicable business interface methods of the class. If the transaction attribute type is not specified, it is assumed to be `REQUIRED`. The absence of a transaction attribute specification on the bean class is equivalent to the specification of `TransactionAttribute(REQUIRED)` on the bean class.

A transaction attribute may be specified on a method of the bean class to override the transaction attribute value explicitly or implicitly specified on the bean class.

If the bean class has superclasses, the following additional rules apply.

- A transaction attribute specified on a superclass *S* applies to the business methods defined by *S*. If a class-level transaction attribute is not specified on *S*, it is equivalent to specification of `TransactionAttribute(REQUIRED)` on *S*.

- A transaction attribute may be specified on a business method *M* defined by class *S* to override for method *M* the transaction attribute value explicitly or implicitly specified on the class *S*.

- If a method *M* of class *S* overrides a business method defined by a superclass of *S*, the transaction attribute of *M* is determined by the above rules as applied to class *S*.

Example:

```
@TransactionAttribute(SUPPORTS)
public class SomeClass {
     public void aMethod () {...}
     public void bMethod () {...}
     ...
}

@Stateless public class ABean extends SomeClass implements A {

     public void aMethod () {...}

     @TransactionAttribute(REQUIRES_NEW)
     public void cMethod () {...}
  ...
}
```

Assuming `aMethod`, `bMethod`, `cMethod` are methods of interface `A`, their transaction attributes are `REQUIRED`, `SUPPORTS`, and `REQUIRES_NEW` respectively.

12.3.7.2 Specification of Transaction Attributes in the Deployment Descriptor

The following is the description of the rules for the specification of transaction attributes in the deployment descriptor. (See Section 18.5 for the complete syntax of the deployment descriptor.)

Note that even in the absence of the use of annotations, it is not necessary to explicitly specify transaction attributes for all of the methods listed in section 12.3.7. If a transaction attribute is not specified for a method in an EJB 3.0 deployment descriptor, the transaction attribute defaults to `Required`.

12.3.7.2.1 Use of the container-transaction element

The `container-transaction` element may be used to define the transaction attributes for business, home, component, and message-listener interface methods; web service endpoint methods; and timeout callback methods. Each `container-transaction` element consists of a list of one or more `method` elements, and the `trans-attribute` element. The `container-transaction` element specifies that all the listed methods are assigned the specified transaction attribute value. It is required that all the methods specified in a single `container-transaction` element be methods of the same enterprise bean.

The `method` element uses the `ejb-name`, `method-name`, and `method-params` elements to denote one or more methods. There are three legal styles of composing the `method` element:

Style 1:

```
<method>
      <ejb-name>EJBNAME</ejb-name>
      <method-name>*</method-name>
</method>
```

This style is used to specify a default value of the transaction attribute for the methods for

which there is no Style 2 or Style 3 element specified. There must be at most one `container-transaction` element that uses the Style 1 `method` element for a given enterprise bean.

Style 2:

```
<method>
      <ejb-name>EJBNAME</ejb-name>
      <method-name>METHOD</method-name>
</method>
```

This style is used for referring to a specified method of a business, home, component or message listener interface method; web service endpoint method; or timeout callback method of the specified enterprise bean. If there are multiple methods with the same overloaded name, this style refers to all the methods with the same name. There must be at most one `container-transaction` element that uses the Style 2 `method` element for a given method name. If there is also a `container-transaction` element that uses Style 1 element for the same bean, the value specified by the Style 2 element takes precedence.

Style 3:

```
<method>
     <ejb-name>EJBNAME</ejb-name>
     <method-name>METHOD</method-name>
     <method-params>
          <method-param>PARAMETER_1</method-param>
          ...
          <method-param>PARAMETER_N</method-param>
     </method-params>
</method>
```

This style is used to refer to a single method within a set of methods with an overloaded name. If there is also a `container-transaction` element that uses the Style 2 element for the method name, or the Style 1 element for the bean, the value specified by the Style 3 element takes precedence.

The optional `method-intf` element can be used to differentiate between methods with the same name and signature that are multiply defined across the business, component, and home interfaces, and/or web service endpoint. However, if the same method is a method of both the local business interface and local component interface, the same transaction attribute applies to the method for both interfaces. Likewise, if the same method is a method of both the remote business interface and remote component interface, the same transaction attribute applies to the method for both interfaces.

The following is an example of the specification of the transaction attributes in the deployment descriptor. The `updatePhoneNumber` method of the `EmployeeRecord` enterprise bean is assigned the transaction attribute `Mandatory`; all other methods of the `EmployeeRecord` bean are assigned the attribute `Required`. All the methods of the enterprise bean `AardvarkPayroll` are assigned the attribute `RequiresNew`.

```
<ejb-jar>
     ...
     <assembly-descriptor>
          ...
          <container-transaction>
               <method>
                    <ejb-name>EmployeeRecord</ejb-name>
                    <method-name>*</method-name>
               </method>
               <trans-attribute>Required</trans-attribute>
          </container-transaction>

          <container-transaction>
               <method>
                    <ejb-name>EmployeeRecord</ejb-name>
                    <method-name>updatePhoneNumber</method-name>
               </method>
               <trans-attribute>Mandatory</trans-attribute>
          </container-transaction>

          <container-transaction>
               <method>
                    <ejb-name>AardvarkPayroll</ejb-name>
                    <method-name>*</method-name>
               </method>
               <trans-attribute>RequiresNew</trans-attribute>
          </container-transaction>
     </assembly-descriptor>
</ejb-jar>
```

12.4 Application Assembler's Responsibilities

This section describes the view and responsibilities of the Application Assembler.

There is no mechanism for an Application Assembler to affect enterprise beans with bean-managed transaction demarcation. The Application Assembler must not define transaction attributes for an enterprise bean with bean-managed transaction demarcation.

The Application Assembler can use the deployment descriptor transaction attribute mechanism described above to override or change the transaction attributes for enterprise beans using container-managed transaction demarcation.

> *The Application Assembler should exercise caution in the changing the transaction attributes, as the behavior specified by the transaction attributes is typically an intrinsic part of the semantics of an application.*

12.5 Deployer's Responsibilities

The Deployer is permitted to override or change the values of transaction attributes at deployment time.

> *The Deployer should exercise caution in the changing the transaction attributes, as the behavior specified by the transaction attributes is typically an intrinsic part of the semantics of an application.*

Compatibility Note: For applications written to the EJB 2.1 specification (and earlier), the deployer is responsible for ensuring that the methods of the deployed enterprise beans with container-managed transaction demarcation have been assigned a transaction attribute if this has not be specified in the deployment descriptor.

12.6 Container Provider Responsibilities

This section defines the responsibilities of the Container Provider.

Every client method invocation on a session or entity bean via the bean's business interface (and/or home and component interface), web service endpoint, and every invocation of a message listener method on a message-driven bean is interposed by the container, and every connection to a resource manager used by an enterprise bean is obtained via the container. This managed execution environment allows the container to affect the enterprise bean's transaction management.

This does not imply that the container must interpose on every resource manager access performed by the enterprise bean. Typically, the container interposes only on the resource manager connection factory (e.g. a JDBC data source) JNDI look up by registering the container-specific implementation of the resource manager connection factory object. The resource manager connection factory object allows the container to obtain the `javax.transaction.xa.XAResource` *interface as described in the JTA specification and pass it to the transaction manager. After the set up is done, the enterprise bean communicates with the resource manager without going through the container.*

12.6.1 Bean-Managed Transaction Demarcation

This subsection defines the container's responsibilities for the transaction management of enterprise beans with bean-managed transaction demarcation.

Note that only session and message-driven beans can be used with bean-managed transaction demarcation.

The container must manage client invocations to an enterprise bean instance with bean-managed transaction demarcation as follows. When a client invokes a business method via one of the enterprise bean's client view interfaces, the container suspends any transaction that may be associated with the client request. If there is a transaction associated with the instance (this would happen if a stateful session bean instance started the transaction in some previous business method), the container associates the method execution with this transaction. If there are interceptor methods associated with the bean instances, these actions are taken before the interceptor methods are invoked.

The container must make the `javax.transaction.UserTransaction` interface available to the enterprise bean's business method, message listener method, interceptor method, or timeout callback method via dependency injection into the enterprise bean class or interceptor class, and through lookup via the `javax.ejb.EJBContext` interface, and in the JNDI naming context under `java:comp/UserTransaction`. When an instance uses the `javax.transaction.UserTransaction` interface to demarcate a transaction, the container must enlist all the resource managers used by the instance between the `begin` and `commit`—or `rollback`—methods with the transaction. When the instance attempts to commit the transaction, the container is responsible for the global coordination of the transaction commit[52].

In the case of a *stateful* session bean, it is possible that the business method that started a transaction completes without committing or rolling back the transaction. In such a case, the container must retain the association between the transaction and the instance across multiple client calls until the instance commits or rolls back the transaction. When the client invokes the next business method, the container must invoke the business method (and any applicable interceptor methods for the bean) in this transaction context.

If a *stateless* session bean instance starts a transaction in a business method or interceptor method, it must commit the transaction before the business method (or all its interceptor methods) returns. The container must detect the case in which a transaction was started, but not completed, in the business method or interceptor method for the business method, and handle it as follows:

- Log this as an application error to alert the System Administrator.
- Roll back the started transaction.
- Discard the instance of the session bean.
- Throw the `javax.ejb.EJBException`[53]. If the EJB 2.1 client view is used, the container should throw `java.rmi.RemoteException` if the client is a remote client, or throw the `javax.ejb.EJBException` if the client is a local client.

If a message-driven bean instance starts a transaction in a message listener method or interceptor method, it must commit the transaction before the message listener method (or all its interceptor methods) returns. The container must detect the case in which a transaction was started, but not completed, in a message listener method or interceptor method for the message listener method, and handle it as follows:

- Log this as an application error to alert the System Administrator.
- Roll back the started transaction.
- Discard the instance of the message-driven bean.

---

[52] The container typically relies on a transaction manager that is part of the EJB server to perform the two-phase commit across all the enlisted resource managers. If only a single resource manager is involved in the transaction and the deployment descriptor indicates that connection sharing may be used, the container may use the local transaction optimization. See [12] and [15] for further discussion.

[53] If the business interface is a remote business interface that extends `java.rmi.Remote`, the `java.rmi.RemoteException` is thrown to the client instead.

If a session bean or message-driven bean instance starts a transaction in a timeout callback method, it must commit the transaction before the timeout callback method returns. The container must detect the case in which a transaction was started, but not completed, in a timeout callback method, and handle it as follows:

- Log this as an application error to alert the System Administrator.
- Roll back the started transaction.
- Discard the instance of the bean.

The actions performed by the container for an instance with bean-managed transaction are summarized by the following table. T1 is a transaction associated with a client request, T2 is a transaction that is currently associated with the instance (i.e. a transaction that was started but not completed by a previous business method).

Table 9 Container's Actions for Methods of Beans with Bean-Managed Transaction

Client's transaction	Transaction currently associated with instance	Transaction associated with the method
none	none	none
T1	none	none
none	T2	T2
T1	T2	T2

The following items describe each entry in the table:

- If the client request is not associated with a transaction and the instance is not associated with a transaction, or if the bean is a message-driven bean, the container invokes the instance with an unspecified transaction context.
- If the client request is associated with a transaction T1, and the instance is not associated with a transaction, the container suspends the client's transaction association and invokes the method with an unspecified transaction context. The container resumes the client's transaction association (T1) when the method (together with any associated interceptor methods) completes. This case can never happen for a message-driven bean or for the invocation of a web service endpoint method of a stateless session bean.
- If the client request is not associated with a transaction and the instance is already associated with a transaction T2, the container invokes the instance with the transaction that is associated with the instance (T2). This case can never happen for a stateless session bean or a message-driven bean: it can only happen for a stateful session bean.
- If the client is associated with a transaction T1, and the instance is already associated with a transaction T2, the container suspends the client's transaction association and invokes the

method with the transaction context that is associated with the instance (T2). The container resumes the client's transaction association (T1) when the method (together with any associated interceptor methods) completes. This case can never happen for a stateless session bean or a message-driven bean: it can only happen for a stateful session bean.

The container must allow the enterprise bean instance to serially perform several transactions in a method.

When an instance attempts to start a transaction using the `begin` method of the `javax.transaction.UserTransaction` interface while the instance has not committed the previous transaction, the container must throw the `javax.transaction.NotSupportedException` in the `begin` method.

The container must throw the `java.lang.IllegalStateException` if an instance of a bean with bean-managed transaction demarcation attempts to invoke the `setRollbackOnly` or `getRollbackOnly` method of the `javax.ejb.EJBContext` interface.

12.6.2 Container-Managed Transaction Demarcation for Session and Entity Beans

The container is responsible for providing the transaction demarcation for the session beans declared with container-managed transaction demarcation, entity beans with bean-managed persistence, and for EJB 2.1 and EJB 1.1 entity beans with container-managed persistence. For these enterprise beans, the container must demarcate transactions as specified by the transaction attribute values specified using metadata annotations in the bean class or specified in the deployment descriptor.

The following subsections define the responsibilities of the container for managing the invocation of an enterprise bean business method when the method is invoked via the enterprise bean's business interface (and/or home or component interface), or web service endpoint. The container's responsibilities depend on the value of the transaction attribute.

12.6.2.1 NOT_SUPPORTED

The container invokes an enterprise bean method whose transaction attribute is set to the `NOT_SUPPORTED` value with an unspecified transaction context.

If a client calls with a transaction context, the container suspends the association of the transaction context with the current thread before invoking the enterprise bean's business method. The container resumes the suspended association when the business method has completed. The suspended transaction context of the client is not passed to the resource managers or other enterprise bean objects that are invoked from the business method.

If the business method invokes other enterprise beans, the container passes no transaction context with the invocation.

Refer to Subsection 12.6.5 for more details of how the container can implement this case.

12.6.2.2 REQUIRED

The container must invoke an enterprise bean method whose transaction attribute is set to the `REQUIRED` value with a valid transaction context.

If a client invokes the enterprise bean's method while the client is associated with a transaction context, the container invokes the enterprise bean's method in the client's transaction context.

If the client invokes the enterprise bean's method while the client is not associated with a transaction context, the container automatically starts a new transaction before delegating a method call to the enterprise bean business method. The container automatically enlists all the resource managers accessed by the business method with the transaction. If the business method invokes other enterprise beans, the container passes the transaction context with the invocation. The container attempts to commit the transaction when the business method has completed. The container performs the commit protocol before the method result is sent to the client.

12.6.2.3 SUPPORTS

The container invokes an enterprise bean method whose transaction attribute is set to `SUPPORTS` as follows.

- If the client calls with a transaction context, the container performs the same steps as described in the `REQUIRED` case.
- If the client calls without a transaction context, the container performs the same steps as described in the `NOT_SUPPORTED` case.

The SUPPORTS transaction attribute must be used with caution. This is because of the different transactional semantics provided by the two possible modes of execution. Only the enterprise beans that will execute correctly in both modes should use the `SUPPORTS` transaction attribute.

12.6.2.4 REQUIRES_NEW

The container must invoke an enterprise bean method whose transaction attribute is set to `REQUIRES_NEW` with a new transaction context.

If the client invokes the enterprise bean's method while the client is not associated with a transaction context, the container automatically starts a new transaction before delegating a method call to the enterprise bean business method. The container automatically enlists all the resource managers accessed by the business method with the transaction. If the business method invokes other enterprise beans, the container passes the transaction context with the invocation. The container attempts to commit the transaction when the business method has completed. The container performs the commit protocol before the method result is sent to the client.

If a client calls with a transaction context, the container suspends the association of the transaction context with the current thread before starting the new transaction and invoking the business method. The container resumes the suspended transaction association after the business method and the new transaction have been completed.

12.6.2.5 MANDATORY

The container must invoke an enterprise bean method whose transaction attribute is set to `MANDATORY` in a client's transaction context. The client is required to call with a transaction context.

- If the client calls with a transaction context, the container performs the same steps as described in the `REQUIRED` case.
- If the client calls without a transaction context, the container throws the `javax.ejb.EJBTransactionRequiredException`[54]. If the EJB 2.1 client view is used, the container throws the `javax.transaction.TransactionRequiredException` exception if the client is a remote client, or the `javax.ejb.TransactionRequiredLocalException` if the client is a local client.

12.6.2.6 NEVER

The container invokes an enterprise bean method whose transaction attribute is set to `NEVER` without a transaction context defined by the EJB specification. The client is required to call without a transaction context.

- If the client calls with a transaction context, the container throws the `javax.ejb.EJBException`[55]. If the EJB 2.1 client view is used, the container throws the `java.rmi.RemoteException` exception if the client is a remote client, or the `javax.ejb.EJBException` if the client is a local client.
- If the client calls without a transaction context, the container performs the same steps as described in the `NOT_SUPPORTED` case.

12.6.2.7 Transaction Attribute Summary

The following table provides a summary of the transaction context that the container passes to the business method and resource managers used by the business method, as a function of the transaction attribute and the client's transaction context. T1 is a transaction passed with the client request, while T2 is a transaction initiated by the container.

Table 10 Transaction Attribute Summary

Transaction attribute	Client's transaction	Transaction associated with business method	Transaction associated with resource managers
NOT_SUPPORTED	none	none	none
	T1	none	none

[54] If the business interface is a remote business interface that extends `java.rmi.Remote`, the `javax.transaction.TransactionRequiredException` is thrown to the client instead.

[55] If the business interface is a remote business interface that extends `java.rmi.Remote`, the `java.rmi.RemoteException` is thrown to the client instead.

Table 10 Transaction Attribute Summary

Transaction attribute	Client's transaction	Transaction associated with business method	Transaction associated with resource managers
REQUIRED	none	T2	T2
	T1	T1	T1
SUPPORTS	none	none	none
	T1	T1	T1
REQUIRES_NEW	none	T2	T2
	T1	T2	T2
MANDATORY	none	**error**	N/A
	T1	T1	T1
NEVER	none	none	none
	T1	**error**	N/A

If the enterprise bean's business method invokes other enterprise beans via their business interfaces or home and component interfaces, the transaction indicated in the column "Transaction associated with business method" will be passed as part of the client context to the target enterprise bean.

See Subsection 12.6.5 for how the container handles the "none" case in Table 10.

12.6.2.8 Handling of setRollbackOnly Method

The container must handle the `EJBContext.setRollbackOnly` method invoked from a business method executing with the `REQUIRED`, `REQUIRES_NEW`, or `MANDATORY` transaction attribute as follows:

- The container must ensure that the transaction will never commit. Typically, the container instructs the transaction manager to mark the transaction for rollback.
- If the container initiated the transaction immediately before dispatching the business method to the instance (as opposed to the transaction being inherited from the caller), the container must note that the instance has invoked the `setRollbackOnly` method. When the business method invocation completes, the container must roll back rather than commit the transaction. If the business method has returned normally or with an application exception, the container must pass the method result or the application exception to the client after the container performed the rollback.

The container must throw the `java.lang.IllegalStateException` if the `EJBContext.setRollbackOnly` method is invoked from a business method executing with the `SUPPORTS`, `NOT_SUPPORTED`, or `NEVER` transaction attribute.

12.6.2.9 Handling of getRollbackOnly Method

The container must handle the `EJBContext.getRollbackOnly` method invoked from a business method executing with the `REQUIRED`, `REQUIRES_NEW`, or `MANDATORY` transaction attribute.

The container must throw the `java.lang.IllegalStateException` if the `EJBContext.getRollbackOnly` method is invoked from a business method executing with the `SUPPORTS`, `NOT_SUPPORTED`, or `NEVER` transaction attribute.

12.6.2.10 Handling of getUserTransaction Method

If an instance of an enterprise bean with container-managed transaction demarcation attempts to invoke the `getUserTransaction` method of the `EJBContext` interface, the container must throw the `java.lang.IllegalStateException`.

12.6.2.11 javax.ejb.SessionSynchronization Callbacks

If a session bean class implements the `javax.ejb.SessionSynchronization` interface, the container must invoke the `afterBegin`, `beforeCompletion`, and `afterCompletion` callbacks on the instance as part of the transaction commit protocol.

The container invokes the `afterBegin` method on an instance before it invokes the first business method in a transaction.

The container invokes the `beforeCompletion` method to give the enterprise bean instance the last chance to cause the transaction to rollback. The instance may cause the transaction to roll back by invoking the `EJBContext.setRollbackOnly` method.

The container invokes the `afterCompletion(Boolean committed)` method after the completion of the transaction commit protocol to notify the enterprise bean instance of the transaction outcome.

12.6.3 Container-Managed Transaction Demarcation for Message-Driven Beans

The container is responsible for providing the transaction demarcation for the message-driven beans that the Bean Provider declared as with container-managed transaction demarcation. For these enterprise beans, the container must demarcate transactions as specified by annotations on the bean class or in the deployment descriptor. (See Chapter 18 for more information about the deployment descriptor.)

The following subsections define the responsibilities of the container for managing the invocation of a message-driven bean's message listener method. The container's responsibilities depend on the value of the transaction attribute.

Only the `NOT_SUPPORTED` and `REQUIRED` transaction attributes may be used for message-driven bean message listener methods. The use of the other transaction attributes is not meaningful for message-driven bean message listener methods because there is no pre-existing client transaction context (`REQUIRES_NEW`, `SUPPORTS`) and no client to handle exceptions (`MANDATORY`, `NEVER`).

12.6.3.1 NOT_SUPPORTED

The container invokes a message-driven bean message listener method whose transaction attribute is set to `NOT_SUPPORTED` with an unspecified transaction context.

If the message listener method invokes other enterprise beans, the container passes no transaction context with the invocation.

12.6.3.2 REQUIRED

The container must invoke a message-driven bean message listener method whose transaction attribute is set to `REQUIRED` with a valid transaction context. The resource managers accessed by the message listener method within the transaction are enlisted with the transaction. If the message listener method invokes other enterprise beans, the container passes the transaction context with the invocation. The container attempts to commit the transaction when the message listener method has completed.

Messaging systems may differ in quality of service with regard to reliability and transactionality of the dequeuing of messages.

The requirement for JMS are as follows:

> *A transaction must be started before the dequeuing of the JMS message and, hence, before the invocation of the message-driven bean's `onMessage` method. The resource manager associated with the arriving message is enlisted with the transaction as well as all the resource managers accessed by the `onMessage` method within the transaction. If the `onMessage` method invokes other enterprise beans, the container passes the transaction context with the invocation. The transaction is committed when the `onMessage` method has completed. If the `onMessage` method does not successfully complete or the transaction is rolled back, message redelivery semantics apply.*

12.6.3.3 Handling of setRollbackOnly Method

The container must handle the `EJBContext.setRollbackOnly` method invoked from a message listener method executing with the `REQUIRED` transaction attribute as follows:

- The container must ensure that the transaction will never commit. Typically, the container instructs the transaction manager to mark the transaction for rollback.
- The container must note that the instance has invoked the `setRollbackOnly` method. When the method invocation completes, the container must roll back rather than commit the transaction.

The container must throw and log the `java.lang.IllegalStateException` if the `EJBContext.setRollbackOnly` method is invoked from a message listener method executing with the `NotSupported` transaction attribute

12.6.3.4 Handling of getRollbackOnly Method

The container must handle the `EJBContext.getRollbackOnly()` method invoked from a message listener method executing with the `REQUIRED` transaction attribute.

The container must throw and log the `java.lang.IllegalStateException` if the `EJBContext.getRollbackOnly` method is invoked from a message listener method executing with the `NOT_SUPPORTED` transaction attribute.

12.6.3.5 Handling of getUserTransaction Method

If an instance of a message-driven bean with container-managed transaction demarcation attempts to invoke the `getUserTransaction` method of the `EJBContext` interface, the container must throw and log the `java.lang.IllegalStateException`.

12.6.4 Local Transaction Optimization

The container may use a local transaction optimization for enterprise beans whose metadata annotations or deployment descriptor indicates that connections to a resource manager are shareable (see Section 15.7.1.3, “Declaration of Resource Manager Connection Factory References in Deployment Descriptor”). The container manages the use of the local transaction optimization transparent to the application.

The container may use the optimization for transactions initiated by the container for a bean with container-managed transaction demarcation and for transactions initiated by a bean with bean-managed transaction demarcation with the `UserTransaction` interface. The container cannot apply the optimization for transactions imported from a different container.

The use of local transaction optimization approach is discussed in [12] and [15].

12.6.5 Handling of Methods that Run with “an unspecified transaction context”

The term “an unspecified transaction context” is used in the EJB specification to refer to the cases in which the EJB architecture does not fully define the transaction semantics of an enterprise bean method execution.

This includes the following cases:

- The execution of a method of an enterprise bean with container-managed transaction demarcation for which the value of the transaction attribute is `NOT_SUPPORTED`, `NEVER`, or `SUPPORTS`.
- The execution of a `PostConstruct`, `PreDestroy`, `PostActivate`, or `PrePassivate` callback method of a session bean with container-managed transaction demarcation.[56]

[56] See Chapter 4, “Session Bean Component Contract”.

- The execution of a `PostConstruct` or `PreDestroy` callback method of a message-driven bean with container-managed transaction demarcation.[57]

The EJB specification does not prescribe how the container should manage the execution of a method with an unspecified transaction context—the transaction semantics are left to the container implementation. Some techniques for how the container may choose to implement the execution of a method with an unspecified transaction context are as follows (the list is not inclusive of all possible strategies):

- The container may execute the method and access the underlying resource managers without a transaction context.
- The container may treat each call of an instance to a resource manager as a single transaction (e.g. the container may set the auto-commit option on a JDBC connection).
- The container may merge multiple calls of an instance to a resource manager into a single transaction.
- The container may merge multiple calls of an instance to multiple resource managers into a single transaction.
- If an instance invokes methods on other enterprise beans, and the invoked methods are also designated to run with an unspecified transaction context, the container may merge the resource manager calls from the multiple instances into a single transaction.
- Any combination of the above.

Since the enterprise bean does not know which technique the container implements, the enterprise bean must be written conservatively not to rely on any particular container behavior.

A failure that occurs in the middle of the execution of a method that runs with an unspecified transaction context may leave the resource managers accessed from the method in an unpredictable state. The EJB architecture does not define how the application should recover the resource managers' state after such a failure.

12.7 Access from Multiple Clients in the Same Transaction Context

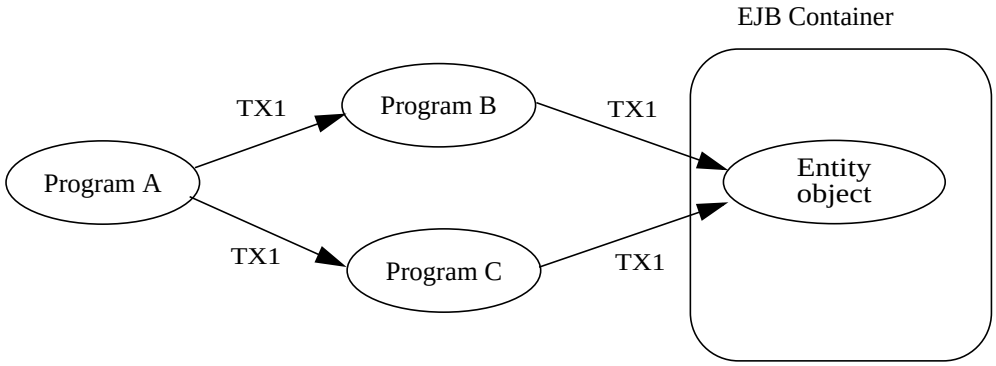
This section describes a more complex distributed transaction scenario, and specifies the container's behavior required for this scenario.

[57] See Chapter 5, "Message-Driven Bean Component Contract".

12.7.1 Transaction "Diamond" Scenario with an Entity Object

An entity object may be accessed by multiple clients in the same transaction. For example, program A may start a transaction, call program B and program C in the transaction context, and then commit the transaction. If programs B and C access the same entity object, the topology of the transaction creates a diamond.

Figure 28 Transaction Diamond Scenario with Entity Object



An example (not realistic in practice) is a client program that tries to perform two purchases at two different stores within the same transaction. At each store, the program that is processing the client's purchase request debits the client's bank account.

It is difficult to implement an EJB server that handles the case in which programs B and C access an entity object through different network paths. This case is challenging because many EJB servers implement the EJB container as a collection of multiple processes, running on the same or multiple machines. Each client is typically connected to a single process. If clients B and C connect to different EJB container processes, and both B and C need to access the same entity object in the same transaction, the issue is how the container can make it possible for B and C to see a consistent state of the entity object within the same transaction[58].

The above example illustrates a simple diamond. We use the term diamond to refer to any distributed transaction scenario in which an entity object is accessed in the same transaction through multiple network paths.

Note that in the diamond scenario the clients B and C access the entity object serially. Concurrent access to an entity object in the same transaction context would be considered an application programming error, and it would be handled in a container-specific way.

Note that the issue of handling diamonds is not unique to the EJB architecture. This issue exists in all distributed transaction processing systems.

[58] This diamond problem applies only to the case when B and C are in the same transaction.

The following subsections define the responsibilities of the EJB Roles when handling distributed transaction topologies that may lead to a diamond involving an entity object.

12.7.2 Container Provider's Responsibilities

This Subsection specifies the EJB container's responsibilities with respect to the diamond case involving an entity object.

The EJB specification requires that the container provide support for local diamonds. In a local diamond, components A, B, C, and D are deployed in the same EJB container.

The EJB specification does not require an EJB container to support distributed diamonds. In a distributed diamond, a target entity object is accessed from multiple clients in the same transaction through multiple network paths, and the clients (programs B and C) are not enterprise beans deployed in the same EJB container as the target entity object.

If the Container Provider chooses not to support distributed diamonds, and if the container can detect that a client invocation would lead to a diamond, the container should throw the `javax.ejb.EJBException` (or `java.rmi.RemoteException` if the EJB 2.1 remote client view is used).

12.7.3 Bean Provider's Responsibilities

This Subsection specifies the Bean Provider's responsibilities with respect to the diamond case involving an entity object.

The diamond case is transparent to the Bean Provider—the Bean Provider does not have to code the enterprise bean differently for the bean to participate in a diamond. Any solution to the diamond problem implemented by the container is transparent to the bean and does not change the semantics of the bean.

12.7.4 Application Assembler and Deployer's Responsibilities

This Subsection specifies the Application Assembler and Deployer's responsibilities with respect to the diamond case involving an entity object.

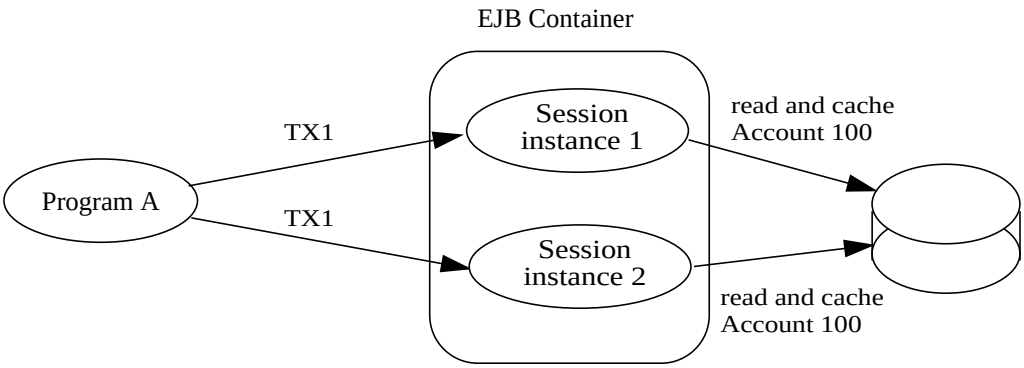
The Application Assembler and Deployer should be aware that distributed diamonds might occur. In general, the Application Assembler should try to avoid creating unnecessary distributed diamonds.

If a distributed diamond is necessary, the Deployer should advise the container (using a container-specific API) that an entity objects of the entity bean may be involved in distributed diamond scenarios.

12.7.5 Transaction Diamonds involving Session Objects

While it is illegal for two clients to access the same session object, it is possible for applications that use session beans to encounter the diamond case. For example, program A starts a transaction and then invokes two different session objects.

Figure 29 Transaction Diamond Scenario with a Session Bean



If the session bean instances cache the same data item (e.g. the current balance of Account 100) across method invocations in the same transaction, most likely the program is going to produce incorrect results.

The problem may exist regardless of whether the two session objects are the same or different session beans. The problem may exist (and may be harder to discover) if there are intermediate objects between the transaction initiator and the session objects that cache the data.

There are no requirements for the Container Provider because it is impossible for the container to detect this problem.

The Bean Provider and Application Assembler must avoid creating applications that would result in inconsistent caching of data in the same transaction by multiple session objects.

Chapter 13 Exception Handling

13.1 Overview and Concepts

13.1.1 Application Exceptions

An *application exception* is an exception defined by the Bean Provider as part of the business logic of an application. *Application exceptions* are distinguished from *system exceptions* in this specification.

Enterprise bean business methods use application exceptions to inform the client of abnormal application-level conditions, such as unacceptable values of the input arguments to a business method. A client can typically recover from an application exception. Application exceptions are not intended for reporting system-level problems.

For example, the `Account` *enterprise bean may throw an application exception to report that a* `debit` *operation cannot be performed because of an insufficient balance. The* `Account` *bean should not use an application exception to report, for example, the failure to obtain a database connection.*

An application exception may be defined in the `throws` clause of a method of an enterprise bean's business interface, home interface, component interface, message listener interface, or web service endpoint.

An application exception may be a subclass (direct or indirect) of `java.lang.Exception` (i.e., a "checked exception"), or an application exception class may be defined as a subclass of the `java.lang.RuntimeException` (an "unchecked exception"). An application exception may not be a subclass of the `java.rmi.RemoteException`. The `java.rmi.RemoteException` and its subclasses are reserved for system exceptions.

The `javax.ejb.CreateException`, `javax.ejb.RemoveException`, `javax.ejb.FinderException`, and subclasses thereof are considered to be application exceptions. These exceptions are used as standard application exceptions to report errors to the client from the `create`, `remove`, and `finder` methods of the EJBHome and/or EJBLocalHome interfaces of components written to the EJB 2.1 client view (see Subsections 8.5.10 and 9.1.11). These exceptions are covered by the rules on application exceptions that are defined in this chapter.

13.1.2 Goals for Exception Handling

The EJB specification for exception handling is designed to meet these high-level goals:

- An application exception thrown by an enterprise bean instance should be reported to the client *precisely* (i.e., the client gets the same exception)[59].
- An application exception thrown by an enterprise bean instance should not automatically rollback a client's transaction unless the application exception was defined to cause transaction rollback. The client should typically be given a chance to recover a transaction from an application exception.
- An unexpected exception that may have left the instance's state variables and/or underlying persistent data in an inconsistent state can be handled safely.

13.2 Bean Provider's Responsibilities

This section describes the view and responsibilities of the Bean Provider with respect to exception handling.

13.2.1 Application Exceptions

The Bean Provider defines application exceptions. Application exceptions that are checked exceptions may be defined as such by being listed in the `throws` clauses of the methods of the bean's business interface, home interface, component interface, and web service endpoint. An application exception that is an unchecked exception is defined as an application exception by annotating it with the `ApplicationException` metadata annotation, or denoting it in the deployment descriptor with the `application-exception` element.

[59] This may not be the case where web services protocols are used. See [25].

Because application exceptions are intended to be handled by the client, and not by the System Administrator, they should be used only for reporting business logic exceptions, not for reporting system level problems.

> *Certain messaging types may define application exceptions in their message listener interfaces. The resource adapter in use for the particular messaging type determines how the exception is processed. See [15].*

The Bean Provider is responsible for throwing the appropriate application exception from the business method to report a business logic exception to the client.

An application exception does not automatically result in marking the transaction for rollback unless the `ApplicationException` annotation is applied to the exception class and is specified with the `rollback` element value `true`, or the `application-exception` deployment descriptor element for the exception specifies the `rollback` element as `true`.

The Bean Provider must do one of the following to ensure data integrity before throwing an application exception from an enterprise bean instance:

- Ensure that the instance is in a state such that a client's attempt to continue and/or commit the transaction does not result in loss of data integrity. For example, the instance throws an application exception indicating that the value of an input parameter was invalid before the instance performed any database updates.

- If the application exception is not specified to cause transaction rollback, mark the transaction for rollback using the `EJBContext.setRollbackOnly` method before throwing the application exception. Marking the transaction for rollback will ensure that the transaction can never commit.

The Bean Provider is also responsible for using the standard EJB application exceptions (`javax.ejb.CreateException`, `javax.ejb.RemoveException`, `javax.ejb.FinderException`, and subclasses thereof) for beans written to the EJB 2.1 and earlier client view as described in Subsections 8.5.10 and 9.1.11.

Bean Providers may define subclasses of the standard EJB application exceptions and throw instances of the subclasses in the enterprise bean methods. A subclass will typically provide more information to the client that catches the exception.

13.2.2 System Exceptions

A system exception is an exception that is a `java.rmi.RemoteException` (or one of its subclasses) or a `RuntimeException` that is not an application exception.

This subsection describes how the Bean Provider should handle various system-level exceptions and errors that an enterprise bean instance may encounter during the execution of a session or entity bean business method, a message-driven bean message listener method, an interceptor method, or a callback method (e.g. `ejbLoad`).

An enterprise bean business method, message listener method, business method interceptor method, or lifecycle callback interceptor method may encounter various exceptions or errors that prevent the method from successfully completing. Typically, this happens because the exception or error is unexpected, or the exception is expected but the EJB Provider does not know how to recover from it. Examples of such exceptions and errors are: failure to obtain a database connection, JNDI exceptions, unexpected `RemoteException` from invocation of other enterprise beans[60], unexpected `RuntimeException`, JVM errors, and so on.

If the enterprise bean method encounters a system-level exception or error that does not allow the method to successfully complete, the method should throw a suitable non-application exception that is compatible with the method's `throws` clause. While the EJB specification does not prescribe the exact usage of the exception, it encourages the Bean Provider to follow these guidelines:

- If the bean method encounters a system exception or error, it should simply propagate the error from the bean method to the container (i.e., the bean method does not have to catch the exception).
- If the bean method performs an operation that results in a checked exception[61] that the bean method cannot recover, the bean method should throw the `javax.ejb.EJBException` that wraps the original exception.
- Any other unexpected error conditions should be reported using the `javax.ejb.EJBException`.

Note that the `javax.ejb.EJBException` is a subclass of the `java.lang.RuntimeException`, and therefore it does not have to be listed in the `throws` clauses of the business methods.

The container catches a non-application exception; logs it (which can result in alerting the System Administrator); and, unless the bean is a message-driven bean, throws the `javax.ejb.EJBException`[62] or, if the web service client view is used, the `java.rmi.RemoteException`. If the EJB 2.1 client view is used, the container throws the `java.rmi.RemoteException` (or subclass thereof) to the client if the client is a remote client, or throws the `javax.ejb.EJBException` (or subclass thereof) to the client if the client is a local client. In the case of a message-driven bean, the container logs the exception and then throws a `javax.ejb.EJBException` that wraps the original exception to the resource adapter. (See [15]).

The exception that is seen by the client is described in section 13.3. It is determined both by the exception that is thrown by the container and/or bean and the client view.

The Bean Provider can rely on the container to perform the following tasks when catching a non-application exception:

- The transaction in which the bean method participated will be rolled back.

---

[60] Note that the enterprise bean business method may attempt to recover from a `RemoteException`. The text in this subsection applies only to the case when the business method does not wish to recover from the `RemoteException`.

[61] A checked exception is one that is not a subclass of `java.lang.RuntimeException`.

[62] If the business interface is a remote business interface that extends `java.rmi.Remote`, the `java.rmi.RemoteException` is thrown to the client instead.

- No other method will be invoked on an instance that threw a non-application exception.

This means that the Bean Provider does not have to perform any cleanup actions before throwing a non-application exception. It is the container that is responsible for the cleanup.

13.2.2.1 javax.ejb.NoSuchEntityException

The `NoSuchEntityException` is a subclass of `EJBException`. It should be thrown by the EJB 2.1 entity bean class methods to indicate that the underlying entity has been removed from the database.

A bean-managed persistence entity bean class typically throws this exception from the `ejbLoad` and `ejbStore` methods, and from the methods that implement the business methods defined in the component interface.

13.3 Container Provider Responsibilities

This section describes the responsibilities of the Container Provider for handling exceptions. The EJB architecture specifies the container's behavior for the following exceptions:

- Exceptions from the business methods of session and entity beans.
- Exceptions from message-driven bean methods
- Exceptions from timeout callback methods
- Exceptions from other container-invoked callbacks on the enterprise bean.
- Exceptions from management of container-managed transaction demarcation.

13.3.1 Exceptions from a Session Bean's Business Interface Methods

Table 13 specifies how the container must handle the exceptions thrown by the methods of the business interface for beans with container-managed transaction demarcation, including the exceptions thrown by business method interceptor methods which intercept the invocation of business methods. The table specifies the container's action as a function of the condition under which the business interface method executes and the exception thrown by the method. The table also illustrates the exception that the client will receive and how the client can recover from the exception. (Section 13.4 describes the client's view of exceptions in detail.) The notation "AppException" denotes an application exception.

Table 11 Handling of Exceptions Thrown by a Business Interface Method of a Bean with Container-Managed Transaction Demarcation

Method condition	Method exception	Container's action	Client's view
Bean method runs in the context of the caller's transaction [Note A]. This case may happen with `Required`, `Mandatory`, and `Supports` attributes.	AppException	Re-throw AppException. Mark the transaction for rollback if the application exception is specified as causing rollback.	Receives AppException. Can attempt to continue computation in the transaction, and eventually commit the transaction unless the application exception is specified as causing rollback (the commit would fail if the instance called `setRollbackOnly`).
	all other exceptions and errors	Log the exception or error [Note B]. Mark the transaction for rollback. Discard instance [Note C]. Throw `javax.ejb.EJBTransactionRolledbackException` to client. [Note D]	Receives `javax.ejb.EJBTransactionRolledbackException` Continuing transaction is fruitless.

Table 11 Handling of Exceptions Thrown by a Business Interface Method of a Bean with Container-Managed Transaction Demarcation

Method condition	Method exception	Container's action	Client's view
Bean method runs in the context of a transaction that the container started immediately before dispatching the business method. This case may happen with `Required` and `RequiresNew` attributes.	AppException	If the instance called `setRollbackOnly()`, then rollback the transaction, and re-throw AppException. Mark the transaction for rollback if the application exception is specified as causing rollback, and then re-throw AppException. Otherwise, attempt to commit the transaction, and then re-throw AppException.	Receives AppException. If the client executes in a transaction, the client's transaction is not marked for rollback, and client can continue its work.
	all other exceptions	Log the exception or error. Rollback the container-started transaction. Discard instance. Throw `EJBException` to client.[Note E]	Receives `EJBException`. If the client executes in a transaction, the client's transaction may or may not be marked for rollback.
Bean method runs with an unspecified transaction context. This case may happen with the `NotSupported`, `Never`, and `Supports` attributes.	AppException	Re-throw AppException.	Receives AppException. If the client executes in a transaction, the client's transaction is not marked for rollback, and client can continue its work.
	all other exceptions	Log the exception or error. Discard instance. Throw `EJBException` to client.[Note F]	Receives `EJBException`. If the client executes in a transaction, the client's transaction may or may not be marked for rollback.

Notes:

[A] The caller can be another enterprise bean or an arbitrary client program.

[B] *Log the exception or error* means that the container logs the exception or error so that the System Administrator is alerted of the problem.

[C] *Discard instance* means that the container must not invoke any business methods or container callbacks on the instance.

[D] If the business interface is a remote business interface that extends java.rmi.Remote, the javax.transaction.TransactionRolledbackException is thrown to the client, which will receive this exception.

[E] If the business interface is a remote business interface that extends java.rmi.Remote, the java.rmi.RemoteException is thrown to the client, which will receive this exception.

[F] If the business interface is a remote business interface that extends java.rmi.Remote, the java.rmi.RemoteException is thrown to the client, which will receive this exception.

Table 12 specifies how the container must handle the exceptions thrown by the methods of the business interface for beans with bean-managed transaction demarcation, including the exceptions thrown by business method interceptor methods which intercept the invocation of business methods. The table specifies the container's action as a function of the condition under which the business interface method executes and the exception thrown by the method. The table also illustrates the exception that the client will receive and how the client can recover from the exception. (Section 13.4 describes the client's view of exceptions in detail.)

Table 12 Handling of Exceptions Thrown by a Business Interface Method of a Session Bean with Bean-Managed Transaction Demarcation

Bean method condition	Bean method exception	Container action	Client receives
Bean is stateful or stateless session.	AppException	Re-throw AppException	Receives AppException.
	all other exceptions	Log the exception or error. Mark for rollback a transaction that has been started, but not yet completed, by the instance. Discard instance. Throw `EJBException` to client. [Note A]	Receives `EJBException`.

Notes:

[A] If the business interface is a remote business interface that extends java.rmi.Remote, the java.rmi.RemoteException is thrown to the client, which will receive this exception.

13.3.2 Exceptions from Method Invoked via Session or Entity Bean's 2.1 Client View or through Web Service Client View

Business methods in this context are considered to be the methods defined in the enterprise bean's business interface, home interface, component interface, or web service endpoint (including superinterfaces of these); and the following session bean or entity bean methods: `ejbCreate<METHOD>`, `ejbPostCreate<METHOD>`, `ejbRemove`, `ejbHome<METHOD>`, and `ejbFind<METHOD>` methods.

Table 13 specifies how the container must handle the exceptions thrown by the business methods for beans with container-managed transaction demarcation, including the exceptions thrown by business method interceptor methods which intercept the invocation of business methods. The table specifies the container's action as a function of the condition under which the business method executes and the exception thrown by the business method. The table also illustrates the exception that the client will receive and how the client can recover from the exception. (Section 13.4 describes the client's view of exceptions in detail.) The notation "AppException" denotes an application exception.

Table 13 Handling of Exceptions Thrown by Methods of Web Service Client View or EJB 2.1 Client View of a Bean with Container-Managed Transaction Demarcation

Method condition	Method exception	Container's action	Client's view
Bean method runs in the context of the caller's transaction [Note A]. This case may happen with `Required`, `Mandatory`, and `Supports` attributes.	AppException	Re-throw AppException Mark the transaction for rollback if the application exception is specified as causing rollback.	Receives AppException. Can attempt to continue computation in the transaction, and eventually commit the transaction unless the application exception is specified as causing rollback (the commit would fail if the instance called `setRollbackOnly`).
	all other exceptions and errors	Log the exception or error [Note B]. Mark the transaction for rollback. Discard instance [Note C]. Throw `javax.transaction.TransactionRolledbackException` to remote client; throw `javax.ejb.TransactionRolledbackLocalException` to local client.	Receives `javax.transaction.TransactionRolledbackException` or `javax.ejb.TransactionRolledbackLocalException` Continuing transaction is fruitless.

Table 13 Handling of Exceptions Thrown by Methods of Web Service Client View or EJB 2.1 Client View of a Bean with Container-Managed Transaction Demarcation

Method condition	Method exception	Container's action	Client's view
Bean method runs in the context of a transaction that the container started immediately before dispatching the business method. This case may happen with `Required` and `RequiresNew` attributes.	AppException	If the instance called `setRollbackOnly()`, then rollback the transaction, and re-throw AppException. Mark the transaction for rollback if the application exception is specified as causing rollback, and then re-throw AppException. Otherwise, attempt to commit the transaction, and then re-throw AppException.	Receives AppException. If the client executes in a transaction, the client's transaction is not marked for rollback, and client can continue its work.
	all other exceptions	Log the exception or error. Rollback the container-started transaction. Discard instance. Throw `RemoteException` to remote or web service client [Note D]; throw `EJBException` to local client.	Receives `RemoteException` or `EJBException`. If the client executes in a transaction, the client's transaction may or may not be marked for rollback.
Bean method runs with an unspecified transaction context. This case may happen with the `NotSupported`, `Never`, and `Supports` attributes.	AppException	Re-throw AppException.	Receives AppException. If the client executes in a transaction, the client's transaction is not marked for rollback, and client can continue its work.
	all other exceptions	Log the exception or error. Discard instance. Throw `RemoteException` to remote or web service client; throw `EJBException` to local client.	Receives `RemoteException` or `EJBException`. If the client executes in a transaction, the client's transaction may or may not be marked for rollback.

Notes:

[A] The caller can be another enterprise bean or an arbitrary client program. This case is not applicable for methods of the web service endpoint.

[B] *Log the exception or error* means that the container logs the exception or error so that the System Administrator is alerted of the problem.

[C] *Discard instance* means that the container must not invoke any business methods or container callbacks on the instance.

[D] Throw `RemoteException` to web service client means that the container maps the `RemoteException` to the appropriate SOAP fault. See [25].

Table 14 specifies how the container must handle the exceptions thrown by the business methods for beans with bean-managed transaction demarcation, including the exceptions thrown by business method interceptor methods which intercept the invocation of business methods. The table specifies the container's action as a function of the condition under which the business method executes and the exception thrown by the business method. The table also illustrates the exception that the client will receive and how the client can recover from the exception. (Section 13.4 describes the client's view of exceptions in detail.)

Table 14 Handling of Exceptions Thrown by a Business Method of a Session Bean with Bean-Managed Transaction Demarcation

Bean method condition	Bean method exception	Container action	Client receives
Bean is stateful or stateless session.	AppException	Re-throw AppException	Receives AppException.
	all other exceptions	Log the exception or error. Mark for rollback a transaction that has been started, but not yet completed, by the instance. Discard instance. Throw `RemoteException` to remote or web service client [Note A]; throw `EJBException` to local client.	Receives `RemoteException` or `EJBException`.

Notes:

[A] Throw `RemoteException` to web service client means that the container maps the `RemoteException` to the appropriate SOAP fault. See [25].

13.3.3 Exceptions from PostConstruct and PreDestroy Methods of a Stateless Session Bean with Web Service Client View

Table 15 specifies how the container must handle the exceptions thrown by the `PostConstruct` and `PreDestroy` methods for stateless session beans with a web service client view.

Table 15 Handling of Exceptions Thrown by a PostConstruct or PreDestroy Method of a Stateless Session Bean with Web Service Client View.

Bean method condition	Bean method exception	Container action
Bean is stateless session bean with web service client view	system exceptions	Log the exception or error. Discard instance.

13.3.4 Exceptions from Message-Driven Bean Message Listener Methods

This section specifies the container's handling of exceptions thrown from a message-driven bean's message listener method.

Table 16 specifies how the container must handle the exceptions thrown by a message listener method of a message-driven bean with container-managed transaction demarcation, including the exceptions thrown by business method interceptor methods which intercept the invocation of message listener methods. The table specifies the container's action as a function of the condition under which the method executes and the exception thrown by the method.

Table 16 Handling of Exceptions Thrown by a Message Listener Method of a Message-Driven Bean with Container-Managed Transaction Demarcation.

Method condition	Method exception	Container's action
Bean method runs in the context of a transaction that the container started immediately before dispatching the method. This case happens with `Required` attribute.	AppException Mark the transaction for rollback if the application exception is specified as causing rollback.	If the instance called `setRollbackOnly`, rollback the transaction and re-throw AppException to resource adapter. Otherwise, attempt to commit the transaction unless the application exception is specified as causing rollback and re-throw AppException to resource adapter.
	system exceptions	Log the exception or error[Note A]. Rollback the container-started transaction. Discard instance[Note B]. Throw EJBException that wraps the original exception to resource adapter.
Bean method runs with an unspecified transaction context. This case happens with the `NotSupported` attribute.	AppException	Re-throw AppException to resource adapter.
	system exceptions	Log the exception or error. Discard instance. Throw EJBException that wraps the original exception to resource adapter

Notes:

[A] *Log the exception or error* means that the container logs the exception or error so that the System Administrator is alerted of the problem.

[B] *Discard instance* means that the container must not invoke any methods on the instance.

Table 17 specifies how the container must handle the exceptions thrown by a message listener method of a message-driven bean with bean-managed transaction demarcation. The table specifies the container's action as a function of the condition under which the method executes and the exception thrown by tthe method.

Table 17 Handling of Exceptions Thrown by a Message Listener Method of a Message-Driven Bean with Bean-Managed Transaction Demarcation.

Bean method condition	Bean method exception	Container action
Bean is message-driven bean	AppException	Re-throw AppException to resource adapter.
	system exceptions	Log the exception or error. Mark for rollback a transaction that has been started, but not yet completed, by the instance. Discard instance. Throw EJBException that wraps the original exception to resource adapter.

13.3.5 Exceptions from PostConstruct and PreDestroy Methods of a Message-Driven Bean

Table 18 specifies how the container must handle the exceptions thrown by the `PostConstruct` and `PreDestroy` methods of message-driven beans.

Table 18 Handling of Exceptions Thrown by a PostConstruct or PreDestroy Method of a Message-Driven Bean.

Bean method condition	Bean method exception	Container action
Bean is message-driven bean	system exceptions	Log the exception or error. Discard instance.

13.3.6 Exceptions from an Enterprise Bean's Timeout Callback Method

This section specifies the container's handling of exceptions thrown from an enterprise bean's timeout callback method.

Table 19 and Table 20 specify how the container must handle the exceptions thrown by the timeout callback method of an enterprise bean. The timeout callback method does not throw application exceptions and cannot throw exceptions to the client.

Table 19 Handling of Exceptions Thrown by the Timeout Callback Method of an Enterprise Bean with Container-Managed Transaction Demarcation.

Method condition	Method exception	Container's action
Bean timeout callback method runs in the context of a transaction that the container started immediately before dispatching the method.	system exceptions	Log the exception or error[Note A]. Rollback the container-started transaction. Discard instance[Note B].

Notes:

[A] *Log the exception or error* means that the container logs the exception or error so that the System Administrator is alerted of the problem.

[B] *Discard instance* means that the container must not invoke any methods on the instance.

Table 20 Handling of Exceptions Thrown by the Timeout Callback Method of an Enterprise Bean with Bean-Managed Transaction Demarcation.

Method condition	Method exception	Container's action
The bean timeout callback method may make use of UserTransaction.	system exceptions	Log the exception or error[Note A]. Mark for rollback a transaction that has been started, but not yet completed, by the instance. Discard instance[Note B].

Notes:

[A] *Log the exception or error* means that the container logs the exception or error so that the System Administrator is alerted of the problem.

[B] *Discard instance* means that the container must not invoke any methods on the instance.

13.3.7 Exceptions from Other Container-invoked Callbacks

This subsection specifies the container's handling of exceptions thrown from the other container-invoked callbacks on the enterprise bean. This subsection applies to the following callback methods:

- Dependency injection methods.
- The `ejbActivate`, `ejbLoad`, `ejbPassivate`, `ejbStore`, `setEntityContext`, and `unsetEntityContext` methods of the `EntityBean` interface.
- The `PostActivate` and `PrePassivate` callback methods, and/or `ejbActivate`, `ejbPassivate`, and `setSessionContext` methods of the `SessionBean` interface.
- The `setMessageDrivenContext` method of the `MessageDrivenBean` interface.
- The `afterBegin, beforeCompletion` and `afterCompletion` methods of the `SessionSynchronization` interface.

The container must handle all exceptions or errors from these methods as follows:

- Log the exception or error to bring the problem to the attention of the System Administrator.
- If the instance is in a transaction, mark the transaction for rollback.
- Discard the instance (i.e., the container must not invoke any business methods or container callbacks on the instance).
- If the exception or error happened during the processing of a client invoked method, throw the `javax.ejb.EJBException`[63]. If the EJB 2.1 client view or web service client view is used, throw the `java.rmi.RemoteException` to the client if the client is a remote client or throw the `javax.ejb.EJBException` to the client if the client is a local client. If the instance executed in the client's transaction, the container should throw the `javax.ejb.EJBTransactionRolledbackException`[64]. If the EJB 2.1 client view or web service client view is used, the container should throw the `javax.transaction.TransactionRolledbackException` to a remote client or the `javax.ejb.TransactionRolledbackLocalException` to a local client, because it provides more information to the client. (The client knows that it is fruitless to continue the transaction.)

13.3.8 javax.ejb.NoSuchEntityException

The `NoSuchEntityException` is a subclass of `EJBException`. If it is thrown by a method of an entity bean class, the container must handle the exception using the rules for `EJBException` described in Sections 13.3.2, 13.3.4, and 13.3.7.

To give the client a better indication of the cause of the error, the container should throw the `java.rmi.NoSuchObjectException` (which is a subclass of `java.rmi.RemoteException`) to a remote client, or the `javax.ejb.NoSuchObjectLocalException` to a local client.

---

[63] If the business interface is a remote business interface that extends `java.rmi.Remote`, the `java.rmi.RemoteException` is thrown to the client instead.

[64] If the business interface is a remote business interface that extends `java.rmi.Remote`, the `javax.transaction.TransactionRolledbackException` is thrown to the client instead.

13.3.9 Non-existing Stateful Session or Entity Object

If a client makes a call to a stateful session or entity object that has been removed, the container should throw the `javax.ejb.NoSuchEJBException`[65]. If the EJB 2.1 client view is used, the container should throw the `java.rmi.NoSuchObjectException` (which is a subclass of `java.rmi.RemoteException`) to a remote client, or the `javax.ejb.NoSuchObjectLocalException` to a local client.

13.3.10 Exceptions from the Management of Container-Managed Transactions

The container is responsible for starting and committing the container-managed transactions, as described in Subsection 12.6.2. This subsection specifies how the container must deal with the exceptions that may be thrown by the transaction start and commit operations.

If the container fails to start or commit a container-managed transaction, the container must throw the `javax.ejb.EJBException`[66]. If the web service client view or EJB 2.1 client view is used, the container must throw the `java.rmi.RemoteException` to a remote or web service client and the `javax.ejb.EJBException` to a local client. In the case where the container fails to start or commit a container-managed transaction on behalf of a message-driven bean or a timeout callback method, the container must throw and log the `javax.ejb.EJBException`.

However, the container should not throw the `javax.ejb.EJBException` or `java.rmi.RemoteException` or if the container performs a transaction rollback because the instance has invoked the `setRollbackOnly` method on its `EJBContext` object. In this case, the container must rollback the transaction and pass the business method result or the application exception thrown by the business method to the client.

Note that some implementations of the container may retry a failed transaction transparently to the client and enterprise bean code. Such a container would throw the `javax.ejb.EJBException` *or* `java.rmi.RemoteException` *or after a number of unsuccessful tries.*

13.3.11 Release of Resources

When the container discards an instance because of a system exception, the container should release all the resources held by the instance that were acquired through the resource factories declared in the enterprise bean environment (See Subsection 15.7).

Note: While the container should release the connections to the resource managers that the instance acquired through the resource factories declared in the enterprise bean environment, the container cannot, in general, release "unmanaged" resources that the instance may have acquired through the JDK APIs. For example, if the instance has opened a TCP/IP connection, most container implementations will not be able to release the connection. The connection will be eventually released by the JVM garbage collector mechanism.

---

[65] If the business interface is a remote business interface that extends `java.rmi.Remote`, the `java.rmi.NoSuchObjectException` is thrown to the client instead.

[66] If the business interface is a remote business interface that extends `java.rmi.Remote`, the `java.rmi.RemoteException` is thrown to the client instead.

13.3.12 Support for Deprecated Use of `java.rmi.RemoteException`

The EJB 1.0 specification allowed the business methods, `ejbCreate`, `ejbPostCreate`, `ejbFind<METHOD>`, `ejbRemove`, and the container-invoked callbacks (i.e., the methods defined in the `EntityBean`, `SessionBean`, and `SessionSynchronization` interfaces) implemented in the enterprise bean class to use the `java.rmi.RemoteException` to report non-application exceptions to the container.

This use of the `java.rmi.RemoteException` was deprecated in EJB 1.1—enterprise beans written for the EJB 1.1 specification should use the `javax.ejb.EJBException` instead, and enterprise beans written for the EJB 2.0 or later specification must use the `javax.ejb.EJBException` instead.

The EJB 1.1 and EJB 2.0 or later specifications require that a container support the deprecated use of the `java.rmi.RemoteException`. The container should treat the `java.rmi.RemoteException` thrown by an enterprise bean method in the same way as it is specified for the `javax.ejb.EJBException`.

13.4 Client's View of Exceptions

This section describes the client's view of exceptions received from an enterprise bean invocation.

A client accesses an enterprise bean either through the enterprise bean's business interface (whether local or remote), through the enterprise bean's remote home and remote interfaces, through the enterprise bean's local home and local interfaces, or through the enterprise bean's web service client view depending on whether the client is written to the EJB 3.0 API or earlier API and whether the client is a remote client, a local client, or a web service client.

The methods of the business interface typically do not throw the `java.rmi.RemoteException`, regardless of whether the interface is a remote or local interface.

The remote home interface, the remote interface, and the web service endpoint interface are Java RMI interfaces, and therefore the `throws` clauses of all their methods (including those inherited from superinterfaces) include the mandatory `java.rmi.RemoteException`. The `throws` clauses may include an arbitrary number of application exceptions.

The local home and local interfaces are both Java local interfaces, and the `throws` clauses of all their methods (including those inherited from superinterfaces) must not include the `java.rmi.RemoteException`. The `throws` clauses may include an arbitrary number of application exceptions.

13.4.1 Application Exception

13.4.1.1 Local and Remote Clients

If a client program receives an application exception from an enterprise bean invocation, the client can continue calling the enterprise bean. An application exception does not result in the removal of the EJB object.

Although the container does not automatically mark for rollback a transaction because of a thrown application exception, the transaction might have been marked for rollback by the enterprise bean instance before it threw the application exception or the application exception may have been specified to require the container to rollback the transaction. There are two ways to learn if a particular application exception results in transaction rollback or not:

- Statically. Programmers can check the documentation of the enterprise bean's client view interface. The Bean Provider may have specified (although he or she is not required to) the application exceptions for which the enterprise bean marks the transaction for rollback before throwing the exception.

- Dynamically. Clients that are enterprise beans with container-managed transaction demarcation can use the `getRollbackOnly` method of the `javax.ejb.EJBContext` object to learn if the current transaction has been marked for rollback; other clients may use the `getStatus` method of the `javax.transaction.UserTransaction` interface to obtain the transaction status.

13.4.1.2 Web Service Clients

If a stateless session bean throws an application exception from one of its web service methods, it is the responsibility of the container to map the exception to the SOAP fault specified in the WSDL that describes the port type that the stateless session bean implements. For Java clients, the exceptions received by the client are described by the mapping rules in [25].

13.4.2 java.rmi.RemoteException and javax.ejb.EJBException

As described above, a client receives the `javax.ejb.EJBException` or the `java.rmi.RemoteException` as an indication of a failure to invoke an enterprise bean method or to properly complete its invocation. The exception can be thrown by the container or by the communication subsystem between the client and the container.

If the client receives the `javax.ejb.EJBException` or the `java.rmi.RemoteException` exception from a method invocation, the client, in general, does not know if the enterprise bean's method has been completed or not.

If the client executes in the context of a transaction, the client's transaction may, or may not, have been marked for rollback by the communication subsystem or target bean's container.

For example, the transaction would be marked for rollback if the underlying transaction service or the target bean's container doubted the integrity of the data because the business method may have been partially completed. Partial completion could happen, for example, when the target bean's method returned with a `RuntimeException` *exception, or if the remote server crashed in the middle of executing the business method.*

The transaction may not necessarily be marked for rollback. This might occur, for example, when the communication subsystem on the client-side has not been able to send the request to the server.

When a client executing in a transaction context receives an `EJBException` or a `RemoteException` from an enterprise bean invocation, the client may use either of the following strategies to deal with the exception:

- Discontinue the transaction. If the client is the transaction originator, it may simply rollback its transaction. If the client is not the transaction originator, it can mark the transaction for rollback or perform an action that will cause a rollback. For example, if the client is an enterprise bean, the enterprise bean may throw a `RuntimeException` which will cause the container to rollback the transaction.

- Continue the transaction. The client may perform additional operations on the same or other enterprise beans, and eventually attempt to commit the transaction. If the transaction was marked for rollback at the time the `EJBException` or `RemoteException` was thrown to the client, the commit will fail.

If the client chooses to continue the transaction, the client can first inquire about the transaction status to avoid fruitless computation on a transaction that has been marked for rollback. A client that is an enterprise bean with container-managed transaction demarcation can use the `EJBContext.getRollbackOnly` method to test if the transaction has been marked for rollback; a client that is an enterprise bean with bean-managed transaction demarcation, and other client types, can use the `UserTransaction.getStatus` method to obtain the status of the transaction.

Some implementations of EJB servers and containers may provide more detailed exception reporting by throwing an appropriate subclass of the `javax.ejb.EJBException` or `java.rmi.RemoteException` to the client. The following subsections describe the several subclasses of the `javax.ejb.EJBException` and `java.rmi.RemoteException` that may be thrown by the container to give the client more information.

13.4.2.1 javax.ejb.EJBTransactionRolledbackException, javax.ejb.TransactionRolledbackLocalException, and javax.transaction.TransactionRolledbackException

The `javax.ejb.EJBTransactionRolledbackException` and `javax.ejb.TransactionRolledbackLocalException` are subclasses of the `javax.ejb.EJBException`. The `javax.transaction.TransactionRolledbackException` is a subclass of the `java.rmi.RemoteException`. It is defined in the JTA standard extension.

If a client receives one of these exceptions, the client knows for certain that the transaction has been marked for rollback. It would be fruitless for the client to continue the transaction because the transaction can never commit.

13.4.2.2 javax.ejb.EJBTransactionRequiredException, javax.ejb.TransactionRequiredLocalException, and javax.transaction.TransactionRequiredException

The `javax.ejb.EJBTransactionRequiredException` and `javax.ejb.TransactionRequiredLocalException` are subclasses of the `javax.ejb.EJBException`. The `javax.transaction.TransactionRequiredException` is a subclass of the `java.rmi.RemoteException`. It is defined in the JTA standard extension.

The `javax.ejb.EJBTransactionRequiredException`, `javax.ejb.TransactionRequiredLocalException`, or `javax.transaction.TransactionRequiredException` informs the client that the target enterprise bean must be invoked in a client's transaction, and that the client invoked the enterprise bean without a transaction context.

This error usually indicates that the application was not properly formed.

13.4.2.3 javax.ejb.NoSuchEJBException, javax.ejb.NoSuchObjectLocalException, and java.rmi.NoSuchObjectException

The `javax.ejb.NoSuchEJBException` is a subclass of the `javax.ejb.EJBException`. It is thrown to the client of a session bean's business interface if a local business method cannot complete because the EJB object no longer exists.

The `javax.ejb.NoSuchObjectLocalException` and the `java.rmi.NoSuchObjectException` apply to the business methods of the EJB 2.1 local and remote client views respectively.

- The `javax.ejb.NoSuchObjectLocalException` is a subclass of the `javax.ejb.EJBException`. It is thrown to the client if a local business method cannot complete because the EJB object no longer exists.
- The `java.rmi.NoSuchObjectException` is a subclass of the `java.rmi.RemoteException`. It is thrown to the client if a remote business method cannot complete because the EJB object no longer exists.

13.5 System Administrator's Responsibilities

The System Administrator is responsible for monitoring the log of the non-application exceptions and errors logged by the container, and for taking actions to correct the problems that caused these exceptions and errors.

Chapter 14 Support for Distributed Interoperability

This chapter describes the interoperability support for accessing an enterprise bean through the EJB 2.1 remote client view from clients distributed over a network, and the distributed interoperability requirements for invocations on enterprise beans from remote clients that are Java Platform, Enterprise Edition (Java EE) components.

14.1 Support for Distribution

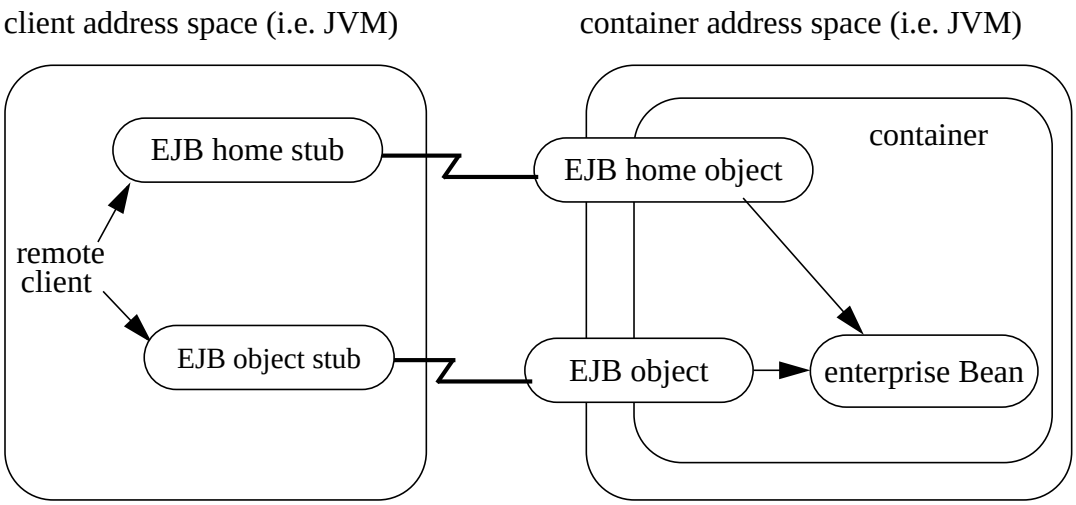
The remote home and remote interfaces of an enterprise bean's remote client view are defined as Java™ RMI [6] interfaces. This allows the container to implement the remote home and remote interfaces as *distributed objects*. A client using the remote home and remote interfaces can reside on a different machine than the enterprise bean (location transparency), and the object references of the remote home and remote interfaces can be passed over the network to other applications.

The EJB specification further constrains the Java RMI types that can be used by enterprise beans to be legal RMI-IIOP types [10]. This makes it possible for EJB container implementors to use RMI-IIOP as the object distribution protocol.

14.1.1 Client-Side Objects in a Distributed Environment

When the RMI-IIOP protocol or similar distribution protocols are used, the remote client communicates with the enterprise bean using *stubs* for the server-side objects. The stubs implement the remote home and remote interfaces.

Figure 30 Location of EJB Client Stubs.



The communication stubs used on the client side are artifacts generated at the enterprise bean's deployment time by the Container Provider's tools. The stubs used on the client are specific to the wire protocol used for the remote invocation.

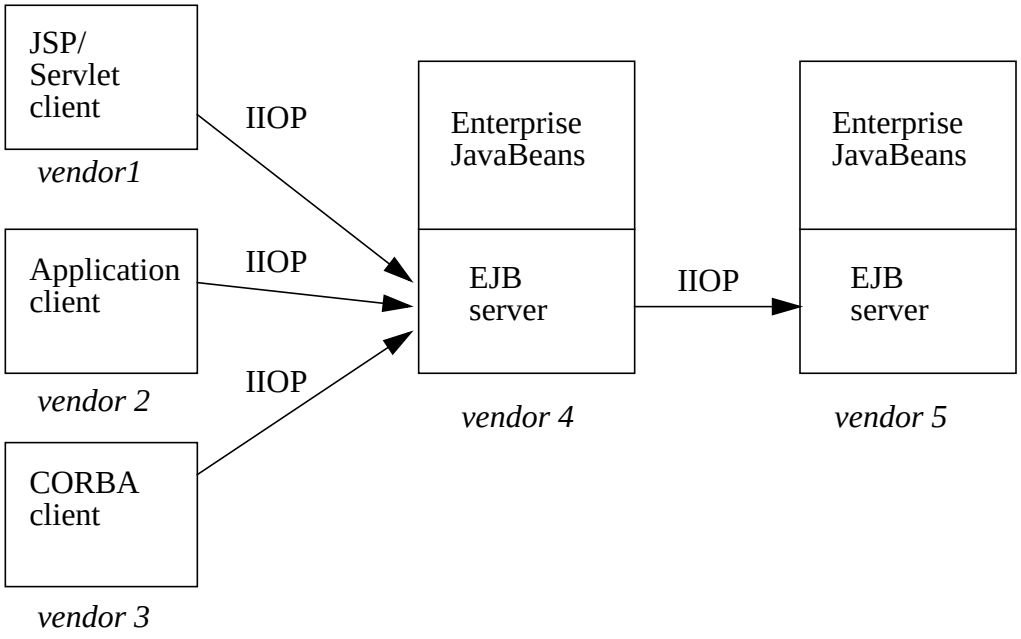
14.2 Interoperability Overview

Session beans and entity beans that are deployed in one vendor's server product may need to be accessed from Java EE client components that are deployed in another vendor's product through the remote client view. EJB defines a standard interoperability protocol based on CORBA/IIOP to address this need.

The interoperability protocols described here must be supported by compatible EJB products. Additional vendor-specific protocols may also be supported.

Figure 31 shows a heterogeneous environment that includes systems from several vendors to illustrate the interoperability enabled by EJB.

Figure 31 Heterogeneous EJB Environment



The following sections in this chapter

- describe the goals for EJB invocation interoperability
- provide illustrative scenarios
- describe the interoperability requirements for remote invocations, transactions, naming, and security.

14.2.1 Interoperability Goals

The goals of the interoperability requirements specified in this chapter are as follows:

- To allow clients in one application deployed in Java EE containers from one server provider to access services from session and entity beans in another application that is deployed in an EJB container from a different server provider. For example, web components (JavaServer Pages and servlets) that are deployed on a Java EE compliant web server provided by one server provider must be able to invoke the business methods of enterprise beans that are deployed on a Java EE compliant EJB server from another server provider.

- To achieve interoperability without any new requirements on the Java EE application developer.
- To ensure out-of-the-box interoperability between compliant Java EE products. It must be possible for an enterprise customer to install multiple Java EE server products from different server providers (on potentially different operating systems), deploy applications in the Java EE servers, and have the multiple applications interoperate.
- To leverage the interoperability work done by standards bodies (including the IETF, W3C, and OMG) where possible, so that customers can work with industry standards and use standard protocols to access enterprise beans.

This specification does not address interoperability issues between enterprise beans and non-Java-EE components. The Java EE platform specification [12] and the JAX-RPC and JAX-WS specifications [25], [32] describe requirements for interoperability with Internet clients (using HTTP and XML) and interoperability with enterprise information systems (using the Connector architecture [15]).

Since the interoperability protocol described here is based on CORBA/IIOP, CORBA clients written in Java, C++, or other languages can also invoke methods on enterprise beans.

This chapter subsumes the previous EJB1.1-to-CORBA mapping document [16].

14.3 Interoperability Scenarios

This section presents a number of interoperability scenarios that motivate the interoperability mechanisms described in later sections of this chapter. These scenarios are illustrative rather than prescriptive. This section does not specify requirements for a Java EE product to support these scenarios in exactly the manner described here.

Java EE applications are multi-tier, web-enabled applications. Each application consists of one or more components, which are deployed in containers. The four types of containers are:

- EJB containers, which host enterprise beans.
- Web containers, which host JavaServer Pages (JSPs) and servlet components as well as static documents, including HTML pages.
- Application client containers, which host standalone applications.
- Applet containers, which host applets which may be downloaded from a web site. At this time, there is no requirement for an applet to be able to directly invoke the remote methods of enterprise beans.

The scenarios below describe interactions between components hosted in these various container types.

14.3.1 Interactions Between Web Containers and EJB Containers for E-Commerce

Applications

This scenario occurs for business-to-business and business-to-consumer interactions over the Internet.

Scenario 1: *A customer wants to buy a book from an Internet bookstore. The bookstore's web site consists of a Java EE application containing JSPs that form the presentation layer, and another Java EE application containing enterprise beans that have the business logic and database access code. The JSPs and enterprise beans are deployed in containers from different vendors.*

At deployment time: *The enterprise beans are deployed, and their EJBHome objects are published in the EJB server's name service. The Deployer links the EJB reference in the JSP's deployment descriptor to the URL of the enterprise bean's EJBHome object, which can be looked up from the name service. The transaction attribute specified in the enterprise bean's deployment descriptor is* `RequiresNew` *for all business methods. Because the "checkout" JSP requires secure access to set up payments for purchases, the bookstore's administrator configures the "checkout" JSP to require access over HTTPS with only server authentication. Customer authentication is done using form-based login. The "book search" JSP is accessed over normal HTTP. Both JSPs talk with enterprise beans that access the book database. The web and EJB containers use the same customer realm and have a trust relationship with each other. The network between the web and EJB servers is not guaranteed to be secure from attacks.*

At runtime: *The customer accesses the book search JSP using a browser. The JSP looks up the enterprise bean's EJBHome object in a name service, and calls* `findBooks(...)` *with the search criteria as parameters. The web container establishes a secure session with the EJB container with mutual authentication between the containers, and invokes the enterprise bean. The customer then decides to buy a book, and accesses the "checkout" JSP. The customer enters the necessary information in the login form, which is used by the web server to authenticate the customer. The JSP invokes the enterprise bean to update the book and customer databases. The customer's principal is propagated to the EJB container and used for authorization checks. The enterprise bean completes the updates and commits the transaction. The JSP sends back a confirmation page to the customer.*

14.3.2 Interactions Between Application Client Containers and EJB Containers Within an Enterprise's Intranet

Scenario 2.1: *An enterprise has an expense accounting application used by employees from their desktops. The server-side consists of a Java EE application containing enterprise beans that are deployed on one vendor's Java EE product, which is hosted in a datacenter. The client side consists of another Java EE application containing an application client deployed using another vendor's Java EE infrastructure. The network between the application client and the EJB container is insecure and needs to be protected against spoofing and other attacks.*

At deployment time: *The enterprise beans are deployed and their EJBHome objects are published in the enterprise's name service. The application clients are configured with the names of the EJBHome objects. The Deployer maps employees to roles that are allowed access to the enterprise beans. The System Administrator configures the security settings of the application client and EJB container to use client and server authentication and message protection. The System Administrator also does the necessary client-side configuration to allow client authentication.*

At runtime: *The employee logs on using username and password. The application client container may interact with the enterprise's authentication service infrastructure to set up the employee's credentials. The client application does a remote invocation to the name server to look up the enterprise bean's EJBHome object, and creates the enterprise beans. The application client container uses a secure transport protocol to interact with the name server and EJB server, which does mutual authentication and also guarantees the confidentiality and integrity of messages. The employee then enters the expense information and submits it. This causes remote business methods of the enterprise beans to be invoked. The EJB container performs authorization checks and, if they succeed, executes the business methods.*

Scenario 2.2: *This is the same as Scenario 2.1, except that there is no client-side authentication infrastructure set up by the System Administrator which can authenticate at the transport protocol layer. At runtime the client container needs to send the user's password to the server during the method invocation to authenticate the employee.*

14.3.3 Interactions Between Two EJB Containers in an Enterprise's Intranet

Scenario 3: *An enterprise has an expense accounting application which needs to communicate with a payroll application. The applications use enterprise beans and are deployed on Java EE servers from different vendors. The Java EE servers and naming/authentication services may be in the enterprise's datacenter with a physically secure private network between them, or they may need to communicate across the intranet, which may be less secure. The applications need to update accounts and payroll databases. The employee (client) accesses the expense accounting application as described in Scenario 2.*

At deployment time: *The Deployer configures both applications with the appropriate database resources. The accounts application is configured with the name of the EJBHome object of the payroll application. The payroll bean's deployment descriptor specifies the* `RequiresNew` *transaction attribute for all methods. The applications use the same principal-to-role mappings (e.g. the roles may be* `Employee`, `PayrollDept`, `AccountsDept`*). The Deployer of these two applications has administratively set up a trust relationship between the two EJB containers, so that the containers do not need to authenticate principals propagated on calls to enterprise beans from the other container. The System Administrator also sets up the message protection parameters of the two containers if the network is not physically secure.*

At runtime: *An employee makes a request to the accounts application which requires it to access the payroll application. The accounts application does a lookup of the payroll application's EJBHome object in the naming/directory service and creates enterprise beans. It updates the accounts database and invokes a remote method of the payroll bean. The accounts bean's container propagates the employee's principal on the method call. The payroll bean's container maps the propagated employee principal to a role, does authorization checks, and sets up the payroll bean's transaction context. The container starts a new transaction, then the payroll bean updates the payroll database, and the container commits the transaction. The accounts bean receives a status reply from the payroll bean. If an error occurs in the payroll bean, the accounts bean executes code to recover from the error and restore the databases to a consistent state.*

14.3.4 Intranet Application Interactions Between Web Containers and EJB Containers

Scenario 4: *This is the same as scenario 2.1, except that instead of using a "fat-client" desktop application to access the enterprise's expense accounting application, employees use a web browser and connect to a web server in the intranet that hosts JSPs. The JSPs gather input from the user (e.g., through an HTML form), invoke enterprise beans that contain the actual business logic, and format the results returned by the enterprise beans (using HTML).*

At deployment time: *The enterprise Deployer configures its expense accounting JSPs to require access over HTTPS with mutual authentication. The web and EJB containers use the same customer realm and have a trust relationship with each other.*

At run-time: *The employee logs in to the client desktop, starts the browser, and accesses the expense accounting JSP. The browser establishes an HTTPS session with the web server. Client authentication is performed (for example) using the employee's credentials which have been established by the operating system at login time (the browser interacts with the operating system to obtain the employee's credentials). The JSP looks up the enterprise bean's EJBHome object in a name service. The web container establishes a secure session with the EJB container with mutual authentication and integrity/confidentiality protection between the containers, and invokes methods on the enterprise beans.*

14.4 Overview of Interoperability Requirements

The interoperability requirements used to support the above scenarios are:

1. Remote method invocation on an enterprise bean's EJBObject and EJBHome object references (scenarios 1,2,3,4), described in section 14.5.

2. Name service lookup of the enterprise bean's EJBHome object (scenarios 1,2,3,4), described in section 14.7.

3. Integrity and confidentiality protection of messages (scenarios 1,2,3,4), described in section 14.8.

4. Authentication between an application client and EJB container (described in section 14.8):

- **4.1** Mutual authentication at the transport protocol layer when there is client-side authentication infrastructure such as certificates (scenario 2.1).
- **4.2** Transfer of the user's authentication data from application client to EJB container to allow the EJB container to authenticate the client when there is no client-side authentication infrastructure (scenario 2.2).

5. Mutual authentication between two EJB containers or between a web and EJB container to establish trust before principals are propagated (scenarios 1,3,4), described in section 14.8.

6. Propagation of the Internet or intranet user's principal name for invocations on enterprise beans from web or EJB containers when the client and server containers have a trust relationship (scenarios 1,3,4), described in section 14.8.

EJB, web, and application client containers must support the above requirements separately as well as in combinations.

14.5 Remote Invocation Interoperability

This section describes the interoperability mechanisms that enable remote invocations on EJBObject and EJBHome object references when client containers and EJB containers are provided by different vendors. This is needed to satisfy interoperability requirement (1) in section 14.4.

All EJB, web, and application client containers must support the IIOP 1.2 protocol for remote invocations on EJBObject and EJBHome references. EJB containers must be capable of servicing IIOP 1.2 based invocations on EJBObject and EJBHome objects. IIOP 1.2 is part of the CORBA 2.3.1 specification [17] from the OMG[67]. Containers may additionally support vendor-specific protocols.

CORBA Interoperable Object References (IORs) for EJBObject and EJBHome object references must include the GIOP version number 1.2. The IIOP infrastructure in all Java EE containers must be able to accept fragmented GIOP messages, although sending fragmented messages is optional. Bidirectional GIOP messages may optionally be supported by Java EE clients and servers: if a Java EE server receives an IIOP message from a client which contains the `BiDirIIOPServiceContext` structure, it may or may not use the same connection for sending requests back to the client.

Since Java applications use Unicode characters by default, Java EE containers are required to support the Unicode UTF16 code set for transmission of character and string data (in the IDL `wchar` and `wstring` datatypes). Java EE containers may optionally support additional code sets. EJBObject and EJBHome IORs must have the `TAG_CODE_SETS` tagged component which declares the codesets supported by the EJB container. IIOP messages which include `wchar` and `wstring` datatypes must have the code sets service context field. The CORBA 2.3.1 requirements for code set support must be followed by Java EE containers.

EJB containers are required to translate Java types to their on-the-wire representation in IIOP messages using the Java Language to IDL mapping specification [10] with the wire formats for IDL types as described in the GIOP specification in CORBA 2.3. The following subsections describe the mapping details for Java types.

[67] CORBA APIs and earlier versions of the IIOP protocol are already included in the J2SE 1.2, J2SE 1.3 and J2EE 1.2 platforms through JavaIDL and RMI-IIOP.

14.5.1 Mapping Java Remote Interfaces to IDL

The Java Language to IDL Mapping specification [10] describes precisely how the remote home and remote interfaces of a session bean or entity bean are mapped to IDL. This mapping to IDL is typically implicit when Java RMI over IIOP is used to invoke enterprise beans. Java EE clients use only the Java RMI APIs to invoke enterprise beans. The client container may use the CORBA portable Stub APIs for the client-side stubs. EJB containers may create CORBA Tie objects for each EJBObject or EJBHome object.

14.5.2 Mapping Value Objects to IDL

The Java interfaces that are passed by value during remote invocations on enterprise beans are `javax.ejb.Handle`, `javax.ejb.HomeHandle`, and `javax.ejb.EJBMetaData`. The `Enumeration` or `Collection` objects returned by entity bean finder methods are value types. There may also be application-specific value types that are passed as parameters or return values on enterprise bean invocations. In addition, several Java exception classes that are thrown by remote methods also result in concrete IDL value types. All these value types are mapped to IDL abstract value types or abstract interfaces using the rules in the Java Language to IDL Mapping.

14.5.3 Mapping of System Exceptions

Java system exceptions, including the `java.rmi.RemoteException` and its subclasses, may be thrown by the EJB container. If the client's invocation was made over IIOP, the EJB server is required to map these exceptions to CORBA system exceptions and send them in the IIOP reply message to the client, as specified in the following table

System exception thrown by EJB container	CORBA system exception received by client ORB
javax.transaction. TransactionRolledbackException	TRANSACTION_ROLLEDBACK
javax.transaction. TransactionRequiredException	TRANSACTION_REQUIRED
javax.transaction. InvalidTransactionException	INVALID_TRANSACTION
java.rmi.NoSuchObjectException	OBJECT_NOT_EXIST
java.rmi.AccessException	NO_PERMISSION
java.rmi.MarshalException	MARSHAL
java.rmi.RemoteException	UNKNOWN

For EJB clients, the ORB's unmarshaling machinery maps CORBA system exceptions received in the IIOP reply message to the appropriate Java exception as specified in the Java Language to IDL mapping. This results in the original Java exception being received by the client Java EE component.

14.5.4 Obtaining Stub and Client View Classes

When a Java EE component (application client, JSP, servlet or enterprise bean) receives a reference to an EJBObject or EJBHome object through JNDI lookup or as a parameter or return value of an invocation on an enterprise bean, an instance of an RMI-IIOP stub class (proxy) for the enterprise bean's remote home or remote RMI interface needs to be created. When a component receives a value object as a parameter or return value of an enterprise bean invocation, an instance of the value class needs to be created. The stub class, value class, and other client view classes must be available to the referencing container (the container hosting the component that receives the reference or value type).

The client view classes, including application value classes, must be packaged with the referencing component's application, as described in Section 19.3.

Stubs for invoking on EJBHome and EJBObject references must be provided by the referencing container, for example, by generating stub classes at deployment time for the EJBHome and EJBObject interfaces of the referenced beans that are packaged with the referencing component's application. Stub classes may or may not follow the standard RMI-IIOP portable stub architecture.

Containers may optionally support run-time downloading of stub and value classes needed by the referencing container. The CORBA 2.3.1 specification and the Java Language to IDL Mapping specify how stub and value type implementations are to be downloaded: using codebase URLs that are either embedded in the EJBObject or EJBHome's IOR, or sent in the IIOP message service context, or marshalled with the value type. The URLs for downloading may optionally include an HTTPS URL for secure downloading.

14.5.5 System Value Classes

System value classes are serializable value classes implementing the `javax.ejb.Handle`, `javax.ejb.HomeHandle`, `javax.ejb.EJBMetaData`, `java.util.Enumeration`, `java.util.Collection`, and `java.util.Iterator` interfaces. These value classes are provided by the EJB container vendor. They must be provided in the form of a JAR file by the container hosting the referenced bean. For interoperability scenarios, if a referencing component would use such system value classes at runtime, the Deployer must ensure that these system value classes provided by the container hosting the referenced bean are available to the referencing component. This may be done, for example, by including these system value classes in the classpath of the referencing container, or by deploying the system value classes with the referencing component's application by providing them to the deployment tool.

Implementations of these system value classes must be portable (they must use only J2SE and Java EE APIs) so that they can be instantiated in another vendor's container. If the system value class implementation needs to load application-specific classes (such as remote home or remote interfaces) at runtime, it must use the thread context class loader. The referencing container must make application-specific classes available to the system value class instance at runtime through the thread context class loader.

14.5.5.1 HandleDelegate SPI

The `javax.ejb.spi.HandleDelegate` service provider interface defines methods that enable portable implementations of `Handle` and `HomeHandle` that are instantiated in a different vendor's container to serialize and deserialize EJBObject and EJBHome references. The `HandleDelegate` interface is not used by enterprise beans or Java EE application components directly.

EJB, web and application client containers must provide implementations of the `HandleDelegate` interface. The HandleDelegate object must be accessible in the client Java EE component's JNDI namespace at the reserved name "`java:comp/HandleDelegate`". The HandleDelegate object is not exported outside the container that provides it.

Portable implementations of `Handle` and `HomeHandle` must look up the HandleDelegate object of the container in which they are instantiated using JNDI at the name "`java:comp/HandleDelegate`" and use the HandleDelegate object to serialize and deserialize EJBObject and EJBHome references as follows:

- Handle and HomeHandle implementation classes must define `writeObject` and `readObject` methods to control their serialization and deserialization. These methods must not wrap or substitute the stream objects that are passed to the HandleDelegate methods.

- The `writeObject` method of Handle implementations must call `HandleDelegate.writeEJBObject` with the Handle's EJBObject reference and the serialization output stream object as parameters. The HandleDelegate implementation (which is provided by the client container in which the Handle was instantiated, potentially from a different vendor) then writes the EJBObject to the output stream. If the output stream corresponds to an IIOP message, the HandleDelegate must use the standard IIOP abstract interface format for writing the EJBObject reference.

- The `readObject` method of Handle implementations must call `HandleDelegate.readEJBObject` with the serialization input stream object as parameter, and with the stream positioned at the location where the EJBObject can be read. The HandleDelegate implementation then reads the EJBObject from the input stream and returns it to the Handle. If the input stream corresponds to an IIOP message, the HandleDelegate must use the standard abstract interface format for reading the EJBObject reference. The HandleDelegate must ensure that the EJBObject reference is capable of performing invocations immediately after deserialization. The Handle maintains a reference to the EJBObject as a transient instance variable and returns it when the Java EE component calls `Handle.getEJBObject`.

- The `writeObject` and `readObject` methods of HomeHandle implementation classes must be implemented similarly, by using `HandleDelegate.writeEJBHome` and `HandleDelegate.readEJBHome` respectively.

14.6 Transaction Interoperability

Transaction interoperability between containers provided by different vendors is an optional feature in this version of the EJB specification. Vendors may choose to not implement transaction interoperability. However, vendors who choose to implement transaction interoperability must follow the requirements in sections 14.6.1 and 14.6.2, and vendors who choose not to implement transaction interoperability must follow the requirements in section 14.6.2.

14.6.1 Transaction Interoperability Requirements

A distributed transaction started by a web or EJB container must be able to propagate in a remote invocation to an enterprise bean in an EJB container provided by a different vendor, and the containers must participate in the distributed two-phase commit protocol.

14.6.1.1 Transaction Context Wire Format

Transaction context propagation from client to EJB container uses the implicit propagation mechanism described in the CORBA Object Transaction Service (OTS) v1.2 specification [11].

The transaction context format in IIOP messages is specified in the `CosTransactions::PropagationContext` structure described in the OTS specification. EJB containers that support transaction interoperability are required to be capable of producing and consuming transaction contexts in IIOP messages in the format described in the OTS specification. Web containers that support transaction interoperability are required to include client-side libraries which can produce the OTS transaction context for sending over IIOP.

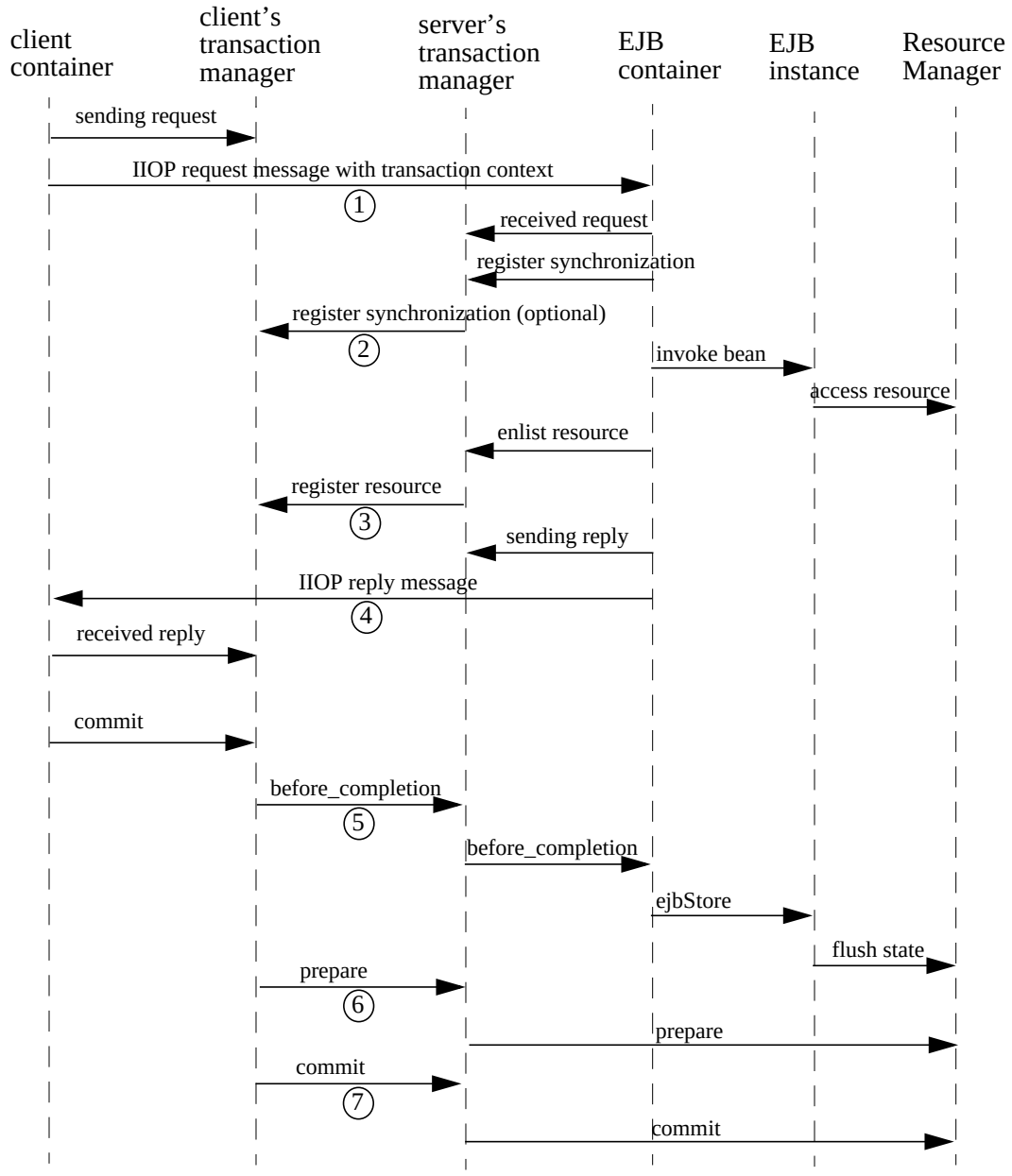
Note that it is not necessary for containers to include the Java mappings of the OTS APIs. A container may implement the requirements in the OTS specification in any manner, for example using a non-Java OTS implementation, or an on-the-wire bridge between an existing transaction manager and the OTS protocol, or an OTS wrapper around an existing transaction manager.

The `CosTransactions::PropagationContext` structure must be included in IIOP messages sent by web or EJB containers when required by the rules described in the OTS 1.2 specification. The target EJB container must process IIOP invocations based on the transaction policies of EJBObject or EJBHome references using the rules described in the OTS 1.2 specification [11].

14.6.1.2 Two-Phase Commit Protocol

The object interaction diagram in Figure 32 illustrates the interactions between the client and server transaction managers for transaction context propagation, resource and synchronization object registration, and two-phase commit. This diagram is an example of the interactions between the various entities; it is not intended to be prescriptive.

Figure 32 Transaction Context Propagation



Containers that perform transactional work within the scope of a transaction must register an OTS Resource object with the transaction coordinator whose object reference is included in the propagated transaction context (step 3), and may also register an OTS Synchronization object (step 2). If the server container does not register an OTS Synchronization object, it must still ensure that the `beforeCompletion` method of session beans and `ejbStore` method of entity beans are called with the proper transaction context. Containers must participate in the two-phase commit and recovery procedures performed by the transaction coordinator / terminator (steps 6,7), as described by the OTS specification.

Compliant Java EE containers must not use nested transactions in interoperability scenarios.

14.6.1.3 Transactional Policies of Enterprise Bean References

The OTS1.2 specification describes the `CosTransactions::OTSPolicy` and `CosTransactions::InvocationPolicy` structures that are encoded in IORs as tagged components. EJBObject and EJBHome references must contain these tagged components[68] with policy values as described below.

The transaction attributes of enterprise beans can be specified per method, while in OTS the entire CORBA object has the same OTS transaction policy. The rules below ensure that the transaction context will be propagated if any method of an enterprise bean needs to execute in the client's transaction context. However, in some cases there may be extra performance overhead of propagating the client's transaction context even if it will not be used by the enterprise bean method.

EJBObject and EJBHome references may have the InvocationPolicy value as either `CosTransactions::SHARED` or `CosTransactions::EITHER`[69].

All EJBObject and EJBHome references must have the OTSPolicy value as `CosTransactions::ADAPTS`. This is necessary to allow clients to invoke methods of the `javax.ejb.EJBObject` and `javax.ejb.EJBHome` with or without a transaction.

The `CosTransactions::Synchronization` object registered by the EJB container with the transaction coordinator should have the OTSPolicy value `CosTransactions::ADAPTS` and InvocationPolicy value `CosTransactions::SHARED` to allow enterprise beans to do transactional work during the `beforeCompletion` notification from the transaction coordinator.

14.6.1.4 Exception Handling Behavior

The exception handling behavior described in the OTS1.2 specification must be followed. In particular, if an application exception (an exception which is not a CORBA system exception and does not extend `java.rmi.RemoteException`) is returned by the server, the request is defined as being successful; hence the client-side OTS library must not roll back the transaction. This allows application exceptions to be propagated back to the client without rolling back the transaction, as required by the exception handling rules in Chapter 13.

14.6.2 Interoperating with Containers that do not Implement Transaction

[68] One way to include the tagged components in IORs is to create the object references using a Portable Object Adapter (POA) which is initialized with the appropriate transaction policies. Note that POA APIs are not required to be supported by server containers.

[69] If the InvocationPolicy is not present in the IOR, it is interpreted by the client as if the policy value was `CosTransactions::EITHER`.

Interoperability

The requirements in this subsection are designed to ensure that when a Java EE container does not support transaction interoperability, the failure modes are well defined so that the integrity of an application's data is not compromised: at worst the transaction is rolled back. When a Java EE client component expects the client's transaction to propagate to the enterprise bean but the client or EJB container cannot satisfy this expectation, a `java.rmi.RemoteException` or subclass is thrown, which ensures that the client's transaction will roll back.

In addition, the requirements below allow a container that does not support transaction propagation to interoperate with a container that does support transaction propagation in the cases where the enterprise bean method's transaction attribute indicates that the method would not be executed in the client's transaction.

14.6.2.1 Client Container Requirements

If the client in another container invokes an enterprise bean's method when there is no active global transaction associated with the client's thread, the client container does not include a transaction context in the IIOP request message to the EJB server, i.e., there is no `CosTransactions::PropagationContext` structure in the IIOP request header.

The client application component expects a global transaction to be propagated to the server only if the client's thread has an active global transaction. In this scenario, if the client container does not support transaction interoperability, it has two options:

1. If the client container does not support transaction propagation or uses a non-OTS protocol, it must include the OTS `CosTransactions::PropagationContext` structure in the IIOP request to the server (step 1 in the object interaction diagram above), with the `CosTransactions::Coordinator` and `CosTransactions::Terminator` object references as null. The remaining fields in this "null transaction context," such as the transaction identifier, are not interpreted and may have any value. The "null transaction context" indicates that there is a global client transaction active but the client container is not capable of propagating it to the server. The presence of this "null transaction context" allows the EJB container to determine whether the Java EE client component expects the client's global transaction to propagate to the server.

2. Client containers that use the OTS transaction context format but still do not support transaction interoperability with other vendor's containers must reject the `Coordinator::register_resource` call (step 3 in the object interaction diagram above) by throwing a CORBA system exception if the server's Resource object reference indicates that it belongs to another vendor's container.

14.6.2.2 EJB container requirements

All EJB containers (including those that do not support transaction propagation) must include the `CosTransactions::OTSPolicy` and optionally the `CosTransactions::InvocationPolicy` tagged component in the IOR for EJBObject and EJBHome references as described in section 14.6.1.3.

14.6.2.2.1 Requirements for EJB Containers Supporting Transaction Interoperability

When an EJB container that supports transaction propagation receives the IIOP request message, it must behave as follows:

- If there is no OTS transaction context in the IIOP message, the container must follow the behavior described in Section 12.6.
- If there is a valid, complete OTS transaction context in the IIOP message, the container must follow the behavior described in Section 12.6.
- If there is a null transaction context (as defined in section 14.6.2.1 above) in the IIOP message, the container's required behavior is described in the table below. The entry "throw RemoteException" indicates that the EJB container must throw the `java.rmi.RemoteException` to the client after the "received request" interaction with the server's transaction manager (after step 1 in the object interaction diagram above).

EJB method's Transaction Attribute	**EJB container behavior on receiving null OTS transaction context**
Mandatory	throw RemoteException
Required	throw RemoteException
RequiresNew	follow Section 12.6
Supports	throw RemoteException
NotSupported	follow Section 12.6
Never	follow Section 12.6
Bean Managed	follow Section 12.6

14.6.2.2.2 Requirements for EJB Containers not Supporting Transaction Interoperability

When an EJB container that does not support transaction interoperability receives the IIOP request message, it must behave as follows:

- If there is no OTS transaction context in the IIOP message, the container must follow the behavior described in Section 12.6.
- If there is a valid, complete OTS transaction context in the IIOP message, the container's required behavior is described in the table below.

- If there is a null transaction context (as defined in section 14.6.2.1) in the IIOP message, the container's required behavior is described in the table below. Note that the container may not know whether the received transaction context in the IIOP message is valid or null.

EJB method's Transaction Attribute	EJB container behavior on receiving null or valid OTS transaction context
Mandatory	throw RemoteException
Required	throw RemoteException
RequiresNew	follow Section 12.6
Supports	throw RemoteException
NotSupported	follow Section 12.6
Never	follow Section 12.6
Bean Managed	follow Section 12.6

EJB containers that accept the OTS transaction context format but still do not support interoperability with other vendors' client containers must follow the column in the table above for "null or valid OTS transaction context" if the transaction identity or the Coordinator object reference in the propagated client transaction context indicate that the client belongs to a different vendor's container.

14.7 Naming Interoperability

This section describes the requirements for supporting interoperable access to naming services for looking up EJBHome object references (interoperability requirement two in section 14.4).

EJB containers are required to be able to publish EJBHome object references in a CORBA CosNaming service [18]. The CosNaming service must implement the IDL interfaces in the CosNaming module defined in [18] and allow clients to invoke the `resolve` and `list` operations over IIOP.

The CosNaming service must follow the requirements in the CORBA Interoperable Name Service specification [19] for providing the host, port, and object key for its root `NamingContext` object. The CosNaming service must be able to service IIOP invocations on the root `NamingContext` at the advertised host, port, and object key.

Client containers (i.e., EJB, web, or application client containers) are required to include a JNDI CosNaming service provider that uses the mechanisms defined in the Interoperable Name Service specification to contact the server's CosNaming service, and to resolve the EJBHome object using standard CosNaming APIs. The JNDI CosNaming service provider may or may not use the JNDI SPI architecture. The JNDI CosNaming service provider must access the root NamingContext of the server's CosNaming service by creating an object reference from the URL `corbaloc:iiop:1.2@<host>:<port>/<objectkey>` (where `<host>`, `<port>`, and `<objectkey>` are the values corresponding to the root NamingContext advertised by the server's CosNaming service), or by using an equivalent mechanism.

At deployment time, the Deployer of the client container should obtain the host, port and object key of the server's CosNaming service and the CosNaming name of the server EJBHome object (e.g. by browsing the server's namespace) for each such `EJB` annotation or `ejb-ref` element in the client component's deployment descriptor. The `ejb-ref-name` (which is used by the client code in the JNDI lookup call) should then be linked to the EJBHome object's CosNaming name. At run-time, the client component's JNDI lookup call uses the CosNaming service provider, which contacts the server's CosNaming service, resolves the CosNaming name, and returns the EJBHome object reference to the client component.

Since the EJBHome object's name is scoped within the namespace of the CosNaming service that is accessible at the provided host and port, it is not necessary to federate the namespaces of the client and server containers.

The advantage of using CosNaming is better integration with the IIOP infrastructure that is already required for interoperability, as well as interoperability with non-Java-EE CORBA clients and servers. Since CosNaming stores only CORBA objects it is likely that vendors will use other enterprise directory services for storing other resources.

Security of CosNaming service access is achieved using the security interoperability protocol described in Section 14.8. The CosNaming service must support this protocol. Clients which construct the root NamingContext object reference from a URL should send an IIOP `LocateRequest` message to the CosNaming service to obtain the complete IOR (with SSL information) of the root NamingContext, and then initiate an SSL session with the CosNaming service, as determined by the client policy.

14.8 Security Interoperability

This section describes the interoperable mechanisms that support secure invocations on enterprise beans in intranets. These mechanisms are based on the CORBA/IIOP protocol.

EJB containers are required to follow the protocol rules prescribed by the CSIv2 specification Conformance Level 0.

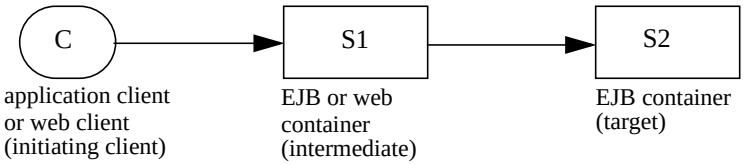
14.8.1 Introduction

The goal of the secure invocation mechanisms is to support the interoperability requirements described earlier in this chapter, as well as be capable of supporting security technologies that are expected to be widely deployed in enterprises, including Kerberos-based secret key mechanisms and X.509 certificate-based public key mechanisms.

The authentication identity (i.e. principal) associated with a Java EE component is usually that of the user on whose behalf the component is executing[70]. The principal under which an enterprise bean invocation is performed is either that of the bean's caller or the run-as principal which was configured by the Deployer. When there is a chain of invocations across a web component and enterprise beans, an intermediate component may use the principal of the caller (the initiating client) or the intermediate component may use its run-as principal to perform an invocation on the callee, depending on the security identity specified for the intermediate component in its deployment descriptor.

The security principal associated with a container depends on the type of container. Application client containers usually do not have a separate principal associated with them (they operate under the user's principal). Web and EJB containers are typically associated with a security principal of their own (e.g., the operating system user for the container's process) which may be configured by the System Administrator at deployment time. When the client is a web or EJB container, the difference between the client component's principal and the client container's principal is significant for interoperability considerations.

14.8.1.1 Trust Relationships Between Containers, Principal Propagation



When there is a chain of multiple invocations across web components and enterprise beans, intermediate components may not have access to the authentication data of the initiating client to provide proof of the client's identity to the target. In such cases, the target's authentication requirements can be satisfied if the target container trusts the intermediate container to vouch for the authenticity of the propagated principal. The call is made using the intermediate container's principal and authentication data, while also carrying the propagated principal of the initiating client. The invocation on the target enterprise bean is authorized and performed using the propagated principal. This procedure also avoids the overhead associated with authentication of clients on every remote invocation in a chain.

EJB containers are required to provide the Deployer or Administrator with the tools to configure trust relationships for interactions with intermediate web or EJB containers[71]. If a trust relationship is set up, the containers are usually configured to perform mutual authentication, unless the security of the network can be ensured by some physical means. If the network is physically secure, the target EJB container may be configured to trust all client containers. After a trust relationship is set up, the target EJB container does not need to independently authenticate the initiating client principal sent by the intermediate container on invocations. Thus only the principal name of the initiating client (which may include a realm) needs to be propagated. After a trust relationship has been established, the target EJB container must be able to accept invocations carrying only the principal name of the initiating client.

[70] When there are concurrent invocations on a component from multiple clients, a different principal may be associated with the thread of execution for each invocation.

[71] One way to achieve this is to configure a "trusted container list" for each EJB container which contains the list of intermediate client containers that are trusted. If the list is empty, then the target EJB container does not have a trust relationship with any intermediate container.

For the current interoperability needs of Java EE, it is assumed that trust relationships are transitive, such that if a target container trusts an intermediate container, it implicitly trusts all containers trusted by the intermediate container.

If no trust relationship has been set up between a target EJB container and an intermediate web or EJB container, the target container must not accept principals propagated from that intermediate container, hence the target container needs to have access to and independently verify the initiating client principal's authentication data.

Web and EJB containers are required to support caller propagation mode (where the initiating client's principal is propagated down the chain of calls on enterprise beans) and run-as mode (where the web/EJB component's run-as identity is propagated). This is needed for scenarios 1, 3 and 4 where the internet or intranet user's principal needs to be propagated to the target EJB container.

14.8.1.2 Application Client Authentication

Application client containers that have authentication infrastructure (such as certificates, Kerberos) can:

- authenticate the user by interacting with an authentication service (e.g. the Kerberos KDC) in the enterprise
- inherit an authentication context which was established at system login time from the operating system process, or
- obtain the user's certificate from a client-side store.

These may be achieved by plugging in a Java Authentication and Authorization Service (JAAS) login module for the particular authentication service. After authentication is completed, a credential is associated with the client's thread of execution, which is used for all invocations on enterprise beans made from that thread.

If there is no authentication infrastructure installed in the client's environment, or the authentication infrastructure is not capable of authenticating at the transport protocol layer, the client may send its private credentials (e.g. password) over a secure connection to the EJB server, which authenticates the user by interacting with an authentication service (e.g. a secure user/password database). This is similar to the basic authentication feature of HTTP.

14.8.2 Securing EJB Invocations

This subsection describes the interoperable protocol requirements for providing authentication, protection of integrity and confidentiality, and principal propagation for invocations on enterprise beans. The invocation takes place over an enterprise's intranet as described in the scenarios in section 14.3. Since EJB invocations use the IIOP protocol, we need to secure IIOP messages between client and server containers. The client container may be any of the Java EE containers; the server container is an EJB container.

The secure interoperability requirements for EJB 2.0 (and later) and other J2EE 1.3 (and later) containers are based on Conformance Level 0 of the Common Secure Interoperability version 2 (CSIv2) Final Available specification [23], which was developed by the OMG. EJB, web, and application client containers must support all requirements of Conformance Level 0 of the CSIv2 specification. The following subsections describe how the CSIv2 features are used to realize the scenarios described in section 14.3.

14.8.2.1 Secure Transport Protocol

The Secure Sockets Layer (SSL 3.0) protocol [22] and the related IETF standard Transport Layer Security (TLS 1.0) protocol [20] provide authentication and message protection (that is, integrity and/or confidentiality) at the transport layer. The original SSL and TLS specifications supported only X.509 certificates for authenticating principals. Recently, Kerberos-based authentication mechanisms and cipher suites have been defined for TLS (RFC 2712 [21]). Thus the TLS specification is capable of supporting the two main security technologies that are expected to be widely deployed in enterprises.

EJB, web and application client containers are required to support both SSL 3.0 and TLS 1.0 as security protocols for IIOP. This satisfies interoperability requirement 3 in section 14.4. Compliant containers must be capable of using the following public key SSL/TLS ciphersuites based on policies set by the System Administrator:

- TLS_RSA_WITH_RC4_128_MD5
- SSL_RSA_WITH_RC4_128_MD5
- TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA[72]
- SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA
- TLS_RSA_EXPORT_WITH_RC4_40_MD5
- SSL_RSA_EXPORT_WITH_RC4_40_MD5
- TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
- SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA

Support for Kerberos ciphersuites is not specified.

When using IIOP over SSL, a secure channel between client and server containers is established at the SSL layer. The SSL handshake layer handles authentication (either mutual or server-only) between containers, negotiation of cipher suite for bulk data encryption, and optionally provides a compression method. The SSL record layer performs confidentiality and integrity protection on application data. Since compliant Java EE products are already required to support SSL (HTTPS for Internet communication), the use of SSL/TLS provides a relatively easy route to interoperable security at the transport layer.

[72] This ciphersuite is mandatory for compliant TLS implementations as specified in [20].

14.8.2.2 Security Information in IORs

Before initiating a secure connection to the EJB container, the client needs to know the hostname and port number at which the server is listening for SSL connections, and the security protocols supported or required by the server object. This information is obtained from the EJBObject or EJBHome reference's IOR.

The CSIv2 specification [23] describes the `TAG_CSI_SEC_MECH_LIST` tagged component which is included in the IORs of secured objects. This component contains a sequence of `CSIIOP::CompoundSecMech` structures (in decreasing order of the server's preference) that contain the target object's security information for transport layer and service context layer mechanisms. This information includes the server's SSL/TLS port, its security principal and supported/required security mechanisms.

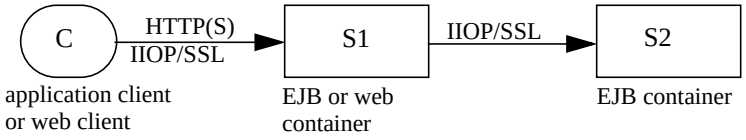
EJB containers must be capable of inserting the CSIv2 tagged components into the IORs for EJBObject and EJBHome references, based on the Deployer or System Administrator's security policy settings. Compliant EJB containers must follow the Conformance Level 0 rules described in the CSIv2 specification for constructing these IORs.

EJB containers must also be capable of creating IORs that allow access to enterprise beans over unprotected IIOP, based on the security policies set by the Deployer or System Administrator.

14.8.2.3 Propagating Principals and Authentication Data in IIOP Messages

In scenarios where client authentication does not occur at the transport layer it is necessary to support transfer of authentication data between two containers in the IIOP message service context. When an intermediate client container does not have authentication data for the initiating client, it is necessary to support propagation of client principals in the IIOP message service context.

It is assumed that all information exchanged between client and server at the transport layer is also available to the containers: e.g. the certificates used for authentication at the SSL layer may be used by the server container for authorization.



The following cases are required to be supported:

1. Application client invocations on enterprise beans with mutual authentication between the application client and EJB container (C and S1) at the SSL layer (scenario 2.1 in section 14.3.2, interoperability requirement 4.1 in section 14.4). For example, this is possible when the enterprise has a Kerberos-based authentication infrastructure or when client-side certificates have been installed. In this case the security context of the IIOP message sent from C to S1 should not contain any additional information.

2. Application client invocations on enterprise beans with server-only authentication between the application client and EJB container (C and S1) at the SSL layer (scenario 2.2 in section 14.3.2, interoperability requirement 4.2 in section 14.4). This usually happens when the client cannot authenticate in the transport. In this case, the client container must be capable of inserting into the IIOP message a CSIv2 security context with a client authentication token that contains the client C's authentication data. Once the EJB container S1 has authenticated the client, it may or may not maintain state about the client, so subsequent invocations from the client on the same network connection may need to be authenticated again. The client and server containers must follow the Conformance Level 0 rules in the CSIv2 specification for client authentication. In particular, support for the GSSUP username-password authentication mechanism is required. Support for other GSSAPI mechanisms (such as Kerberos) to perform client authentication at the IIOP layer is optional.

3. Invocations from Web/EJB clients to enterprise beans with a trust relationship between the client container S1 and server container S2 (scenarios 1,3 and 4 in section 14.3.3, interoperability requirements five and six in section 14.4). S2 does not need to independently authenticate the initiating client C. In this case the client container S1 must insert into the IIOP message a security context with an identity token in the format described in the CSIv2 specification. The principal may be propagated as an X.509 certificate chain or as a X.501 distinguished name or as a principal name encoded in the GSS exported name format, as described in the CSIv2 specification. The identity propagated is determined as follows:
    - If the client Web/EJB component is configured to use caller identity, and the caller C authenticated itself to S1, then the identity token contains the initiating client C's identity.
    - If the client component is configured to use caller identity, and the caller C did not authenticate itself to S1, then the identity token contains the anonymous type.
    - If the client component is configured to use a run-as identity then the identity token contains the run-as identity.

Java EE containers are required to support the stateless mode of propagating principal and authentication information defined in CSIv2 (where the server does not store any state for a particular client principal across invocations), and may optionally support the stateful mode.

The caller principal String provided by `EJBContext.getCallerPrincipal().getName()` is defined as follows:

- For case one, the principal should be derived from the distinguished name obtained from the first X.509 certificate in the client's certificate chain that was provided to the server during SSL mutual authentication.

- For case two, the principal should be derived from the username obtained from the client authentication token in the CSIv2 security context of the IIOP message. For the GSSUP username-password mechanism, the principal should be derived from the username in the `GSSUP::InitialContextToken` structure.

- For case three, the principal depends on the identity token type in the CSIv2 security context:
    - If the type is X.509 certificate chain, then the principal should be derived from the distinguished name from the first certificate in the chain.

- If the type is distinguished name, then the principal should be derived from the distinguished name.
- If the type is principal name propagated as a GSS exported name, then the principal should be derived from the mechanism-specific principal name.
- If the anonymous principal type was propagated or the identity token was absent, then `EJBContext.getCallerPrincipal().getName()` returns a product-specific unauthenticated principal name.

14.8.2.4 Security Configuration for Containers

Since the interoperability scenarios involve IIOP/SSL usage in intranets, it is assumed that client and server container administrators cooperatively configure a consistent set of security policies for the enterprise.

At product installation or application deployment time, client and server container administrators may optionally configure the container and SSL infrastructure as described below. These preferences may be specified at any level of granularity (e.g. per host or per container process or per enterprise bean).

- Configure the list of supported SSL cipher suites in preference order.
- For server containers, configure a list of trusted client container principals with whom the server has a trust relationship.
- Configure authentication preferences and requirements (e.g. if the server prefers authenticated clients to anonymous clients). In particular, if a trust relationship has been configured between two servers, then mutual authentication should be required unless there is physical network security.
- If the client and server are using certificates for authentication, configure a trusted common certificate authority for both client and server. If using Kerberos, configure the client and server with the same KDC or cooperating KDCs.
- Configure a restricted list of trusted server principals that a client container is allowed to interact with, to prevent the client's private credentials such as password from being sent to untrusted servers.

14.8.2.5 Runtime Behavior

Client containers determine whether to use SSL for an enterprise bean invocation by using the security policies configured by the client administrator for interactions with the target host or enterprise bean, and the `target_requires` information in the CSIv2 tagged component in the target enterprise bean's IOR. If either the client configuration requires secure interactions with the enterprise bean, or the enterprise bean requires a secure transport, the client should initiate an SSL connection to the server. The client must follow the rules described in the CSIv2 specification Conformance Level 0 for interpreting security information in IORs and including security context information in IIOP messages.

When an EJB container receives an IIOP message, its behavior depends on deployment time configuration, run-time information exchanged with the client at the SSL layer, and principal/authentication data contained in the IIOP message service context. EJB containers are required to follow the protocol rules prescribed by the CSIv2 specification Conformance Level 0.

When the System Administrator changes the security policies associated with an enterprise bean, the IORs for EJB references should be updated. When the bean has existing clients holding IORs, it is recommended that the security policy change should be handled by the client and server containers transparently to the client application if the old security policy is compatible with the new one. This may be done by using interoperable GIOP 1.2 forwarding mechanisms.

Chapter 15 Enterprise Bean Environment

This chapter specifies how enterprise beans declare dependencies on external resources and other objects in their environment, and how those items can be injected into enterprise beans or accessed in the JNDI naming context.

15.1 Overview

The Application Assembler and Deployer should be able to customize an enterprise bean's business logic without accessing the enterprise bean's source code.

In addition, ISVs typically develop enterprise beans that are, to a large degree, independent from the operational environment in which the application will be deployed. Most enterprise beans must access resource managers and external information. The key issue is how enterprise beans can locate external information without prior knowledge of how the external information is named and organized in the target operational environment. The JNDI naming context and Java language metadata annotations provide this capability.

The enterprise bean environment mechanism attempts to address both of the above issues.

This chapter is organized as follows.

- Section 15.2 defines the general rules for the use of the JNDI naming context and its interaction with Java language annotations that reference entries in the naming context.
- Section 15.3 defines the general responsibilities for each of the EJB roles, such as Bean Provider, Application Assembler, Deployer, and Container Provider.
- Section 15.4 defines the basic mechanisms and interfaces that specify and access the enterprise bean's environment. The section illustrates the use of the enterprise bean's environment for generic customization of the enterprise bean's business logic.
- Section 15.5 defines the means for obtaining the business interface or home interface of another enterprise bean using an *EJB reference*. An EJB reference is a special entry in the enterprise bean's environment.
- Section 15.6 defines the means for obtaining the web service interface using a *web service reference*. A web service reference is a special entry in the enterprise bean's environment.
- Section 15.7 defines the means for obtaining a resource manager connection factory using a *resource manager connection factory reference*. A resource manager connection factory reference is a special entry in the enterprise bean's environment.
- Section 15.8 defines the means for obtaining an administered object that is associated with a resource (e.g., a CCI `InteractionSpec`) using a *resource environment reference*. A resource environment reference is a special entry in the enterprise bean's environment.
- Section 15.9 defines the means for obtaining a message destination associated with a resource using a *message destination reference*. Message destination references allow the flow of messages within an application to be specified. A message destination reference is a special entry in the enterprise bean's environment.
- Section 15.10 describes the means for obtaining an entity manager factory using a *persistence unit reference*.
- Section 15.11 describes the means for obtaining an entity manager using a *persistence context reference*.
- Section 15.12 describes the use by eligible enterprise beans of references to a `UserTransaction` object in the bean's environment to start, commit, and rollback transactions.
- Section 15.13 describes the use of references to a CORBA `ORB` object in the enterprise bean's environment.
- Section 15.14 describes the means for obtaining the TimerService.
- Section 15.15 describes the means for obtaining a bean's EJBContext object.

15.2 Enterprise Bean's Environment as a JNDI Naming Context

The enterprise bean's environment is a mechanism that allows customization of the enterprise bean's business logic during deployment or assembly. The enterprise bean's environment allows the enterprise bean to be customized without the need to access or change the enterprise bean's source code.

Annotations and deployment descriptors are the main vehicles for conveying access information to the application assembler and deployer about beans' requirements for customization of business logic and access to external information.

The container implements the enterprise bean's environment, and provides it as a JNDI naming context. The enterprise bean's environment is used as follows:

1. The enterprise bean makes use of entries from the environment. Entries from the environment may be injected by the container into the bean's fields or methods, or the methods of the bean may access the environment using the EJBContext `lookup` method or the JNDI interfaces. The Bean Provider declares in Java language metadata annotations or in the deployment descriptor all the environment entries that the enterprise bean expects to be provided in its environment at runtime.

2. The container provides an implementation of the JNDI naming context that stores the enterprise bean environment. The container also provides the tools that allow the Deployer to create and manage the environment of each enterprise bean.

3. The Deployer uses the tools provided by the container to create and initialize the environment entries that are declared by means of the enterprise bean's annotations or deployment descriptor. The Deployer can set and modify the values of the environment entries.

4. The container injects entries from the environment into the enterprise bean's fields or methods as specified by the bean's metadata annotations or the deployment descriptor.

5. The container makes the environment naming context available to the enterprise bean instances at runtime. The enterprise bean's instances can use the EJBContext `lookup` method or the JNDI interfaces to obtain the values of the environment entries.

The container must make an enterprise bean's environment available to any interceptor class and any JAX-WS handler for the bean as well. The interceptor and web service handler classes for an enterprise bean share that bean's environment. Within the context of this chapter, the term "bean" should be construed as including a bean's interceptor and handler classes unless otherwise noted.

15.2.1 Sharing of Environment Entries

Each enterprise bean defines its own set of environment entries. All instances of an enterprise bean share the same environment entries; the environment entries are not shared with other enterprise beans. Enterprise bean instances are not allowed to modify the bean's environment at runtime.

Compatibility Note: If an enterprise bean written to the EJB 2.1 API specification is deployed multiple times in the same container, each deployment results in the creation of a distinct home. The Deployer may set different values for the enterprise bean environment entries for each home.

In general, lookups of objects in the JNDI `java:` namespace are required to return a new instance of the requested object every time. Exceptions are allowed for the following:

- The container knows the object is immutable (for example, objects of type `java.lang.String`), or knows that the application can't change the state of the object.
- The object is defined to be a singleton, such that only one instance of the object may exist in the JVM.
- The name used for the lookup is defined to return an instance of the object that might be shared. The name `java:comp/ORB` is such a name.

In these cases, a shared instance of the object may be returned. In all other cases, a new instance of the requested object must be returned on each lookup. Note that, in the case of resource adapter connection objects, it is the resource adapter's `ManagedConnectionFactory` implementation that is responsible for satisfying this requirement.

Each injection of an object corresponds to a JNDI lookup. Whether a new instance of the requested object is injected, or whether a shared instance is injected, is determined by the rules described above.

Terminology warning: The enterprise bean's "environment" should not be confused with the "environment properties" defined in the JNDI documentation.

15.2.2 Annotations for Environment Entries

A field or method of a bean class may be annotated to request that an entry from the bean's environment be injected. Any of the types of resources or other environment entries[73] described in this chapter may be injected. Injection may also be requested using entries in the deployment descriptor corresponding to each of these resource types. The field or method may have any access qualifier (`public`, `private`, etc.) but must not be `static`.

- A field of the bean class may be the target of injection. The field must not be `final`. By default, the name of the field is combined with the name of the class in which the annotation is used and is used directly as the name in the bean's naming context. For example, a field named `myDatabase` in the class `MySessionBean` in the package `com.acme.example` would correspond to the JNDI name `java:comp/env/com.acme.example.MySessionBean/myDatabase`. The annotation also allows the JNDI name to be specified explicitly.
- Environment entries may also be injected into the bean through bean methods that follow the naming conventions for JavaBeans properties. The annotation is applied to the `set` method for the property, which is the method that is called to inject the environment entry. The JavaBeans property name (not the method name) is used as the default JNDI name. For example, a

[73] The term "resource" is used generically in this chapter to refer to these other environment entries as resources as well. Resources in the non-generic sense are described in section 15.7.

method named `setMyDatabase` in the same `MySessionBean` class would correspond to the JNDI name `java:comp/env/com.example.MySessionBean/myDatabase`.

- When a deployment descriptor entry is used to specify injection, the JNDI name and the instance variable name or property name are both specified explicitly. Note that the JNDI name is always relative to the `java:comp/env` naming context.

Each resource may only be injected into a single field or method of the bean. Requesting injection of the `java:comp/env/com.example.MySessionBean/myDatabase` resource into both the `setMyDatabase` method and the `myDatabase` instance variable is an error. Note, however, that either the field or the method could request injection of a resource of a different (non-default) name. By explicitly specifying the JNDI name of a resource, a single resource may be injected into multiple fields or methods of multiple classes.

Annotations may also be applied to the bean class itself. These annotations declare an entry in the bean's environment, but do not cause the resource to be injected. Instead, the bean is expected to use the EJBContext `lookup` method or the methods of the JNDI API to lookup the entry. When the annotation is applied to the bean class, the JNDI name and the environment entry type must be explicitly specified.

Annotations may appear on the bean class, or on any superclass. A resource annotation on any class in the inheritance hierarchy defines a resource needed by the bean. However, injection of such resources follows the Java language overriding rules for the visibility of fields and methods. A method definition that overrides a method on a superclass defines the resource, if any, to be injected into that method. An overriding method may request injection of a different resource than is requested by the superclass, or it may request no injection even though the superclass method requests injection.

In addition, fields or methods that are not visible in or are hidden (as opposed to overridden) by a subclass may still request injection. This allows, for example, a private field to be the target of injection and that field to be used in the implementation of the superclass, even though the subclass has no visibility into that field and doesn't know that the implementation of the superclass is using an injected resource. Note that a declaration of a field in a subclass with the same name as a field in a superclass always causes the field in the superclass to be hidden.

15.2.3 Annotations and Deployment Descriptors

Environment entries may be declared by the use of annotations, without need for any deployment descriptor entries. Environment entries may also be declared by deployment descriptor entries, without need for any annotations. The same environment entry may be declared using both an annotation and a deployment descriptor entry. In this case, the information in the deployment descriptor entry may be used to override some of the information provided in the annotation. This approach may be used by an Application Assembler to override information provided by the Bean Provider. Deployment descriptor entries should not be used to request injection of a resource into a field or method that has not been annotated for injection.

The following rules apply to how a deployment descriptor entry may override a `Resource` annotation:

- The relevant deployment descriptor entry is located based on the JNDI name used with the annotation (either defaulted or provided explicitly).

- The type specified in the deployment descriptor must be assignable to the type of the field or property.
- The description, if specified, overrides the description element of the annotation.
- The injection target, if specified, must name exactly the annotated field or property method.
- The `res-sharing`-scope element, if specified, overrides the `shareable` element of the annotation. In general, the Application Assembler or Deployer should never change the value of this element, as doing so is likely to break the application.
- The `res-auth` element, if specified, overrides the `authenticationType` element of the annotation. In general, the Application Assembler or Deployer should never change the value of this element, as doing so is likely to break the application.

Restrictions on the overriding of environment entry values depend on the type of environment entry.

The rules for how a deployment descriptor entry may override an EJB annotation are described in Section 15.5. The rules for how a deployment descriptor entry may override a `PersistenceUnit` or `PersistenceContext` annotation are described in Sections 15.10 and 15.11. The rules for web services references and how a deployment descriptor entry may override a `WebServiceRef` annotation are included in the Web Services for Java EE specification [31].

15.3 Responsibilities by EJB Role

This section describes the responsibilities of the various EJB roles with regard to the specification and handling of environment entries. The sections that follow describe the responsibilities that are specific to the different types of objects that may be stored in the naming context.

15.3.1 Bean Provider's Responsibilities

The Bean Provider may use Java language annotations or deployment descriptor entries to request injection of a resource from the naming context, or to declare entries that are needed in the naming context. The Bean Provider may also use the EJBContext `lookup` method or the JNDI APIs to access entries in the naming context. Deployment descriptor entries may also be used by the Bean Provider to override information provided by annotations.

> *When using JNDI interfaces directly, an enterprise bean instance creates a `javax.naming.InitialContext` object by using the constructor with no arguments, and looks up the environment naming via the `InitialContext` under the name `java:comp/env`.*

The enterprise bean's environment entries are stored directly in the environment naming context, or in any of its direct or indirect subcontexts.

The value of an environment entry is of the Java type declared by the Bean Provider in the metadata annotation or deployment descriptor, or the type of the instance variable or setter method parameter of the method with which the metadata annotation is associated.

15.3.2 Application Assembler's Responsibility

The Application Assembler is allowed to modify the values of the environment entries set by the Bean Provider, and is allowed to set the values of those environment entries for which the Bean Provider has not specified any initial values. The Application Assembler uses the deployment descriptor to override settings made by the Bean Provider, whether these were defined by the Bean Provider in the deployment descriptor or in the source code using annotations.

15.3.3 Deployer's Responsibility

The Deployer must ensure that the values of all the environment entries declared by an enterprise bean are created and/or set to meaningful values.

The Deployer can modify the values of the environment entries that have been previously set by the Bean Provider and/or Application Assembler, and must set the values of those environment entries for which no value has been specified.

The `description` elements provided by the Bean Provider or Application Assembler help the Deployer with this task.

15.3.4 Container Provider Responsibility

The Container Provider has the following responsibilities:

- Provide a deployment tool that allows the Deployer to set and modify the values of the enterprise bean's environment entries.
- Implement the `java:comp/env` environment naming context, and provide it to the enterprise bean instances at runtime. The naming context must include all the environment entries declared by the Bean Provider, with their values supplied in the deployment descriptor or set by the Deployer. The environment naming context must allow the Deployer to create subcontexts if they are needed by an enterprise bean.
- Inject entries from the naming environment, as specified by annotations or by the deployment descriptor.
- The container must ensure that the enterprise bean instances have only read access to their environment variables. The container must throw the `javax.naming.OperationNotSupportedException` from all the methods of the `javax.naming.Context` interface that modify the environment naming context and its subcontexts.

15.4 Simple Environment Entries

A simple environment entry is a configuration parameter used to customize an enterprise bean's business logic. The environment entry values may be one of the following Java types: `String`, `Character`, `Integer`, `Boolean`, `Double`, `Byte`, `Short`, `Long`, and `Float`.

The following subsections describe the responsibilities of each EJB role.

15.4.1 Bean Provider's Responsibilities

This section describes the Bean Provider's view of the bean's environment, and defines his or her responsibilities. The first subsection describes annotations for injecting simple environment entries; the second describes the API for accessing simple environment entries; and the third describes syntax for declaring the environment entries in a deployment descriptor.

15.4.1.1 Injection of Simple Environment Entries Using Annotations

The Bean Provider uses the `Resource` annotation to annotate a field or method of the bean class as a target for the injection of a simple environment entry. The name of the environment entry is as described in Section 15.2.2; the type is as described in Section 15.4. Note that the container will unbox the environment entry as required to match it to a primitive type used for the injection field or method. The `authenticationType` and `shareable` elements of the `Resource` annotation must not be specified; simple environment entries are not shareable and do not require authentication.

The following code example illustrates how an enterprise bean uses annotations for the injection of environment entries.

```
@Stateless public class EmployeeServiceBean
         implements EmployeeService {

     ...
     // The maximum number of tax exemptions, configured by Deployer
     @Resource int maxExemptions;

     // The minimum number of tax exemptions, configured by Deployer
     @Resource int minExemptions;

     public void setTaxInfo(int numberOfExemptions,...)
              throws InvalidNumberOfExemptionsException {
          ...
          // Use the environment entries to customize business logic.
          if (numberOfExemptions > maxExemptions ||
               numberOfExemptions < minExemptions)
                    throw new InvalidNumberOfExemptionsException();

     }
}
```

15.4.1.2 Programming Interfaces for Accessing Simple Environment Entries

In addition to the use of injection as described above, an enterprise bean may access environment entries dynamically. This may be done by means of the EJBContext `lookup` method or by direct use of the JNDI interfaces. The environment entries are declared by the Bean Provider by means of annotations on the bean class or in the deployment descriptor.

When the JNDI interfaces are used directly, the bean instance creates a `javax.naming.InitialContext` object by using the constructor with no arguments, and looks up the naming environment via the `InitialContext` under the name `java:comp/env`. The bean's environmental entries are stored directly in the environment naming context, or its direct or indirect subcontexts.

The following code example illustrates how an enterprise bean accesses its environment entries when the JNDI APIs are used directly. In this example, the names under which the entries are accessed are defined by the deployment descriptor, as shown in the example of section 15.4.1.3.

```
@Stateless public class EmployeeServiceBean
         implements EmployeeService {

     ...
     public void setTaxInfo(int numberOfExemptions, ...)
              throws InvalidNumberOfExemptionsException {
          ...

          // Obtain the enterprise bean's environment naming context.
          Context initCtx = new InitialContext();
          Context myEnv = (Context)initCtx.lookup("java:comp/env");

          // Obtain the maximum number of tax exemptions
          // configured by the Deployer.
          Integer maxExemptions =
              (Integer)myEnv.lookup("maxExemptions");

          // Obtain the minimum number of tax exemptions
          // configured by the Deployer.
          Integer minExemptions =
              (Integer)myEnv.lookup("minExemptions");

          // Use the environment entries to customize business logic.
          if (numberOfExeptions > maxExemptions ||
              numberOfExemptions < minExemptions)
                   throw new InvalidNumberOfExemptionsException();

          // Get some more environment entries. These environment
          // entries are stored in subcontexts.
          String val1 = (String)myEnv.lookup("foo/name1");
          Boolean val2 = (Boolean)myEnv.lookup("foo/bar/name2");

          // The enterprise bean can also lookup using full pathnames.
          Integer val3 = (Integer)
              initCtx.lookup("java:comp/env/name3");
          Integer val4 = (Integer)
              initCtx.lookup("java:comp/env/foo/name4");
          ...
     }
}
```

15.4.1.3 Declaration of Simple Environment Entries in the Deployment Descriptor

The Bean Provider must declare all the simple environment entries accessed from the enterprise bean's code. The simple environment entries are declared either using annotations in the bean class code or using the `env-entry` elements in the deployment descriptor.

Each `env-entry` deployment descriptor element describes a single environment entry. The `env-entry` element consists of an optional description of the environment entry, the environment entry name relative to the `java:comp/env` context, the expected Java type of the environment entry value (i.e., the type of the object returned from the EJBContext or JNDI `lookup` method), and an optional environment entry value.

An environment entry is scoped to the enterprise bean whose deployment descriptor element contains the given `env-entry` element. This means that the environment entry is inaccessible from other enterprise beans at runtime, and that other enterprise beans may define `env-entry` elements with the same `env-entry-name` without causing a name conflict.

If the Bean Provider provides a value for an environment entry using the `env-entry-value` element, the value can be changed later by the Application Assembler or Deployer. The value must be a string that is valid for the constructor of the specified type that takes a single `String` parameter, or for `java.lang.Character`, a single character.

The following example is the declaration of environment entries used by the `EmployeeService-Bean` whose code was illustrated in the previous subsection.

```
<enterprise-beans>
     <session>
          ...
          <ejb-name>EmployeeService</ejb-name>
          <ejb-class>com.wombat.empl.EmployeeServiceBean</ejb-class>
          ...
          <env-entry>
               <description>
                    The maximum number of tax exemptions
                    allowed to be set.
               </description>
               <env-entry-name>maxExemptions</env-entry-name>
               <env-entry-type>java.lang.Integer</env-entry-type>
               <env-entry-value>15</env-entry-value>
          </env-entry>
          <env-entry>
               <description>
                    The minimum number of tax exemptions
                    allowed to be set.
               </description>
               <env-entry-name>minExemptions</env-entry-name>
               <env-entry-type>java.lang.Integer</env-entry-type>
               <env-entry-value>1</env-entry-value>
          </env-entry>
          <env-entry>
               <env-entry-name>foo/name1</env-entry-name>
               <env-entry-type>java.lang.String</env-entry-type>
               <env-entry-value>value1</env-entry-value>
          </env-entry>
          <env-entry>
               <env-entry-name>foo/bar/name2</env-entry-name>
               <env-entry-type>java.lang.Boolean</env-entry-type>
               <env-entry-value>true</env-entry-value>
          </env-entry>
          <env-entry>
               <description>Some description.</description>
               <env-entry-name>name3</env-entry-name>
               <env-entry-type>java.lang.Integer</env-entry-type>
          </env-entry>
          <env-entry>
               <env-entry-name>foo/name4</env-entry-name>
               <env-entry-type>java.lang.Integer</env-entry-type>
               <env-entry-value>10</env-entry-value>
          </env-entry>
          ...
     </session>
</enterprise-beans>
...
```

Injection of environment entries may also be specified using the deployment descriptor, without need for Java language annotations. The following is an example of the declaration of environment entries corresponding to the example of section 15.4.1.1.

```
<enterprise-beans>
     <session>
          ...
          <ejb-name>EmployeeService</ejb-name>
          <ejb-class>com.wombat.empl.EmployeeServiceBean</ejb-class>
          ...
          <env-entry>
               <description>
                    The maximum number of tax exemptions
                    allowed to be set.
               </description>
               <env-entry-name>
                    com.wombat.empl.EmployeeService/maxExemptions
               </env-entry-name>
               <env-entry-type>java.lang.Integer</env-entry-type>
               <env-entry-value>15</env-entry-value>
               <injection-target>
                    <injection-target-class>
                         com.wombat.empl.EmployeeServiceBean
                    </injection-target-class>
                    <injection-target-name>
                         maxExemptions
                    </injection-target-name>
               </injection-target>
          </env-entry>
          <env-entry>
               <description>
                    The minimum number of tax exemptions
                    allowed to be set.
               </description>
               <env-entry-name>
                    com.wombat.empl.EmployeeService/minExemptions
               </env-entry-name>
               <env-entry-type>java.lang.Integer</env-entry-type>
               <env-entry-value>1</env-entry-value>
               <injection-target>
                    <injection-target-class>
                         com.wombat.empl.EmployeeServiceBean
                    </injection-target-class>
                    <injection-target-name>
                         minExemptions
                    </injection-target-name>
               </injection-target>
          </env-entry>
          ...
     </session>
</enterprise-beans>
...
```

It is often convenient to declare a field as an injection target, but to specify a default value in the code, as illustrated in the following example.

```
// The maximum number of tax exemptions, configured by the Deployer.
@Resource int maxExemptions = 4; // defaults to 4
```

To support this case, the container must only inject a value for the environment entry if the application assembler or deployer has specified a value to override the default value. The `env-entry-value` element in the deployment descriptor is optional when an injection target is specified. If the element is not specified, no value will be injected. In addition, if the element is not specified, the named resource is not initialized in the naming context, and explicit lookups of the named resource will fail.

15.4.2 Application Assembler's Responsibility

The Application Assembler is allowed to modify the values of the simple environment entries set by the Bean Provider, and is allowed to set the values of those environment entries for which the Bean Provider has not specified any initial values. The Application Assembler may use the deployment descriptor to override settings made by the Bean Provider, whether in the deployment descriptor or using annotations.

15.4.3 Deployer's Responsibility

The Deployer must ensure that the values of all the simple environment entries declared by an enterprise bean are set to meaningful values.

The Deployer can modify the values of the environment entries that have been previously set by the Bean Provider and/or Application Assembler, and must set the values of those environment entries for which no value has been specified.

The `description` elements provided by the Bean Provider or Application Assembler help the Deployer with this task.

15.4.4 Container Provider Responsibility

The Container Provider has the following responsibilities:

- Provide a deployment tool that allows the Deployer to set and modify the values of the enterprise bean's environment entries.
- Implement the `java:comp/env` environment naming context, and provide it to the enterprise bean instances at runtime. The naming context must include all the environment entries declared by the Bean Provider, with their values supplied in the deployment descriptor or set by the Deployer. The environment naming context must allow the Deployer to create subcontexts if they are needed by an enterprise bean.
- Inject entries from the naming environment into the bean instance, as specified by the annotations on the bean class or by the deployment descriptor.
- The container must ensure that the enterprise bean instances have only read access to their environment variables. The container must throw the `javax.naming.OperationNotSupportedException` from all the methods of the `javax.naming.Context` interface that modify the environment naming context and its subcontexts.

15.5 EJB References

This section describes the programming and deployment descriptor interfaces that allow the Bean Provider to refer to the business interfaces or homes of other enterprise beans using "logical" names called *EJB references*. The EJB references are special entries in the enterprise bean's environment. The Deployer binds the EJB references to the enterprise bean business interfaces or homes in the target operational environment, as appropriate.

The deployment descriptor also allows the Application Assembler to link an EJB reference declared in one enterprise bean to another enterprise bean contained in the same ejb-jar file, or in another ejb-jar file in the same Java EE application unit. The link is an instruction to the tools used by the Deployer that the EJB reference should be bound to the business interface or home of the specified target enterprise bean. This linking can also be specified by the Bean Provider using annotations in the source code of the bean class.

15.5.1 Bean Provider's Responsibilities

This section describes the Bean Provider's view and responsibilities with respect to EJB references. The first subsection describes annotations for injecting EJB references; the second describes the API for accessing EJB references; and the third describes syntax for declaring the EJB references in a deployment descriptor.

15.5.1.1 Injection of EJB References

The Bean Provider uses the `EJB` annotation to annotate a field or setter property method of the bean class as a target for the injection of an EJB reference. The reference may be to a session bean's business interface or to the local home interface or remote home interface of a session bean or entity bean.

The following example illustrates how an enterprise bean uses the `EJB` annotation to reference another enterprise bean. The enterprise bean reference will have the name `java:comp/env/com.acme.example.ExampleBean/myCart` in the referencing bean's naming context, where `ExampleBean` is the name of the class of the referencing bean and `com.acme.example` its package. The target of the reference must be resolved by the Deployer.

```
package com.acme.example;

@Stateless public class ExampleBean implements Example {
     ...
     @EJB private ShoppingCart myCart;
     ...
}
```

The following example illustrates use of all portable elements of the `EJB` annotation. In this case, the enterprise bean reference would have the name `java:comp/env/ejb/shopping-cart` in the referencing bean's naming context. This reference is linked to a bean named `cart1`.

```
@EJB(
     name="ejb/shopping-cart",
     beanInterface=ShoppingCart.class,
     beanName="cart1",
     description="The shopping cart for this application"
)
private ShoppingCart myCart;
```

If the `ShoppingCart` bean were instead written to the EJB 2.1 client view, the EJB reference would be to the bean's home interface. For example:

```
@EJB(
     name="ejb/shopping-cart",
     beanInterface=ShoppingCartHome.class,
     beanName="cart1",
     description="The shopping cart for this application"
)
private ShoppingCartHome myCartHome;
```

15.5.1.2 EJB Reference Programming Interfaces

The Bean Provider may use EJB references to locate the business interfaces or home interfaces of other enterprise beans as follows.

- Assign an entry in the enterprise bean's environment to the reference. (See subsection 15.5.1.3 for information on how EJB references are declared in the deployment descriptor.)
- The EJB specification recommends, but does not require, that all references to other enterprise beans be organized in the `ejb` subcontext of the bean's environment (i.e., in the `java:comp/env/ejb` JNDI context). Note that enterprise bean references declared by means of annotations will not, by default, be in any subcontext.
- Look up the business interface or home interface of the referenced enterprise bean in the enterprise bean's environment using the EJBContext `lookup` method or the JNDI API.

The following example illustrates how an enterprise bean uses an EJB reference to locate the remote home interface of another enterprise bean using the JNDI APIs.

```
@EJB(name="ejb/EmplRecord", beanInterface=EmployeeRecordHome.class)
@Stateless public class EmployeeServiceBean
         implements EmployeeService {

     public void changePhoneNumber(...) {
          ...

          // Obtain the default initial JNDI context.
          Context initCtx = new InitialContext();

          // Look up the home interface of the EmployeeRecord
          // enterprise bean in the environment.
          Object result = initCtx.lookup(
               "java:comp/env/ejb/EmplRecord");

          // Convert the result to the proper type.
          EmployeeRecordHome emplRecordHome = (EmployeeRecordHome)
               javax.rmi.PortableRemoteObject.narrow(result,
                        EmployeeRecordHome.class);
          ...
     }
}
```

In the example, the Bean Provider of the `EmployeeServiceBean` enterprise bean assigned the environment entry `ejb/EmplRecord` as the EJB reference name to refer to the remote home of another enterprise bean.

15.5.1.3 Declaration of EJB References in Deployment Descriptor

Although the EJB reference is an entry in the enterprise bean's environment, the Bean Provider must not use a `env-entry` element to declare it. Instead, the Bean Provider must declare all the EJB references using the `ejb-ref` and `ejb-local-ref` elements of the deployment descriptor. This allows the ejb-jar consumer (i.e. Application Assembler or Deployer) to discover all the EJB references used by the enterprise bean. Deployment descriptor entries may also be used to specify injection of an EJB reference into a bean.

Each `ejb-ref` or `ejb-local-ref` element describes the interface requirements that the referencing enterprise bean has for the referenced enterprise bean. The `ejb-ref` element is used for referencing an enterprise bean that is accessed through its remote business interface or remote home and component interfaces. The `ejb-local-ref` element is used for referencing an enterprise bean that is accessed through its local business interface or local home and component interfaces.

The `ejb-ref` element contains the `description`, `ejb-ref-name`, `ejb-ref-type`, `home`, and `remote` elements.

The `ejb-local-ref` element contains the `description`, `ejb-ref-name`, `ejb-ref-type`, `local-home`, and `local` elements.

The `ejb-ref-name` element specifies the EJB reference name: its value is the environment entry name used in the enterprise bean code. The `ejb-ref-name` must be specified. The optional `ejb-ref-type` element specifies the expected type of the enterprise bean: its value must be either `Entity` or `Session`. The `home` and `remote` or `local-home` and `local` elements specify the expected Java types of the referenced enterprise bean's interface(s). If the reference is to an EJB 2.1 remote client view interface, the `home` element is required. Likewise, if the reference is to an EJB 2.1 local client view interface, the `local-home` element is required. The `remote` element of the `ejb-ref` element refers to either the business interface type or the component interface, depending on whether the reference is to a bean's EJB 3.0 or EJB 2.1 remote client view. Likewise, the `local` element of the `ejb-local-ref` element refers to either the business interface type or the component interface, depending on whether the reference is to a bean's EJB 3.0 or EJB 2.1 local client view.

An EJB reference is scoped to the enterprise bean whose declaration contains the `ejb-ref` or `ejb-local-ref` element. This means that the EJB reference is not accessible to other enterprise beans at runtime, and that other enterprise beans may define `ejb-ref` and/or `ejb-local-ref` elements with the same `ejb-ref-name` without causing a name conflict.

The following example illustrates the declaration of EJB references in the deployment descriptor.

```
...
<enterprise-beans>
     <session>
          ...
          <ejb-name>EmployeeService</ejb-name>
          <ejb-class>com.wombat.empl.EmployeeServiceBean</ejb-class>
          ...
          <ejb-ref>
               <description>
                    This is a reference to an EJB 2.1 entity bean that
                    encapsulates access to employee records.
               </description>
               <ejb-ref-name>ejb/EmplRecord</ejb-ref-name>
               <ejb-ref-type>Entity</ejb-ref-type>
               <home>com.wombat.empl.EmployeeRecordHome</home>
               <remote>com.wombat.empl.EmployeeRecord</remote>
          </ejb-ref>

          <ejb-ref>
               <description>
                    This is a reference to the local business interface
                    of an EJB 3.0 session bean that provides a payroll
                    service.
               </description>
               <ejb-ref-name>ejb/Payroll</ejb-ref-name>
               <local>com.aardvark.payroll.Payroll</local>
          </ejb-ref>

          <ejb-ref>
               <description>
                    This is a reference to the local business interface
                    of an EJB 3.0 session bean that provides a pension
                    plan service.
               </description>
               <ejb-ref-name>ejb/PensionPlan</ejb-ref-name>
               <local>com.wombat.empl.PensionPlan</local>
          </ejb-ref>
          ...
     </session>
     ...
</enterprise-beans>
...
```

15.5.2 Application Assembler's Responsibilities

The Application Assembler can use the `ejb-link` element in the deployment descriptor to link an EJB reference to a target enterprise bean.

The Application Assembler specifies the link between two enterprise beans as follows:

- The Application Assembler uses the optional `ejb-link` element of the `ejb-ref` or `ejb-local-ref` element of the referencing enterprise bean. The value of the `ejb-link` element is the name of the target enterprise bean. (This is the bean name as defined by metadata annotation (or default) in the bean class or in the `ejb-name` element of the target enter-

prise bean.) The target enterprise bean can be in any ejb-jar file in the same Java EE application as the referencing application component.

- Alternatively, to avoid the need to rename enterprise beans to have unique names within an entire Java EE application, the Application Assembler may use the following syntax in the `ejb-link` element of the referencing application component. The Application Assembler specifies the path name of the ejb-jar file containing the referenced enterprise bean and appends the ejb-name of the target bean separated from the path name by `#`. The path name is relative to the referencing application component jar file. In this manner, multiple beans with the same ejb-name may be uniquely identified when the Application Assembler cannot change ejb-names.

- The Application Assembler must ensure that the target enterprise bean is type-compatible with the declared EJB reference. This means that the target enterprise bean must be of the type indicated in the `ejb-ref-type` element, if present, and that the business interface or home and component interfaces of the target enterprise bean must be Java type-compatible with the interfaces declared in the EJB reference.

The following illustrates an `ejb-link` in the deployment descriptor.

```
...
<enterprise-beans>
     <session>
          ...
          <ejb-name>EmployeeService</ejb-name>
          <ejb-class>com.wombat.empl.EmployeeServiceBean</ejb-class>
          ...
          <ejb-ref>
               <ejb-ref-name>ejb/EmplRecord</ejb-ref-name>
               <ejb-ref-type>Entity</ejb-ref-type>
               <home>com.wombat.empl.EmployeeRecordHome</home>
               <remote>com.wombat.empl.EmployeeRecord</remote>
               <ejb-link>EmployeeRecord</ejb-link>
          </ejb-ref>
          ...
     </session>
     ...


     <entity>
          <ejb-name>EmployeeRecord</ejb-name>
          <home>com.wombat.empl.EmployeeRecordHome</home>
          <remote>com.wombat.empl.EmployeeRecord</remote>
          ...
     </entity>
     ...
</enterprise-beans>
...
```

The Application Assembler uses the `ejb-link` element to indicate that the EJB reference `EmplRecord` declared in the `EmployeeService` enterprise bean has been linked to the `EmployeeRecord` enterprise bean.

The following example illustrates using the `ejb-link` element to indicate an enterprise bean reference to the `ProductEJB` enterprise bean that is in the same Java EE application unit but in a different ejb-jar file.

```
<entity>
     ...
     <ejb-name>OrderEJB</ejb-name>
     <ejb-class>com.wombat.orders.OrderBean</ejb-class>
     ...
     <ejb-ref>
          <ejb-ref-name>ejb/Product</ejb-ref-name>
          <ejb-ref-type>Entity</ejb-ref-type>
          <home>com.acme.orders.ProductHome</home>
          <remote>com.acme.orders.Product</remote>
          <ejb-link>../products/product.jar#ProductEJB</ejb-link>
     </ejb-ref>
     ...
</entity>
```

The following example illustrates using the `ejb-link` element to indicate an enterprise bean reference to the `ShoppingCart` enterprise bean that is in the same Java EE application unit but in a different ejb-jar file. The reference was originally declared in the bean's code using an annotation. The Application Assembler provides only the link to the bean.

```
...
<ejb-ref>
     <ejb-ref-name>ShoppingService/myCart</ejb-ref-name>
     <ejb-link>../products/product.jar#ShoppingCart</ejb-link>
</ejb-ref>
```

15.5.2.1 Overriding Rules

The following rules apply to how a deployment descriptor entry may override an `EJB` annotation:

- The relevant deployment descriptor entry is located based on the JNDI name used with the annotation (either defaulted or provided explicitly).
- The type specified in the deployment descriptor via the `remote`, `local`, `remote-home`, or `local-home` element and any bean referenced by the `ejb-link` element must be assignable to the type of the field or property.
- The description, if specified, overrides the description element of the annotation.
- The injection target, if specified, must name exactly the annotated field or property method.

15.5.3 Deployer's Responsibility

The Deployer is responsible for the following:

- The Deployer must ensure that all the declared EJB references are bound to the business interfaces or homes of enterprise beans that exist in the operational environment. The Deployer

may use, for example, the JNDI `LinkRef` mechanism to create a symbolic link to the actual JNDI name of the target enterprise bean.

- The Deployer must ensure that the target enterprise bean is type-compatible with the types declared for the EJB reference. This means that the target enterprise bean must be of the type indicated by the use of the `EJB` annotation, by the `ejb-ref-type` element (if specified), and that the business interface and/or home and component interfaces of the target enterprise bean must be Java type-compatible with the interface type of the injection target or the interface types declared in the EJB reference.

- If an EJB annotation includes the `beanName` element or the reference declaration includes the `ejb-link` element, the Deployer should bind the enterprise bean reference to the enterprise bean specified as the target.

15.5.4 Container Provider's Responsibility

The Container Provider must provide the deployment tools that allow the Deployer to perform the tasks described in the previous subsection. The deployment tools provided by the EJB Container Provider must be able to process the information supplied in the `ejb-ref` and `ejb-local-ref` elements in the deployment descriptor.

At the minimum, the tools must be able to:

- Preserve the application assembly information in annotations or in the `ejb-link` elements by binding an EJB reference to the business interface or the home interface of the specified target bean.

- Inform the Deployer of any unresolved EJB references, and allow him or her to resolve an EJB reference by binding it to a specified compatible target bean.

15.6 Web Service References

Web service references allow the Bean Provider to refer to external web services. The web service references are special entries in the enterprise bean's environment. The Deployer binds the web service references to the web service classes or interfaces in the target operational environment.

The specification of web service references and their usage is defined in the Java API for XML Web Services (JAX-WS) [32] and Web Services for Java EE specifications [31].

A web service reference is scoped to the enterprise bean whose definition contains the `WebServiceRef` annotation or whose deployment descriptor declaration contains the `service-ref` element. The EJB specification recommends, but does not require, that all references to web services be organized in the `service` subcontext of the bean's environment (i.e., in the `java:comp/env/service` JNDI context).

15.7 Resource Manager Connection Factory References

A resource manager connection factory is an object that is used to create connections to a resource manager. For example, an object that implements the `javax.sql.DataSource` interface is a resource manager connection factory for `java.sql.Connection` objects that implement connections to a database management system.

This section describes the metadata annotations and deployment descriptor elements that allow the enterprise bean code to refer to resource factories using logical names called *resource manager connection factory references*. The resource manager connection factory references are special entries in the enterprise bean's environment. The Deployer binds the resource manager connection factory references to the actual resource manager connection factories that are configured in the container. Because these resource manager connection factories allow the container to affect resource management, the connections acquired through the resource manager connection factory references are called *managed resources* (e.g., these resource manager connection factories allow the container to implement connection pooling and automatic enlistment of the connection with a transaction).

15.7.1 Bean Provider's Responsibilities

This subsection describes the Bean Provider's view of locating resource factories and defines his or her responsibilities. The first subsection describes annotations for injecting references to resource manager connection factories; the second describes the API for accessing resource manager connection references; and the third describes syntax for declaring the resource manager connection references in a deployment descriptor.

15.7.1.1 Injection of Resource Manager Connection Factory References

A field or a method of an enterprise bean may be annotated with the `Resource` annotation. The name and type of the factory are as described above in Section 15.2.2. The `authenticationType` and `shareable` elements of the `Resource` annotation may be used to control the type of authentication desired for the resource and the shareability of connections acquired from the factory, as described in the following sections.

The following code example illustrates how an enterprise bean uses annotations to declare resource manager connection factory references.

```
//The employee database.
@Resource javax.sql.DataSource employeeAppDB;
...
public void changePhoneNumber(...) {
     ...
     // Invoke factory to obtain a resource. The security
     // principal for the resource is not given, and
     // therefore it will be configured by the Deployer.
     java.sql.Connection con = employeeAppDB.getConnection();
     ...
}
```

15.7.1.2 Programming Interfaces for Resource Manager Connection Factory References

The Bean Provider must use resource manager connection factory references to obtain connections to resources as follows.

- Assign an entry in the enterprise bean's environment to the resource manager connection factory reference. (See subsection 15.7.1.3 for information on how resource manager connection factory references are declared in the deployment descriptor.)

- The EJB specification recommends, but does not require, that all resource manager connection factory references be organized in the subcontexts of the bean's environment, using a different subcontext for each resource manager type. For example, all JDBC™ DataSource references might be declared in the `java:comp/env/jdbc` subcontext, and all JMS connection factories in the `java:comp/env/jms` subcontext. Also, all JavaMail connection factories might be declared in the `java:comp/env/mail` subcontext and all URL connection factories in the `java:comp/env/url` subcontext. Note that resource manager connection factory references declared via annotations will not, by default, appear in any subcontext.

- Lookup the resource manager connection factory object in the enterprise bean's environment using the EJBContext `lookup` method or using the JNDI API.

- Invoke the appropriate method on the resource manager connection factory to obtain a connection to the resource. The factory method is specific to the resource type. It is possible to obtain multiple connections by calling the factory object multiple times.

The Bean Provider can control the shareability of the connections acquired from the resource manager connection factory. By default, connections to a resource manager are shareable across other enterprise beans in the application that use the same resource in the same transaction context. The Bean Provider can specify that connections obtained from a resource manager connection factory reference are not shareable by specifying the value of the `shareable` annotation element to `false` or the `res-sharing-scope` deployment descriptor element to be `Unshareable`. The sharing of connections to a resource manager allows the container to optimize the use of connections and enables the container's use of local transaction optimizations.

The Bean Provider has two choices with respect to dealing with associating a principal with the resource manager access:

- Allow the Deployer to set up principal mapping or resource manager sign-on information. In this case, the enterprise bean code invokes a resource manager connection factory method that has no security-related parameters.

- Sign on to the resource manager from the bean code. In this case, the enterprise bean invokes the appropriate resource manager connection factory method that takes the sign-on information as method parameters.

The Bean Provider uses the `authenticationType` annotation element or the `res-auth` deployment descriptor element to indicate which of the two resource manager authentication approaches is used.

We expect that the first form (i.e., letting the Deployer set up the resource manager sign-on information) will be the approach used by most enterprise beans.

The following code sample illustrates obtaining a JDBC connection when the EJBContext `lookup` method is used.

```
@Resource(name="jdbc/EmployeeAppDB", type=javax.sql.DataSource)
@Stateless public class EmployeeServiceBean
        implements EmployeeService {
    @Resource SessionContext ctx;

    public void changePhoneNumber(...) {
        ...
        // use context lookup to obtain resource manager
        // connection factory
        javax.sql.DataSource ds = (javax.sql.DataSource)
            ctx.lookup("jdbc/EmployeeAppDB");

        // Invoke factory to obtain a connection. The security
        // principal is not given, and therefore
        // it will be configured by the Deployer.
        java.sql.Connection con = ds.getConnection();
        ...
    }
}
```

The following code sample illustrates obtaining a JDBC connection when the JNDI APIs are used directly.

```
@Resource(name="jdbc/EmployeeAppDB", type=javax.sql.DataSource)
@Stateless public class EmployeeServiceBean
        implements EmployeeService {
    EJBContext ejbContext;

    public void changePhoneNumber(...) {
        ...
        // obtain the initial JNDI context
        Context initCtx = new InitialContext();

        // perform JNDI lookup to obtain resource manager
        // connection factory
        javax.sql.DataSource ds = (javax.sql.DataSource)
            initCtx.lookup("java:comp/env/jdbc/EmployeeAppDB");

        // Invoke factory to obtain a connection. The security
        // principal is not given, and therefore
        // it will be configured by the Deployer.
        java.sql.Connection con = ds.getConnection();
        ...
    }
}
```

15.7.1.3 Declaration of Resource Manager Connection Factory References in Deployment

Descriptor

Although a resource manager connection factory reference is an entry in the enterprise bean's environment, the Bean Provider must not use an `env-entry` element to declare it.

Instead, if metadata annotations are not used, the Bean Provider must declare all the resource manager connection factory references in the deployment descriptor using the `resource-ref` elements. This allows the ejb-jar consumer (i.e. Application Assembler or Deployer) to discover all the resource manager connection factory references used by an enterprise bean. Deployment descriptor entries may also be used to specify injection of a resource manager connection factor reference into a bean.

Each `resource-ref` element describes a single resource manager connection factory reference. The `resource-ref` element consists of the `description` element; the mandatory `res-ref-name element`; and the optional `res-type`, `res-auth` and `res-sharing-scope` elements. The `res-ref-name` element contains the name of the environment entry used in the enterprise bean's code. The name of the environment entry is relative to the `java:comp/env` context (e.g., the name should be `jdbc/EmployeeAppDB` rather than `java:comp/env/jdbc/EmployeeAppDB`). The `res-type` element contains the Java type of the resource manager connection factory that the enterprise bean code expects. The `res-type` element is optional if an injection target is specified for the resource; in this case, the `res-type` defaults to the type of the injection target. The `res-auth` element indicates whether the enterprise bean code performs resource manager sign-on programmatically, or whether the container signs on to the resource manager using the principal mapping information supplied by the Deployer. The Bean Provider indicates the sign-on responsibility by setting the value of the `res-auth` element to `Application` or `Container`. If the `res-auth` element is not specified, `Container` sign-on is assumed. The `res-sharing-scope` element indicates whether connections to the resource manager obtained through the given resource manager connection factory reference are to be shared or whether connections are unshareable. The value of the `res-sharing-scope` element is `Shareable` or `Unshareable`. If the `res-sharing-scope` element is not specified, connections are assumed to be shareable.

A resource manager connection factory reference is scoped to the enterprise bean whose declaration contains the `resource-ref` element. This means that the resource manager connection factory reference is not accessible from other enterprise beans at runtime, and that other enterprise beans may define `resource-ref` elements with the same `res-ref-name` without causing a name conflict.

The type declaration allows the Deployer to identify the type of the resource manager connection factory.

Note that the indicated type is the Java type of the resource factory, not the Java type of the resource.

The following example is the declaration of resource manager connection factory references used by the `EmployeeService` enterprise bean illustrated in the previous subsection.

```
...
<enterprise-beans>
     <session>
          ...
          <ejb-name>EmployeeService</ejb-name>
          <ejb-class>com.wombat.empl.EmployeeServiceBean</ejb-class>
          ...
          <resource-ref>
               <description>
                    A data source for the database in which
                    the EmployeeService enterprise bean will
                    record a log of all transactions.
               </description>
               <res-ref-name>jdbc/EmployeeAppDB</res-ref-name>
               <res-type>javax.sql.DataSource</res-type>
               <res-auth>Container</res-auth>
               <res-sharing-scope>Shareable</res-sharing-scope>
          </resource-ref>
          ...
     </session>
</enterprise-beans>
...
```

The following example illustrates the declaration of the JMS resource manager connection factory references used by the example on page 293.

```
...
<enterprise-beans>
     <session>
          ...
          <resource-ref>
               <description>
                    A queue connection factory used by the
                    MySession enterprise bean to send
                    notifications.
               </description>
               <res-ref-name>jms/qConnFactory</res-ref-name>
               <res-type>javax.jms.QueueConnectionFactory</res-type>
               <res-auth>Container</res-auth>
               <res-sharing-scope>Unshareable</res-sharing-scope>
          </resource-ref>
          ...
     </session>
</enterprise-beans>
...
```

15.7.1.4 Standard Resource Manager Connection Factory Types

The Bean Provider must use the `javax.sql.DataSource` resource manager connection factory type for obtaining JDBC connections, and the `javax.jms.ConnectionFactory`, `javax.jms.QueueConnectionFactory`, or `javax.jms.TopicConnectionFactory` for obtaining JMS connections.

The Bean Provider must use the `javax.mail.Session` resource manager connection factory type for obtaining JavaMail connections, and the `java.net.URL` resource manager connection factory type for obtaining URL connections.

It is recommended that the Bean Provider names JDBC data sources in the `java:comp/env/jdbc` subcontext, and JMS connection factories in the `java:comp/env/jms` subcontext. It is also recommended that the Bean Provider name all JavaMail connection factories in the `java:comp/env/mail` subcontext, and all URL connection factories in the `java:comp/env/url` subcontext. Note that resource manager connection factory references declared via annotations will not, by default, appear in any subcontext.

The Connector architecture [15] allows an enterprise bean to use the API described in this section to obtain resource objects that provide access to additional back-end systems.

15.7.2 Deployer's Responsibility

The Deployer uses deployment tools to bind the resource manager connection factory references to the actual resource factories configured in the target operational environment.

The Deployer must perform the following tasks for each resource manager connection factory reference declared in the metadata annotations or deployment descriptor:

- Bind the resource manager connection factory reference to a resource manager connection factory that exists in the operational environment. The Deployer may use, for example, the JNDI `LinkRef` mechanism to create a symbolic link to the actual JNDI name of the resource manager connection factory. The resource manager connection factory type must be compatible with the type declared in the source code or in the `res-type` element.

- Provide any additional configuration information that the resource manager needs for opening and managing the resource. The configuration mechanism is resource-manager specific, and is beyond the scope of this specification.

- If the value of the `Resource` annotation `authenticationType` element is `AuthenticationType.CONTAINER` or the deployment descriptor `res-auth` element is `Container`, the Deployer is responsible for configuring the sign-on information for the resource manager. This is performed in a manner specific to the EJB container and resource manager; it is beyond the scope of this specification.

For example, if principals must be mapped from the security domain and principal realm used at the enterprise beans application level to the security domain and principal realm of the resource manager, the Deployer or System Administrator must define the mapping. The mapping is performed in a manner specific to the EJB container and resource manager; it is beyond the scope of the current EJB specification.

15.7.3 Container Provider Responsibility

The EJB Container Provider is responsible for the following:

- Provide the deployment tools that allow the Deployer to perform the tasks described in the previous subsection.

- Provide the implementation of the resource manager connection factory classes for the resource managers that are configured with the EJB container.

- If the Bean Provider sets the `authenticationType` element of the `Resource` annotation to `AuthenticationType.APPLICATION` or the `res-auth` deployment descriptor entry for a resource manager connection factory reference to `Application`, the container must allow the bean to perform explicit programmatic sign-on using the resource manager's API.

- If the Bean Provider sets the `shareable` element of the `Resource` annotation to `false` or sets the `res-sharing-scope` deployment descriptor entry for a resource manager connection factory reference to `Unshareable`, the container must not attempt to share the connections obtained from the resource manager connection factory *reference*[74]. If the Bean Provider sets the `res-sharing-scope` of a resource manager connection factory reference to `Shareable` or does not specify `res-sharing-scope`, the container must share the connections obtained from the resource manager connection factory according to the requirements defined in [12].

- The container must provide tools that allow the Deployer to set up resource manager sign-on information for the resource manager references whose annotation element `authenticationType` is set to `AuthenticationType.CONTAINER` or whose `res-auth` deployment descriptor element element is set to `Container`. The minimum requirement is that the Deployer must be able to specify the user/password information for each resource manager connection factory reference declared by the enterprise bean, and the container must be able to use the user/password combination for user authentication when obtaining a connection to the resource by invoking the resource manager connection factory.

Although not required by the EJB specification, we expect that containers will support some form of a single sign-on mechanism that spans the application server and the resource managers. The container will allow the Deployer to set up the resource managers such that the EJB caller principal can be propagated (directly or through principal mapping) to a resource manager, if required by the application.

While not required by the EJB specification, most EJB container providers also provide the following features:

- A tool to allow the System Administrator to add, remove, and configure a resource manager for the EJB server.

- A mechanism to pool connections to the resources for the enterprise beans and otherwise manage the use of resources by the container. The pooling must be transparent to the enterprise beans.

[74] Connections obtained from the same resource manager connection factory through a different resource manager connection factory reference may be shareable.

15.7.4 System Administrator's Responsibility

The System Administrator is typically responsible for the following:

- Add, remove, and configure resource managers in the EJB server environment.

In some scenarios, these tasks can be performed by the Deployer.

15.8 Resource Environment References

This section describes the programming and deployment descriptor interfaces that allow the Bean Provider to refer to administered objects that are associated with resources (e.g., a Connector CCI `InteractionSpec` instance) by using "logical" names called *resource environment references*. Resource environment references are special entries in the enterprise bean's environment. The Deployer binds the resource environment references to administered objects in the target operational environment.

15.8.1 Bean Provider's Responsibilities

This subsection describes the Bean Provider's view and responsibilities with respect to resource environment references.

15.8.1.1 Injection of Resource Environment References

A field or a method of a bean may be annotated with the `Resource` annotation to request injection of a resource environment reference. The name and type of the resource environment reference are as described in Section 15.2.2. The `authenticationType` and `shareable` elements of the `Resource` annotation must not be specified; resource environment entries are not shareable and do not require authentication. The use of the `Resource` annotation to declare a resource environment reference differs from the use of the `Resource` annotation to declare simple environment references only in that the type of a resource environment reference is not one of the Java language types used for simple environment references.

15.8.1.2 Resource Environment Reference Programming Interfaces

The Bean Provider must use resource environment references to locate administered objects that are associated with resources, as follows.

- Assign an entry in the enterprise bean's environment to the reference. (See subsection 15.8.1.3 for information on how resource environment references are declared in the deployment descriptor.)

- The EJB specification recommends, but does not require, that all resource environment references be organized in the appropriate subcontext of the bean's environment for the resource

type. Note that the resource environment references declared via annotations will not, by default, appear in any subcontext.

- Look up the administered object in the enterprise bean's environment using the EJBContext `lookup` method or the JNDI API.

15.8.1.3 Declaration of Resource Environment References in Deployment Descriptor

Although the resource environment reference is an entry in the enterprise bean's environment, the Bean Provider must not use a `env-entry` element to declare it. Instead, the Bean Provider must declare all references to administered objects associated with resources using either annotations in the bean's source code or the `resource-env-ref` elements of the deployment descriptor. This allows the ejb-jar consumer to discover all the resource environment references used by the enterprise bean. Deployment descriptor entries may also be used to specify injection of a resource environment reference into a bean.

Each `resource-env-ref` element describes the requirements that the referencing enterprise bean has for the referenced administered object. The `resource-env-ref` element contains optional `description` and `resource-env-ref-type` elements, and the mandatory `resource-env-ref-name` element. The `resource-env-ref-type` element is optional if an injection target is specified for the resource environment reference; in this case the `resource-env-ref-type` defaults to the type of the injection target.

The `resource-env-ref-name` element specifies the resource environment reference name: its value is the environment entry name used in the enterprise bean code. The name of the environment entry is relative to the `java:comp/env` context. The `resource-env-ref-type` element specifies the expected type of the referenced object.

A resource environment reference is scoped to the enterprise bean whose declaration contains the `resource-env-ref` element. This means that the resource environment reference is not accessible to other enterprise beans at runtime, and that other enterprise beans may define `resource-env-ref` elements with the same `resource-env-ref-name` without causing a name conflict.

15.8.2 Deployer's Responsibility

The Deployer is responsible for the following:

- The Deployer must ensure that all the declared resource environment references are bound to administered objects that exist in the operational environment. The Deployer may use, for example, the JNDI `LinkRef` mechanism to create a symbolic link to the actual JNDI name of the target object.

- The Deployer must ensure that the target object is type-compatible with the type declared for the resource environment reference. This means that the target object must be of the type indicated in the `Resource` annotation or the `resource-env-ref-type` element.

15.8.3 Container Provider's Responsibility

The Container Provider must provide the deployment tools that allow the Deployer to perform the tasks described in the previous subsection. The deployment tools provided by the EJB Container Provider must be able to process the information supplied in the class file annotations and `resource-env-ref` elements in the deployment descriptor.

At the minimum, the tools must be able to inform the Deployer of any unresolved resource environment references, and allow him or her to resolve a resource environment reference by binding it to a specified compatible target object in the environment.

15.9 Message Destination References

This section describes the programming and deployment descriptor interfaces that allow the Bean Provider to refer to message destination objects by using "logical" names called *message destination references*. Message destination references are special entries in the enterprise bean's environment. The Deployer binds the message destination references to administered message destinations in the target operational environment.

15.9.1 Bean Provider's Responsibilities

This subsection describes the Bean Provider's view and responsibilities with respect to message destination references.

15.9.1.1 Injection of Message Destination References

A field or a method of a bean may be annotated with the `Resource` annotation to request injection of a message destination reference. The name and type of the resource environment reference are as described in Section 15.2.2. The `authenticationType` and `shareable` elements of the `Resource` annotation must not be specified.

Note that when using the `Resource` annotation to declare a message destination reference it is not possible to link the reference to other references to the same message destination, or to specify whether the destination is used to produce or consume messages. The deployment descriptor entries described in Section 15.9.1.3 provide a way to associate multiple message destination references with a single message destination and to specify whether each message destination reference is used to produce, consume, or both produce and consume messsages, so that the entire message flow of an application may be specified. The Application Assembler may use these message destination links to link together message destination references that have been declared using the `Resource` annotation. A message destination reference declared via the `Resource` annotation is assumed to be used to both produce and consume messages; this default may be overridden using a deployment descriptor entry.

The following example illustrates how an enterprise bean uses the `Resource` annotation to request injection of a message destination reference.

```
@Resource javax.jms.Queue stockQueue;
```

15.9.1.2 Message Destination Reference Programming Interfaces

The Bean Provider uses message destination references to locate message destinations, as follows.

- Assign an entry in the enterprise bean's environment to the reference. (See subsection 15.9.1.3 for information on how message destination references are declared in the deployment descriptor.)
- The EJB specification recommends, but does not require, that all message destination references be organized in the appropriate subcontext of the bean's environment for the messaging resource type (e.g. in the `java:comp/env/jms` JNDI context for JMS Destinations). Note that message destination references declared via annotations will not, by default, appear in any subcontext.
- Look up the destination in the enterprise bean's environment using the EJBContext `lookup` method or the JNDI APIs.

The following example illustrates how an enterprise bean uses a message destination reference to locate a JMS Destination.

```
@Resource(name="jms/StockQueue", type=javax.jms.Queue)
@Stateless public class StockServiceBean implements StockService {

     @Resource SessionContext ctx;

     public void processStockInfo(...) {
          ...
          // Look up the JMS StockQueue in the environment.
          Object result = ctx.lookup("jms/StockQueue");

          // Convert the result to the proper type.
          javax.jms.Queue queue = (javax.jms.Queue)result;
     }
}
```

In the example, the Bean Provider of the `StockServiceBean` enterprise bean has assigned the environment entry `jms/StockQueue` as the message destination reference name to refer to a JMS queue.

If the JNDI APIs were used directly, the example would be as follows.

```
@Resource(name=”jms/StockQueue”, type=javax.jms.Queue)
@Stateless public class StockServiceBean implements StockService {

     public void processStockInfo(...) {
          ...
          // Obtain the default initial JNDI context.
          Context initCtx = new InitialContext();

          // Look up the JMS StockQueue in the environment.
          Object result = initCtx.lookup(
               "java:comp/env/jms/StockQueue");

          // Convert the result to the proper type.
          javax.jms.Queue queue = (javax.jms.Queue)result;
     }
}
```

15.9.1.3 Declaration of Message Destination References in Deployment Descriptor

Although the message destination reference is an entry in the enterprise bean's environment, the Bean Provider must not use a `env-entry` element to declare it. Instead, the Bean Provider should declare all references to message destinations using either the `Resource` annotation in the bean's code or the the `message-destination-ref` elements of the deployment descriptor. This allows the ejb-jar consumer to discover all the message destination references used by the enterprise bean. Deployment descriptor entries may also be used to specify injection of a message destination reference into a bean.

Each `message-destination-ref` element describes the requirements that the referencing enterprise bean has for the referenced destination. The `message-destination-ref` element contains optional `description`, `message-destination-type`, and `message-destination-usage` elements, and the mandatory `message-destination-ref-name` element.

The `message-destination-ref-name` element specifies the message destination reference name: its value is the environment entry name used in the enterprise bean code. The name of the message destination reference is relative to the `java:comp/env` context (e.g., the name should be `jms/StockQueue` rather than `java:comp/env/jms/StockQueue`). The `message-destination-type` element specifies the expected type of the referenced destination. For example, in the case of a JMS Destination, its value might be `javax.jms.Queue`. The `message-destination-type` element is optional if an injection target is specified for the message destination reference; in this case the `message-destination-type` defaults to the type of the injection target. The `message-destination-usage` element specifies whether messages are consumed from the message destination, produced for the destination, or both. If the `message-destination-usage` element is not specified, messages are assumed to be both consumed and produced.

A message destination reference is scoped to the enterprise bean whose declaration contains the `message-destination-ref` element. This means that the message destination reference is not accessible to other enterprise beans at runtime, and that other enterprise beans may define `message-destination-ref` elements with the same `message-destination-ref-name` without causing a name conflict.

The following example illustrates the declaration of message destination references in the deployment descriptor.

```
...
<message-destination-ref>
     <description>
     This is a reference to a JMS queue used in processing Stock info
     </description>
     <message-destination-ref-name>
          jms/StockInfo
     </message-destination-ref-name>
     <message-destination-type>
          javax.jms.Queue
     </message-destination-type>
     <message-destination-usage>
          Produces<
     /message-destination-usage>
</message-destination-ref>
...
```

15.9.2 Application Assembler's Responsibilities

By means of linking message consumers and producers to one or more common logical destinations specified in the deployment descriptor, the Application Assembler can specify the flow of messages within an application. The Application Assembler uses the `message-destination` element, the `message-destination-link` element of the `message-destination-ref` element, and the `message-destination-link` element of the `message-driven` element to link message destination references to a common logical destination.

The Application Assembler specifies the link between message consumers and producers as follows:

- The Application Assembler uses the `message-destination` element to specify a logical message destination within the application. The `message-destination` element defines a `message-destination-name`, which is used for the purpose of linking.

- The Application Assembler uses the `message-destination-link` element of the `message-destination-ref` element of an enterprise bean that produces messages to link it to the target destination. The value of the `message-destination-link` element is the name of the target destination, as defined in the `message-destination-name` element of the `message-destination` element. The `message-destination` element can be in any module in the same Java EE application as the referencing component. The Application Assembler uses the `message-destination-usage` element of the `message-destination-ref` element to indicate that the referencing enterprise bean produces messages to the referenced destination.

- If the consumer of messages from the common destination is a message-driven bean, the Application Assembler uses the `message-destination-link` element of the `message-driven` element to reference the logical destination. If the Application Assembler links a message-driven bean to its source destination, he or she should use the `message-destination-type` element of the `message-driven` element to specify the expected destination type.

- If an enterprise bean is otherwise a message consumer, the Application Assembler uses the `message-destination-link` element of the `message-destination-ref` element of the enterprise bean that consumes messages to link to the common destination. In the latter case, the Application Assembler uses the `message-destination-usage` element of the `message-destination-ref` element to indicate that the enterprise bean consumes messages from the referenced destination.

- To avoid the need to rename message destinations to have unique names within an entire Java EE application, the Application Assembler may use the following syntax in the `message-destination-link` element of the referencing application component. The Application Assembler specifies the path name of the ejb-jar file containing the referenced message destination and appends the `message-destination-name` of the target destination separated from the path name by `#`. The path name is relative to the referencing application component jar file. In this manner, multiple destinations with the same `message-destination-name` may be uniquely identified.

- When linking message destinations, the Application Assembler must ensure that the consumers and producers for the destination require a message destination of the same or compatible type, as determined by the messaging system.

The following example illustrates the use of message destination linking in the deployment descriptor.

```
...
<enterprise-beans>
<session>
     ...
     <ejb-name>EmployeeService</ejb-name>
     <ejb-class>com.wombat.empl.EmployeeServiceBean</ejb-class>
     ...
     <message-destination-ref>
          <message-destination-ref-name>
               jms/EmployeeReimbursements
          </message-destination-ref-name>
          <message-destination-type>
               javax.jms.Queue
          </message-destination-type>
          <message-destination-usage>
               Produces
          </message-destination-usage>
          <message-destination-link>
               ExpenseProcessingQueue
          </message-destination-link>
     </message-destination-ref>
</session>
...

<message-driven>
     <ejb-name>ExpenseProcessing</ejb-name>
     <ejb-class>com.wombat.empl.ExpenseProcessingBean</ejb-class>
     <messaging-type>javax.jms.MessageListener</messaging-type>
     ...
     <message-destination-type>
          javax.jms.Queue
     </message-destination-type>
     <message-destination-link>
          ExpenseProcessingQueue
     </message-destination-link>
     ...
</message-driven>
     ...
</enterprise-beans>
...
<assembly-descriptor>
     ...
     <message-destination>
          <message-destination-name>
               ExpenseProcessingQueue
          </message-destination-name>
     </message-destination>
     ...
</assembly-descriptor>
```

The Application Assembler uses the `message-destination-link` element to indicate that the message destination reference `EmployeeReimbursement` declared in the `EmployeeService` enterprise bean is linked to the `ExpenseProcessing` message-driven bean by means of the commen destination `ExpenseProcessingQueue`.

The following example illustrates using the `message-destination-link` element to indicate an enterprise bean reference to the `ExpenseProcessingQueue` that is in the same Java EE application unit but in a different ejb-jar file.

```
<session>
     ...
     <ejb-name>EmployeeService</ejb-name>
     <ejb-class>com.wombat.empl.EmployeeServiceBean</ejb-class>
     ...
     <message-destination-ref>
          <message-destination-ref-name>
               jms/EmployeeReimbursements
          </message-destination-ref-name>
          <message-destination-type>
               javax.jms.Queue
          </message-destination-type>
          <message-destination-usage>
               Produces
          </message-destination-usage>
          <message-destination-link>
               finance.jar#ExpenseProcessingQueue
          </message-destination-link>
     </message-destination-ref>
</session>
```

15.9.3 Deployer's Responsibility

The Deployer is responsible for the following:

- The Deployer must ensure that all the declared message destination references are bound to destination objects that exist in the operational environment. The Deployer may use, for example, the JNDI `LinkRef` mechanism to create a symbolic link to the actual JNDI name of the target object.
- The Deployer must ensure that the target object is type-compatible with the type declared for the message destination reference.
- The Deployer must observe the message destination links specified by the Application Assembler.

15.9.4 Container Provider's Responsibility

The Container Provider must provide the deployment tools that allow the Deployer to perform the tasks described in the previous subsection. The deployment tools provided by the EJB Container Provider must be able to process the information supplied in the `message-destination-ref` and `message-destination-link` elements in the deployment descriptor.

The tools must be able to inform the Deployer of the message flow between consumers and producers sharing common message destinations. The tools must also be able to inform the Deployer of any unresolved message destination references, and allow him or her to resolve a message destination reference by binding it to a specified compatible target object in the environment.

15.10 Persistence Unit References

This section describes the metadata annotations and deployment descriptor elements that allow the enterprise bean code to refer to the entity manager factory for a persistence unit using a logical name called a *persistence unit reference*. Persistence unit references are special entries in the enterprise bean's environment. The Deployer binds the persistence unit references to entity manager factories that are configured in accordance with the `persistence.xml` specification for the persistence unit, as described in the document "*Java Persistence*" of this specification [2].

15.10.1 Bean Provider's Responsibilities

This subsection describes the Bean Provider's view of locating the entity manager factory for a persistence unit and defines his or her responsibilities. The first subsection describes annotations for injecting references to an entity manager factory for a persistence unit; the second describes the API for accessing an entity manager factory using a persistence unit reference; and the third describes syntax for declaring persistence unit references in a deployment descriptor.

15.10.1.1 Injection of Persistence Unit References

A field or a method of an enterprise bean may be annotated with the `PersistenceUnit` annotation. The `name` element specifies the name under which the entity manager factory for the referenced persistence unit may be located in the JNDI naming context. The `unitName` element specifies the name of the persistence unit as declared in the `persistence.xml` file that defines the persistence unit, and is required if multiple persistence units exist in the scope of the referencing component, as described in [2].

The following code example illustrates how an enterprise bean uses annotations to declare persistence unit references.

```
@PersistenceUnit
EntityManagerFactory emf;
```

If multiple persistence units exist in the scope of the referencing component, the `unitName` element must be specified:

```
@PersistenceUnit(unitName="InventoryManagement")
EntityManagerFactory emf;
```

15.10.1.2 Programming Interfaces for Persistence Unit References

The Bean Provider must use persistence unit references to obtain references to entity manager factories as follows.

- Assign an entry in the enterprise bean's environment to the persistence unit reference. (See subsection 15.10.1.3 for information on how persistence unit references are declared in the deployment descriptor.)
- The EJB specification recommends, but does not require, that all persistence unit references be organized in the `java:comp/env/persistence` subcontexts of the bean's environment.

- Lookup the entity manager factory for the persistence unit in the enterprise bean's environment using the EJBContext `lookup` method or using the JNDI API.
- Invoke the appropriate method on the entity manager factory to obtain an entity manager instance.

The following code sample illustrates obtaining an entity manager factory when the EJBContext `lookup` method is used.

```
@PersistenceUnit(
     name="persistence/InventoryAppDB",
     unitName="InventoryManagement"
)
@Stateless
public class InventoryManagerBean implements InventoryManager {
     @Resource SessionContext ctx;

     public void updateInventory(...) {
          ...
          // use context lookup to obtain entity manager factory
          // for the InventoryManagement persistence unit
          javax.persistence.EntityManagerFactory emf =
               (javax.persistence.EntityManagerFactory)
               ctx.lookup("persistence/InventoryAppDB");

          // Invoke factory to obtain an entity manager.
          javax.persistence.EntityManager em = emf.getEntityManager();
          ...
     }
}
```

The following code sample illustrates obtaining an entity manager factory when the JNDI APIs are used directly.

```
@PersistenceUnit(
     name=”persistence/InventoryAppDB”,
     unitName=InventoryManagement)
@Stateless
public class InventoryManagerBean implements InventoryManager {
     EJBContext ejbContext;
     ...
     public void updateInventory(...) {
          ...
          // obtain the initial JNDI context
          Context initCtx = new InitialContext();

          // perform JNDI lookup to obtain entity manager factory
          // for the persistence unit
          javax.persistence.EntityManagerFactory = (
               javax.persistence.EntityManagerFactory)
               initCtx.lookup("java:comp/env/persistence/InventoryAp-
pDB");

          // Invoke factory to obtain an entity manager.
          javax.persistence.EntityManager em = emf.getEntityManager();
          ...
     }
}
```

15.10.1.3 Declaration of Persistence Unit References in Deployment Descriptor

Although a persistence unit reference is an entry in the enterprise bean’s environment, the Bean Provider must not use an `env-entry` element to declare it.

Instead, if metadata annotations are not used, the Bean Provider must declare all the persistence unit references in the deployment descriptor using the `persistence-unit-ref` elements. This allows the ejb-jar consumer (i.e. Application Assembler or Deployer) to discover all the persistence unit references used by an enterprise bean. Deployment descriptor entries may also be used to specify injection of a persistence unit reference into a bean.

Each `persistence-unit-ref` element describes a single entity manager factory reference for the persistence unit. The `persistence-unit-ref` element consists of the optional `description` and `persistence-unit-name` elements, and the mandatory `persistence-unit-ref-name` element.

The `persistence-unit-ref-name` element contains the name of the environment entry used in the enterprise bean’s code. The name of the environment entry is relative to the `java:comp/env` context (e.g., the name should be `persistence/InventoryAppDB` rather than `java:comp/env/persistence/InventoryAppDB`). The `persistence-unit-name` element is the name of the persistence unit, as specified in the `persistence.xml` file for the persistence unit, and must be specified if multiple persistence units exist in the scope of the referencing component.

The following example is the declaration of a persistence unit reference used by the `InventoryManager` enterprise bean illustrated in the previous subsection.

```
...
<enterprise-beans>
    <session>
        ...
        <ejb-name>InventoryManagerBean</ejb-name>
        <ejb-class>
            com.wombat.empl.InventoryManagerBean
        </ejb-class>
        ...
        <persistence-unit-ref>
            <description>
                Persistence unit for the inventory management
                application.
            </description>
            <persistence-unit-ref-name>
                persistence/InventoryAppDB
            </persistence-unit-ref-name>
            <persistence-unit-name>
                InventoryManagement
            </persistence-unit-name>
        </persistence-unit-ref>
        ...
    </session>
</enterprise-beans>
...
```

15.10.2 Application Assembler's Responsibilities

The Application Assembler can use the `persistence-unit-name` element in the deployment descriptor to disambiguate a reference to a persistence unit. The Application Assembler (or Bean Provider) may use the following syntax in the `persistence-unit-name` element of the referencing application component to avoid the need to rename persistence units to have unique names within a Java EE application. The Application Assembler specifies the path name of the root of the referenced persistence unit and appends the name of the persistence unit separated from the path name by `#`. The path name is relative to the referencing application component jar file. In this manner, multiple persistence units with the same persistence unit name may be uniquely identified when the Application Assembler cannot change persistence unit names.

For example,

```
...
<enterprise-beans>
     <session>
          ...
          <ejb-name>InventoryManagerBean</ejb-name>
          <ejb-class>
               com.wombat.empl.InventoryManagerBean
          </ejb-class>
          ...
          <persistence-unit-ref>
               <description>
                    Persistence unit for the inventory management
                    application.
               </description>
               <persistence-unit-ref-name>
                    persistence/InventoryAppDB
               </persistence-unit-ref-name>
               <persistence-unit-name>
                    ../inventory.jar#InventoryManagement
               </persistence-unit-name>
          </persistence-unit-ref>
          ...
     </session>
</enterprise-beans>
...
```

The Application Assembler uses the `persistence-unit-name` element to link the persistence unit name `InventoryManagement` declared in the `InventoryManagerBean` to the persistence unit named `InventoryManagement` defined in `inventory.jar`.

15.10.3 Deployer's Responsibility

The Deployer uses deployment tools to bind a persistence unit reference to the actual entity manager factory configured for the persistence in the target operational environment.

The Deployer must perform the following tasks for each persistence unit reference declared in the meta-data annotations or deployment descriptor:

- Bind the persistence unit reference to an entity manager factory configured for the persistence unit as specified in the `persistence.xml` file for the persistence unit that exists in the operational environment. The Deployer may use, for example, the JNDI `LinkRef` mechanism to create a symbolic link to the actual JNDI name of the entity manager factory.

- Provide any additional configuration information that the entity manager factory needs for managing the persistence unit, as described in [2].

15.10.4 Container Provider Responsibility

The EJB Container Provider is responsible for the following:

- Provide the deployment tools that allow the Deployer to perform the tasks described in the previous subsection.
- Provide the implementation of the entity manager factory classes for the persistence units that are configured with the EJB container. The implementation of the entity manager factory classes may be provided by the container directory or by the container in conjunction with a third-party persistence provider, as described in [2].

15.10.5 System Administrator's Responsibility

The System Administrator is typically responsible for the following:

- Add, remove, and configure entity manager factories in the EJB server environment.

In some scenarios, these tasks can be performed by the Deployer.

15.11 Persistence Context References

This section describes the metadata annotations and deployment descriptor elements that allow the enterprise bean code to refer to a container-managed entity manager of a specified persistence context type using a logical name called a *persistence context reference*. Persistence context references are special entries in the enterprise bean's environment. The Deployer binds the persistence context references to container-managed entity managers for persistence contexts of the specified type and configured in accordance with their persistence unit, as described in the document "*Java Persistence*" of this specification [2].

15.11.1 Bean Provider's Responsibilities

This subsection describes the Bean Provider's view of locating container-managed entity manager instances and defines his or her responsibilities. The first subsection describes annotations for injecting references to container-managed entity managers for persistence contexts; the second describes the API for accessing references to container-managed entity managers for persistence contexts; and the third describes syntax for declaring these references in a deployment descriptor.

15.11.1.1 Injection of Persistence Context References

A field or a method of an enterprise bean may be annotated with the `PersistenceContext` annotation. The `name` element specifies the name under which a container-managed entity manager for the referenced persistence unit may be located in the JNDI naming context. The `unitName` element specifies the name of the persistence unit as declared in the `persistence.xml` file that defines the persistence unit, and is required if multiple persistence units exist in the scope of the referencing component, as described in [2]. The optional `type` element specifies whether a transaction-scoped or extended persistence context is to be used. If the type is not specified, a transaction-scoped persistence context will be used. References to container-managed entity managers with extended persistence contexts can only be injected into stateful session beans.

The following code example illustrates how an enterprise bean uses annotations to declare persistence context references.

```
@PersistenceContext(type=EXTENDED)
EntityManager em;
```

If multiple persistence units exist in the scope of the referencing component, the `unitName` element must be specified:

```
@PersistenceContext(unitName="InventoryManagement", type=EXTENDED)
EntityManager em;
```

15.11.1.2 Programming Interfaces for Persistence Context References

The Bean Provider must use persistence context references to obtain references to a container-managed entity manager configured for a persistence unit as follows:

- Assign an entry in the enterprise bean's environment to the persistence context reference. (See subsection 15.11.1.3 for information on how persistence context references are declared in the deployment descriptor.)
- The EJB specification recommends, but does not require, that all persistence context references be organized in the `java:comp/env/persistence` subcontexts of the bean's environment.
- Lookup the container-managed entity manager for the persistence unit in the enterprise bean's environment using the EJBContext `lookup` method or using the JNDI API.

The following code sample illustrates obtaining an entity manager for a persistence context when the EJBContext `lookup` method is used.

```
@PersistenceContext(
     name=”persistence/InventoryAppMgr”,
     unitName=”InventoryManagement”
)
@Stateless
public class InventoryManagerBean implements InventoryManager {
     @Resource SessionContext ctx;

     public void updateInventory(...) {
          ...
          // use context lookup to obtain container-managed entity
manager
          javax.persistence.EntityManager em =
               (javax.persistence.EntityManager)
               ctx.lookup("persistence/InventoryAppMgr");
          ...
     }
}
```

The following code sample illustrates obtaining an entity manager when the JNDI APIs are used directly.

```
@PersistenceContext(
     name=”persistence/InventoryAppMgr”,
     unitName=InventoryManagement)
@Stateless
public class InventoryManagerBean implements InventoryManager {
     EJBContext ejbContext;

     public void updateInventory(...) {
          ...
          // obtain the initial JNDI context
          Context initCtx = new InitialContext();

          // perform JNDI lookup to obtain container-managed entity
manager
          javax.persistence.EntityManager = (
               javax.persistence.EntityManager)
               initCtx.lookup("java:comp/env/persistence/InventoryApp-
Mgr");
          ...
     }
}
```

15.11.1.3 Declaration of Persistence Context References in Deployment Descriptor

Although a persistence context reference is an entry in the enterprise bean’s environment, the Bean Provider must not use an `env-entry` element to declare it.

Instead, if metadata annotations are not used, the Bean Provider must declare all the persistence context references in the deployment descriptor using the `persistence-context-ref` elements. This allows the ejb-jar consumer (i.e. Application Assembler or Deployer) to discover all the persistence context references used by an enterprise bean. Deployment descriptor entries may also be used to specify injection of a persistence context reference into a bean.

Each `persistence-context-ref` element describes a single container-managed entity manager reference for the persistence unit. The `persistence-context-ref` element consists of the optional `description`, `persistence-unit-name`, and `persistence-context-type` elements, and the mandatory `persistence-context-ref-name` element.

The `persistence-context-ref-name` element contains the name of the environment entry used in the enterprise bean’s code. The name of the environment entry is relative to the `java:comp/env` context (e.g., the name should be `persistence/InventoryAppMgr` rather than `java:comp/env/persistence/InventoryAppMgr`). The `persistence-unit-name` element is the name of the persistence unit, as specified in the `persistence.xml` file for the persistence unit, and must be specified if multiple persistence units exist in the scope of the referencing component. The `persistence-context-type` element specifies whether a transaction-scoped or extended persistence context is to be used. Its value is either `Transaction` or `Extended`. If the persistence context type is not specified, a transaction-scoped persistence context will be used.

The following example is the declaration of a persistence context reference used by the `Inventory-Manager` enterprise bean illustrated in the previous subsection.

```
...
<enterprise-beans>
     <session>
          ...
          <ejb-name>InventoryManagerBean</ejb-name>
          <ejb-class>
               com.wombat.empl.InventoryManagerBean
          </ejb-class>
          ...
          <persistence-context-ref>
               <description>
                    Persistence context for the inventory management
                    application.
               </description>
               <persistence-context-ref-name>
                    persistence/InventoryAppMgr
               </persistence-context-ref-name>
               <persistence-unit-name>
                    InventoryManagement
               </persistence-unit-name>
          </persistence-context-ref>
          ...
     </session>
</enterprise-beans>
...
```

15.11.2 Application Assembler's Responsibilities

The Application Assembler can use the `persistence-unit-name` element in the deployment descriptor to disambiguate a reference to a persistence unit using the syntax described in section 15.10.2. In this manner, multiple persistence units with the same persistence unit name may be uniquely identified when the Application Assembler cannot change persistence unit names.

For example,

```
...
<enterprise-beans>
     <session>
          ...
          <ejb-name>InventoryManagerBean</ejb-name>
          <ejb-class>
               com.wombat.empl.InventoryManagerBean
          </ejb-class>
          ...
          <persistence-context-ref>
               <description>
                    Persistence context for the inventory management
                    application.
               </description>
               <persistence-context-ref-name>
                    persistence/InventoryAppMgr
               </persistence-context-ref-name>
               <persistence-unit-name>
                    ../inventory.jar#InventoryManagement
               </persistence-unit-name>
          </persistence-context-ref>
          ...
     </session>
</enterprise-beans>
...
```

The Application Assembler uses the `persistence-unit-name` element to link the persistence unit name `InventoryManagement` declared in the `InventoryManagerBean` to the persistence unit named `InventoryManagement` defined in `inventory.jar`.

15.11.3 Deployer's Responsibility

The Deployer uses deployment tools to bind a persistence context reference to the container-managed entity manager for the persistence context of the specified type and configured for the persistence unit in the target operational environment.

The Deployer must perform the following tasks for each persistence context reference declared in the metadata annotations or deployment descriptor:

- Bind the persistence context reference to a container-managed entity manager for a persistence context of the specified type and configured for the persistence unit as specified in the `persistence.xml` file for the persistence unit that exists in the operational environment. The Deployer may use, for example, the JNDI `LinkRef` mechanism to create a symbolic link to the actual JNDI name of the entity manager.

- Provide any additional configuration information that the entity manager factory needs for creating such an entity manager and for managing the persistence unit, as described in [2].

15.11.4 Container Provider Responsibility

The EJB Container Provider is responsible for the following:

- Provide the deployment tools that allow the Deployer to perform the tasks described in the previous subsection.
- Provide the implementation of the entity manager classes for the persistence units that are configured with the EJB container. This implementation may be provided by the container directory or by the container in conjunction with a third-party persistence provider, as described in [2].

15.11.5 System Administrator's Responsibility

The System Administrator is typically responsible for the following:

- Add, remove, and configure entity manager factories in the EJB server environment.

In some scenarios, these tasks can be performed by the Deployer.

15.12 UserTransaction Interface

The container must make the `UserTransaction` interface available to the enterprise beans that are allowed to use this interface (only session and message-driven beans with bean-managed transaction demarcation are allowed to use this interface) either through injection using the `Resource` annotation or in JNDI under the name `java:comp/UserTransaction`, in addition to through the EJBContext interface. The `authenticationType` and `shareable` elements of the `Resource` annotation must not be specified.

The container must not make the `UserTransaction` interface available to the enterprise beans that are not allowed to use this interface. The container should throw `javax.naming.NameNotFoundException` if an instance of an enterprise bean that is not allowed to use the `UserTransaction` interface attempts to look up the interface in JNDI using the JNDI APIs.

The following example illustrates how an enterprise bean acquires and uses a `UserTransaction` object via injection.

```
@Resource UserTransaction tx;
...
public void updateData(...) {
     ...
     // Start a transaction.
     tx.begin();
     ...
     // Perform transactional operations on data.
     ...
     // Commit the transaction.
     tx.commit();
     ...
}
```

The following code example

```
public MySessionBean implements SessionBean {
     ...
     public someMethod()
     {
          Context initCtx = new InitialContext();
          UserTransaction utx = (UserTransaction)initCtx.lookup(
               “java:comp/UserTransaction”);
          utx.begin();
          ...
          utx.commit();
     }
     ...
}
```

is functionally equivalent to

```
public MySessionBean implements SessionBean {
     ...
     SessionContext ctx;
     ...
     public someMethod()
     {
          UserTransaction utx = ctx.getUserTransaction();
          utx.begin();
          ...
          utx.commit();
     }
     ...
}
```

A `UserTransaction` object reference may also be declared in a deployment descriptor in the same way as a resource environment reference. Such a deployment descriptor entry may be used to specify injection of a `UserTransaction` object.

15.12.1 Bean Provider's Responsibility

The Bean Provider is responsible for requesting injection of a `UserTransaction` object using a `Resource` annotation or for using the defined name to lookup the `UserTransaction` object.

15.12.2 Container Provider's Responsibility

The Container Provider is responsible for providing an appropriate `UserTransaction` object as required by this specification.

15.13 ORB References

Enterprise beans that need to make use of the CORBA ORB to perform certain operations can find an appropriate object implementing the ORB interface by requesting injection of an `ORB` object or by looking up the JNDI name `java:comp/ORB`. Any such reference to an `ORB` object is only valid within the bean instance that performed the lookup.

The following example illustrates how an application component acquires and uses an `ORB` object via injection.

```
@Resource ORB orb;

public void method(...) {
     ...
     // Get the POA to use when creating object references.
     POA rootPOA = (POA)orb.resolve_initial_references("RootPOA");
     ...
}
```

The following example illustrates how an enterprise bean acquires and uses an ORB object using a JNDI lookup.

```
@Resource ORB orb;

public void method(...) {
     ...
     // Obtain the default initial JNDI context.
     Context initCtx = new InitialContext();

     // Look up the ORB object.
     ORB orb = (ORB)initCtx.lookup("java:comp/ORB");

     // Get the POA to use when creating object references.
     POA rootPOA = (POA)orb.resolve_initial_references("RootPOA");
     ...
}
```

An `ORB` reference may also be declared in a deployment descriptor in the same way as a resource manager connection factory reference. Such a deployment descriptor entry may be used to specify injection of an `ORB` object.

The ORB instance available under the JNDI name `java:comp/ORB` may always be a shared instance. By default, the `ORB` instance injected into an enterprise bean or declared via a deployment descriptor entry may also be a shared instance. However, the application may set the `shareable` element of the `Resource` annotation to `false`, or may set the `res-sharing-scope` element in the deployment descriptor to `Unshareable`, to request a non-shared `ORB` instance.

15.13.1 Bean Provider's Responsibility

The Bean Provider is responsible for requesting injection of the `ORB` object using the `Resource` annotation, or using the defined name to look up the `ORB` object. If the `shareable` element of the `Resource` annotation is set to `false`, the ORB object injected will not be the shared instance used by other components in the application but instead will be a private `ORB` instance used only by the given component.

15.13.2 Container Provider's Responsibility

The Container Provider is responsible for providing an appropriate `ORB` object as required by this specification.

15.14 TimerService References

The container must make the `TimerService` interface available either through injection using the `Resource` annotation or in JNDI under the name `java:comp/TimerService`, in addition to through the EJBContext interface. The `authenticationType` and `shareable` elements of the `Resource` annotation must not be specified.

A `TimerService` object reference may also be declared in a deployment descriptor in the same way as a resource environment reference. Such a deployment descriptor entry may be used to specify injection of a `TimerService` object.

15.14.1 Bean Provider's Responsibility

The Bean Provider is responsible for requesting injection of a `TimerService` object using a `Resource` annotation, or using the defined name to lookup the `TimerService` object.

15.14.2 Container Provider's Responsibility

The Container Provider is responsible for providing an appropriate `TimerService` object as required by this specification.

15.15 EJBContext References

The container must make a component's `EJBContext` interface available either through injection using the `Resource` annotation or in JNDI under the name `java:comp/EJBContext`. The `authenticationType` and `shareable` elements of the `Resource` annotation must not be specified.

An EJBContext object reference may also be declared in a deployment descriptor in the same way as a resource environment reference. Such a deployment descriptor entry may be used to specify injection of an `EJBContext` object.

15.15.1 Bean Provider's Responsibility

The Bean Provider is responsible for requesting injection of an `EJBContext` object using a `Resource` annotation or using the defined name to lookup the `EJBContext` object.

EJBContext objects accessed through the naming environment are only valid within the bean instance that performed the lookup.

15.15.2 Container Provider's Responsibility

The Container Provider is responsible for providing an appropriate `EJBContext` object to the referencing component. The object returned must be of the appropriate specific type for the bean requesting injection or performing the lookup—that is, the container provider must return an instance of the SessionContext interface to referencing session beans and an instance of the MessageDrivenContext interface to message-driven beans.

15.16 Deprecated EJBContext.getEnvironment Method

The *environment naming context* introduced in EJB 1.1 replaced the EJB 1.0 concept of *environment properties*.

An EJB 1.1 or later compliant container is not required to implement support for the EJB 1.0 style environment properties. If the container does not implement the functionality, it should throw a RuntimeException (or subclass thereof) from the `EJBContext.getEnvironment` method.

If an EJB 1.1 or later compliant container chooses to provide support for the EJB 1.0 style environment properties (so that it can support enterprise beans written to the EJB 1.0 specification), it should implement the support as described below.

When the tools convert the EJB 1.0 deployment descriptor to the EJB 1.1 XML format, they should place the definitions of the environment properties into the `ejb10-properties` subcontext of the environment naming context. The `env-entry` elements should be defined as follows: the `env-entry-name` element contains the name of the environment property, the `env-entry-type` must be `java.lang.String`, and the optional `env-entry-value` contains the environment property value.

For example, an EJB 1.0 enterprise bean with two environment properties `foo` and `bar`, should declare the following `env-entry` elements in its EJB 1.1 format deployment descriptor.

```
...
<env-entry>
     env-entry-name>ejb10-properties/foo</env-entry-name>
     <env-entry-type>java.lang.String</env-entry-type>
</env-entry>
<env-entry>
     <description>bar's description</description>
     <env-entry-name>ejb10-properties/bar</env-entry-name>
     <env-entry-type>java.lang.String</env-entry-type>
     <env-entry-value>bar value</env-entry-value>
</env-entry>
...
```

The container should provide the entries declared in the `ejb10-properties` subcontext to the instances as a `java.util.Properties` object that the instances obtain by invoking the `EJBContext.getEnvironment` method.

The enterprise bean uses the EJB 1.0 API to access the properties, as shown by the following example.

```
public class SomeBean implements SessionBean {
     SessionContext ctx;
     java.util.Properties env;

     public void setSessionContext(SessionContext sc) {
          ctx = sc;
          env = ctx.getEnvironment();
     }

     public someBusinessMethod(...) ... {
          String fooValue = env.getProperty("foo");
          String barValue = env.getProperty("bar");
     }
     ...

}
```

Chapter 16 Security Management

This chapter defines the EJB architecture's support for security management.

16.1 Overview

We set the following goals for the security management in the EJB architecture:

- *Lessen the burden of the application developer (i.e. the Bean Provider) for securing the application by allowing greater coverage from more qualified EJB roles. The EJB Container Provider provides the implementation of the security infrastructure; the Deployer and System Administrator define the security policies.*
- *Allow the security policies to be set by the Application Assembler or Deployer.*
- *Allow the enterprise bean applications to be portable across multiple EJB servers that use different security mechanisms.*

The EJB architecture encourages the Bean Provider to implement the enterprise bean class without hard-coding the security policies and mechanisms into the business methods. In most cases, the enterprise bean's business methods should not contain any security-related logic. This allows the Deployer to configure the security policies for the application in a way that is most appropriate for the operational environment of the enterprise.

To make the Deployer's task easier, the Bean Provider or the Application Assembler (which could be the same party as the Bean Provider) may define *security roles* for an application composed of one or more enterprise beans. A security role is a semantic grouping of permissions that a given type of users of the application must have in order to successfully use the application. The Bean Provider can define declaratively using metadata annotations or the deployment descriptor the *method permissions* for each security role. The Applications Assembler can define, augment, or override the method permissions using the deployment descriptor. A method permission is a permission to invoke a specified group of methods of an enterprise bean's business interface, home interface, component interface, and/or web service endpoint. The security roles defined present a simplified security view of the enterprise beans application to the Deployer—the Deployer's view of the application's security requirements is the small set of security roles rather than a large number of individual methods.

The security principal under which a method invocation is performed is typically that of the component's caller. By specifying a run-as identity, however, it is possible to specify that a different principal be substituted for the execution of the methods of the bean's business interface, home interface, component interface, and/or web service endpoint and any methods of other enterprise beans that the bean may call.

This determines whether the caller principal is propagated from the caller to the callee—that is, whether the called enterprise bean will see the same returned value of the `EJBContext.getCallerPrincipal` as the calling enterprise bean—or whether a security principal that has been assigned to the specified security role will be used for the execution of the bean's methods and will be visible as the caller principal in the bean's callee.

The Bean Provider can use metadata annotations or the deployment descriptor to specify whether the caller's security identity or a run-as security identity should be used for the execution of the bean's methods.

- By default, the caller principal will be propagated as the caller identity. The Bean Provider can use the `RunAs` annotation to specify that a security principal that has been assigned to a specified security role be used instead. See Section 16.3.4.

- If the deployment descriptor is used to specify the security principal, the Bean Provider or the Application Assembler can use the `security-identity` deployment descriptor element to specify the security identity. If the `security-identity` deployment descriptor element is not specified and if a run-as identity has not been specified by the use of the `RunAs` annotation or if `use-caller-identity` is specified as the value of the `security-identity` element, the caller principal is propagated from the caller to the callee. If the `run-as` element is specified, a security principal that has been assigned to the specified security role will be used. The Application Assembler is permitted to override a security identity value set by the Bean Provider.

The Deployer is responsible for assigning principals, or groups of principals, which are defined in the target operational environment, to the security roles defined by the Bean Provider or Application Assembler. The Deployer is also responsible for assigning principals for the run-as identities specified. The Deployer is further responsible for configuring other aspects of the security management of the enterprise beans, such as principal mapping for inter-enterprise bean calls, and principal mapping for resource manager access.

At runtime, a client will be allowed to invoke a business method only if the principal associated with the client call has been assigned by the Deployer to have at least one security role that is allowed to invoke the business method or if the Bean Provider or Application Assembler has specified that security authorization is not to be checked for the method (i.e., that all roles, including any unauthenticated roles, are permitted). See Section 16.3.2.

The Container Provider is responsible for enforcing the security policies at runtime, providing the tools for managing security at runtime, and providing the tools used by the Deployer to manage security during deployment.

Because not all security policies can be expressed declaratively, the EJB architecture provides a simple programmatic interface that the Bean Provider may use to access the security context from the business methods.

The following sections define the responsibilities of the individual EJB roles with respect to security management.

16.2 Bean Provider's Responsibilities

This section defines the Bean Provider's perspective of the EJB architecture support for security, and defines his or her responsibilities. In addition, the Bean Provider may define the security roles for the application, as defined in Section 16.3.

16.2.1 Invocation of Other Enterprise Beans

An enterprise bean business method can invoke another enterprise bean via the other bean's business interface or home or component interface. The EJB architecture provides no programmatic interfaces for the invoking enterprise bean to control the principal passed to the invoked enterprise bean.

The management of caller principals passed on *inter-enterprise* bean invocations (i.e. principal delegation) is set up by the Deployer and System Administrator in a container-specific way. The Bean Provider and Application Assembler should describe all the requirements for the caller's principal management of inter-enterprise bean invocations as part of the description.

16.2.2 Resource Access

Section 15.7 defines the protocol for accessing resource managers, including the requirements for security management.

16.2.3 Access of Underlying OS Resources

The EJB architecture does not define the operating system principal under which enterprise bean methods execute. Therefore, the Bean Provider cannot rely on a specific principal for accessing the underlying OS resources, such as files. (See Subsection 16.6.8 for the reasons behind this rule.)

We believe that most enterprise business applications store information in resource managers such as relational databases rather than in resources at the operating system levels. Therefore, this rule should not affect the portability of most enterprise beans.

16.2.4 Programming Style Recommendations

The Bean Provider should neither implement security mechanisms nor hard-code security policies in the enterprise beans' business methods. Rather, the Bean Provider should rely on the security mechanisms provided by the EJB container.

The Bean Provider can use metadata annotations and/or the deployment descriptor to convey security-related information to the Deployer. The information helps the Deployer to set up the appropriate security policy for the enterprise bean application.

16.2.5 Programmatic Access to Caller's Security Context

Note: In general, security management should be enforced by the container in a manner that is transparent to the enterprise beans' business methods. The security API described in this section should be used only in the less frequent situations in which the enterprise bean business methods need to access the security context information.

The `javax.ejb.EJBContext` interface provides two methods (plus two deprecated methods that were defined in EJB 1.0) that allow the Bean Provider to access security information about the enterprise bean's caller.

```
public interface javax.ejb.EJBContext {
    ...

    //
    // The following two methods allow the EJB class
    // to access security information.
    //
    java.security.Principal getCallerPrincipal();
    boolean isCallerInRole(String roleName);


    //
    // The following two EJB 1.0 methods are deprecated.
    //
    java.security.Identity getCallerIdentity();
    boolean isCallerInRole(java.security.Identity role);

    ...

}
```

The Bean Provider can invoke the `getCallerPrincipal` and `isCallerInRole` methods only in the enterprise bean's business methods as specified in Table 1 on page 77, Table 2 on page 86, Table 3 on page 114, Table 4 on page 195, and Table 7 on page 237. If they are otherwise invoked when no security context exists, they should throw the `java.lang.IllegalStateException` runtime exception.

The `getCallerIdentity()` and `isCallerInRole(Identity role)` methods were deprecated in EJB 1.1. The Bean Provider must use the `getCallerPrincipal()` and `isCallerInRole(String roleName)` methods for new enterprise beans.

An EJB 1.1 or later compliant container may choose to implement the two deprecated methods as follows.

- A container that does not want to provide support for this deprecated method should throw a `RuntimeException` (or subclass of `RuntimeException`) from the `getCallerIdentity` method.
- A container that wants to provide support for the `getCallerIdentity` method should return an instance of a subclass of the `java.security.Identity` abstract class from the method. The `getName` method invoked on the returned object must return the same value that `getCallerPrincipal().getName()` would return.
- A container that does not want to provide support for this deprecated method should throw a `RuntimeException` (or subclass of `RuntimeException`) from the `isCallerInRole(Identity identity)` method.
- A container that wants to implement the `isCallerInRole(Identity identity)` method should implement it as follows:

```
public isCallerInRole(Identity identity) {
    return isCallerInRole(identity.getName());
}
```

16.2.5.1 Use of getCallerPrincipal

The purpose of the `getCallerPrincipal` method is to allow the enterprise bean methods to obtain the current caller principal's name. The methods might, for example, use the name as a key to information in a database.

An enterprise bean can invoke the `getCallerPrincipal` method to obtain a `java.security.Principal` interface representing the current caller. The enterprise bean can then obtain the distinguished name of the caller principal using the `getName` method of the `java.security.Principal` interface. If the security identity has not been established, `getCallerPrincipal` returns the container's representation of the unauthenticated identity.

> *Note that `getCallerPrincipal` returns the principal that represents the caller of the enterprise bean, not the principal that corresponds to the run-as security identity for the bean, if any.*

The meaning of the *current caller*, the Java class that implements the `java.security.Principal` interface, and the realm of the principals returned by the `getCallerPrincipal` method depend on the operational environment and the configuration of the application.

An enterprise may have a complex security infrastructure that includes multiple security domains. The security infrastructure may perform one or more mapping of principals on the path from an EJB caller to the EJB object. For example, an employee accessing his or her company over the Internet may be identified by a userid and password (basic authentication), and the security infrastructure may authenticate the principal and then map the principal to a Kerberos principal that is used on the enterprise's intranet before delivering the method invocation to the EJB object. If the security infrastructure performs principal mapping, the `getCallerPrincipal` method returns the principal that is the result of the mapping, not the original caller principal. (In the previous example, `getCallerPrincipal` would return the Kerberos principal.) The management of the security infrastructure, such as principal mapping, is performed by the System Administrator role; it is beyond the scope of the EJB specification.

The following code sample illustrates the use of the `getCallerPrincipal()` method.

```
@Stateless public class EmployeeServiceBean
        implements EmployeeService{
    @Resource SessionContext ctx;
    @PersistenceContext EntityManager em;

    public void changePhoneNumber(...) {
        ...

        // obtain the caller principal.
        callerPrincipal = ctx.getCallerPrincipal();

        // obtain the caller principal's name.
        callerKey = callerPrincipal.getName();

        // use callerKey as primary key to find EmployeeRecord
        EmployeeRecord myEmployeeRecord =
            em.findByPrimaryKey(EmployeeRecord.class, callerKey);

        // update phone number
        myEmployeeRecord.setPhoneNumber(...);

        ...
    }
}
```

In the previous example, the enterprise bean obtains the principal name of the current caller and uses it as the primary key to locate an `EmployeeRecord` entity. This example assumes that application has been deployed such that the current caller principal contains the primary key used for the identification of employees (e.g., employee number).

16.2.5.2 Use of isCallerInRole

The main purpose of the `isCallerInRole(String roleName)` *method is to allow the Bean Provider to code the security checks that cannot be easily defined declaratively in the deployment descriptor using method permissions. Such a check might impose a role-based limit on a request, or it might depend on information stored in the database.*

The enterprise bean code can use the `isCallerInRole` method to test whether the current caller has been assigned to a given security role. Security roles are defined by the Bean Provider or the Application Assembler (see Subsection 16.3.1), and are assigned to principals or principal groups that exist in the operational environment by the Deployer.

> *Note that* `isCallerInRole(String roleName)` *tests the principal that represents the caller of the enterprise bean, not the principal that corresponds to the run-as security identity for the bean, if any.*

The following code sample illustrates the use of the `isCallerInRole(String roleName)` method.

```
@Stateless public class PayrollBean implements Payroll {
     @Resource SessionContext ctx;

     public void updateEmployeeInfo(EmplInfo info) {

          oldInfo = ... read from database;

          // The salary field can be changed only by callers
          // who have the security role "payroll"
          if (info.salary != oldInfo.salary &&
               !ctx.isCallerInRole("payroll")) {
                    throw new SecurityException(...);
          }
          ...
     }
     ...
}
```

16.2.5.3 Declaration of Security Roles Referenced from the Bean's Code

The Bean Provider is responsible for using the `DeclareRoles` annotation or the `security-role-ref` elements of the deployment descriptor to declare all the security role names used in the enterprise bean code. The `DeclareRoles` annotation is specified on a bean class, where it serves to declare roles that may be tested by calling `isCallerInRole` from within the methods of the annotated class. Declaring the security roles allows the Bean Provider, Application Assembler, or Deployer to link these security role names used in the code to the security roles defined for an assembled application. In the absence of this linking step, any security role name as used in the code will be assumed to correspond to a security role of the same name.

The Bean Provider declares the security roles referenced in the code using the `DeclareRoles` metadata annotation. When declaring the name of a role used as a parameter to the `isCallerInRole(String roleName)` method, the declared name must be the same as the parameter value. The Bean Provider may optionally provide a description of the named security roles in the `description` element of the `DeclareRoles` annotation.

In the following example, the `DeclareRoles` annotation is used to indicate that the enterprise bean `AardvarkPayroll` makes the security check using `isCallerInRole("payroll")` in its business method.

```
@DeclareRoles("payroll")
@Stateless public class PayrollBean implements Payroll {
     @Resource SessionContext ctx;

     public void updateEmployeeInfo(EmplInfo info) {

          oldInfo = ... read from database;

          // The salary field can be changed only by callers
          // who have the security role "payroll"
          if (info.salary != oldInfo.salary &&
               !ctx.isCallerInRole("payroll")) {
                    throw new SecurityException(...);
          }
          ...
     }
     ...
}
```

If the `DeclareRoles` annotation is not used, the Bean Provider must use the `security-role-ref` elements of the deployment descriptor to declare the security roles referenced in the code. The `security-role-ref` elements are defined as follows:

- Declare the name of the security role using the `role-name` element. The name must be the security role name that is used as a parameter to the `isCallerInRole(String roleName)` method.
- Optionally provide a description of the security role in the `description` element.

The following example illustrates how an enterprise bean's references to security roles are declared in the deployment descriptor.

```
...
<enterprise-beans>
     ...
     <session>
          <ejb-name>AardvarkPayroll</ejb-name>
          <ejb-class>com.aardvark.payroll.PayrollBean</ejb-class>
          ...
          <security-role-ref>
               <description>
                    This security role should be assigned to the
                    employees of the payroll department who are
                    allowed to update employees' salaries.
               </description>
               <role-name>payroll</role-name>
          </security-role-ref>
          ...
     </session>
     ...
</enterprise-beans>
...
```

The deployment descriptor above indicates that the enterprise bean `AardvarkPayroll` makes the security check using `isCallerInRole("payroll")` in its business method.

A security role reference, including the name defined by the reference, is scoped to the component whose bean class contains the `DeclareRoles` metadata annotation or whose deployment descriptor element contains the `security-role-ref` deployment descriptor element.

The Bean Provider (or Application Assembler) may also use the `security-role-ref` elements for those references that were declared in annotations and which the Bean Provider wishes to have linked to a `security-role` whose name differs from the reference value. If a security role reference is not linked to a security role in this way, the container must map the reference name to the security role of the same name. See section 16.3.3 for a description of how security role references are linked to security roles.

16.3 Responsibilities of the Bean Provider and/or Application Assembler

The Bean Provider and Application Assembler (which could be the same party as the Bean Provider) may define a *security view* of the enterprise beans contained in the ejb-jar file. Providing the security view is optional for the Bean Provider and Application Assembler.

The main reason for providing the security view of the enterprise beans is to simplify the Deployer's job. In the absence of a security view of an application, the Deployer needs detailed knowledge of the application in order to deploy the application securely. For example, the Deployer would have to know what each business method does to determine which users can call it. The security view defined by the Bean Provider or Application Assembler presents a more consolidated view to the Deployer, allowing the Deployer to be less familiar with the application.

The security view consists of a set of *security roles*. A security role is a semantic grouping of permissions that a given type of users of an application must have in order to successfully use the application.

The Bean Provider or Application Assembler defines *method permissions* for each security role. A method permission is a permission to invoke a specified group of methods of the enterprise beans' business interface, home interface, component interface, and/or web service endpoint.

It is important to keep in mind that the security roles are used to define the logical security view of an application. They should not be confused with the user groups, users, principals, and other concepts that exist in the target enterprise's operational environment.

In special cases, a qualified Deployer may change the definition of the security roles for an application, or completely ignore them and secure the application using a different mechanism that is specific to the operational environment.

16.3.1 Security Roles

The Bean Provider or Application Assembler can define one or more security roles in the bean's metadata annotations or deployment descriptor. The Bean Provider or Application Assembler then assigns groups of methods of the enterprise beans' business, home, and component interfaces, and/or web service endpoints to the security roles to define the security view of the application.

Because the Bean Provider and Application Assembler do not, in general, know the security environment of the operational environment, the security roles are meant to be *logical* roles (or actors), each representing a type of user that should have the same access rights to the application.

The Deployer then assigns user groups and/or user accounts defined in the operational environment to the security roles defined by the Bean Provider and Application Assembler.

Defining the security roles in the metadata annotations and/or deployment descriptor is optional[75]. Their omission means that the Bean Provider and Application Assembler chose not to pass any security deployment related instructions to the Deployer.

If Java language metadata annotations are used, the Bean Provider uses the `DeclareRoles` and `RolesAllowed` annotations to define the security roles. The set of security roles used by the application is taken to be the aggregation of the security roles defined by the security role names used in the `DeclareRoles` and `RolesAllowed` annotations. The Bean Provider may augment the set of security roles defined for the application by annotations in this way by means of the `security-role` deployment descriptor element.

[75] If the Bean Provider and Application Assembler do not define security roles, the Deployer will have to define security roles at deployment time.

If the deployment descriptor is used, the The Bean Provider and/or Application Assembler uses the `security-role` deployment descriptor element as follows:

- Define each security role using a `security-role` element.
- Use the `role-name` element to define the name of the security role.
- Optionally, use the `description` element to provide a description of a security role.

The following example illustrates security roles definition in a deployment descriptor.

```
     ...
<assembly-descriptor>
     <security-role>
          <description>
               This role includes the employees of the
               enterprise who are allowed to access the
               employee self-service application. This role
               is allowed only to access his/her own
               information.
          </description>
          <role-name>employee</role-name>
     </security-role>

     <security-role>
          <description>
               This role includes the employees of the human
               resources department. The role is allowed to
               view and update all employee records.
          </description>
          <role-name>hr-department</role-name>
     </security-role>

     <security-role>
          <description>
               This role includes the employees of the payroll
               department. The role is allowed to view and
               update the payroll entry for any employee.
          </description>
          <role-name>payroll-department</role-name>
     </security-role>

     <security-role>
          <description>
               This role should be assigned to the personnel
               authorized to perform administrative functions
               for the employee self-service application.
               This role does not have direct access to
               sensitive employee and payroll information.
          </description>
          <role-name>admin</role-name>
     </security-role>
     ...
</assembly-descriptor>
```

16.3.2 Method Permissions

If the Bean Provider and/or Application Assembler have defined security roles for the enterprise beans in the ejb-jar file, they can also specify the methods of the business, home, and component interfaces, and/or web service endpoints that each security role is allowed to invoke.

Metadata annotations and/or the deployment descriptor can be used for this purpose.

Method permissions are defined as a binary relation from the set of security roles to the set of methods of the business interfaces, home interfaces, component interfaces, and/or web service endpoints of session and entity beans, including all their superinterfaces (including the methods of the `EJBHome` and `EJBObject` interfaces and/or `EJBLocalHome` and `EJBLocalObject` interfaces). The method permissions relation includes the pair (*R*, *M*) if and only if the security role *R* is allowed to invoke the method *M*.

16.3.2.1 Specification of Method Permissions with Metadata Annotations

The following is the description of the rules for the specification of method permissions using Java language metadata annotations.

The method permissions for the methods of a bean class may be specified on the class, the business methods of the class, or both.

The `RolesAllowed`, `PermitAll`, and `DenyAll` annotations are used to specify method permissions. The value of the `RolesAllowed` annotation is a list of security role names to be mapped to the security roles that are permitted to execute the specified method(s). The `PermitAll` annotation specifies that all security roles are permitted to execute the specified method(s). The `DenyAll` annotation specifies that no security roles are permitted to execute the specified method(s).

Specifying the `RolesAllowed` or `PermitAll` annotation on the bean class means that it applies to all applicable business methods of the class.

Method permissions may be specified on a method of the bean class to override the method permissions value specified on the bean class.

If the bean class has superclasses, the following additional rules apply.

- A method permissions value specified on a superclass *S* applies to the business methods defined by *S*.
- A method permissions value may be specified on a business method *M* defined by class *S* to override for method *M* the method permissions value explicitly or implicitly specified on the class *S*.
- If a method *M* of class *S* overrides a business method defined by a superclass of *S*, the method permissions value of *M* is determined by the above rules as applied to class *S*.

Example:

```
@RolesAllowed(“admin”)
public class SomeClass {
     public void aMethod () {...}
     public void bMethod () {...}
     ...
}

@Stateless public class MyBean implements A extends SomeClass {

     @RolesAllowed(“HR”)
     public void aMethod () {...}

     public void cMethod () {...}
  ...
}
```

Assuming `aMethod`, `bMethod`, `cMethod` are methods of business interface `A`, the method permissions values of methods `aMethod` and `bMethod` are `RolesAllowed(“HR”)` and `RolesAllowed(“admin”)` respectively. The method permissions for method `cMethod` have not been specified (see Section 16.3.2.2) .

16.3.2.2 Specification of Method Permissions in the Deployment Descriptor

The Bean Provider may use the deployment descriptor as an alternative to metadata annotations to specify the method permissions (or as a means to supplement or override metadata annotations for method permission values). The application assembler is permitted to override the method permission values using the bean’s deployment descriptor.

Any values explicitly specified in the deployment descriptor override any values specified in annotations. If a value for a method has not be specified in the deployment descriptor, and a value has been specified for that method by means of the use of annotations, the value specified in annotations will apply. The granularity of overriding is on the per-method basis.

The Bean Provider or Application Assembler defines the method permissions relation in the deployment descriptor using the `method-permission` elements as follows.

- Each `method-permission` element includes a list of one or more security roles and a list of one or more methods. All the listed security roles are allowed to invoke all the listed methods. Each security role in the list is identified by the `role-name` element, and each method (or a set of methods, as described below) is identified by the `method` element. An optional description can be associated with a `method-permission` element using the `description` element.

- The method permissions relation is defined as the union of all the method permissions defined in the individual `method-permission` elements.

- A security role or a method may appear in multiple `method-permission` elements.

The Bean Provider or Application Assembler can indicate that all roles are permitted to execute one or more specified methods (i.e., the methods should not be "checked" for authorization prior to invocation by the container). The `unchecked` element is used instead of a role name in the `method-permission` element to indicate that all roles are permitted.

If the method permission relation specifies both the `unchecked` element for a given method and one or more security roles, all roles are permitted for the specified methods.

The `exclude-list` element can be used to indicate the set of methods that should not be called. The Deployer should configure the enterprise bean's security such that no access is permitted to any method contained in the `exclude-list`.

If a given method is specified both in the `exclude-list` element and in the method permission relation, the Deployer should configure the enterprise bean's security such that no access is permitted to the method.

The `method` element uses the `ejb-name`, `method-name`, and `method-params` elements to denote one or more methods of an enterprise bean's business , home, and component interface, and/or web service endpoint. There are three legal styles for composing the `method` element:

Style 1:

```
<method>
      <ejb-name>EJBNAME</ejb-name>
      <method-name>*</method-name>
</method>
```

This style is used for referring to all of the methods of the business, home, and component interfaces, and web service endpoint of a specified enterprise bean.

Style 2:

```
<method>
      <ejb-name>EJBNAME</ejb-name>
      <method-name>METHOD</method-name>
</method>
```

This style is used for referring to a specified method of the business, home, or component interface, or web service endpoint of the specified enterprise bean. If there are multiple methods with the same overloaded name, this style refers to all of the overloaded methods.

Style 3:

```
<method>
     <ejb-name>EJBNAME</ejb-name>
     <method-name>METHOD</method-name>
     <method-params>
          <method-param>PARAMETER_1</method-param>
          ...
          <method-param>PARAMETER_N</method-param>
     </method-params>
</method>
```

This style is used to refer to a specified method within a set of methods with an overloaded name. The method must be defined in the specified enterprise bean's business, home, or component interface, or web service endpoint. If there are multiple methods with the same overloaded name, however, this style refers to all of the overloaded methods.

The optional `method-intf` element can be used to differentiate between methods with the same name and signature that are multiply defined across the business, component, or home interfaces, and/or web service endpoint. If the same method is a method of both the local business interface and local component interface, the same method permission values apply to the method for both interfaces. Likewise, if the same method is a method of both the remote business interface and remote component interface, the same method permission values apply to the method for both interfaces.

The following example illustrates how security roles are assigned method permissions in the deployment descriptor:

```
...
<method-permission>
     <role-name>employee</role-name>
     <method>
          <ejb-name>EmployeeService</ejb-name>
          <method-name>*</method-name>
     </method>
</method-permission>

<method-permission>
     <role-name>employee</role-name>
     <method>
          <ejb-name>AardvarkPayroll</ejb-name>
          <method-name>findByPrimaryKey</method-name>
     </method>
     <method>
          <ejb-name>AardvarkPayroll</ejb-name>
          <method-name>getEmployeeInfo</method-name>
     </method>
     <method>
          <ejb-name>AardvarkPayroll</ejb-name>
          <method-name>updateEmployeeInfo</method-name>
     </method>
</method-permission>

<method-permission>
     <role-name>payroll-department</role-name>
     <method>
          <ejb-name>AardvarkPayroll</ejb-name>
          <method-name>findByPrimaryKey</method-name>
     </method>
     <method>
          <ejb-name>AardvarkPayroll</ejb-name>
          <method-name>getEmployeeInfo</method-name>
     </method>
     <method>
          <ejb-name>AardvarkPayroll</ejb-name>
          <method-name>updateEmployeeInfo</method-name>
     </method>
     <method>
          <ejb-name>AardvarkPayroll</ejb-name>
          <method-name>updateSalary</method-name>
     </method>
</method-permission>

<method-permission>
     <role-name>admin</role-name>
     <method>
          <ejb-name>EmployeeServiceAdmin</ejb-name>
          <method-name>*</method-name>
     </method>
</method-permission>
...
```

16.3.2.3 Unspecified Method Permissions

It is possible that some methods are not assigned to any security roles nor annotated as `DenyAll` or contained in the `exclude-list` element. In this case, the Deployer should assign method permissions for all of the unspecified methods, either by assigning them to security roles, or by marking them as unchecked. If the Deployer does not assigned method permissions to the unspecified methods, those methods must be treated by the container as `unchecked`.

16.3.3 Linking Security Role References to Security Roles

The security role references used in the components of the application are linked to the security roles defined for the application. In the absence of any explicit linking, a security role reference will be linked to a security role having the same name.

The Application Assembler may explicitly link all the security role references declared in the `DeclareRoles` annotation or `security-role-ref` elements for a component to the security roles defined by the use of annotations (see section 16.3.1) and/or in the `security-role` elements.

The Application Assembler links each security role reference to a security role using the `role-link` element. The value of the `role-link` element must be the name of one of the security roles defined in a `security-role` element or by means of the `DeclareRoles` annotations or `RolesAllowed` annotations (as described in section 16.3.1), but need not be specified when the `role-name` used in the code is the same as the name of the `security-role` (to be linked).

The following deployment descriptor example shows how to link the security role reference named `payroll` to the security role named `payroll-department`.

```
...
<enterprise-beans>
     ...
     <session>
          <ejb-name>AardvarkPayroll</ejb-name>
          <ejb-class>com.aardvark.payroll.PayrollBean</ejb-class>
          ...
          <security-role-ref>
               <description>
                    This role should be assigned to the
                    employees of the payroll department.
                    Members of this role have access to
                    anyone's payroll record.
                    The role has been linked to the
                    payroll-department role.
               </description>
               <role-name>payroll</role-name>
               <role-link>payroll-department</role-link>
          </security-role-ref>
          ...
     </session>
     ...
</enterprise-beans>
...
```

16.3.4 Specification of Security Identities in the Deployment Descriptor

The Bean Provider or Application Assembler typically specifies whether the caller's security identity should be used for the execution of the methods of an enterprise bean or whether a specific run-as identity should be used.

By default the caller's security identity is used. The Bean Provider can use the `RunAs` metadata annotation to specify a run-as identity for the execution of the bean's methods. If the deployment descriptor is used, the Bean Provider or the Application Assembler can use the `security-identity` deployment descriptor element for this purpose or to override a security identity specified in metadata. The value of the `security-identity` element is either `use-caller-identity` or `run-as`.

Defining the security identities in the deployment descriptor is optional for the Application Assembler. Their omission in the deployment descriptor means that the Application Assembler chose not to pass any instructions related to security identities to the Deployer in the deployment descriptor.

If a run-as security identity is not specified by the Deployer, the container should use the caller's security identity for the execution of the bean's methods.

16.3.4.1 Run-as

The Bean Provider can use the `RunAs` metadata annotation or the Bean Provider or Application Assembler can use the `run-as` deployment descriptor element to define a run-as identity for an enterprise bean in the deployment descriptor. The run-as identity applies to the enterprise bean as a whole, that is, to all methods of the enterprise bean's business, home, and component interfaces, and/or web service endpoint; to the message listener methods of a message-driven bean; and to the timeout callback method of an enterprise bean; and all internal methods of the bean that they might in turn call.

Establishing a run-as identity for an enterprise bean does not affect the identities of its callers, which are the identities tested for permission to access the methods of the enterprise bean. The run-as identity establishes the identity the enterprise bean will use when it makes calls.

Because the Bean Provider and Application Assembler do not, in general, know the security environment of the operational environment, the run-as identity is designated by a *logical* role-name, which corresponds to one of the security roles defined by the Bean Provider or Application Assembler in the metadata annotations or deployment descriptor.

The Deployer then assigns a security principal defined in the operational environment to be used as the principal for the run-as identity. The security principal assigned by the Deployer should be a principal that has been assigned to the security role specified by `RunAs` annotation or by the `role-name` element of the `run-as` deployment descriptor element.

The Bean Provider and/or Application Assembler is responsible for the following in the specification of run-as identities:

- Use the `RunAs` metadata annotation or `role-name` element of the `run-as` deployment descriptor element to define the name of the security role.
- Optionally, use the description element to provide a description of the principal that is expected to be bound to the run-as identity in terms of its security role.

The following example illustrates the definition of a run-as identity using metadata annotations.

```
@RunAs("admin")
@Stateless public class EmployeeServiceBean
         implements EmployeeService{
     ...
}
```

Using the deployment descriptor, this can be specified as follows.

```
...
<enterprise-beans>
     ...
     <session>
          <ejb-name>EmployeeService</ejb-name>
          ...
          <security-identity>
               <run-as>
                    <role-name>admin</role-name>
               </run-as>
          </security-identity>
          ...
     </session>
     ...
</enterprise-beans>
...
```

16.4 Deployer's Responsibilities

The Deployer is responsible for ensuring that an assembled application is secure after it has been deployed in the target operational environment. This section defines the Deployer's responsibility with respect to EJB security management.

The Deployer uses deployment tools provided by the EJB Container Provider to read the security view of the application supplied by the Bean Provider and/or Application Assembler in the metadata annotations and/or deployment descriptor. The Deployer's job is to map the security view that was specified by the Bean Provider and/or Application Assembler to the mechanisms and policies used by the security domain in the target operational environment. The output of the Deployer's work includes an application security policy descriptor that is specific to the operational environment. The format of this descriptor and the information stored in the descriptor are specific to the EJB container.

The following subsections describe the security related tasks performed by the Deployer.

16.4.1 Security Domain and Principal Realm Assignment

The Deployer is responsible for assigning the security domain and principal realm to an enterprise bean application.

Multiple principal realms within the same security domain may exist, for example, to separate the realms of employees, trading partners, and customers. Multiple security domains may exist, for example, in application hosting scenarios.

16.4.2 Assignment of Security Roles

The Deployer assigns principals and/or groups of principals (such as individual users or user groups) used for managing security in the operational environment to the security roles defined by means of the `DeclareRoles` and `RolesAllowed` metadata annotations and/or `security-role` elements of the deployment descriptor.

The Deployer does not assign principals and/or principal groups to the security role references—the principals and/or principals groups assigned to a security role apply also to all the linked security role references. For example, the Deployer of the `AardvarkPayroll` enterprise bean in subsection 16.3.3 would assign principals and/or principal groups to the security-role `payroll-department`, and the assigned principals and/or principal groups would be implicitly assigned also to the linked security role `payroll`.

The EJB architecture does not specify how an enterprise should implement its security architecture. Therefore, the process of assigning the logical security roles defined in the application's deployment descriptor to the operational environment's security concepts is specific to that operational environment. Typically, the deployment process consists of assigning to each security role one or more user groups (or individual users) defined in the operational environment. This assignment is done on a per-application basis. (That is, if multiple independent ejb-jar files use the same security role name, each may be assigned differently.) If the deployer does not assign the logical security roles defined by the application to groups in the operational environment, it must be assumed that a logical role maps to a principal or principal group of the same name.

16.4.3 Principal Delegation

The Deployer is responsible for configuring the principal delegation for inter-component calls. The Deployer must follow any instructions supplied by the Bean Provider and/or Application Assembler (for example, provided in the `RunAs` metadata annotations, the `run-as` elements of the deployment descriptor, in the `description` elements of the annotations or deployment descriptor, or in a deployment manual).

If the security identity is defaulted, or it is explicitly specified that the caller identity be used (e.g., `use-caller-identity` deployment descriptor element is specified), the caller principal is propagated from one component to another (i.e., the caller principal of the first enterprise bean in a call-chain is passed to the enterprise beans down the chain).

If the Bean Provider or Application Assembler specifies that a run-as identity be used on behalf of a particular enterprise bean, the Deployer must configure the enterprise beans such that the run-as principal is used as the caller principal on any calls that the enterprise bean makes to other beans, and that the run-as principal is propagated along the call-chain of those other beans (in the absence of the specification of any further run-as elements).

16.4.4 Security Management of Resource Access

The Deployer's responsibilities with respect to securing resource managers access are defined in subsection 15.7.2.

16.4.5 General Notes on Deployment Descriptor Processing

The Deployer can use the security view defined in the deployment descriptor by the Bean Provider and Application Assembler merely as “hints” and may change the information whenever necessary to adapt the security policy to the operational environment.

Since providing the security information is optional for the Bean Provider and Application Assembler, the Deployer is responsible for performing any tasks that have not been done by the Bean Provider or Application Assembler. (For example, if the definition of security roles and method permissions is missing in the metadata annotations and in deployment descriptor, the Deployer must define the security roles and method permissions for the application.) It is not required that the Deployer store the output of this activity in the standard ejb-jar file format.

16.5 EJB Client Responsibilities

This section defines the rules that the EJB client program must follow to ensure that the security context passed on the client calls, and possibly imported by the enterprise bean, do not conflict with the EJB server’s capabilities for association between a security context and transactions.

These rules are:

- A transactional client cannot change its principal association within a transaction. This rule ensures that all calls from the client within a transaction are performed with the same security context.
- A session bean’s client must not change its principal association for the duration of the communication with the session object. This rule ensures that the server can associate a security identity with the session instance at instance creation time, and never have to change the security association during the session instance lifetime.
- If transactional requests within a single transaction arrive from multiple clients (this could happen if there are intermediary objects or programs in the transaction call-chain), all requests within the same transaction must be associated with the same security context.

16.6 EJB Container Provider’s Responsibilities

This section describes the responsibilities of the EJB Container and Server Provider.

16.6.1 Deployment Tools

The EJB Container Provider is responsible for providing the deployment tools that the Deployer can use to perform the tasks defined in Section 16.4.

The deployment tools read the information from the beans' metadata annotations and/or deployment descriptor and present the information to the Deployer. The tools guide the Deployer through the deployment process, and present him or her with choices for mapping the security information in the metadata annotations and deployment descriptor to the security management mechanisms and policies used in the target operational environment.

The deployment tools' output is stored in an EJB container-specific manner, and is available at runtime to the EJB container.

16.6.2 Security Domain(s)

The EJB container provides a security domain and one or more principal realms to the enterprise beans. The EJB architecture does not specify how an EJB server should implement a security domain, and does not define the scope of a security domain.

A security domain can be implemented, managed, and administered by the EJB server. For example, the EJB server may store X509 certificates or it might use an external security provider such as Kerberos.

The EJB specification does not define the scope of the security domain. For example, the scope may be defined by the boundaries of the application, EJB server, operating system, network, or enterprise.

The EJB server can, but is not required to, provide support for multiple security domains, and/or multiple principal realms.

The case of multiple domains on the same EJB server can happen when a large server is used for application hosting. Each hosted application can have its own security domain to ensure security and management isolation between applications owned by multiple organizations.

16.6.3 Security Mechanisms

The EJB Container Provider must provide the security mechanisms necessary to enforce the security policies set by the Deployer. The EJB specification does not specify the exact mechanisms that must be implemented and supported by the EJB server.

The typical security functions provided by the EJB server include:

- *Authentication of principals.*
- *Access authorization for EJB calls and resource manager access.*
- *Secure communication with remote clients (privacy, integrity, etc.).*

16.6.4 Passing Principals on EJB Calls

The EJB Container Provider is responsible for providing the deployment tools that allow the Deployer to configure the principal delegation for calls from one enterprise bean to another. The EJB container is responsible for performing the principal delegation as specified by the Deployer.

The EJB container must be capable of allowing the Deployer to specify that, for all calls from a single application within a single Java EE product, the caller principal is propagated on calls from one enterprise bean to another (i.e., the multiple beans in the call chain will see the same return value from `getCallerPrincipal`).

This requirement is necessary for applications that need a consistent return value of `getCallerPrincipal` across a chain of calls between enterprise beans.

The EJB container must be capable of allowing the Deployer to specify that a run-as principal be used for the execution of the business, home, and component interfaces, and/or web service endpoint methods of a session or entity bean, or for the message listener methods of a message-driven bean.

16.6.5 Security Methods in javax.ejb.EJBContext

The EJB container must provide access to the caller's security context information from the enterprise beans' instances via the `getCallerPrincipal()` and `isCallerInRole(String roleName)` methods. The EJB container must provide the caller's security context information during the execution of a business method invoked via the enterprise bean's business, home, component, or message listener interface, web service endpoint, and/or `TimedObject` interface, as defined in Table 1 on page 77, Table 2 on page 86, Table 3 on page 114, Table 4 on page 195, and Table 7 on page 237. The container must ensure that all enterprise bean method invocations received through these interfaces are associated with some principal. If the security identity of the caller has not been established, the container returns the container's representation of the unauthenticated identity. The container must never return a null from the `getCallerPrincipal` method.

16.6.6 Secure Access to Resource Managers

The EJB Container Provider is responsible for providing secure access to resource managers as described in Subsection 15.7.3.

16.6.7 Principal Mapping

If the application requires that its clients are deployed in a different security domain, or if multiple applications deployed across multiple security domains need to interoperate, the EJB Container Provider is responsible for the mechanism and tools that allow mapping of principals. The tools are used by the System Administrator to configure the security for the application's environment.

16.6.8 System Principal

The EJB specification does not define the "system" principal under which the JVM running an enterprise bean's method executes.

Leaving the principal undefined makes it easier for the EJB container vendors to provide runtime support for EJB on top of their existing server infrastructures. For example, while one EJB container implementation can execute all instances of all enterprise beans in a single JVM, another implementation can use a separate JVM per ejb-jar per client. Some EJB containers may make the system principal the same as the application-level principal. Others may use different principals, potentially from different principal realms and even security domains.

16.6.9 Runtime Security Enforcement

The EJB container is responsible for enforcing the security policies defined by the Deployer. The implementation of the enforcement mechanism is EJB container implementation-specific. The EJB container may, but does not have to, use the Java programming language security as the enforcement mechanism.

For example, to isolate multiple executing enterprise bean instances, the EJB container can load the multiple instances into the same JVM and isolate them via using multiple class loaders, or it can load each instance into its own JVM and rely on the address space protection provided by the operating system.

The general security enforcement requirements for the EJB container follow:

- The EJB container must provide enforcement of the client access control per the policy defined by the Deployer. A caller is allowed to invoke a method if, and only if, the method is specified as `PermitAll` or the caller is assigned *at least one* of the security roles that includes the method in its method permissions definition. (That is, it is not meant that the caller must be assigned *all* the roles associated with the method.) If the container denies a client access to a business method, the container must throw the `javax.ejb.EJBAccessException`[76]. If the EJB 2.1 client view is used, the container must throw the `java.rmi.RemoteException` (or its subclass, the `java.rmi.AccessException`) to the client if the client is a remote client, or the `javax.ejb.EJBException` (or its subclass, the `javax.ejb.AccessLocalException`) if the client is a local client.

- The EJB container must isolate an enterprise bean instance from other instances and other application components running on the server. The EJB container must ensure that other enterprise bean instances and other application components are allowed to access an enterprise bean only via the enterprise bean's business interface, component interface, home interface, and/or web service endpoint.

- The EJB container must isolate an enterprise bean instance at runtime such that the instance does not gain unauthorized access to privileged system information. Such information includes the internal implementation classes of the container, the various runtime state and context maintained by the container, object references of other enterprise bean instances, or resource managers used by other enterprise bean instances. The EJB container must ensure that the interactions between the enterprise beans and the container are only through the EJB architected interfaces.

- The EJB container must ensure the security of the persistent state of the enterprise beans.

- The EJB container must manage the mapping of principals on calls to other enterprise beans or on access to resource managers according to the security policy defined by the Deployer.

[76] If the business interface is a remote business interface that extends `java.rmi.Remote`, the `java.rmi.AccessException` is thrown to the client instead.

- The container must allow the same enterprise bean to be deployed independently multiple times, each time with a different security policy[77]. The container must allow multiple-deployed enterprise beans to co-exist at runtime.

16.6.10 Audit Trail

The EJB container may provide a security audit trail mechanism. A security audit trail mechanism typically logs all `java.security.Exceptions`. It also logs all denials of access to EJB servers, EJB container, EJB component interfaces, EJB home interfaces, and EJB web service endpoints.

16.7 System Administrator's Responsibilities

This section defines the security-related responsibilities of the System Administrator. Note that some responsibilities may be carried out by the Deployer instead, or may require cooperation of the Deployer and the System Administrator.

16.7.1 Security Domain Administration

The System Administrator is responsible for the administration of principals. Security domain administration is beyond the scope of the EJB specification.

Typically, the System Administrator is responsible for creating a new user account, adding a user to a user group, removing a user from a user group, and removing or freezing a user account.

16.7.2 Principal Mapping

If the client is in a different security domain than the target enterprise bean, the System Administrator is responsible for mapping the principals used by the client to the principals defined for the enterprise bean. The result of the mapping is available to the Deployer.

The specification of principal mapping techniques is beyond the scope of the EJB architecture.

16.7.3 Audit Trail Review

If the EJB container provides an audit trail facility, the System Administrator is responsible for its management.

[77] For example, the enterprise bean may be installed each time using a different bean name (as specified by means of the deployment descriptor).

Chapter 17 Timer Service

This chapter defines the EJB container-managed timer service. The EJB timer service is a container-provided service that allows the Bean Provider to register enterprise beans for timer callbacks to occur at a specified time, after a specified elapsed time, or at specified intervals.

17.1 Overview

Enterprise applications that model workflow-type business processes are dependent on notifications that certain temporal events have occurred in order to manage the semantic state transitions that are intrinsic to the business processes that they model.

The EJB Timer Service is a container-managed service that provides methods to allow callbacks to be scheduled for time-based events. The container provides a reliable and transactional notification service for timed events. Timer notifications may be scheduled to occur at a specific time, after a specific elapsed duration, or at specific recurring intervals.

The Timer Service is implemented by the EJB container. An enterprise bean accesses this service by means of dependency injection, through the `EJBContext` interface, or through lookup in the JNDI namespace.

The EJB Timer Service is a coarse-grained timer notification service that is designed for use in the modeling of application-level processes. It is not intended for the modeling of real-time events.

> *While timer durations are expressed in millisecond units, this is because the millisecond is the unit of time granularity used by the APIs of the Java SE platform. It is expected that most timed events will correspond to hours, days, or longer periods of time.*

The following sections describe the Timer Service with respect to the various individual EJB roles.

17.2 Bean Provider's View of the Timer Service

The EJB Timer Service is a container-provided service that allows enterprise beans to be registered for timer callback methods to occur at a specified time, after a specified elapsed time, or after specified intervals. The timer service provides methods for the creation and cancellation of timers, as well as for locating the timers that are associated with a bean.

A timer is created to schedule timed callbacks. The bean class of an enterprise bean that uses the timer service must provide a timeout callback method. This method may be a method that is annotated with the `Timeout` annotation, or the bean may implement the `javax.ejb.TimedObject` interface. The `javax.ejb.TimedObject` interface has a single method, the timer callback method `ejbTimeout`. Timers can be created for stateless session beans, message-driven beans, and 2.1 entity beans. Timers cannot be created for stateful session beans[78] or EJB 3.0 entities.

A timer that is created for a 2.1 entity bean is associated with the entity bean's identity. The timeout callback method invocation for a timer that is created for a stateless session bean or a message-driven bean may be called on any bean instance in the pooled state.

When the time specified at timer creation elapses, the container invokes the timeout callback method of the bean. A timer may be cancelled by a bean before its expiration. If a timer is cancelled, the timeout callback method is not called[79]. A timer is cancelled by calling its `cancel` method.

Invocations of the methods to create and cancel timers and of the timeout callback method are typically made within a transaction.

The timer service is intended for the modelling of long-lived business processes. Timers survive container crashes, server shutdown, and the activation/passivation and load/store cycles of the enterprise beans that are registered with them.

[78] This functionality may be added in a future release of this specification.

[79] In the event of race conditions, extraneous calls to the timeout callback method may occur.

17.2.1 The Timer Service Interface

The Timer Service is accessed via dependency injection, through the `getTimerService` method of the `EJBContext` interface, or through lookup in the JNDI namespace. The `TimerService` interface has the following methods:

```
public interface javax.ejb.TimerService {

    public Timer createTimer(long duration,
            java.io.Serializable info);

    public Timer createTimer(long initialDuration,
        long intervalDuration, java.io.Serializable info);

    public Timer createTimer(java.util.Date expiration,
        java.io.Serializable info);

    public Timer createTimer(java.util.Date initialExpiration,
        long intervalDuration, java.io.Serializable info);

    public Collection getTimers();

    }
```

The timer creation methods allow a timer to be created as a single-event timer or as an interval timer. The timer expiration (initial expiration in the case of an interval timer) may be expressed either in terms of a duration or as an absolute time.

The bean may pass some client-specific information at timer creation to help it recognize the significance of the timer's expiration. This information is stored by the timer service and available through the timer. The information object must be serializable. [80]

The timer duration is expressed in terms of milliseconds. The timer service begins counting down the timer duration upon timer creation.

The `createTimer` methods return a `Timer` object that allows the bean to cancel the timer or to obtain information about the timer prior to its cancellation and/or expiration (if it is a single-event timer).

The `getTimers` method returns the active timers associated with the bean. For an EJB 2.1 entity bean, the result of `getTimers` is a collection of those timers that are associated with the bean's identity.

17.2.2 Timeout Callbacks

The enterprise bean class of a bean that is to be registered with the timer service for timer callbacks must provide a timeout callback method.

[80] There is currently no way to set the information object other than through the createTimer method. An API to do this may be added in a future release of this specification.

This method may be a method annotated with the `Timeout` annotation (or a method specified as a timeout method in the deployment descriptor) or the bean may implement the `javax.ejb.TimedObject` interface. This interface has a single method, `ejbTimeout`. If the bean implements the `TimedObject` interface, the `Timeout` annotation or `timeout-method` deployment descriptor element can only be used to specify the `ejbTimeout` method. A bean can have at most one timeout method.

```
public interface javax.ejb.TimedObject {
     public void ejbTimeout(Timer timer);
}
```

Any method annotated as a `Timeout` method (or designated in the deployment descriptor as such) must have the signature below, where <METHOD> designates the method name[81]:

```
public void <METHOD>(Timer timer)
```

Timeout callback methods must not throw application exceptions.

When the timer expires (i.e., after the number of milliseconds specified at its creation expires or after the absolute time specified has passed), the container calls the timeout method of the bean that was registered for the timer. The timeout method contains the business logic that the Bean Provider supplies to handle the timeout event. The container calls the timeout method with the timer that has expired. The Bean Provider can use the `getInfo` method to retrieve the information that was supplied when the timer was created. This information may be useful in enabling the timed object to recognize the significance of the timer expiration.

> *The container interleaves calls to the timeout callback method with the calls to the business methods and the life cycle callback methods of the bean. The time at which the timeout callback method is called may therefore not correspond exactly to the time specified at timer creation. If multiple timers have been created for a bean and will expire at approximately the same times, the Bean Provider must be prepared to handle timeout callbacks that are out of sequence. The Bean Provider must be prepared to handle extraneous calls to the timeout callback method in the event that a timer expiration is outstanding when a call to the cancellation method has been made.*

In general, the timeout callback method can perform the same operations as business methods from the component interface or methods from the message listener interface. See Tables 2, 3, 4, and 7 for the specification of the operations that may be performed by the timeout callback method.

Since the timeout callback method is an internal method of the bean class, it has no client security context. When `getCallerPrincipal` is called from within the timeout callback method, it returns the container's representation of the unauthenticated identity.

If the timed object needs to make use of the identity of the timer to recognize the significance of the timer expiration, it may use the `equals` method to compare it with any other timer references it might have outstanding.

---

[81] If the bean implements the `TimedObject` interface, the `Timeout` annotation may optionally be applied to the `ejbTimeout` method.

If the timer is a single-action timer, the container removes the timer after the timeout callback method has been successfully invoked (e.g., when the transaction that has been started for the invocation of the timeout callback method commits). If the bean invokes a method on the timer after the termination of the timeout callback method, the `NoSuchObjectLocalException` is thrown.

17.2.3 The Timer and TimerHandle Interfaces

The `javax.ejb.Timer` interface allows the Bean Provider to cancel a timer and to obtain information about the timer.

The `javax.ejb.TimerHandle` interface allows the Bean Provider to obtain a serializable timer handle that may be persisted. Since timers are local objects, a TimerHandle must not be passed through a bean's remote business interface, remote interface or web service interface.

The methods of these interfaces are as follows:

```
public interface javax.ejb.Timer {

     public void cancel();

     public long getTimeRemaining();

     public java.util.Date getNextTimeout();

     public javax.ejb.TimerHandle getHandle();

     public java.io.Serializable getInfo();
}

public interface javax.ejb.TimerHandle extends java.io.Serializable {

     public javax.ejb.Timer getTimer();

}
```

17.2.4 Timer Identity

The Bean Provider cannot rely on the `==` operator to compare timers for "object equality". The Bean Provider must use the `Timer.equals(Object obj)` method.

17.2.5 Transactions

An enterprise bean typically creates a timer within the scope of a transaction. If the transaction is then rolled back, the timer creation is rolled back.

An enterprise bean typically cancels a timer within a transaction. If the transaction is rolled back, the container rescinds the timer cancellation.

The timeout callback method is typically has transaction attribute `REQUIRED` or `REQUIRES_NEW` (`Required` or `RequiresNew` if the deployment descriptor is used to specify the transaction attribute). If the transaction is rolled back, the container retries the timeout.

The container must start a new transaction if the `REQUIRED` *(*`Required`*) transaction attribute is used. This transaction attribute value is allowed so that specification of a transaction attribute for the timeout callback method can be defaulted.*

17.3 Bean Provider's Responsibilities

This section defines the Bean Provider's responsibilities.

17.3.1 Enterprise Bean Class

An enterprise bean that is to be registered with the Timer Service must have a timeout callback method. The enterprise bean class may have superclasses and/or superinterfaces. If the bean class has superclasses, the timeout method may be defined in the bean class, or in any of its superclasses.

17.3.2 TimerHandle

Since the `TimerHandle` interface extends `java.io.Serializable`, a client may serialize the handle. The serialized handle may be used later to obtain a reference to the timer identified by the handle. A TimerHandle is intended to be storable in persistent storage.

A TimerHandle must not be passed as an argument or result of an enterprise bean's remote business interface, remote interface, or web service method.

17.4 Container's Responsibilities

This section describes the responsibilities of the Container Provider to support the EJB Timer Service.

17.4.1 TimerService, Timer, and TimerHandle Interfaces

The container must provide the implementation of the `TimerService`, `Timer`, and `TimerHandle` interfaces.

Timer instances must not be serializable.

The container must implement a timer handle to be usable over the lifetime of the timer.

The container must provide suitable implementations of the `Timer equals(Object obj)` and `hashCode()` methods.

17.4.2 Timer Expiration and Timeout Callback Method

The container must call the timeout callback method after the timed duration or the absolute time specification in the timer creation method has passed. The timer service must begin to count down the timer duration upon timer creation. The container must call the timeout callback method with the expired Timer.

If container-managed transaction demarcation is used and the `REQUIRED` or `REQUIRES_NEW` transaction attribute is specified or defaulted (`Required` or `RequiresNew` if the deployment descriptor is used), the container must begin a new transaction prior to invoking the timeout callback method. If the transaction fails or is rolled back, the container must retry the timeout at least once.

If a timer for an EJB 2.1 entity bean expires, and the bean has been passivated, the container must call the `ejbActivate` and `ejbLoad` methods on the entity bean class before calling the timeout callback method, as described in Sections 8.5.3 and 9.1.4.2.

If the timer is a single-event timer, the container must cause the timer to no longer exist. If a method is subsequently invoked on the timer after the completion of the timeout callback method, the container must throw the `javax.ejb.NoSuchObjectLocalException`.

If the Bean Provider invokes the `setRollbackOnly` method from within the timeout callback method, the container must rollback the transaction in which the timeout callback method is invoked. This has the effect of rescinding the timer expiration. The container must retry the timeout after the transaction rollback.

Timers are persistent objects. In the event of a container crash, any single-event timers that have expired during the intervening time before container restart must cause the timeout callback method to be invoked upon restart. Any interval timers that have expired during the intervening time must cause the timeout callback method to be invoked at least once upon restart.

17.4.3 Timer Cancellation

When a timer's `cancel` method has been called, the container must cause the timer to no longer exist. If a method is subsequently invoked on the timer, the container must throw the `javax.ejb.NoSuchObjectLocalException`.

If the transaction in which the timer cancellation occurs is rolled back, the container must restore the duration of the timer to the duration it would have had if it had not been cancelled. If the timer would have expired by the time that the transaction failed, the failure of the transaction should result in the expired timer providing an expiration notification after the transaction rolls back.

17.4.4 Entity Bean Removal

If an entity bean is removed, the container must remove the timers for that bean.

Chapter 18 Deployment Descriptor

This chapter defines the deployment descriptor that is part of the ejb-jar file. Section 18.1 provides an overview of the deployment descriptor. Sections 18.2 through 18.4 describe the information in the deployment descriptor from the perspective of the EJB roles responsible for providing the information. Section 18.5 defines the deployment descriptor's XML Schema elements that are specific to the EJB architecture. The XML Schema elements that are common to the Java EE Platform specifications are provided in [12].

18.1 Overview

The deployment descriptor is part of the contract between the ejb-jar file producer and consumer. This contract covers both the passing of enterprise beans from the Bean Provider to the Application Assembler, and from the Application Assembler to the Deployer.

An ejb-jar file produced by the Bean Provider contains one or more enterprise beans and typically does not contain application assembly instructions. An ejb-jar file produced by an Application Assembler contains one or more enterprise beans, plus application assembly information describing how the enterprise beans are combined into a single application deployment unit.

The Java EE specification defines how enterprise beans and other application components contained in multiple ejb-jar files can be assembled into an application.

The role of the deployment descriptor is to capture declarative information that is not included directly in the enterprise beans' code and that is intended for the consumer of the ejb-jar file.

There are two basic kinds of information in the deployment descriptor:

- *Enterprise beans' structural information*. Structural information describes the structure of an enterprise bean and declares an enterprise bean's external dependencies. Providing structural information for the ejb-jar file producer. Structural information may be provided using metadata annotations in the beans' code or in the deployment descriptor. The structural information cannot, in general, be changed because doing so could break the enterprise bean's function.

- *Application assembly information*. Application assembly information describes how the enterprise bean (or beans) in the ejb-jar file is composed into a larger application deployment unit. Providing assembly information—whether in metadata annotations or in the deployment descriptor—is optional for the ejb-jar file producer. Assembly level information can be changed without breaking the enterprise bean's function, although doing so may alter the behavior of an assembled application.

18.2 Bean Provider's Responsibilities

The Bean Provider is responsible for providing in the deployment descriptor the following structural information for each enterprise bean if this information has not be provided in metadata annotations or is to be defaulted.

The Bean Provider uses the `enterprise-beans` element to list all the enterprise beans in the ejb-jar file.

The Bean Provider must provide the following information for each enterprise bean:

- **Enterprise bean's name**. A logical name is assigned to each enterprise bean in the ejb-jar file. There is no architected relationship between this name and the JNDI name that the Deployer will assign to the enterprise bean. The Bean Provider can specify the enterprise bean's name in the `ejb-name` element. If the enterprise bean's name is not explicitly specified in metadata annotations or in the deployment descriptor, it defaults to the unqualified name of the bean class.

- **Enterprise bean's class**. If the bean class has not been annotated with the `Stateless`, `Stateful`, or `Message-driven` annotation, the Bean Provider must use the `ejb-class` element of the `session` or `message-driven` deployment descriptor element to specify the fully-qualified name of the Java class that implements the enterprise bean's business methods. The Bean Provider specifies the enterprise bean's class name in the `ejb-class` element. The Bean Provider must use this element for an EJB 2.1 and earlier entity bean.

- **Enterprise bean's local business interface**. If the bean class has a local business interface and neither implements the business interface nor specifies it as a local business interface using metadata annotations on the bean class, the Bean Provider must specify the fully-qualified name of the enterprise bean's local business interface in the `business-local` element.

- **Enterprise bean's remote business interface**. If the bean class has a remote business interface and neither implements nor specifies it as a remote business interface using metadata annotations on the bean class, the Bean Provider must specify the fully-qualified name of the enterprise bean's remote business interface in the `business-remote` element.

- **Enterprise bean's remote home interface**. If the bean class has a remote home interface, and the remote home interface has not been specified using metadata annotations, the Bean Provider must specify the fully-qualified name of the enterprise bean's remote home interface in the `home` element.

- **Enterprise bean's remote interface**. If the bean class has a remote interface, and the remote home interface has not been specified using metadata annotations, the Bean Provider must specify the fully-qualified name of the enterprise bean's remote interface in the `remote` element.

- **Enterprise bean's local home interface**. If the bean class has a local home interface, and the local home interface has not been specified using metadata annotations, the Bean Provider must specify the fully-qualified name of the enterprise bean's local home interfacein the `local-home` element.

- **Enterprise bean's local interface**. If the bean class has a local interface, and the local home interface has not been specified using metadata annotations, the Bean Provider must specify the fully-qualified name of the enterprise bean's local interface in the `local` element.

- **Enterprise bean's web service endpoint interface.** If the bean class has a web service endpoint interface, and the interface has not been specified using metadata annotations on the bean class, the Bean Provider must specify the fully-qualified name of the enterprise bean's web service endpoint interface, in the `service-endpoint` element. This element may only be used for stateless session beans.

- **Enterprise bean's type**. The enterprise bean types are: session, entity, and message-driven. The Bean Provider must use the appropriate `session`, `entity`, or `message-driven` element to declare the enterprise bean's structural information if annotations have not been used for this purpose.

- **Re-entrancy indication**. The Bean Provider must specify whether an EJB 2.1 entity bean is re-entrant or not. Session beans and message-driven beans are never re-entrant.

- **Session bean's state management type**. If the enterprise bean is a session bean and the bean class has not been annotated with the `Stateful` or `Stateless` **annotation,** the Bean Provider must use the `session-type` element to declare whether the session bean is stateful or stateless.

- **Session or message-driven bean's transaction demarcation type**. If the enterprise bean is a session bean or message-driven bean, the Bean Provider may use the `transaction-type` element to declare whether transaction demarcation is performed by the enterprise bean or by the container. If the neither the `TransactionType` annotation is used nor the `transaction-type` deployment descriptor element, the bean will have container managed transaction demarcation.

- **Entity bean's persistence management**. If the enterprise bean is an EJB 2.1 entity bean, the Bean Provider must use the `persistence-type` element to declare whether persistence management is performed by the enterprise bean or by the container.

- **Entity bean's primary key class**. If the enterprise bean is an EJB 2.1 entity bean, the Bean Provider specifies the fully-qualified name of the entity bean's primary key class in the `prim-key-class` element. The Bean Provider *must* specify the primary key class for an entity with bean-managed persistence.

- **Entity bean's abstract schema name.** If the enterprise bean is an entity bean with container-managed persistence and `cmp-version 2.x`, the Bean Provider must specify the abstract schema name of the entity bean using the `abstract-schema-name` element.

- **Container-managed fields**. If the enterprise bean is an entity bean with container-managed persistence, the Bean Provider must specify the container-managed fields using the `cmp-field` elements.

- **Container-managed relationships.** If the enterprise bean is an entity bean with container-managed persistence and `cmp-version 2.x`, the Bean Provider must specify the container-managed relationships of the entity bean using the `relationships` element.

- **Finder and select queries.** If the enterprise bean is an entity bean with container-managed persistence and `cmp-version 2.x`, the Bean Provider must use the `query` element to specify any EJB QL finder or select query for the entity bean other than a query for the `findByPrimaryKey` method.

- **Environment entries**. The Bean Provider must declare any enterprise bean's environment entries that have not been defined by means of metadata annotations, as specified in Subsection 15.3.1.

- **Resource manager connection factory references**. The Bean Provider must declare any enterprise bean's resource manager connection factory references that have not been defined by means of metadata annotations, as specified in Subsection 15.7.1.

- **Resource environment references**. The Bean Provider must declare any enterprise bean's references to administered objects that are associated with resources and that have not been defined by means of metadata annotations, as specified in Subsection 15.8.1.

- **EJB references**. The Bean Provider must declare any enterprise bean's references to the remote homes of other enterprise beans that have not been defined by means of metadata annotations, as specified in Subsection 15.5.1.

- **EJB local references**. The Bean Provider must declare any enterprise bean's references to the local homes of other enterprise beans that have not been defined by means of metadata annotations, as specified in Subsection 15.5.1.

- **Web service references**. The Bean Provider must declare any enterprise bean's references to web service interfaces that have not been defined by means of metadata annotations, as specified in Subsection 15.6.

- **Persistence unit references**. The Bean Provider must declare any enterprise bean's references to an entity manager factory for a persistence unit that have not been defined by means of metadata annotations, as specified in Subsection 15.10.

- **Persistence context references**. The Bean Provider must declare any enterprise bean's references to an entity manager for a persistence context that have not been defined by means of metadata annotations, as specified in Subsection 15.11.

- **Message destination references**. The Bean Provider must declare any enterprise bean's references to message destinations that have not been defined by means of metadata annotations, as specified in Subsection 15.9.1.

- **Security role references**. The Bean Provider must declare any enterprise bean's references to security roles that have not been defined by means of metadata annotations, as specified in Subsection 16.2.5.3.

- **Message-driven bean's configuration properties.** The Bean Provider may provide input to the Deployer as to how a message-driven bean should be configured upon activation in its operational environment. Activation configuration properties for a JMS message-driven bean include information about a bean's intended destination type, its message selector, and its acknowledgement mode. Other bean types may make use of different properties. See [15].

- **Message-driven bean's destination.** The Bean Provider may provide advice to the Application Assembler as to the destination type to which a message-driven bean should be assigned when linking message destinations

- **Interceptors.** The Bean Provider must declare any interceptor classes and methods that have not been declared by means of metadata annotations.

The deployment descriptor produced by the Bean Provider must conform to the XML Schema defintion in Section 18.5 or the DTD definition from a previous version of this specification. The content of the deployment descriptor must conform to the semantics rules specified in the XML Schema or DTD comments and elsewhere in this specification.

18.3 Application Assembler's Responsibility

The Application Assembler assembles enterprise beans into a single deployment unit. The Application Assembler's input is one or more ejb-jar files provided by one or more Bean Providers, and the output is also one or more ejb-jar files. The Application Assembler can combine multiple input ejb-jar files into a single output ejb-jar file, or split an input ejb-jar file into multiple output ejb-jar files. Each output ejb-jar file is either a deployment unit intended for the Deployer, or a partially assembled application that is intended for another Application Assembler.

The Bean Provider and Application Assembler may be the same person or organization. In such a case, the person or organization performs the responsibilities described both in this and the previous sections.

The Application Assembler may modify the following information that was specified by the Bean Provider:

- **Values of environment entries**. The Application Assembler may change existing and/or define new values of environment properties.
- **Description fields**. The Application Assembler may change existing or create new `description` elements.
- **Relationship names for EJB 2.x entity beans.** If multiple ejb-jar files use the same names for relationships and are merged into a single ejb-jar file, it is the responsibility of the Application Assembler to modify the relationship names defined in the `ejb-relation-name` elements.
- **Message-driven bean message selector**. The Application Assembler may further restrict, but not replace, the value of the `messageSelector` `activation-config-property` element of a JMS message-driven bean—whether this was defined in metadata annotations or the deployment descriptor.

In general, the Application Assembler should never modify any of the following.

- **Enterprise bean's abstract schema name**. The Application Assembler should not change the enterprise bean's name defined in the `abstract-schema-name` element since EJB QL queries may depend on the content of this element.
- **Relationship role source element**. The Application Assembler should not change the value of an `ejb-name` element in the `relationship-role-source` element.

If any of these elements must be modified by the Application Assembler in order to resolve name clashes during the merging two ejb-jar files into one, the Application Assembler must also modify all `ejb-ql` query strings that depend on the value of the modified element(s).

The Application Assembler must not, in general, modify any other information listed in Section 18.2 that was provided in the input ejb-jar file.

The Application Assembler may, but is not required to, specify any of the following application assembly information:

- **Binding of enterprise bean references**. The Application Assembler may link an enterprise bean reference to another enterprise bean in the ejb-jar file or in the same Java EE application unit. The Application Assembler creates the link by adding the `ejb-link` element to the referencing bean. The Application Assembler uses the `ejb-name` of the referenced bean for the link. If there are multiple enterprise beans with the same `ejb-name`, the Application Assembler uses the path name specifying the location of the ejb-jar file that contains the referenced component. The path name is relative to the referencing ejb-jar file. The Application Assembler appends the `ejb-name` of the referenced bean to the path name separated by #. This allows multiple beans with the same name to be uniquely identified.
- **Linking of message destination references**. The Application Assembler may link message consumers and producers through common message destinations specified in the ejb-jar file or

in the same Java EE application unit. The Application Assembler creates the link by adding the `message-destination-link` element to the referencing bean.

- **Security roles**. The Application Assembler may define one or more security roles. The security roles define the *recommended* security roles for the clients of the enterprise beans. The Application Assembler defines the security roles using the `security-role` elements.

- **Method permissions**. The Application Assembler may define method permissions. Method permission is a binary relation between the security roles and the methods of the business interfaces, home interfaces, component interfaces, and/or web service endpoints of the enterprise beans. The Application Assembler defines method permissions using the `method-permission` elements. The Application Assembler may augment or ovrride method permissions defined by the Bean Provider—whether in metadata annotations or in the deployment descriptor.

- **Linking of security role references**. If the Application Assembler defines security roles in the deployment descriptor, the Application Assembler may link the security role references declared by the Bean Provider to the security roles. The Application Assembler defines these links using the `role-link` element.

- **Security identity**. The Application Assembler may specify whether the caller's security identity should be used for the execution of the methods of an enterprise bean or whether a specific run-as security identity should be used. The Application Assembler may ovrride a security identity defined by the Bean Provider—whether in metadata annotations or in the deployment descriptor

- **Transaction attributes**. The Application Assembler may define the value of the transaction attributes for the methods of the business interface, home interface, component interface, web service endpoint, and `TimedObject` interface of the enterprise beans that require container-managed transaction demarcation. All entity beans and the session and message-driven beans declared by the Bean Provider as transaction-type `Container` require container-managed transaction demarcation. The Application Assembler uses the `container-transaction` elements to declare the transaction attributes.

- **Interceptors**. The Application Assembler may override, augment, and/or recorder the interceptor methods defined by the Bean Provider—whether in metadata annotations or in the deployment descriptor.

If an input ejb-jar file contains application assembly information, the Application Assembler is allowed to change the application assembly information supplied in the input ejb-jar file. (This could happen when the input ejb-jar file was produced by another Application Assembler.)

The deployment descriptor produced by the Bean Provider must conform to the XML Schema definition in Section 18.5 or the DTD definition from a previous version of this specification. The content of the deployment descriptor must conform to the semantics rules specified in the XML Schema or DTD comments and elsewhere in this specification.

18.4 Container Provider's Responsibilities

The Container Provider provides tools that read and import the information contained in the XML deployment descriptor.

All EJB 3.0 implementations must support EJB 2.1, EJB 2.0, and EJB 1.1 as well as EJB 3.0 deployment descriptors. The definitions of the EJB 2.1, EJB 2.0, and EJB 1.1 deployment descriptors can be found in the Enterprise JavaBeans 2.1 specification [3].

18.5 Deployment Descriptor XML Schema

This section provides the XML Schema for the EJB 3.0 deployment descriptor. The comments in the XML Schema specify additional requirements for the syntax and semantics that cannot be easily expressed by the XML Schema mechanism.

The content of the XML elements is in general case sensitive (i.e., unless stated otherwise). This means, for example, that

```
<transaction-type>Container</transaction-type>
```

must be used, rather than:

```
<transaction-type>container</transaction-type>.
```

All valid ejb-jar deployment descriptors must conform to the XML Schema definition below or to the DTD definition from a previous version of this specification.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://java.sun.com/xml/ns/javaee"
            xmlns:javaee="http://java.sun.com/xml/ns/javaee"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            elementFormDefault="qualified"
            attributeFormDefault="unqualified"
            version="3.0">
  <xsd:annotation>
    <xsd:documentation>
      @(#)ejb-jar_3_0.xsds  1.47 11/22/05
    </xsd:documentation>
  </xsd:annotation>

  <xsd:annotation>
    <xsd:documentation>

      Copyright 2003-2005 Sun Microsystems, Inc.
      4150 Network Circle
      Santa Clara, California 95054
      U.S.A
      All rights reserved.

      Sun Microsystems, Inc. has intellectual property rights
      relating to technology described in this document. In
      particular, and without limitation, these intellectual
      property rights may include one or more of the U.S. patents
      listed at http://www.sun.com/patents and one or more
      additional patents or pending patent applications in the
      U.S. and other countries.

      This document and the technology which it describes are
      distributed under licenses restricting their use, copying,
      distribution, and decompilation. No part of this document
      may be reproduced in any form by any means without prior
      written authorization of Sun and its licensors, if any.

      Third-party software, including font technology, is
      copyrighted and licensed from Sun suppliers.

      Sun, Sun Microsystems, the Sun logo, Solaris, Java, J2EE,
      JavaServer Pages, Enterprise JavaBeans and the Java Coffee
      Cup logo are trademarks or registered trademarks of Sun
      Microsystems, Inc. in the U.S. and other countries.

      Federal Acquisitions: Commercial Software - Government Users
      Subject to Standard License Terms and Conditions.

    </xsd:documentation>
  </xsd:annotation>

  <xsd:annotation>
    <xsd:documentation>
      <![CDATA[

        This is the XML Schema for the EJB 3.0 deployment descriptor.
        The deployment descriptor must be named "META-INF/ejb-jar.xml" in
        the EJB's jar file.  All EJB deployment descriptors must indicate
        the ejb-jar schema by using the Java EE namespace:

        http://java.sun.com/xml/ns/javaee
```

```
            and by indicating the version of the schema by
            using the version element as shown below:

                <ejb-jar xmlns="http://java.sun.com/xml/ns/javaee"
                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance
                  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                  http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd"
                  version="3.0">
                  ...
                </ejb-jar>

            The instance documents may indicate the published version of
            the schema using the xsi:schemaLocation attribute for the
            Java EE namespace with the following location:

            http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd

            ]]>
        </xsd:documentation>
      </xsd:annotation>

      <xsd:annotation>
        <xsd:documentation>

          The following conventions apply to all Java EE
          deployment descriptor elements unless indicated otherwise.

          - In elements that specify a pathname to a file within the
            same JAR file, relative filenames (i.e., those not
            starting with "/") are considered relative to the root of
            the JAR file's namespace.  Absolute filenames (i.e., those
            starting with "/") also specify names in the root of the
            JAR file's namespace.  In general, relative names are
            preferred.  The exception is .war files where absolute
            names are preferred for consistency with the Servlet API.

        </xsd:documentation>
      </xsd:annotation>

      <xsd:include schemaLocation="javaee_5.xsd"/>


    <!-- **************************************************** -->

      <xsd:element name="ejb-jar" type="javaee:ejb-jarType">
        <xsd:annotation>
          <xsd:documentation>

            This is the root of the ejb-jar deployment descriptor.

          </xsd:documentation>
        </xsd:annotation>

        <xsd:key name="ejb-name-key">
          <xsd:annotation>
            <xsd:documentation>

              The ejb-name element contains the name of an enterprise
              bean. The name must be unique within the ejb-jar file.

            </xsd:documentation>
          </xsd:annotation>
          <xsd:selector xpath="javaee:enterprise-beans/*"/>
          <xsd:field    xpath="javaee:ejb-name"/>
        </xsd:key>
```

```
    <xsd:keyref name="ejb-name-references"
                refer="javaee:ejb-name-key">
      <xsd:annotation>
        <xsd:documentation>

          The keyref indicates the references from
          relationship-role-source must be to a specific ejb-name
          defined within the scope of enterprise-beans element.

        </xsd:documentation>
      </xsd:annotation>
      <xsd:selector
xpath=".//javaee:ejb-relationship-role/javaee:relationship-role-source"/>
      <xsd:field
           xpath="javaee:ejb-name"/>
    </xsd:keyref>

    <xsd:key name="role-name-key">
      <xsd:annotation>
        <xsd:documentation>

          A role-name-key is specified to allow the references
          from the security-role-refs.

        </xsd:documentation>
      </xsd:annotation>
      <xsd:selector xpath="javaee:assembly-descriptor/javaee:security-role"/>
      <xsd:field    xpath="javaee:role-name"/>
    </xsd:key>

    <xsd:keyref name="role-name-references"
                refer="javaee:role-name-key">
      <xsd:annotation>
        <xsd:documentation>

          The keyref indicates the references from
          security-role-ref to a specified role-name.

        </xsd:documentation>
      </xsd:annotation>
      <xsd:selector xpath="javaee:enterprise-beans/*/javaee:secu-
rity-role-ref"/>
      <xsd:field    xpath="javaee:role-link"/>
    </xsd:keyref>
  </xsd:element>


<!-- **************************************************** -->

  <xsd:complexType name="activation-config-propertyType">
    <xsd:annotation>
      <xsd:documentation>

        The activation-config-propertyType contains a name/value
        configuration property pair for a message-driven bean.

        The properties that are recognized for a particular
        message-driven bean are determined by the messaging type.

      </xsd:documentation>
    </xsd:annotation>

    <xsd:sequence>
      <xsd:element name="activation-config-property-name"
                   type="javaee:xsdStringType">
        <xsd:annotation>
```

```
            <xsd:documentation>

              The activation-config-property-name element contains
              the name for an activation configuration property of
              a message-driven bean.

              For JMS message-driven beans, the following property
              names are recognized: acknowledgeMode,
              messageSelector, destinationType, subscriptionDurability

            </xsd:documentation>
          </xsd:annotation>
        </xsd:element>
        <xsd:element name="activation-config-property-value"
                     type="javaee:xsdStringType">
          <xsd:annotation>
           <xsd:documentation>

              The activation-config-property-value element
              contains the value for an activation configuration
              property of a message-driven bean.

            </xsd:documentation>
          </xsd:annotation>
        </xsd:element>
      </xsd:sequence>
      <xsd:attribute name="id" type="xsd:ID"/>
    </xsd:complexType>

  <!-- **************************************************** -->

    <xsd:complexType name="activation-configType">
      <xsd:annotation>
        <xsd:documentation>

          The activation-configType defines information about the
          expected configuration properties of the message-driven bean
          in its operational environment. This may include information
          about message acknowledgement, message selector, expected
          destination type, etc.

          The configuration information is expressed in terms of
          name/value configuration properties.

          The properties that are recognized for a particular
          message-driven bean are determined by the messaging type.

        </xsd:documentation>
      </xsd:annotation>

      <xsd:sequence>
        <xsd:element name="description"
                     type="javaee:descriptionType"
                     minOccurs="0"
                     maxOccurs="unbounded"/>
        <xsd:element name="activation-config-property"
                     type="javaee:activation-config-propertyType"
                     maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="id" type="xsd:ID"/>
    </xsd:complexType>

  <!-- **************************************************** -->

    <xsd:complexType name="application-exceptionType">
      <xsd:annotation>
```

```
      <xsd:documentation>

        The application-exceptionType declares an application
        exception. The declaration consists of:

            - the exception class. When the container receives
              an exception of this type, it is required to
              forward this exception as an applcation exception
              to the client regardless of whether it is a checked
              or unchecked exception.
            - an optional rollback element. If this element is
              set to true, the container must rollback the current
              transaction before forwarding the exception to the
              client.

      </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element name="exception-class"
                   type="javaee:fully-qualified-classType"/>
      <xsd:element name="rollback"
                   type="javaee:true-falseType"
                   minOccurs="0"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID"/>
  </xsd:complexType>

<!-- ********************************************** -->

  <xsd:complexType name="assembly-descriptorType">
    <xsd:annotation>
      <xsd:documentation>

        The assembly-descriptorType defines
        application-assembly information.

        The application-assembly information consists of the
        following parts: the definition of security roles, the
        definition of method permissions, the definition of
        transaction attributes for enterprise beans with
        container-managed transaction demarcation, the definition
        of interceptor bindings, a list of
        methods to be excluded from being invoked, and a list of
        exception types that should be treated as application exceptions.

        All the parts are optional in the sense that they are
        omitted if the lists represented by them are empty.

        Providing an assembly-descriptor in the deployment
        descriptor is optional for the ejb-jar file producer.

      </xsd:documentation>
    </xsd:annotation>

    <xsd:sequence>
      <xsd:element name="security-role"
                   type="javaee:security-roleType"
                   minOccurs="0"
                   maxOccurs="unbounded"/>
      <xsd:element name="method-permission"
                   type="javaee:method-permissionType"
                   minOccurs="0"
                   maxOccurs="unbounded"/>
      <xsd:element name="container-transaction"
                   type="javaee:container-transactionType"
                   minOccurs="0"
```

```
                    maxOccurs="unbounded"/>
        <xsd:element name="interceptor-binding"
                    type="javaee:interceptor-bindingType"
                    minOccurs="0"
                    maxOccurs="unbounded"/>
        <xsd:element name="message-destination"
                    type="javaee:message-destinationType"
                    minOccurs="0"
                    maxOccurs="unbounded"/>
        <xsd:element name="exclude-list"
                    type="javaee:exclude-listType"
                    minOccurs="0"/>
        <xsd:element name="application-exception"
                    type="javaee:application-exceptionType"
                    minOccurs="0"
                    maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="id" type="xsd:ID"/>
    </xsd:complexType>

  <!-- **************************************************** -->

    <xsd:complexType name="cmp-fieldType">
      <xsd:annotation>
        <xsd:documentation>

          The cmp-fieldType describes a container-managed field. The
          cmp-fieldType contains an optional description of the field,
          and the name of the field.

        </xsd:documentation>
      </xsd:annotation>
      <xsd:sequence>
        <xsd:element name="description"
                    type="javaee:descriptionType"
                    minOccurs="0"
                    maxOccurs="unbounded"/>
        <xsd:element name="field-name"
                    type="javaee:java-identifierType">
           <xsd:annotation>
            <xsd:documentation>

              The field-name element specifies the name of a
              container managed field.

              The name of the cmp-field of an entity bean with
              cmp-version 2.x must begin with a lowercase
              letter. This field is accessed by methods whose
              names consists of the name of the field specified by
              field-name in which the first letter is uppercased,
              prefixed by "get" or "set".

              The name of the cmp-field of an entity bean with
              cmp-version 1.x must denote a public field of the
              enterprise bean class or one of its superclasses.

            </xsd:documentation>
          </xsd:annotation>
        </xsd:element>
      </xsd:sequence>
      <xsd:attribute name="id" type="xsd:ID"/>
    </xsd:complexType>

  <!-- **************************************************** -->

    <xsd:complexType name="cmp-versionType">
```

```
    <xsd:annotation>
      <xsd:documentation>

        The cmp-versionType specifies the version of an entity bean
        with container-managed persistence. It is used by
        cmp-version elements.

        The value must be one of the two following:

            1.x
            2.x

      </xsd:documentation>
    </xsd:annotation>
    <xsd:simpleContent>
      <xsd:restriction base="javaee:string">
        <xsd:enumeration value="1.x"/>
        <xsd:enumeration value="2.x"/>
      </xsd:restriction>
    </xsd:simpleContent>
  </xsd:complexType>

<!-- **************************************************** -->

  <xsd:complexType name="cmr-field-typeType">
    <xsd:annotation>
      <xsd:documentation>

        The cmr-field-type element specifies the class of a
        collection-valued logical relationship field in the entity
        bean class. The value of an element using cmr-field-typeType
        must be either: java.util.Collection or java.util.Set.

      </xsd:documentation>
    </xsd:annotation>
    <xsd:simpleContent>
      <xsd:restriction base="javaee:string">
        <xsd:enumeration value="java.util.Collection"/>
        <xsd:enumeration value="java.util.Set"/>
      </xsd:restriction>
    </xsd:simpleContent>
  </xsd:complexType>

<!-- **************************************************** -->

  <xsd:complexType name="cmr-fieldType">
    <xsd:annotation>
      <xsd:documentation>

        The cmr-fieldType describes the bean provider's view of
        a relationship. It consists of an optional description, and
        the name and the class type of a field in the source of a
        role of a relationship. The cmr-field-name element
        corresponds to the name used for the get and set accessor
        methods for the relationship. The cmr-field-type element is
        used only for collection-valued cmr-fields. It specifies the
        type of the collection that is used.

      </xsd:documentation>
    </xsd:annotation>

    <xsd:sequence>
      <xsd:element name="description"
                   type="javaee:descriptionType"
                   minOccurs="0"
                   maxOccurs="unbounded"/>
```

```
      <xsd:element name="cmr-field-name"
                   type="javaee:string">
        <xsd:annotation>
          <xsd:documentation>

            The cmr-field-name element specifies the name of a
            logical relationship field in the entity bean
            class. The name of the cmr-field must begin with a
            lowercase letter. This field is accessed by methods
            whose names consist of the name of the field
            specified by cmr-field-name in which the first
            letter is uppercased, prefixed by "get" or "set".

          </xsd:documentation>
        </xsd:annotation>
      </xsd:element>
      <xsd:element name="cmr-field-type"
                   type="javaee:cmr-field-typeType"
                   minOccurs="0"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID"/>
  </xsd:complexType>

<!-- **************************************************** -->

  <xsd:complexType name="container-transactionType">
    <xsd:annotation>
      <xsd:documentation>

        The container-transactionType specifies how the container
        must manage transaction scopes for the enterprise bean's
        method invocations. It defines an optional description, a
        list of method elements, and a transaction attribute. The
        transaction attribute is to be applied to all the specified
        methods.

      </xsd:documentation>
    </xsd:annotation>

    <xsd:sequence>
      <xsd:element name="description"
                   type="javaee:descriptionType"
                   minOccurs="0"
                   maxOccurs="unbounded"/>
      <xsd:element name="method"
                   type="javaee:methodType"
                   maxOccurs="unbounded"/>
      <xsd:element name="trans-attribute"
                   type="javaee:trans-attributeType"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID"/>
  </xsd:complexType>

<!-- **************************************************** -->

  <xsd:complexType name="ejb-classType">
    <xsd:annotation>
      <xsd:documentation>
        <![CDATA[

          The ejb-classType contains the fully-qualified name of the
          enterprise bean's class. It is used by ejb-class elements.

          Example:

            <ejb-class>com.wombat.empl.EmployeeServiceBean</ejb-class>
```

```
            ]]>
          </xsd:documentation>
        </xsd:annotation>
        <xsd:simpleContent>
          <xsd:restriction base="javaee:fully-qualified-classType"/>
        </xsd:simpleContent>
      </xsd:complexType>

    <!-- **************************************************** -->

      <xsd:complexType name="ejb-jarType">
        <xsd:annotation>
          <xsd:documentation>

            The ejb-jarType defines the root element of the EJB
            deployment descriptor. It contains

                - an optional description of the ejb-jar file
                - an optional display name
                - an optional icon that contains a small and a large
                  icon file name
                - structural information about all included
                  enterprise beans that is not specified through
                  annotations
                - structural information about interceptor classes
                - a descriptor for container managed relationships,
                  if any.
                - an optional application-assembly descriptor
                - an optional name of an ejb-client-jar file for the
                  ejb-jar.

          </xsd:documentation>
        </xsd:annotation>

        <xsd:sequence>
          <xsd:group ref="javaee:descriptionGroup"/>
          <xsd:element name="enterprise-beans"
                       type="javaee:enterprise-beansType"
                       minOccurs="0"/>
          <xsd:element name="interceptors"
                       type="javaee:interceptorsType"
                       minOccurs="0"/>
          <xsd:element name="relationships"
                       type="javaee:relationshipsType"
                       minOccurs="0">
            <xsd:unique name="relationship-name-uniqueness">
              <xsd:annotation>
                <xsd:documentation>

                  The ejb-relation-name contains the name of a
                  relation. The name must be unique within
                  relationships.

                </xsd:documentation>
              </xsd:annotation>
              <xsd:selector xpath="javaee:ejb-relation"/>
              <xsd:field    xpath="javaee:ejb-relation-name"/>
            </xsd:unique>
          </xsd:element>
          <xsd:element name="assembly-descriptor"
                       type="javaee:assembly-descriptorType"
                       minOccurs="0">
            <xsd:annotation>
              <xsd:documentation>
```

```
            Providing an assembly-descriptor in the deployment
            descriptor is optional for the ejb-jar file
            producer.

          </xsd:documentation>
        </xsd:annotation>
      </xsd:element>
      <xsd:element name="ejb-client-jar"
                   type="javaee:pathType"
                   minOccurs="0">
        <xsd:annotation>
          <xsd:documentation>
            <![CDATA[

              The optional ejb-client-jar element specifies a JAR
              file that contains the class files necessary for a
              client program to access the
              enterprise beans in the ejb-jar file.

              Example:

                  <ejb-client-jar>employee_service_client.jar
                  </ejb-client-jar>

              ]]>
          </xsd:documentation>
        </xsd:annotation>
      </xsd:element>
    </xsd:sequence>
    <xsd:attribute name="version"
                   type="javaee:dewey-versionType"
                   fixed="3.0"
                   use="required">
      <xsd:annotation>
        <xsd:documentation>

          The version specifies the version of the
          EJB specification that the instance document must
          comply with. This information enables deployment tools
          to validate a particular EJB Deployment
          Descriptor with respect to a specific version of the EJB
          schema.

        </xsd:documentation>
      </xsd:annotation>
    </xsd:attribute>
    <xsd:attribute name="full" type="xsd:boolean">
      <xsd:annotation>
        <xsd:documentation>

          The full attribute defines whether this deployment
          descriptor is complete, or whether the class files
          of the jar file should be examined for annotations
          that specify deployment information.

          If full is set to "true", the deployment tool
          must ignore any EJB annotations present in the
          class files of the application.

          If full is not specified or is set to "false", the
          deployment tool must examine the class files of the
          application for annotations, as specified by the EJB
          specifications.

        </xsd:documentation>
      </xsd:annotation>
```

```
    </xsd:attribute>

    <xsd:attribute name="id" type="xsd:ID"/>
  </xsd:complexType>

<!-- **************************************************** -->

  <xsd:complexType name="ejb-nameType">
    <xsd:annotation>
      <xsd:documentation>
        <![CDATA[

          The ejb-nameType specifies an enterprise bean's name. It is
          used by ejb-name elements. This name is assigned by the
          ejb-jar file producer to name the enterprise bean in the
          ejb-jar file's deployment descriptor. The name must be
          unique among the names of the enterprise beans in the same
          ejb-jar file.

          There is no architected relationship between the used
          ejb-name in the deployment descriptor and the JNDI name that
          the Deployer will assign to the enterprise bean's home.

          The name for an entity bean must conform to the lexical
          rules for an NMTOKEN.

          Example:

          <ejb-name>EmployeeService</ejb-name>

          ]]>
      </xsd:documentation>
    </xsd:annotation>
    <xsd:simpleContent>
      <xsd:restriction base="javaee:xsdNMTOKENType"/>
    </xsd:simpleContent>
  </xsd:complexType>

<!-- **************************************************** -->

  <xsd:complexType name="ejb-relationType">
    <xsd:annotation>
      <xsd:documentation>

        The ejb-relationType describes a relationship between two
        entity beans with container-managed persistence.  It is used
        by ejb-relation elements. It contains a description; an
        optional ejb-relation-name element; and exactly two
        relationship role declarations, defined by the
        ejb-relationship-role elements. The name of the
        relationship, if specified, is unique within the ejb-jar
        file.

      </xsd:documentation>
    </xsd:annotation>

    <xsd:sequence>
      <xsd:element name="description"
                   type="javaee:descriptionType"
                   minOccurs="0"
                   maxOccurs="unbounded"/>
      <xsd:element name="ejb-relation-name"
                   type="javaee:string"
                   minOccurs="0">
        <xsd:annotation>
```

```
            <xsd:documentation>

              The ejb-relation-name element provides a unique name
              within the ejb-jar file for a relationship.

            </xsd:documentation>
          </xsd:annotation>
        </xsd:element>
        <xsd:element name="ejb-relationship-role"
                     type="javaee:ejb-relationship-roleType"/>
        <xsd:element name="ejb-relationship-role"
                     type="javaee:ejb-relationship-roleType"/>
      </xsd:sequence>
      <xsd:attribute name="id" type="xsd:ID"/>
    </xsd:complexType>

  <!-- **************************************************** -->

    <xsd:complexType name="ejb-relationship-roleType">
      <xsd:annotation>
        <xsd:documentation>
          <![CDATA[

            The ejb-relationship-roleType describes a role within a
            relationship. There are two roles in each relationship.

            The ejb-relationship-roleType contains an optional
            description; an optional name for the relationship role; a
            specification of the multiplicity of the role; an optional
            specification of cascade-delete functionality for the role;
            the role source; and a declaration of the cmr-field, if any,
            by means of which the other side of the relationship is
            accessed from the perspective of the role source.

            The multiplicity and role-source element are mandatory.

            The relationship-role-source element designates an entity
            bean by means of an ejb-name element. For bidirectional
            relationships, both roles of a relationship must declare a
            relationship-role-source element that specifies a cmr-field
            in terms of which the relationship is accessed. The lack of
            a cmr-field element in an ejb-relationship-role specifies
            that the relationship is unidirectional in navigability and
            the entity bean that participates in the relationship is
            "not aware" of the relationship.

            Example:

            <ejb-relation>
                <ejb-relation-name>Product-LineItem</ejb-relation-name>
                <ejb-relationship-role>
                    <ejb-relationship-role-name>product-has-lineitems
                    </ejb-relationship-role-name>
                    <multiplicity>One</multiplicity>
                    <relationship-role-source>
                    <ejb-name>ProductEJB</ejb-name>
                    </relationship-role-source>
                </ejb-relationship-role>
            </ejb-relation>

            ]]>
        </xsd:documentation>
      </xsd:annotation>

      <xsd:sequence>
        <xsd:element name="description"
```

```
                    type="javaee:descriptionType"
                    minOccurs="0"
                    maxOccurs="unbounded"/>
      <xsd:element name="ejb-relationship-role-name"
                    type="javaee:string"
                    minOccurs="0">
        <xsd:annotation>
          <xsd:documentation>

            The ejb-relationship-role-name element defines a
            name for a role that is unique within an
            ejb-relation. Different relationships can use the
            same name for a role.

          </xsd:documentation>
        </xsd:annotation>
      </xsd:element>
      <xsd:element name="multiplicity"
                    type="javaee:multiplicityType"/>
      <xsd:element name="cascade-delete"
                    type="javaee:emptyType"
                    minOccurs="0">
        <xsd:annotation>
          <xsd:documentation>

            The cascade-delete element specifies that, within a
            particular relationship, the lifetime of one or more
            entity beans is dependent upon the lifetime of
            another entity bean. The cascade-delete element can
            only be specified for an ejb-relationship-role
            element contained in an ejb-relation element in
            which the other ejb-relationship-role
            element specifies a multiplicity of One.

          </xsd:documentation>
        </xsd:annotation>
      </xsd:element>
      <xsd:element name="relationship-role-source"
                    type="javaee:relationship-role-sourceType"/>
      <xsd:element name="cmr-field"
                    type="javaee:cmr-fieldType"
                    minOccurs="0"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID"/>
  </xsd:complexType>

<!-- ********************************************** -->

  <xsd:complexType name="enterprise-beansType">
    <xsd:annotation>
      <xsd:documentation>

        The enterprise-beansType declares one or more enterprise
        beans. Each bean can be a session, entity or message-driven
        bean.

      </xsd:documentation>
    </xsd:annotation>

    <xsd:choice maxOccurs="unbounded">
      <xsd:element name="session"
                    type="javaee:session-beanType">
        <xsd:unique name="session-ejb-local-ref-name-uniqueness">
          <xsd:annotation>
            <xsd:documentation>
```

```
        The ejb-ref-name element contains the name of
        an EJB reference. The EJB reference is an entry in
        the component's environment and is relative to the
        java:comp/env context.  The name must be unique within
        the component.

        It is recommended that name be prefixed with "ejb/".

      </xsd:documentation>
    </xsd:annotation>
    <xsd:selector xpath="javaee:ejb-local-ref"/>
    <xsd:field    xpath="javaee:ejb-ref-name"/>
  </xsd:unique>

  <xsd:unique name="session-ejb-ref-name-uniqueness">
    <xsd:annotation>
      <xsd:documentation>

        The ejb-ref-name element contains the name of an EJB
        reference. The EJB reference is an entry in the
        component's environment and is relative to the
        java:comp/env context. The name must be unique
        within the component.

        It is recommended that name is prefixed with "ejb/".

      </xsd:documentation>
    </xsd:annotation>
    <xsd:selector xpath="javaee:ejb-ref"/>
    <xsd:field    xpath="javaee:ejb-ref-name"/>
  </xsd:unique>

  <xsd:unique name="session-resource-env-ref-uniqueness">
    <xsd:annotation>
      <xsd:documentation>

        The resource-env-ref-name element specifies the name
        of a resource environment reference; its value is
        the environment entry name used in the component
        code. The name is a JNDI name relative to the
        java:comp/env context and must be unique within an
        component.

      </xsd:documentation>
    </xsd:annotation>
    <xsd:selector xpath="javaee:resource-env-ref"/>
    <xsd:field    xpath="javaee:resource-env-ref-name"/>
  </xsd:unique>

  <xsd:unique name="session-message-destination-ref-uniqueness">
    <xsd:annotation>
      <xsd:documentation>

        The message-destination-ref-name element specifies the name
        of a message destination reference; its value is
        the message destination reference name used in the component
        code. The name is a JNDI name relative to the
        java:comp/env context and must be unique within an
        component.

      </xsd:documentation>
    </xsd:annotation>
    <xsd:selector xpath="javaee:message-destination-ref"/>
    <xsd:field    xpath="javaee:message-destination-ref-name"/>
  </xsd:unique>
```

```
    <xsd:unique name="session-res-ref-name-uniqueness">
      <xsd:annotation>
        <xsd:documentation>

          The res-ref-name element specifies the name of a
          resource manager connection factory reference.  The name
          is a JNDI name relative to the java:comp/env context.
          The name must be unique within an component.

        </xsd:documentation>
      </xsd:annotation>
      <xsd:selector xpath="javaee:resource-ref"/>
      <xsd:field    xpath="javaee:res-ref-name"/>
    </xsd:unique>

    <xsd:unique name="session-env-entry-name-uniqueness">
      <xsd:annotation>
        <xsd:documentation>

          The env-entry-name element contains the name of a
          component's environment entry.  The name is a JNDI
          name relative to the java:comp/env context.  The
          name must be unique within an component.

        </xsd:documentation>
      </xsd:annotation>
      <xsd:selector xpath="javaee:env-entry"/>
      <xsd:field    xpath="javaee:env-entry-name"/>
    </xsd:unique>
  </xsd:element>

  <xsd:element name="entity"
               type="javaee:entity-beanType">
    <xsd:unique name="entity-ejb-local-ref-name-uniqueness">
      <xsd:annotation>
        <xsd:documentation>

          The ejb-ref-name element contains the name of
          an EJB reference. The EJB reference is an entry in
          the component's environment and is relative to the
          java:comp/env context.  The name must be unique within
          the component.

          It is recommended that name be prefixed with "ejb/".

        </xsd:documentation>
      </xsd:annotation>
      <xsd:selector xpath="javaee:ejb-local-ref"/>
      <xsd:field    xpath="javaee:ejb-ref-name"/>
    </xsd:unique>

    <xsd:unique name="entity-ejb-ref-name-uniqueness">
      <xsd:annotation>
        <xsd:documentation>

          The ejb-ref-name element contains the name of an EJB
          reference. The EJB reference is an entry in the
          component's environment and is relative to the
          java:comp/env context. The name must be unique
          within the component.

          It is recommended that name is prefixed with "ejb/".

        </xsd:documentation>
      </xsd:annotation>
      <xsd:selector xpath="javaee:ejb-ref"/>
```

```
  <xsd:field    xpath="javaee:ejb-ref-name"/>
</xsd:unique>

<xsd:unique name="entity-resource-env-ref-uniqueness">
  <xsd:annotation>
    <xsd:documentation>

      The resource-env-ref-name element specifies the name
      of a resource environment reference; its value is
      the environment entry name used in the component
      code. The name is a JNDI name relative to the
      java:comp/env context and must be unique within an
      component.

    </xsd:documentation>
  </xsd:annotation>
  <xsd:selector xpath="javaee:resource-env-ref"/>
  <xsd:field    xpath="javaee:resource-env-ref-name"/>
</xsd:unique>

<xsd:unique name="entity-message-destination-ref-uniqueness">
  <xsd:annotation>
    <xsd:documentation>

      The message-destination-ref-name element specifies the name
      of a message destination reference; its value is
      the message destination reference name used in the component
      code. The name is a JNDI name relative to the
      java:comp/env context and must be unique within an
      component.

    </xsd:documentation>
  </xsd:annotation>
  <xsd:selector xpath="javaee:message-destination-ref"/>
  <xsd:field    xpath="javaee:message-destination-ref-name"/>
</xsd:unique>

<xsd:unique name="entity-res-ref-name-uniqueness">
  <xsd:annotation>
    <xsd:documentation>

      The res-ref-name element specifies the name of a
      resource manager connection factory reference.  The name
      is a JNDI name relative to the java:comp/env context.
      The name must be unique within an component.

    </xsd:documentation>
  </xsd:annotation>
  <xsd:selector xpath="javaee:resource-ref"/>
  <xsd:field    xpath="javaee:res-ref-name"/>
</xsd:unique>

<xsd:unique name="entity-env-entry-name-uniqueness">
  <xsd:annotation>
    <xsd:documentation>

      The env-entry-name element contains the name of a
      component's environment entry.  The name is a JNDI
      name relative to the java:comp/env context.  The
      name must be unique within an component.

    </xsd:documentation>
  </xsd:annotation>
  <xsd:selector xpath="javaee:env-entry"/>
  <xsd:field    xpath="javaee:env-entry-name"/>
</xsd:unique>
```

```
</xsd:element>

<xsd:element name="message-driven"
             type="javaee:message-driven-beanType">
  <xsd:unique name="messaged-ejb-local-ref-name-uniqueness">
    <xsd:annotation>
      <xsd:documentation>

        The ejb-ref-name element contains the name of
        an EJB reference. The EJB reference is an entry in
        the component's environment and is relative to the
        java:comp/env context.  The name must be unique within
        the component.

        It is recommended that name be prefixed with "ejb/".

      </xsd:documentation>
    </xsd:annotation>
    <xsd:selector xpath="javaee:ejb-local-ref"/>
    <xsd:field    xpath="javaee:ejb-ref-name"/>
  </xsd:unique>

  <xsd:unique name="messaged-ejb-ref-name-uniqueness">
    <xsd:annotation>
      <xsd:documentation>

        The ejb-ref-name element contains the name of an EJB
        reference. The EJB reference is an entry in the
        component's environment and is relative to the
        java:comp/env context. The name must be unique
        within the component.

        It is recommended that name is prefixed with "ejb/".

      </xsd:documentation>
    </xsd:annotation>
    <xsd:selector xpath="javaee:ejb-ref"/>
    <xsd:field    xpath="javaee:ejb-ref-name"/>
  </xsd:unique>

  <xsd:unique name="messaged-resource-env-ref-uniqueness">
    <xsd:annotation>
      <xsd:documentation>

        The resource-env-ref-name element specifies the name
        of a resource environment reference; its value is
        the environment entry name used in the component
        code. The name is a JNDI name relative to the
        java:comp/env context and must be unique within an
        component.

      </xsd:documentation>
    </xsd:annotation>
    <xsd:selector xpath="javaee:resource-env-ref"/>
    <xsd:field    xpath="javaee:resource-env-ref-name"/>
  </xsd:unique>

  <xsd:unique name="messaged-message-destination-ref-uniqueness">
    <xsd:annotation>
      <xsd:documentation>

        The message-destination-ref-name element specifies the name
        of a message destination reference; its value is
        the message destination reference name used in the component
        code. The name is a JNDI name relative to the
        java:comp/env context and must be unique within an
```

```
                component.

              </xsd:documentation>
            </xsd:annotation>
            <xsd:selector xpath="javaee:message-destination-ref"/>
            <xsd:field    xpath="javaee:message-destination-ref-name"/>
          </xsd:unique>

          <xsd:unique name="messaged-res-ref-name-uniqueness">
            <xsd:annotation>
              <xsd:documentation>

                The res-ref-name element specifies the name of a
                resource manager connection factory reference.  The name
                is a JNDI name relative to the java:comp/env context.
                The name must be unique within an component.

              </xsd:documentation>
            </xsd:annotation>
            <xsd:selector xpath="javaee:resource-ref"/>
            <xsd:field    xpath="javaee:res-ref-name"/>
          </xsd:unique>

          <xsd:unique name="messaged-env-entry-name-uniqueness">
            <xsd:annotation>
              <xsd:documentation>

                The env-entry-name element contains the name of a
                component's environment entry.  The name is a JNDI
                name relative to the java:comp/env context.  The
                name must be unique within an component.

              </xsd:documentation>
            </xsd:annotation>
            <xsd:selector xpath="javaee:env-entry"/>
            <xsd:field    xpath="javaee:env-entry-name"/>
          </xsd:unique>
        </xsd:element>

      </xsd:choice>
      <xsd:attribute name="id" type="xsd:ID"/>
    </xsd:complexType>

  <!-- **************************************************** -->

    <xsd:complexType name="entity-beanType">
      <xsd:annotation>
        <xsd:documentation>

          The entity-beanType declares an entity bean. The declaration
          consists of:

              - an optional description
              - an optional display name
              - an optional icon element that contains a small and a large
                icon file name
              - a unique name assigned to the enterprise bean
                in the deployment descriptor
              - an optional mapped-name element that can be used to provide
                vendor-specific deployment information such as the physical
                jndi-name of the entity bean's remote home interface. This
                element is not required to be supported by all implementations.
                Any use of this element is non-portable.
              - the names of the entity bean's remote home
                and remote interfaces, if any
              - the names of the entity bean's local home and local
```

```
          interfaces, if any
        - the entity bean's implementation class
        - the optional entity bean's persistence management type. If
          this element is not specified it is defaulted to Container.
        - the entity bean's primary key class name
        - an indication of the entity bean's reentrancy
        - an optional specification of the
          entity bean's cmp-version
        - an optional specification of the entity bean's
          abstract schema name
        - an optional list of container-managed fields
        - an optional specification of the primary key
          field
        - an optional declaration of the bean's environment
          entries
        - an optional declaration of the bean's EJB
          references
        - an optional declaration of the bean's local
          EJB references
        - an optional declaration of the bean's web
          service references
        - an optional declaration of the security role
          references
        - an optional declaration of the security identity
          to be used for the execution of the bean's methods
        - an optional declaration of the bean's
          resource manager connection factory references
        - an optional declaration of the bean's
          resource environment references
        - an optional declaration of the bean's message
          destination references
        - an optional set of query declarations
          for finder and select methods for an entity
          bean with cmp-version 2.x.

      The optional abstract-schema-name element must be specified
      for an entity bean with container-managed persistence and
      cmp-version 2.x.

      The optional primkey-field may be present in the descriptor
      if the entity's persistence-type is Container.

      The optional cmp-version element may be present in the
      descriptor if the entity's persistence-type is Container. If
      the persistence-type is Container and the cmp-version
      element is not specified, its value defaults to 2.x.

      The optional home and remote elements must be specified if
      the entity bean cmp-version is 1.x.

      The optional home and remote elements must be specified if
      the entity bean has a remote home and remote interface.

      The optional local-home and local elements must be specified
      if the entity bean has a local home and local interface.

      Either both the local-home and the local elements or both
      the home and the remote elements must be specified.

      The optional query elements must be present if the
      persistence-type is Container and the cmp-version is 2.x and
      query methods other than findByPrimaryKey have been defined
      for the entity bean.

      The other elements that are optional are "optional" in the
      sense that they are omitted if the lists represented by them
```

```
      are empty.

      At least one cmp-field element must be present in the
      descriptor if the entity's persistence-type is Container and
      the cmp-version is 1.x, and none must not be present if the
      entity's persistence-type is Bean.

    </xsd:documentation>
  </xsd:annotation>

  <xsd:sequence>
    <xsd:group ref="javaee:descriptionGroup"/>
    <xsd:element name="ejb-name"
                 type="javaee:ejb-nameType"/>
    <xsd:element name="mapped-name"
                 type="javaee:xsdStringType"
                 minOccurs="0"/>
    <xsd:element name="home"
                 type="javaee:homeType"
                 minOccurs="0"/>
    <xsd:element name="remote"
                 type="javaee:remoteType"
                 minOccurs="0"/>
    <xsd:element name="local-home"
                 type="javaee:local-homeType"
                 minOccurs="0"/>
    <xsd:element name="local"
                 type="javaee:localType"
                 minOccurs="0"/>
    <xsd:element name="ejb-class"
                 type="javaee:ejb-classType"/>
    <xsd:element name="persistence-type"
                 type="javaee:persistence-typeType"/>
    <xsd:element name="prim-key-class"
                 type="javaee:fully-qualified-classType">
      <xsd:annotation>
        <xsd:documentation>

          The prim-key-class element contains the
          fully-qualified name of an
          entity bean's primary key class.

          If the definition of the primary key class is
          deferred to deployment time, the prim-key-class
          element should specify java.lang.Object.

        </xsd:documentation>
      </xsd:annotation>
    </xsd:element>
    <xsd:element name="reentrant"
                 type="javaee:true-falseType">
      <xsd:annotation>
        <xsd:documentation>

          The reentrant element specifies whether an entity
          bean is reentrant or not.

          The reentrant element must be one of the two
          following: true or false

        </xsd:documentation>
      </xsd:annotation>
    </xsd:element>
    <xsd:element name="cmp-version"
                 type="javaee:cmp-versionType"
                 minOccurs="0"/>
```

```
      <xsd:element name="abstract-schema-name"
                   type="javaee:java-identifierType"
                   minOccurs="0">
        <xsd:annotation>
          <xsd:documentation>

            The abstract-schema-name element specifies the name
            of the abstract schema type of an entity bean with
            cmp-version 2.x. It is used in EJB QL queries.

            For example, the abstract-schema-name for an entity
            bean whose local interface is
            com.acme.commerce.Order might be Order.

          </xsd:documentation>
        </xsd:annotation>
      </xsd:element>
      <xsd:element name="cmp-field"
                   type="javaee:cmp-fieldType"
                   minOccurs="0"
                   maxOccurs="unbounded"/>
      <xsd:element name="primkey-field"
                   type="javaee:string"
                   minOccurs="0">
        <xsd:annotation>
          <xsd:documentation>

            The primkey-field element is used to specify the
            name of the primary key field for an entity with
            container-managed persistence.

            The primkey-field must be one of the fields declared
            in the cmp-field element, and the type of the field
            must be the same as the primary key type.

            The primkey-field element is not used if the primary
            key maps to multiple container-managed fields
            (i.e. the key is a compound key). In this case, the
            fields of the primary key class must be public, and
            their names must correspond to the field names of
            the entity bean class that comprise the key.

          </xsd:documentation>
        </xsd:annotation>
      </xsd:element>
      <xsd:group ref="javaee:jndiEnvironmentRefsGroup"/>
      <xsd:element name="security-role-ref"
                   type="javaee:security-role-refType"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="security-identity"
                   type="javaee:security-identityType"
                   minOccurs="0"/>
      <xsd:element name="query"
                   type="javaee:queryType"
                   minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID"/>
  </xsd:complexType>

<!-- ********************************************** -->

  <xsd:complexType name="exclude-listType">
    <xsd:annotation>
      <xsd:documentation>

        The exclude-listType specifies one or more methods which
```

```
            the Assembler marks to be uncallable.

            If the method permission relation contains methods that are
            in the exclude list, the Deployer should consider those
            methods to be uncallable.

          </xsd:documentation>
        </xsd:annotation>

        <xsd:sequence>
          <xsd:element name="description"
                       type="javaee:descriptionType"
                       minOccurs="0"
                       maxOccurs="unbounded"/>
          <xsd:element name="method"
                       type="javaee:methodType"
                       maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="id" type="xsd:ID"/>
      </xsd:complexType>

    <!-- **************************************************** -->

      <xsd:complexType name="init-methodType">
        <xsd:sequence>
          <xsd:element name="create-method"
                       type="javaee:named-methodType"/>
          <xsd:element name="bean-method"
                       type="javaee:named-methodType"/>
        </xsd:sequence>
        <xsd:attribute name="id" type="xsd:ID"/>
      </xsd:complexType>

    <!-- **************************************************** -->

      <xsd:complexType name="interceptor-bindingType">
        <xsd:annotation>
          <xsd:documentation>

            The interceptor-bindingType element describes the binding of
            interceptor classes to beans within the ejb-jar.
            It consists of :

              - An optional description.
              - The name of an ejb within the ejb-jar or the wildcard value "*",
                which is used to define interceptors that are bound to all
                beans in the ejb-jar.
              - A list of interceptor classes that are bound to the contents of
                the ejb-name element or a specification of the total ordering
                over the interceptors defined for the given level and above.
             - An optional exclude-default-interceptors element.  If set to true,
                specifies that default interceptors are not to be applied to
                a bean-class and/or business method.
              - An optional exclude-class-interceptors element.  If set to true,
                specifies that class interceptors are not to be applied to
                a business method.
             - An optional set of method elements for describing the name/params
                of a method-level interceptor.

             Interceptors bound to all classes using the wildcard syntax
             "*" are default interceptors for the components in the ejb-jar.
             In addition, interceptors may be bound at the level of the bean
             class (class-level interceptors) or business methods (method-level
             interceptors ).

             The binding of interceptors to classes is additive.  If interceptors
```

```
are bound at the class-level and/or default-level as well as the
method-level, both class-level and/or default-level as well as
method-level will apply.

There are four possible styles of the interceptor element syntax :

1.
<interceptor-binding>
  <ejb-name>*</ejb-name>
  <interceptor-class>INTERCEPTOR</interceptor-class>
</interceptor-binding>

Specifying the ejb-name as the wildcard value "*" designates
default interceptors (interceptors that apply to all session and
message-driven beans contained in the ejb-jar).

2.
<interceptor-binding>
  <ejb-name>EJBNAME</ejb-name>
  <interceptor-class>INTERCEPTOR</interceptor-class>
</interceptor-binding>

This style is used to refer to interceptors associated with the
specified enterprise bean(class-level interceptors).

3.
<interceptor-binding>
  <ejb-name>EJBNAME</ejb-name>
  <interceptor-class>INTERCEPTOR</interceptor-class>
  <method>
    <method-name>METHOD</method-name>
  </method>
</interceptor-binding>

This style is used to associate a method-level interceptor with
the specified enterprise bean.  If there are multiple methods
with the same overloaded name, the element of this style refers
to all the methods with the overloaded name.  Method-level
interceptors can only be associated with business methods of the
bean class.   Note that the wildcard value "*" cannot be used
to specify method-level interceptors.

4.
<interceptor-binding>
  <ejb-name>EJBNAME</ejb-name>
  <interceptor-class>INTERCEPTOR</interceptor-class>
  <method>
    <method-name>METHOD</method-name>
    <method-params>
      <method-param>PARAM-1</method-param>
      <method-param>PARAM-2</method-param>
      ...
      <method-param>PARAM-N</method-param>
    </method-params>
  </method>
</interceptor-binding>

This style is used to associate a method-level interceptor with
the specified method of the specified enterprise bean.  This
style is used to refer to a single method within a set of methods
with an overloaded name.  The values PARAM-1 through PARAM-N
are the fully-qualified Java types of the method's input parameters
(if the method has no input arguments, the method-params element
contains no method-param elements). Arrays are specified by the
array element's type, followed by one or more pair of square
brackets (e.g. int[][]).
```

```
      </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element name="description"
                   type="javaee:descriptionType"
                   minOccurs="0"
                   maxOccurs="unbounded"/>
      <xsd:element name="ejb-name"
                   type="javaee:string"/>
      <xsd:choice>
        <xsd:element name="interceptor-class"
                     type="javaee:fully-qualified-classType"
                     minOccurs="1"
                     maxOccurs="unbounded"/>
        <xsd:element name="interceptor-order"
                     type="javaee:interceptor-orderType"
                     minOccurs="1"/>
      </xsd:choice>
      <xsd:element name="exclude-default-interceptors"
                   type="javaee:true-falseType"
                   minOccurs="0"/>
      <xsd:element name="exclude-class-interceptors"
                   type="javaee:true-falseType"
                   minOccurs="0"/>
      <xsd:element name="method"
                   type="javaee:named-methodType"
                   minOccurs="0"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID"/>
  </xsd:complexType>

<!-- **************************************************** -->

  <xsd:complexType name="interceptor-orderType">
    <xsd:annotation>
      <xsd:documentation>
        The interceptor-orderType element describes a total ordering
        of interceptor classes.
      </xsd:documentation>
    </xsd:annotation>

    <xsd:sequence>

      <xsd:element name="interceptor-class"
                   type="javaee:fully-qualified-classType"
                   minOccurs="1"
                   maxOccurs="unbounded"/>

    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID"/>
  </xsd:complexType>

<!-- **************************************************** -->

  <xsd:complexType name="interceptorType">
    <xsd:annotation>
      <xsd:documentation>

        The interceptorType element declares information about a single
        interceptor class.  It consists of :

            - An optional description.
            - The fully-qualified name of the interceptor class.
            - An optional list of around invoke methods declared on the
              interceptor class and/or its super-classes.
```

```
                - An optional list environment dependencies for the interceptor
                  class and/or its super-classes.
                - An optional list of post-activate methods declared on the
                  interceptor class and/or its super-classes.
                - An optional list of pre-passivate methods declared on the
                  interceptor class and/or its super-classes.

            </xsd:documentation>
          </xsd:annotation>
          <xsd:sequence>
            <xsd:element name="description"
                         type="javaee:descriptionType"
                         minOccurs="0"
                         maxOccurs="unbounded"/>
            <xsd:element name="interceptor-class"
                         type="javaee:fully-qualified-classType"/>
            <xsd:element name="around-invoke-method"
                         type="javaee:lifecycle-callbackType"
                         minOccurs="0"
                         maxOccurs="unbounded"/>
            <xsd:group ref="javaee:jndiEnvironmentRefsGroup"/>
            <xsd:element name="post-activate-method"
                         type="javaee:lifecycle-callbackType"
                         minOccurs="0"
                         maxOccurs="unbounded"/>
            <xsd:element name="pre-passivate-method"
                         type="javaee:lifecycle-callbackType"
                         minOccurs="0"
                         maxOccurs="unbounded"/>
          </xsd:sequence>
          <xsd:attribute name="id" type="xsd:ID"/>
        </xsd:complexType>

      <!-- **************************************************** -->

        <xsd:complexType name="interceptorsType">
          <xsd:annotation>
            <xsd:documentation>

              The interceptorsType element declares one or more interceptor
              classes used by components within this ejb-jar.  The declaration
              consists of :

                  - An optional description.
                  - One or more interceptor elements.

            </xsd:documentation>
          </xsd:annotation>
          <xsd:sequence>
            <xsd:element name="description"
                         type="javaee:descriptionType"
                         minOccurs="0"
                         maxOccurs="unbounded"/>
            <xsd:element name="interceptor"
                         type="javaee:interceptorType"
                         maxOccurs="unbounded"/>
          </xsd:sequence>
          <xsd:attribute name="id" type="xsd:ID"/>
        </xsd:complexType>

      <!-- **************************************************** -->

        <xsd:complexType name="message-driven-beanType">
          <xsd:annotation>
            <xsd:documentation>
```

```
        The message-driven element declares a message-driven
        bean. The declaration consists of:

            - an optional description
            - an optional display name
            - an optional icon element that contains a small and a large
              icon file name.
            - a name assigned to the enterprise bean in
              the deployment descriptor
            - an optional mapped-name element that can be used to provide
              vendor-specific deployment information such as the physical
              jndi-name of destination from which this message-driven bean
              should consume.  This element is not required to be supported
             by all implementations.  Any use of this element is non-portable.
            - the message-driven bean's implementation class
            - an optional declaration of the bean's messaging
              type
            - an optional declaration of the bean's timeout method.
            - the optional message-driven bean's transaction management
              type. If it is not defined, it is defaulted to Container.
            - an optional declaration of the bean's
              message-destination-type
            - an optional declaration of the bean's
              message-destination-link
            - an optional declaration of the message-driven bean's
              activation configuration properties
            - an optional list of the message-driven bean class and/or
              superclass around-invoke methods.
            - an optional declaration of the bean's environment
              entries
            - an optional declaration of the bean's EJB references
            - an optional declaration of the bean's local EJB
              references
            - an optional declaration of the bean's web service
              references
            - an optional declaration of the security
              identity to be used for the execution of the bean's
              methods
            - an optional declaration of the bean's
              resource manager connection factory
              references
            - an optional declaration of the bean's resource
              environment references.
            - an optional declaration of the bean's message
              destination references

      </xsd:documentation>
    </xsd:annotation>

    <xsd:sequence>
      <xsd:group ref="javaee:descriptionGroup"/>
      <xsd:element name="ejb-name"
                   type="javaee:ejb-nameType"/>
      <xsd:element name="mapped-name"
                   type="javaee:xsdStringType"
                   minOccurs="0"/>
      <xsd:element name="ejb-class"
                   type="javaee:ejb-classType"/>
      <xsd:element name="messaging-type"
                   type="javaee:fully-qualified-classType"
                   minOccurs="0">
        <xsd:annotation>
          <xsd:documentation>

            The messaging-type element specifies the message
            listener interface of the message-driven bean. If
```

```
                the messaging-type element is not specified, it is
                assumed to be javax.jms.MessageListener.

              </xsd:documentation>
            </xsd:annotation>
          </xsd:element>
          <xsd:element name="timeout-method"
                       type="javaee:named-methodType"
                       minOccurs="0"/>
          <xsd:element name="transaction-type"
                       type="javaee:transaction-typeType"
                       minOccurs="0"/>
          <xsd:element name="message-destination-type"
                       type="javaee:message-destination-typeType"
                       minOccurs="0"/>
          <xsd:element name="message-destination-link"
                       type="javaee:message-destination-linkType"
                       minOccurs="0"/>
          <xsd:element name="activation-config"
                       type="javaee:activation-configType"
                       minOccurs="0"/>
          <xsd:element name="around-invoke-method"
                       type="javaee:lifecycle-callbackType"
                       minOccurs="0"
                       maxOccurs="unbounded"/>
          <xsd:group ref="javaee:jndiEnvironmentRefsGroup"/>
          <xsd:element name="security-identity"
                       type="javaee:security-identityType"
                       minOccurs="0"/>
        </xsd:sequence>
        <xsd:attribute name="id" type="xsd:ID"/>
      </xsd:complexType>

    <!-- **************************************************** -->

      <xsd:complexType name="method-intfType">

        <xsd:annotation>
          <xsd:documentation>

            The method-intf element allows a method element to
            differentiate between the methods with the same name and
            signature that are multiply defined across the home and
            component interfaces (e.g, in both an enterprise bean's
            remote and local interfaces or in both an enterprise bean's
            home and remote interfaces, etc.); the component and web
            service endpoint interfaces, and so on. The Local applies to
            both local component interface and local business interface.
            Similarly, Remote applies to both remote component interface
            and the remote business interface.

            The method-intf element must be one of the following:

                Home
                Remote
                LocalHome
                Local
                ServiceEndpoint

          </xsd:documentation>
        </xsd:annotation>
        <xsd:simpleContent>
          <xsd:restriction base="javaee:string">
            <xsd:enumeration value="Home"/>
            <xsd:enumeration value="Remote"/>
            <xsd:enumeration value="LocalHome"/>
```

```
        <xsd:enumeration value="Local"/>
        <xsd:enumeration value="ServiceEndpoint"/>
      </xsd:restriction>
    </xsd:simpleContent>
  </xsd:complexType>

<!-- ********************************************** -->

  <xsd:complexType name="method-nameType">
    <xsd:annotation>
      <xsd:documentation>

        The method-nameType contains a name of an enterprise
        bean method or the asterisk (*) character. The asterisk is
        used when the element denotes all the methods of an
        enterprise bean's client view interfaces.

      </xsd:documentation>
    </xsd:annotation>
    <xsd:simpleContent>
      <xsd:restriction base="javaee:string"/>
    </xsd:simpleContent>
  </xsd:complexType>

<!-- ********************************************** -->

  <xsd:complexType name="method-paramsType">
    <xsd:annotation>
      <xsd:documentation>

        The method-paramsType defines a list of the
        fully-qualified Java type names of the method parameters.

      </xsd:documentation>
    </xsd:annotation>

    <xsd:sequence>
      <xsd:element name="method-param"
                   type="javaee:java-typeType"
                   minOccurs="0"
                   maxOccurs="unbounded">
        <xsd:annotation>
          <xsd:documentation>

            The method-param element contains a primitive
            or a fully-qualified Java type name of a method
            parameter.

          </xsd:documentation>
        </xsd:annotation>
      </xsd:element>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID"/>
  </xsd:complexType>

<!-- ********************************************** -->

  <xsd:complexType name="method-permissionType">
    <xsd:annotation>
      <xsd:documentation>

        The method-permissionType specifies that one or more
        security roles are allowed to invoke one or more enterprise
        bean methods. The method-permissionType consists of an
        optional description, a list of security role names or an
        indicator to state that the method is unchecked for
```

```
            authorization, and a list of method elements.

            The security roles used in the method-permissionType
            must be defined in the security-role elements of the
            deployment descriptor, and the methods must be methods
            defined in the enterprise bean's business, home, component
            and/or web service endpoint interfaces.

          </xsd:documentation>
        </xsd:annotation>

        <xsd:sequence>
          <xsd:element name="description"
                       type="javaee:descriptionType"
                       minOccurs="0"
                       maxOccurs="unbounded"/>
          <xsd:choice>
            <xsd:element name="role-name"
                         type="javaee:role-nameType"
                         maxOccurs="unbounded"/>
            <xsd:element name="unchecked"
                         type="javaee:emptyType">
              <xsd:annotation>
                <xsd:documentation>

                  The unchecked element specifies that a method is
                  not checked for authorization by the container
                  prior to invocation of the method.

                </xsd:documentation>
              </xsd:annotation>
            </xsd:element>
          </xsd:choice>
          <xsd:element name="method"
                       type="javaee:methodType"
                       maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="id" type="xsd:ID"/>
      </xsd:complexType>

    <!-- ********************************************************** -->

      <xsd:complexType name="methodType">
        <xsd:annotation>
          <xsd:documentation>
            <![CDATA[

              The methodType is used to denote a method of an enterprise
              bean's business, home, component, and/or web service endpoint
              interface, or, in the case of a message-driven bean, the
              bean's message listener method, or a set of such
              methods. The ejb-name element must be the name of one of the
              enterprise beans declared in the deployment descriptor; the
              optional method-intf element allows to distinguish between a
              method with the same signature that is multiply defined
              across the business, home, component, and/or web service
              endpoint nterfaces; the method-name element specifies the
              method name; and the optional method-params elements identify
              a single method among multiple methods with an overloaded
              method name.

              There are three possible styles of using methodType element
              within a method element:

              1.
              <method>
```

```
    <ejb-name>EJBNAME</ejb-name>
    <method-name>*</method-name>
</method>

   This style is used to refer to all the methods of the
   specified enterprise bean's business, home, component,
   and/or web service endpoint interfaces.

2.
<method>
    <ejb-name>EJBNAME</ejb-name>
    <method-name>METHOD</method-name>
</method>

   This style is used to refer to the specified method of
   the specified enterprise bean. If there are multiple
   methods with the same overloaded name, the element of
   this style refers to all the methods with the overloaded
   name.

3.
<method>
    <ejb-name>EJBNAME</ejb-name>
    <method-name>METHOD</method-name>
    <method-params>
        <method-param>PARAM-1</method-param>
        <method-param>PARAM-2</method-param>
        ...
        <method-param>PARAM-n</method-param>
    </method-params>
</method>

   This style is used to refer to a single method within a
   set of methods with an overloaded name. PARAM-1 through
   PARAM-n are the fully-qualified Java types of the
   method's input parameters (if the method has no input
   arguments, the method-params element contains no
   method-param elements). Arrays are specified by the
   array element's type, followed by one or more pair of
   square brackets (e.g. int[][]). If there are multiple
   methods with the same overloaded name, this style refers
   to all of the overloaded methods.

Examples:

Style 1: The following method element refers to all the
methods of the EmployeeService bean's business, home,
component, and/or web service endpoint interfaces:

<method>
    <ejb-name>EmployeeService</ejb-name>
    <method-name>*</method-name>
</method>

Style 2: The following method element refers to all the
create methods of the EmployeeService bean's home
interface(s).

<method>
    <ejb-name>EmployeeService</ejb-name>
    <method-name>create</method-name>
</method>

Style 3: The following method element refers to the
create(String firstName, String LastName) method of the
EmployeeService bean's home interface(s).
```

```
<method>
    <ejb-name>EmployeeService</ejb-name>
    <method-name>create</method-name>
    <method-params>
        <method-param>java.lang.String</method-param>
        <method-param>java.lang.String</method-param>
    </method-params>
</method>
```

The following example illustrates a Style 3 element with more complex parameter types. The method foobar(char s, int i, int[] iar, mypackage.MyClass mycl, mypackage.MyClass[][] myclaar) would be specified as:

```
<method>
    <ejb-name>EmployeeService</ejb-name>
    <method-name>foobar</method-name>
    <method-params>
        <method-param>char</method-param>
        <method-param>int</method-param>
        <method-param>int[]</method-param>
        <method-param>mypackage.MyClass</method-param>
        <method-param>mypackage.MyClass[][]</method-param>
    </method-params>
</method>
```

The optional method-intf element can be used when it becomes necessary to differentiate between a method that is multiply defined across the enterprise bean's business, home, component, and/or web service endpoint interfaces with the same name and signature. However, if the same method is a method of both the local business interface, and the local component interface, the same attribute applies to the method for both interfaces. Likewise, if the same method is a method of both the remote business interface and the remote component interface, the same attribute applies to the method for both interfaces.

For example, the method element

```
<method>
    <ejb-name>EmployeeService</ejb-name>
    <method-intf>Remote</method-intf>
    <method-name>create</method-name>
    <method-params>
        <method-param>java.lang.String</method-param>
        <method-param>java.lang.String</method-param>
    </method-params>
</method>
```

can be used to differentiate the create(String, String) method defined in the remote interface from the create(String, String) method defined in the remote home interface, which would be defined as

```
<method>
    <ejb-name>EmployeeService</ejb-name>
    <method-intf>Home</method-intf>
    <method-name>create</method-name>
    <method-params>
        <method-param>java.lang.String</method-param>
        <method-param>java.lang.String</method-param>
    </method-params>
</method>
```

and the create method that is defined in the local home

```
                interface which would be defined as

                <method>
                    <ejb-name>EmployeeService</ejb-name>
                    <method-intf>LocalHome</method-intf>
                    <method-name>create</method-name>
                    <method-params>
                        <method-param>java.lang.String</method-param>
                        <method-param>java.lang.String</method-param>
                    </method-params>
                </method>

                The method-intf element can be used with all three Styles
                of the method element usage. For example, the following
                method element example could be used to refer to all the
                methods of the EmployeeService bean's remote home interface
                and the remote business interface.

                <method>
                    <ejb-name>EmployeeService</ejb-name>
                    <method-intf>Home</method-intf>
                    <method-name>*</method-name>
                </method>

                ]]>
              </xsd:documentation>
            </xsd:annotation>

            <xsd:sequence>
              <xsd:element name="description"
                           type="javaee:descriptionType"
                           minOccurs="0" maxOccurs="unbounded"/>
              <xsd:element name="ejb-name"
                           type="javaee:ejb-nameType"/>
              <xsd:element name="method-intf"
                           type="javaee:method-intfType"
                           minOccurs="0">
              </xsd:element>
              <xsd:element name="method-name"
                           type="javaee:method-nameType"/>
              <xsd:element name="method-params"
                           type="javaee:method-paramsType"
                           minOccurs="0"/>
            </xsd:sequence>
            <xsd:attribute name="id" type="xsd:ID"/>
          </xsd:complexType>

        <!-- **********************************************  -->

          <xsd:complexType name="multiplicityType">
            <xsd:annotation>
              <xsd:documentation>

                The multiplicityType describes the multiplicity of the
                role that participates in a relation.

                The value must be one of the two following:

                    One
                    Many

              </xsd:documentation>
            </xsd:annotation>
            <xsd:simpleContent>
              <xsd:restriction base="javaee:string">
                <xsd:enumeration value="One"/>
```

```
        <xsd:enumeration value="Many"/>
      </xsd:restriction>
    </xsd:simpleContent>
  </xsd:complexType>

<!-- **************************************************** -->

  <xsd:complexType name="named-methodType">
    <xsd:sequence>
      <xsd:element name="method-name"
                   type="javaee:string"/>
      <xsd:element name="method-params"
                   type="javaee:method-paramsType"
                   minOccurs="0"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID"/>
  </xsd:complexType>

<!-- **************************************************** -->

  <xsd:complexType name="persistence-typeType">
    <xsd:annotation>
      <xsd:documentation>

        The persistence-typeType specifies an entity bean's persistence
        management type.

        The persistence-type element must be one of the two following:

            Bean
            Container

      </xsd:documentation>
    </xsd:annotation>
    <xsd:simpleContent>
      <xsd:restriction base="javaee:string">
        <xsd:enumeration value="Bean"/>
        <xsd:enumeration value="Container"/>
      </xsd:restriction>
    </xsd:simpleContent>
  </xsd:complexType>

<!-- **************************************************** -->

  <xsd:complexType name="query-methodType">
    <xsd:annotation>
      <xsd:documentation>
        <![CDATA[

          The query-method specifies the method for a finder or select
          query.

          The method-name element specifies the name of a finder or select
          method in the entity bean's implementation class.

          Each method-param must be defined for a query-method using the
          method-params element.

          It is used by the query-method element.

          Example:

          <query>
              <description>Method finds large orders</description>
              <query-method>
                  <method-name>findLargeOrders</method-name>
```

```
                    <method-params></method-params>
                  </query-method>
                  <ejb-ql>
                    SELECT OBJECT(o) FROM Order o
                      WHERE o.amount &gt; 1000
                  </ejb-ql>
                </query>

                ]]>
              </xsd:documentation>
            </xsd:annotation>

            <xsd:sequence>
              <xsd:element name="method-name"
                           type="javaee:method-nameType"/>
              <xsd:element name="method-params"
                           type="javaee:method-paramsType"/>
            </xsd:sequence>
            <xsd:attribute name="id" type="xsd:ID"/>
          </xsd:complexType>

        <!-- **************************************************** -->

          <xsd:complexType name="queryType">
            <xsd:annotation>
              <xsd:documentation>

                The queryType defines a finder or select
                query. It contains
                    - an optional description of the query
                    - the specification of the finder or select
                      method it is used by
                        - an optional specification of the result type
                          mapping, if the query is for a select method
                          and entity objects are returned.
                        - the EJB QL query string that defines the query.

                Queries that are expressible in EJB QL must use the ejb-ql
                element to specify the query. If a query is not expressible
                in EJB QL, the description element should be used to
                describe the semantics of the query and the ejb-ql element
                should be empty.

                The result-type-mapping is an optional element. It can only
                be present if the query-method specifies a select method
                that returns entity objects.  The default value for the
                result-type-mapping element is "Local".

              </xsd:documentation>
            </xsd:annotation>

            <xsd:sequence>
              <xsd:element name="description"
                           type="javaee:descriptionType" minOccurs="0"/>
              <xsd:element name="query-method"
                           type="javaee:query-methodType"/>
              <xsd:element name="result-type-mapping"
                           type="javaee:result-type-mappingType"
                           minOccurs="0"/>
              <xsd:element name="ejb-ql"
                           type="javaee:xsdStringType"/>
            </xsd:sequence>
            <xsd:attribute name="id" type="xsd:ID"/>
          </xsd:complexType>

        <!-- **************************************************** -->
```

```
  <xsd:complexType name="relationship-role-sourceType">
    <xsd:annotation>
      <xsd:documentation>

        The relationship-role-sourceType designates the source of a
        role that participates in a relationship. A
        relationship-role-sourceType is used by
        relationship-role-source elements to uniquely identify an
        entity bean.

      </xsd:documentation>
    </xsd:annotation>

    <xsd:sequence>
      <xsd:element name="description"
                   type="javaee:descriptionType"
                   minOccurs="0"
                   maxOccurs="unbounded"/>
      <xsd:element name="ejb-name"
                   type="javaee:ejb-nameType"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID"/>
  </xsd:complexType>

<!-- ********************************************************** -->

  <xsd:complexType name="relationshipsType">
    <xsd:annotation>
      <xsd:documentation>

        The relationshipsType describes the relationships in
        which entity beans with container-managed persistence
        participate. The relationshipsType contains an optional
        description; and a list of ejb-relation elements, which
        specify the container managed relationships.

      </xsd:documentation>
    </xsd:annotation>

    <xsd:sequence>
      <xsd:element name="description"
                   type="javaee:descriptionType"
                   minOccurs="0"
                   maxOccurs="unbounded"/>
      <xsd:element name="ejb-relation"
                   type="javaee:ejb-relationType"
                   maxOccurs="unbounded">

        <xsd:unique name="role-name-uniqueness">
          <xsd:annotation>
            <xsd:documentation>

              The ejb-relationship-role-name contains the name of a
              relationship role. The name must be unique within
              a relationship, but can be reused in different
              relationships.

            </xsd:documentation>
          </xsd:annotation>
          <xsd:selector
               xpath=".//javaee:ejb-relationship-role-name"/>
          <xsd:field
               xpath="."/>
        </xsd:unique>
      </xsd:element>
```

```
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID"/>
  </xsd:complexType>

<!-- **************************************************** -->

  <xsd:complexType name="remove-methodType">
    <xsd:sequence>
      <xsd:element name="bean-method"
                   type="javaee:named-methodType"/>
      <xsd:element name="retain-if-exception"
                   type="javaee:true-falseType"
                   minOccurs="1"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID"/>
  </xsd:complexType>

<!-- **************************************************** -->

  <xsd:complexType name="result-type-mappingType">
    <xsd:annotation>
      <xsd:documentation>

        The result-type-mappingType is used in the query element to
        specify whether an abstract schema type returned by a query
        for a select method is to be mapped to an EJBLocalObject or
        EJBObject type.

        The value must be one of the following:

            Local
            Remote

      </xsd:documentation>
    </xsd:annotation>
    <xsd:simpleContent>
      <xsd:restriction base="javaee:string">
        <xsd:enumeration value="Local"/>
        <xsd:enumeration value="Remote"/>
      </xsd:restriction>
    </xsd:simpleContent>
  </xsd:complexType>

<!-- **************************************************** -->

  <xsd:complexType name="security-identityType">
    <xsd:annotation>
      <xsd:documentation>

        The security-identityType specifies whether the caller's
        security identity is to be used for the execution of the
        methods of the enterprise bean or whether a specific run-as
        identity is to be used. It contains an optional description
        and a specification of the security identity to be used.

      </xsd:documentation>
    </xsd:annotation>

    <xsd:sequence>
      <xsd:element name="description"
                   type="javaee:descriptionType"
                   minOccurs="0"
                   maxOccurs="unbounded"/>
      <xsd:choice>
        <xsd:element name="use-caller-identity"
                     type="javaee:emptyType">
```

```
            <xsd:annotation>
              <xsd:documentation>

                The use-caller-identity element specifies that
                the caller's security identity be used as the
                security identity for the execution of the
                enterprise bean's methods.

              </xsd:documentation>
            </xsd:annotation>
          </xsd:element>
          <xsd:element name="run-as"
                       type="javaee:run-asType"/>
        </xsd:choice>
      </xsd:sequence>
      <xsd:attribute name="id" type="xsd:ID"/>
    </xsd:complexType>

  <!-- **************************************************** -->

    <xsd:complexType name="session-beanType">
      <xsd:annotation>
        <xsd:documentation>

          The session-beanType declares an session bean. The
          declaration consists of:

              - an optional description
              - an optional display name
              - an optional icon element that contains a small and a large
                icon file name
              - a name assigned to the enterprise bean
                in the deployment description
              - an optional mapped-name element that can be used to provide
                vendor-specific deployment information such as the physical
                jndi-name of the session bean's remote home/business interface.
                This element is not required to be supported by all
                implementations. Any use of this element is non-portable.
              - the names of all the remote or local business interfaces,
                if any
              - the names of the session bean's remote home and
                remote interfaces, if any
              - the names of the session bean's local home and
                local interfaces, if any
              - the name of the session bean's web service endpoint
                interface, if any
              - the session bean's implementation class
              - the session bean's state management type
              - an optional declaration of the session bean's timeout method.
              - the optional session bean's transaction management type.
                If it is not present, it is defaulted to Container.
              - an optional list of the session bean class and/or
                superclass around-invoke methods.
              - an optional declaration of the bean's
                environment entries
              - an optional declaration of the bean's EJB references
              - an optional declaration of the bean's local
                EJB references
              - an optional declaration of the bean's web
                service references
              - an optional declaration of the security role
                references
              - an optional declaration of the security identity
                to be used for the execution of the bean's methods
              - an optional declaration of the bean's resource
                manager connection factory references
```

```
            - an optional declaration of the bean's resource
              environment references.
            - an optional declaration of the bean's message
              destination references

          The elements that are optional are "optional" in the sense
          that they are omitted when if lists represented by them are
          empty.

          Either both the local-home and the local elements or both
          the home and the remote elements must be specified for the
          session bean.

          The service-endpoint element may only be specified if the
          bean is a stateless session bean.

        </xsd:documentation>
      </xsd:annotation>

      <xsd:sequence>
        <xsd:group ref="javaee:descriptionGroup"/>
        <xsd:element name="ejb-name"
                     type="javaee:ejb-nameType"/>
        <xsd:element name="mapped-name"
                     type="javaee:xsdStringType"
                     minOccurs="0"/>
        <xsd:element name="home"
                     type="javaee:homeType"
                     minOccurs="0"/>
        <xsd:element name="remote"
                     type="javaee:remoteType"
                     minOccurs="0"/>
        <xsd:element name="local-home"
                     type="javaee:local-homeType"
                     minOccurs="0"/>
        <xsd:element name="local"
                     type="javaee:localType"
                     minOccurs="0"/>
        <xsd:element name="business-local"
                     type="javaee:fully-qualified-classType"
                     minOccurs="0"
                     maxOccurs="unbounded"/>
        <xsd:element name="business-remote"
                     type="javaee:fully-qualified-classType"
                     minOccurs="0"
                     maxOccurs="unbounded"/>
        <xsd:element name="service-endpoint"
                     type="javaee:fully-qualified-classType"
                     minOccurs="0">
          <xsd:annotation>
            <xsd:documentation>

              The service-endpoint element contains the
              fully-qualified name of the enterprise bean's web
              service endpoint interface. The service-endpoint
              element may only be specified for a stateless
              session bean. The specified interface must be a
              valid JAX-RPC service endpoint interface.

            </xsd:documentation>
          </xsd:annotation>
        </xsd:element>
        <xsd:element name="ejb-class"
                     type="javaee:ejb-classType"/>
        <xsd:element name="session-type"
                     type="javaee:session-typeType"
```

```
                minOccurs="0"/>
        <xsd:element name="timeout-method"
                type="javaee:named-methodType"
                minOccurs="0"/>
        <xsd:element name="init-method"
                type="javaee:init-methodType"
                minOccurs="0"
                maxOccurs="unbounded">
          <xsd:annotation>
            <xsd:documentation>

              The init-method element specifies the mappings for
              EJB 2.x style create methods for an EJB 3.0 bean.
              This element can only be specified for stateful
              session beans.

            </xsd:documentation>
          </xsd:annotation>
        </xsd:element>
        <xsd:element name="remove-method"
                type="javaee:remove-methodType"
                minOccurs="0"
                maxOccurs="unbounded">
          <xsd:annotation>
            <xsd:documentation>

              The remove-method element specifies the mappings for
              EJB 2.x style remove methods for an EJB 3.0 bean.
              This element can only be specified for stateful
              session beans.

            </xsd:documentation>
          </xsd:annotation>
        </xsd:element>
        <xsd:element name="transaction-type"
                type="javaee:transaction-typeType"
                minOccurs="0"/>
        <xsd:element name="around-invoke-method"
                type="javaee:lifecycle-callbackType"
                minOccurs="0"
                maxOccurs="unbounded"/>
        <xsd:group ref="javaee:jndiEnvironmentRefsGroup"/>
        <xsd:element name="post-activate-method"
                type="javaee:lifecycle-callbackType"
                minOccurs="0"
                maxOccurs="unbounded"/>
        <xsd:element name="pre-passivate-method"
                type="javaee:lifecycle-callbackType"
                minOccurs="0"
                maxOccurs="unbounded"/>
        <xsd:element name="security-role-ref"
                type="javaee:security-role-refType"
                minOccurs="0"
                maxOccurs="unbounded">
        </xsd:element>
        <xsd:element name="security-identity"
                type="javaee:security-identityType"
                minOccurs="0">
        </xsd:element>
      </xsd:sequence>
      <xsd:attribute name="id" type="xsd:ID"/>
    </xsd:complexType>

  <!-- ********************************************** -->

    <xsd:complexType name="session-typeType">
```

```
      <xsd:annotation>
        <xsd:documentation>

          The session-typeType describes whether the session bean is a
          stateful session or stateless session. It is used by
          session-type elements.

          The value must be one of the two following:

              Stateful
              Stateless

        </xsd:documentation>
      </xsd:annotation>
      <xsd:simpleContent>
        <xsd:restriction base="javaee:string">
          <xsd:enumeration value="Stateful"/>
          <xsd:enumeration value="Stateless"/>
        </xsd:restriction>
      </xsd:simpleContent>
    </xsd:complexType>

  <!-- **************************************************** -->

    <xsd:complexType name="trans-attributeType">
      <xsd:annotation>
        <xsd:documentation>

          The trans-attributeType specifies how the container must
          manage the transaction boundaries when delegating a method
          invocation to an enterprise bean's business method.

          The value must be one of the following:

              NotSupported
              Supports
              Required
              RequiresNew
              Mandatory
              Never

        </xsd:documentation>
      </xsd:annotation>
      <xsd:simpleContent>
        <xsd:restriction base="javaee:string">
          <xsd:enumeration value="NotSupported"/>
          <xsd:enumeration value="Supports"/>
          <xsd:enumeration value="Required"/>
          <xsd:enumeration value="RequiresNew"/>
          <xsd:enumeration value="Mandatory"/>
          <xsd:enumeration value="Never"/>
        </xsd:restriction>
      </xsd:simpleContent>
    </xsd:complexType>

  <!-- **************************************************** -->

    <xsd:complexType name="transaction-typeType">
      <xsd:annotation>
        <xsd:documentation>

          The transaction-typeType specifies an enterprise bean's
          transaction management type.

          The transaction-type must be one of the two following:
```

```
              Bean
              Container

        </xsd:documentation>
      </xsd:annotation>
      <xsd:simpleContent>
        <xsd:restriction base="javaee:string">
          <xsd:enumeration value="Bean"/>
          <xsd:enumeration value="Container"/>
        </xsd:restriction>
      </xsd:simpleContent>
    </xsd:complexType>

</xsd:schema>
```

Chapter 19 Ejb-jar File

The ejb-jar file is the standard format for the packaging of enterprise beans. The ejb-jar file format is used to package un-assembled enterprise beans (the Bean Provider's output), and to package assembled applications (the Application Assembler's output).

19.1 Overview

The ejb-jar file format is the contract between the Bean Provider and the Application Assembler, and between the Application Assembler and the Deployer.

An ejb-jar file produced by the Bean Provider contains one or more enterprise beans that typically do not contain application assembly instructions. The ejb-jar file produced by an Application Assembler (which can be the same person or organization as the Bean Provider) contains one or more enterprise beans, plus application assembly information describing how the enterprise beans are combined into a single application deployment unit.

19.2 Deployment Descriptor

The ejb-jar file must contain the deployment descriptor (if any) in the format defined in Chapter 18. The deployment descriptor must be stored with the name `META-INF/ejb-jar.xml` in the ejb-jar file.

19.3 Ejb-jar File Requirements

The ejb-jar file must contain, either by inclusion or by reference, the class files of each enterprise bean as follows:

- The enterprise bean class.
- The enterprise bean business interfaces, web service endpoint interfaces, and home and component interfaces.
- Interceptor classes.
- The primary key class if the bean is an entity bean.

We say that a jar file contains a second file "by reference" if the second file is named in the Class-Path attribute in the Manifest file of the referencing jar file or is contained (either by inclusion or by reference) in another jar file that is named in the Class-Path attribute in the Manifest file of the referencing jar file.

The ejb-jar file must also contain, either by inclusion or by reference, the class files for all the classes and interfaces that each enterprise bean class and the home interfaces, component interfaces, and/or web service endpoints depend on, except Java EE and J2SE classes. This includes their superclasses and superinterfaces, dependent classes, and the classes and interfaces used as method parameters, results, and exceptions.

The Application Assembler must not package the stubs of the EJBHome and EJBObject interfaces in the ejb-jar file. This includes the stubs for the enterprise beans whose implementations are provided in the ejb-jar file as well as the referenced enterprise beans. Generating the stubs is the responsibility of the container. The stubs are typically generated by the Container Provider's deployment tools for each class that extends the EJBHome or EJBObject interfaces, or they may be generated by the container at runtime.

19.4 The Client View and the ejb-client JAR File

The client view of an enterprise bean is comprised of the business interfaces or home and component interfaces of the referenced enterprise bean and other classes that these interfaces depend on, such as their superclasses and superinterfaces, the classes and interfaces used as method parameters, results, and exceptions. The serializable application value classes, including the classes which may be used as members of a collection in a remote method call to an enterprise bean, are part of the client view. An example of an application value class might be an `Address` class used as a parameter in a method call.

The ejb-jar file producer can create an ejb-client JAR file for the ejb-jar file. The ejb-client JAR file contains all the class files that a client program needs to use the client view of the enterprise beans that are contained in the ejb-jar file. If this option is used, it is the responsibility of the Application Assembler to include all the classes necessary to comprise the client view of an enterprise bean in the ejb-client JAR file.

The ejb-client JAR file is specified in the deployment descriptor of the ejb-jar file using the optional `ejb-client-jar` element. The value of the `ejb-client-jar` element is the path name specifying the location of the ejb-client JAR file in the containing Java EE Enterprise Application Archive (`.ear`) file. The path name is relative to the location of the referencing ejb-jar file.

The EJB specification does not specify whether an ejb-jar file should include by copy or by reference the classes that are in an ejb-client JAR file, but they must be included either one way or the other. If the by-copy approach is used, the producer simply includes all the class files in the ejb-client JAR file also in the ejb-jar file. If the by-reference approach is used, the ejb-jar file producer does not duplicate the content of the ejb-client JAR file in the ejb-jar file, but instead uses a Manifest Class-Path entry in the ejb-jar file to specify that the ejb-jar file depends on the ejb-client JAR at runtime. The use of the Class-Path entries in the JAR files is explained in the Java EE Platform, Enterprise Edition specification [12].

19.5 Requirements for Clients

The Application Assembler must construct the application to insure that the client view classes are available to the client at runtime. The client of an enterprise bean may be another enterprise bean packaged in the same ejb-jar or different ejb-jar file, or the client may be another Java EE component, such as a web component.

When clients packaged in jar files refer to enterprise beans, the jar file that contains the client, e.g. an ejb-jar file, should contain, either by inclusion or by reference, all the client view classes of the referenced beans. The client view classes may have been packaged in an ejb-client JAR file. In other words, the jar file that contains the client should contain one of the following:

- a reference to the ejb-client JAR file
- a reference to the ejb-jar file that contains the client view classes
- a copy of the client view classes

The client may also require the use of system value classes (e.g., the serializable value classes implementing the `javax.ejb.Handle`, `javax.ejb.HomeHandle`, `javax.ejb.EJBMetaData`, `java.util.Enumeration`, `java.util.Collection`, and `java.util.Iterator` interfaces) , although these are not packaged with the application. It is the responsibility of the provider of the container hosting the referenced beans to provide the system value classes and make them available for use when the client is deployed. See Section 14.5.5, “System Value Classes”.

19.6 Example

In this example, the Bean Provider has chosen to package the enterprise bean client view classes in a separate jar file and to reference that jar file from the other jar files that need those classes. Those classes are needed both by `ejb2.jar`, packaged in the same application as `ejb1.jar`, and by `ejb3.jar`, packaged in a different application. Those classes are also needed by `ejb1.jar` itself because they define the remote interface of the enterprise beans in `ejb1.jar`, and the Bean Provider has chosen the *by reference* approach to making these classes available.

The deployment descriptor for `ejb1.jar` names the client view jar file in the `ejb-client-jar` element. Because `ejb2.jar` requires these client view classes, it includes a Class-Path reference to `ejb1_client.jar`.

The Class-Path mechanism must be used by components in `app2.ear` to reference the client view jar file that corresponds to the enterprise beans packaged in `ejb1.jar` of `app1.ear`. Those enterprise beans are referenced by enterprise beans in `ejb3.jar`. Note that the client view jar file must be included directly in the `app2.ear` file.

```
app1.ear:
     META-INF/application.xml
     ejb1.jar      Class-Path: ejb1_client.jar
          deployment descriptor contains:
               <ejb-client-jar>ejb1_client.jar</ejb-client-jar>
     ejb1_client.jar
     ejb2.jar      Class-Path: ejb1_client.jar


app2.ear:
     META-INF/application.xml
     ejb1_client.jar
     ejb3.jar      Class-Path: ejb1_client.jar
```

Chapter 20 Runtime Environment

This chapter defines the application programming interfaces (APIs) that a compliant EJB 3.0 container must make available to the enterprise bean instances at runtime. These APIs can be used by portable enterprise beans because the APIs are guaranteed to be available in all EJB 2.1 containers.

The chapter also defines the restrictions that the EJB 3.0 Container Provider can impose on the functionality that it provides to the enterprise beans. These restrictions are necessary to enforce security and to allow the container to properly manage the runtime environment.

20.1 Bean Provider's Responsibilities

This section describes the view and responsibilities of the Bean Provider.

20.1.1 APIs Provided by Container

The EJB Provider can rely on the EJB 3.0 Container Provider to provide all components with the Java 2 Platform Standard Edition, v5.0 (J2SE) APIs, as required by the Java EE 5.0 specification [12]. The Java SE platform includes the following enterprise APIs:

- JDBC

- RMI-IIOP
- JNDI
- JAXP
- Java IDL
- JAAS

The Java EE platform also requires a number of optional packages. The following optional packages are required to be provided in EJB containers:

- EJB 3.0, including Java Persistence
- JTA 1.1
- JMS 1.1
- JavaMail 1.4 (for sending mail only)
- JAF 1.1[82]
- JAXR 1.0
- SAAJ 1.3
- JAX-RPC 1.1
- JAX-WS 2.0
- Connector 1.5
- Web Services 1.2
- JAXB 2.0
- Java EE Management 1.1
- JACC 1.1
- Web Services Metadata 1.1
- Common Annotations 1.0
- StAX 1.0

[82] See [12] for restrictions.

20.1.2 Programming Restrictions

This section describes the programming restrictions that a Bean Provider must follow to ensure that the enterprise bean is *portable* and can be deployed in any compliant EJB 3.0 container. The restrictions apply to the implementation of the business methods. Section 20.2, which describes the container's view of these restrictions, defines the programming environment that all EJB containers must provide.

- An enterprise bean must not use read/write static fields. Using read-only static fields is allowed. Therefore, it is recommended that all static fields in the enterprise bean class be declared as `final`.

This rule is required to ensure consistent runtime semantics because while some EJB containers may use a single JVM to execute all enterprise bean's instances, others may distribute the instances across multiple JVMs.

- An enterprise bean must not use thread synchronization primitives to synchronize execution of multiple instances.

This is for the same reason as above. Synchronization would not work if the EJB container distributed enterprise bean's instances across multiple JVMs.

- An enterprise bean must not use the AWT functionality to attempt to output information to a display, or to input information from a keyboard.

Most servers do not allow direct interaction between an application program and a keyboard/display attached to the server system.

- An enterprise bean must not use the `java.io` package to attempt to access files and directories in the file system.

The file system APIs are not well-suited for business components to access data. Business components should use a resource manager API, such as JDBC, to store data.

- An enterprise bean must not attempt to listen on a socket, accept connections on a socket, or use a socket for multicast.

The EJB architecture allows an enterprise bean instance to be a network socket client, but it does not allow it to be a network server. Allowing the instance to become a network server would conflict with the basic function of the enterprise bean— to serve the EJB clients.

- The enterprise bean must not attempt to query a class to obtain information about the declared members that are not otherwise accessible to the enterprise bean because of the security rules of the Java language. The enterprise bean must not attempt to use the Reflection API to access information that the security rules of the Java programming language make unavailable.

Allowing the enterprise bean to access information about other classes and to access the classes in a manner that is normally disallowed by the Java programming language could compromise security.

- The enterprise bean must not attempt to create a class loader; obtain the current class loader; set the context class loader; set security manager; create a new security manager; stop the JVM; or change the input, output, and error streams.

These functions are reserved for the EJB container. Allowing the enterprise bean to use these functions could compromise security and decrease the container's ability to properly manage the runtime environment.

- The enterprise bean must not attempt to set the socket factory used by ServerSocket, Socket, or the stream handler factory used by URL.

These networking functions are reserved for the EJB container. Allowing the enterprise bean to use these functions could compromise security and decrease the container's ability to properly manage the runtime environment.

- The enterprise bean must not attempt to manage threads. The enterprise bean must not attempt to start, stop, suspend, or resume a thread, or to change a thread's priority or name. The enterprise bean must not attempt to manage thread groups.

These functions are reserved for the EJB container. Allowing the enterprise bean to manage threads would decrease the container's ability to properly manage the runtime environment.

- The enterprise bean must not attempt to directly read or write a file descriptor.

Allowing the enterprise bean to read and write file descriptors directly could compromise security.

- The enterprise bean must not attempt to obtain the security policy information for a particular code source.

Allowing the enterprise bean to access the security policy information would create a security hole.

- The enterprise bean must not attempt to load a native library.

This function is reserved for the EJB container. Allowing the enterprise bean to load native code would create a security hole.

- The enterprise bean must not attempt to gain access to packages and classes that the usual rules of the Java programming language make unavailable to the enterprise bean.

This function is reserved for the EJB container. Allowing the enterprise bean to perform this function would create a security hole.

- The enterprise bean must not attempt to define a class in a package.

This function is reserved for the EJB container. Allowing the enterprise bean to perform this function would create a security hole.

- The enterprise bean must not attempt to access or modify the security configuration objects (Policy, Security, Provider, Signer, and Identity).

These functions are reserved for the EJB container. Allowing the enterprise bean to use these functions could compromise security.

- The enterprise bean must not attempt to use the subclass and object substitution features of the Java Serialization Protocol.

Allowing the enterprise bean to use these functions could compromise security.

- The enterprise bean must not attempt to pass `this` as an argument or method result. The enterprise bean must pass the result of `SessionContext.getBusinessObject`, `SessionContext.getEJBObject`, `SessionContext.getEJBLocalObject`, `EntityContext.getEJBObject`, or `EntityContext.getEJBLocalObject` instead.

To guarantee portability of the enterprise bean's implementation across all compliant EJB 3.0 containers, the Bean Provider should test the enterprise bean using a container with the security settings defined in Table 21. The tables define the minimal functionality that a compliant EJB container must provide to the enterprise bean instances at runtime.

20.2 Container Provider's Responsibility

This section defines the container's responsibilities for providing the runtime environment to the enterprise bean instances. The requirements described here are considered to be the minimal requirements; a container may choose to provide additional functionality that is not required by the EJB specification.

An EJB 3.0 container must make the following APIs available to the enterprise bean instances at runtime:

- Java 2 Platform, Standard Edition v5 (J2SE) APIs, which include the following APIs:
  - JDBC
  - RMI-IIOP
  - JNDI
  - JAXP
  - Java IDL
- EJB 3.0 APIs, including Java Persistence
- JTA 1.1, the `UserTransaction` interface only
- JMS 1.1
- JavaMail 1.4, sending mail only
- JAF 1.1[83]

[83] See [12] for restrictions.

- JAXP 1.2
- JAXR 1.0
- JAX-RPC 1.1
- JAX-WS 2.0
- JAXB 2.0
- SAAJ 1.3
- Connector 1.5
- Web Services 1.2
- Web Services Metadata 1.1
- Common Annotations 1.0

The following subsections describes the requirements in more detail.

20.2.1 Java 2 APIs Requirements

The container must provide the full set of Java 2 Platform, Standard Edition, v5 (J2SE) APIs. The container is not allowed to subset the Java 2 platform APIs.

The EJB container is allowed to make certain Java 2 platform functionality unavailable to the enterprise bean instances by using the Java 2 platform security policy mechanism. The primary reason for the container to make certain functions unavailable to enterprise bean instances is to protect the security and integrity of the EJB container environment, and to prevent the enterprise bean instances from interfering with the container's functions.

The following table defines the Java 2 platform security permissions that the EJB container must be able to grant to the enterprise bean instances at runtime. The term "grant" means that the container must be able to grant the permission, the term "deny" means that the container should deny the permission.

Table 21 Java 2 Platform Security Policy for a Standard EJB Container

Permission name	EJB Container policy
java.security.AllPermission	deny
java.awt.AWTPermission	deny
java.io.FilePermission	deny
java.net.NetPermission	deny
java.util.PropertyPermission	grant "read", "*" deny all other
java.lang.reflect.ReflectPermission	deny
java.lang.RuntimePermission	grant "queuePrintJob", deny all other
java.lang.SecurityPermission	deny
java.io.SerializablePermission	deny
java.net.SocketPermission	grant "connect", "*" [Note A], deny all other

Notes:

[A] This permission is necessary, for example, to allow enterprise beans to use the client functionality of the Java IDL and RMI-IIOP packages that are part of the Java 2 platform.

Some containers may allow the Deployer to grant more, or fewer, permissions to the enterprise bean instances than specified in Table 21. Support for this is not required by the EJB specification. Enterprise beans that rely on more or fewer permissions will not be portable across all EJB containers.

20.2.2 EJB 3.0 Requirements

The container must implement the EJB 3.0 interfaces as defined in this specification.

The container must implement the semantics of the metadata annotations that are supported by EJB 3.0 as defined by this specification.

The container must implement the `javax.persistence` interfaces and metadata annotations that are defined in the document *"Java Persistence"* [2] of this specification.

20.2.3 JNDI Requirements

At the minimum, the EJB container must provide a JNDI API name space to the enterprise bean instances. The EJB container must make the name space available to an instance when the instance invokes the `javax.naming.InitialContext` default (no-arg) constructor.

The EJB container must make available at least the following objects in the name space:

- The business interfaces of other enterprise beans.
- The resource factories used by the enterprise beans.
- The entity managers and entity manager factories used by the enterprise beans.
- The web service interfaces used by the enterprise beans.
- The home interfaces of other enterprise beans.
- ORB objects
- UserTransaction objects
- EJBContext objects
- TimerService objects

The EJB specification does not require that all the enterprise beans deployed in a container be presented with the same JNDI API name space. However, all the instances of the same enterprise bean must be presented with the same JNDI API name space.

20.2.4 JTA 1.0.1 Requirements

The EJB container must include the JTA 1.1 extension, and it must provide the `javax.transaction.UserTransaction` interface to enterprise beans with bean-managed transaction demarcation through the `javax.ejb.EJBContext` interface, and also in JNDI under the name `java:comp/UserTransaction`, in the cases required by the EJB specification.

The other JTA interfaces are low-level transaction manager and resource manager integration interfaces, and are not intended for direct use by enterprise beans.

20.2.5 JDBC™ 3.0 Extension Requirements

The EJB container must include the JDBC 3.0 extension and provide its functionality to the enterprise bean instances, with the exception of the low-level XA and connection pooling interfaces. These low-level interfaces are intended for integration of a JDBC driver with an application server, not for direct use by enterprise beans.

20.2.6 JMS 1.1 Requirements

The EJB container must include the JMS 1.1 extension and provide its functionality to the enterprise bean instances, with the exception of the low-level interfaces that are intended for integration of a JMS provider with an application server, not for direct use by enterprise beans. These interfaces include: `javax.jms.ServerSession`, `javax.jms.ServerSessionPool`, `javax.jms.ConnectionConsumer`, and all the `javax.jms` XA interfaces.

In addition, the following methods are for use by the container only. Enterprise beans must not call these methods: `javax.jms.Session.setMessageListener`, `javax.jms.Session.getMessageListener`, `javax.jms.Session.run`, `javax.jms.QueueConnection.createConnectionConsumer`, `javax.jms.TopicConnection.createConnectionConsumer`, `javax.jms.TopicConnection.createDurableConnectionConsumer`, `javax.jms.Connection.createConnectionConsumer`, `javax.jms.Connection.createDurableConnectionConsumer`.

The following methods must not be called by enterprise beans because they may interfere with the connection management by the container: `javax.jms.Connection.setExceptionListener`, `javax.jms.Connection.stop`, `javax.jms.Connection.setClientID`.

Enterprise beans must not call the `javax.jms.MessageConsumer.setMessageListener` or `javax.jms.MessageConsumer.getMessageListener` method.

This specification recommends, but does not require, that the container throw the `javax.jms.JMSException` if enterprise beans call any of the methods listed in this section.

20.2.7 Argument Passing Semantics

An enterprise bean's business interfaces annotated as `Remote` (whether in metadata annotations or the deployment descriptor) and/or remote home and remote interfaces are *remote interfaces* for Java RMI. The container must ensure the semantics for passing arguments conforms to Java RMI-IIOP. Non-remote objects must be passed by value.

Specifically, the EJB container is not allowed to pass non-remote objects by reference on inter-EJB invocations when the calling and called enterprise beans are collocated in the same JVM. Doing so could result in the multiple beans sharing the state of a Java object, which would break the enterprise bean's semantics. Any local optimizations of remote interface calls must ensure the semantics for passing arguments conforms to Java RMI-IIOP.

An enterprise bean's local business interfaces and/or local home and local interfaces are *local Java interfaces*. The caller and callee enterprise beans that make use of these local interfaces are typically collocated in the same JVM. The EJB container must ensure the semantics for passing arguments across these interfaces conforms to the standard argument passing semantics of the Java programming language.

20.2.8 Other Requirements

The assertions contained in the Javadoc specification of the EJB interfaces and Persistence interfaces are required functionality and must be implemented by compliant containers.

Chapter 21 Responsibilities of EJB Roles

This chapter provides the summary of the responsibilities of each EJB Role.

21.1 Bean Provider's Responsibilities

This section highlights the requirements for the Bean Provider. Meeting these requirements is necessary to ensure that the enterprise beans developed by the Bean Provider can be deployed in all compliant EJB containers.

21.1.1 API Requirements

The enterprise beans must meet all the API requirements defined in the individual chapters of this document.

21.1.2 Packaging Requirements

The Bean Provider is responsible for packaging the enterprise beans in an ejb-jar file in the format described in Chapter 19.

The deployment descriptor, if present, must conform to the requirements of Chapter 18.

21.2 Application Assembler's Responsibilities

The requirements for the Application Assembler are in defined in Chapter 18 and Chapter 19.

21.3 EJB Container Provider's Responsibilities

The EJB Container Provider is responsible for providing the deployment tools used by the Deployer to deploy enterprise beans packaged in the ejb-jar file. The requirements for the deployment tools are defined in the individual chapters of this document.

The EJB Container Provider is responsible for implementing its part of the EJB contracts, and for providing all the runtime services described in the individual chapters of this document.

21.4 Persistence Provider's Responsibilities

The Persistence Provider is responsible for implementing its part of the contracts described in the document "*Java Persistence*" of this specification [2].

21.5 Deployer's Responsibilities

The Deployer uses the deployment tools provided by the EJB Container Provider to deploy ejb-jar files produced by the Bean Providers and Application Assemblers.

The individual chapters of this document describe the responsibilities of the Deployer in more detail.

21.6 System Administrator's Responsibilities

The System Administrator is responsible for configuring the EJB container and server, setting up security management, integrating resource managers with the EJB container, and runtime monitoring of deployed enterprise beans applications.

The individual chapters of this document describe the responsibilities of the System Administrator in more detail.

21.7 Client Programmer's Responsibilities

The EJB client programmer writes applications that access enterprise beans via their business interfaces, via their web service client view, or via messages, or view their home and component interfaces.

Chapter 22 Related Documents

[1] EnterpriseJavaBeans, version 3.0. EJB 3.0 Simplified API. *http://java.sun.com/products/ejb.*

[2] EnterpriseJavaBeans, version 3.0. Java Persistence. *http://java.sun.com/products/ejb.*

[3] EnterpriseJavaBeans, version 2. (EJB 2.1). *http://java.sun.com/products/ejb.*

[4] JavaBeans. *http://java.sun.com/beans*.

[5] Java Naming and Directory Interface (JNDI). *http://java.sun.com/products/jndi*.

[6] Java Remote Method Invocation (RMI). *http://java.sun.com/products/rmi.*

[7] Java Security. *http://java.sun.com/security.*

[8] Java Transaction API (JTA). *http://java.sun.com/products/jta*.

[9] Java Transaction Service (JTS). *http://java.sun.com/products/jts.*

[10] Java Language to IDL Mapping Specification. *http://www.omg.org/cgi-bin/doc?ptc/00-01-06*.

[11] CORBA Object Transaction Service v1.2. *http://www.omg.org/cgi-bin/doc?ptc/2000-11-07*.

[12] Java Platform, Enterprise Edition (Java EE), v5. *http://jcp.org/en/jsr/detail?id=244*.

[13] Java Message Service (JMS), v 1.1. *http://java.sun.com/products/jms*.

[14] Java API for XML Messaging (JAXM).

[15] Java 2 Enterprise Edition Connector Architecture, v1.5 . *http://java.sun.com/j2ee/connector.*

[16] Enterprise JavaBeans to CORBA Mapping v1.1. *http://java.sun.com/products/ejb/docs.html*.

[17] CORBA 2.3.1 Specification. *http://www.omg.org/cgi-bin/doc?formal/99-10-07*.

[18] CORBA COSNaming Service. *http://www.omg.org/cgi-bin/doc?formal/00-06-19*.

[19] Interoperable Name Service FTF document. *http://www.omg.org/cgi-bin/doc?ptc/00-08-07*.

[20] RFC 2246: The TLS Protocol. *ftp://ftp.isi.edu/in-notes/rfc2246.txt.*

[21] RFC 2712: Addition of Kerberos Cipher Suites to Transport Layer Security. *ftp://ftp.isi.edu/in-notes/rfc2712.txt.*

[22] The SSL Protocol Version 3.0. *http://home.netscape.com/eng/ssl3/draft302.txt.*

[23] Common Secure Interoperability Version 2 Final Available Specification. *http://www.omg.org/cgi-bin/doc?ptc/2001-06-17*.

[24] Database Language SQL. ANSI X3.135-1992 or ISO/IEC 9075:1992.

[25] Java API for XML-based RPC (JAX-RPC) 2.0. *http://jcp.org/en/jsr/detail?id=101*.

[26] Web Services Description Language (WSDL) 1.1. *http://www.w3.org/TR/wsdl.*

[27] W3C: SOAP 1.1. *http://www.w3.org/TR/SOAP/.*

[28] The Java Virtual Machine Specification.

[29] JDBC 3.0 Specification. *http://java.sun.com/products/jdbc*.

[30] Web Services Metadata for the Java Platform. *http://jcp.org/en/jsr/detail?id=181*.

[31] Web Services for Java EE, Version 1.2. *http://jcp.org/en/jsr/detail?id=109.*

[32] Java API for XML Web Services (JAX-WS 2.0). *http://jcp.org/en/jsr/detail?id=224.*

Appendix A Revision History

This appendix lists the significant changes that have been made during the development of the EJB 3.0 Specification.

A.1 Public Draft

Created document from the EJB 2.1 Final Release; removed illustrative material that was specific to the EJB 2.1 and earlier specifications.

A.2 Proposed Final Draft

Merged callback listener functionality into interceptors.

Added support for multiple interceptor classes per bean.

Added support for method-level interceptors.

Added description of role of the Persistence Provider.

Removed support for annotations, dependency injection, interceptors for EJB 2.1 entity beans.

Clarified that timers survive server shutdown.

Clarified passivation requirements if extended persistence context is used.

Updated Security chapter to reflect JSR-250 annotations.

Updated to reflect JAX-WS and Web Services for Java EE specifications.

Added getInvokedBusinessInterface method.

Clarified that web service message handlers are invoked before business method interceptor methods.

Clarified use of equals and hashCode for identity testing of references to session bean business interfaces.

Removed prohibitions against container vendors providing read-only entity beans.

Editorial sweep.