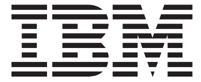
WebSphere™ Application Server

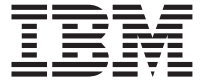


# Enterprise Beans の作成

*バージョン 4.0*

SD88-7343-02
(英文原典：SC09-4431-03)

WebSphere™ Application Server



# Enterprise Beans の作成

*バージョン 4.0*

SD88-7343-02
(英文原典：SC09-4431-03)

**ご注意**

本書の情報およびそれによってサポートされる製品を使用する前に、323ページの『特記事項』に記載する一般情報をお読みください。

本書は、SD88-7343-01 の改訂版です。

本マニュアルに関するご意見やご感想は、次の URL からお送りください。今後の参考にさせていただきます。

http://www.ibm.com/jp/manuals/main/mail.html

なお、日本 IBM 発行のマニュアルはインターネット経由でもご購入いただけます。詳しくは

http://www.ibm.com/jp/manuals/ の「ご注文について」をご覧ください。

(URL は、変更になる場合があります)

**原　典：** SC09-4431-03
WebSphere™ Application Server
Writing Enterprise Beans in WebSphere
Version 4.0

**発　行：** 日本アイ・ビー・エム株式会社

**担　当：** ナショナル・ランゲージ・サポート

第1刷 2001.3

この文書では、平成明朝体™W3、平成明朝体™W9、平成角ゴシック体™W3、平成角ゴシック体™W5、および平成角ゴシック体™W7を使用しています。この (書体*) は、(財) 日本規格協会と使用契約を締結し使用しているものです。フォントとして無断複製することは禁止されています。

注* 平成明朝体™W3、平成明朝体™W9、平成角ゴシック体™W3、
平成角ゴシック体™W5、平成角ゴシック体™W7

**© Copyright International Business Machines Corporation 1999, 2001. All rights reserved.**

Translation: © Copyright IBM Japan 2001

# 目次


**図 . . . . . . . . . . . . . . . . . vii**

**表 . . . . . . . . . . . . . . . . . xi**

**本書について . . . . . . . . . . . . . xiii**
本書の対象読者. . . . . . . . . . . . . xiii
本書の構成 . . . . . . . . . . . . . . xiii
関連情報 . . . . . . . . . . . . . . . xv
本書で使用されている規則 . . . . . . . . xv

**第1章 EJB プログラミング環境のアーキテクチャーの概要 . . . . . . . . . . . . . . 1**
EJB 環境のコンポーネント . . . . . . . . 1
EJB サーバー . . . . . . . . . . . . . 2
　セキュリティー・サービス . . . . . . . 4
　ワークロード管理サービス . . . . . . . 7
　パーシスタンス・サービス . . . . . . . 7
　ネーム解決サービス. . . . . . . . . . 8
　トランザクション・サービス. . . . . . 9
データ・ソース . . . . . . . . . . . . 12
EJB クライアント . . . . . . . . . . . 13
Web サーバー . . . . . . . . . . . . . 15
管理インターフェース. . . . . . . . . . 15

**第2章 Enterprise Bean の概要 . . . . . . 17**
bean の基本 . . . . . . . . . . . . . 17
　エンティティー bean . . . . . . . . . 18
　セッション bean . . . . . . . . . . 20
EJB モジュールの作成 . . . . . . . . . 22
　EJB モジュール. . . . . . . . . . . 22
　デプロイメント・ディスクリプター . . . 22
EJB モジュールの配置 . . . . . . . . . 25
EJB アプリケーションの開発 . . . . . . 26
　例 : 銀行の場合の Enterprise Bean . . . 26
　銀行取引 bean を使用した EJB 銀行取引アプリケーションの開発. . . . . . . . . 28
Enterprise Bean インスタンスのライフ・サイクル. . . . . . . . . . . . . . . . 29
　セッション bean のライフ・サイクル . . 29
　エンティティー bean のライフ・サイクル 31

**第3章 EJB サーバー (AE) 環境での Enterprise Bean の開発用および配置用ツール . . . . . . . . . . . . . . . . 35**
VisualAge for Java の使用 . . . . . . . 35
EJB サーバー (AE) ツールでの Enterprise Bean の開発および配置 . . . . . . . . 36
　EJB サーバー (AE) 用のソフトウェアのインストールおよび構成. . . . . . . . . 37
　EJB サーバー (AE) 環境での CLASSPATH 環境変数の設定 . . . . . . . . . . . 38
　Enterprise Bean のコンポーネントの作成 38
　EJB サーバー (AE) でのファインダー・ロジックの作成. . . . . . . . . . . . 39
　EJB モジュールの作成 . . . . . . . . 39
　エンティティー bean が使用するデータベースの作成 . . . . . . . . . . . . . 41

**第4章 EJB サーバー (CB) 環境での Enterprise Bean の開発用および配置用ツール . . . . . . . . . . . . . . . . 43**
EJB サーバー (CB) ツールによる Enterprise Bean の開発および配置 . . . . . . . . 44
　EJB サーバー (CB) の前提条件ソフトウェア . . . . . . . . . . . . . . . . 46
　EJB サーバー (CB) 環境での CLASSPATH 環境変数の設定 . . . . . . . . . . . 47
　Enterprise Bean のコンポーネントの作成 48
　EJB サーバー (CB) でのファインダー・ロジックの作成. . . . . . . . . . . . 48
　Enterprise Bean 用の EJB JAR ファイルの作成. . . . . . . . . . . . . . . 54
　CBDeployJar ツールを使った Enterprise Bean の配置 . . . . . . . . . . . 70
　CBDeployEar ツールを使った Enterprise Bean の配置 . . . . . . . . . . . 72
　手作業による Enterprise Bean の配置. . . 73
　逐次列挙型を使用可能にするシステム管理の構成 . . . . . . . . . . . . . 100
CBConnector のライフ・サイクル・サービスの使用による EJB ホームの解決 . . . . 101


© Copyright IBM Corp. 1999, 2001

デフォルトのコンテキストとファインダーの関連付け . . . . . . . . . . . . . 102
アプリケーション固有のコンテキストおよび appbind ツール . . . . . . . . . 104
既存の CICS または IMS アプリケーションからの Enterprise Bean の作成 . . . . . 108
MQSeries と通信する Enterprise Bean の作成 109
EJB サーバー (CB) 環境での制約事項 . . . 112

**第5章 Enterprise Bean の開発 . . . . . 117**
CMP を持つエンティティー bean の開発 118
Enterprise Bean クラスの作成 (CMP を持つエンティティー) . . . . . . . . . 118
ホーム・インターフェースの作成 (CMP を持つエンティティー) . . . . . . . . 129
リモート・インターフェースの作成 (CMP を持つエンティティー) . . . . . . . . 133
1 次キー・クラスの作成 (CMP を持つエンティティー) . . . . . . . . . . 134
データベースとの対話 . . . . . . . . 137
セッション bean の開発 . . . . . . . . 138
Enterprise Bean クラスの作成 (セッション) . . . . . . . . . . . . . . . 138
ホーム・インターフェースの作成 (セッション) . . . . . . . . . . . . . . 151
リモート・インターフェースの作成 (セッション) . . . . . . . . . . . . . 152
複数のタイプの Enterprise Bean に共通なインターフェースのインプリメント . . . . 154
javax.ejb.EJBObject から継承されたメソッド . . . . . . . . . . . . . . . 154
javax.ejb.EJBHome インターフェース . . 154
java.io.Serializable インターフェースおよび java.rmi.Remote インターフェース . . 155
Enterprise Bean でのスレッドおよび再入可能性の使用 . . . . . . . . . . . . . 155
Enterprise Bean の EJB モジュールの作成 156
bean コンポーネントを Java パッケージの一部にする . . . . . . . . . . . 157
EJB モジュールおよびデプロイメント・ディスクリプターの作成 . . . . . . . 157

**第6章 Enterprise Bean でのトランザクションおよびセキュリティーの使用可能化 . . 159**
デプロイメント・ディスクリプターのトランザクション属性の設定 . . . . . . . . . 160
トランザクション属性の設定 . . . . . 160
トランザクション分離レベル属性の設定 164
デプロイメント・ディスクリプターのセキュリティー属性の設定 . . . . . . . . . . 167

**第7章 EJB クライアントの開発 . . . . . 169**
必要な Java パッケージのインポート . . . 171
bean の EJB オブジェクトへの参照の作成および取得 . . . . . . . . . . . . . . 172
EJB ホーム・オブジェクトの検索および作成 . . . . . . . . . . . . . . 173
EJB オブジェクトの作成 . . . . . . 178
セッション bean の無効な EJB オブジェクトの処理 . . . . . . . . . . . . . 179
bean の EJB オブジェクトの除去 . . . . 181
EJB クライアントでのトランザクションの管理 . . . . . . . . . . . . . . . . 182
EJB サーバー (CB) 固有の EJB クライアントに関する追加情報 . . . . . . . . . . 184
ActiveX を使用する EJB クライアント 185
Component Broker のセッション・サービスを使用するクライアント . . . . . . 185

**第8章 Enterprise Bean を使用するサーブレットの開発 . . . . . . . . . . . . 189**
標準のサーブレット・メソッドの概要 . . . 189
サーブレットを組み込んだ HTML ページの作成 . . . . . . . . . . . . . . . 190
サーブレットの開発 . . . . . . . . . . 191
サーブレットのインスタンス変数 . . . 192
サーブレットの init メソッド . . . . . 193
サーブレットの doGet メソッド . . . . 196
Enterprise Bean の作成 . . . . . . . 198
ユーザー応答の内容の決定 . . . . . . 198
ユーザー応答の送信 . . . . . . . . 199
スレッド化に関する問題 . . . . . . . 200

**第9章 Enterprise Bean に関する高度なプログラミングの概念 . . . . . . . . . 201**
BMP を持つエンティティー bean の開発 201
Enterprise Bean クラスの作成 (BMP を持つエンティティー) . . . . . . . . . 202
ホーム・インターフェースの作成 (BMP を持つエンティティー) . . . . . . . . 214
リモート・インターフェースの作成 (BMP を持つエンティティー) . . . . . . . . 216

1 次キー・クラスの作成または選択 (BMP を持つエンティティー) . . . . . . . . 218
BMP エンティティー bean によるデータベースの使用 . . . . . . . . . . . . . . 219
EJB サーバー (CB) 環境での接続の管理 220
EJB サーバー (AE) 環境でのデータベース接続の管理 . . . . . . . . . . 223
データベース内のデータの操作 . . . . 226
bean 管理トランザクションの使用 . . . . 228

**第10章 WebSphere プログラミング・モデル拡張機能 . . . . . . . . . . . . . 233**
分散例外パッケージ . . . . . . . . . . 233
概説 . . . . . . . . . . . . . . . 234
DistributedException クラスの拡張 . . . 237
DistributedExceptionEnabled インターフェースのインプリメント . . . . . . . . 238
分散例外の使用法. . . . . . . . . . 243
コマンド・パッケージ . . . . . . . . . 245
概説 . . . . . . . . . . . . . . . 246
コマンド・インターフェースの書き込み 250
コマンド・インターフェースのインプリメント . . . . . . . . . . . . . . 253
コマンドの使用法. . . . . . . . . . 261
コマンド・ターゲットの書き込み (サーバー). . . . . . . . . . . . . . . 263
ターゲットおよびターゲット・ポリシー 266
コマンド・ターゲットの書き込み (クライアント側のアダプター) . . . . . . . . 272
ローカライズ可能テキスト・パッケージ . . 277
概説 . . . . . . . . . . . . . . . 277
ローカライズ可能アプリケーションの作成 286
オプション引き数の使用 . . . . . . . 290
フォーマット Enterprise Bean の配置 . . 300

**付録A. EJB 仕様バージョン 1.1 の変更内容 . . . . . . . . . . . . . . . . . 303**
新しいフィーチャーおよび更新されたフィーチャー . . . . . . . . . . . . . . 303
バージョン 1.0 からバージョン 1.1 への移行 . . . . . . . . . . . . . . . . 303

**付録B. WebSphere Application Server に提供されているコード例 . . . . . . . . 307**
本書に記載する例の説明 . . . . . . . . 307
EJB サーバー (AE) 環境における他の例の説明 . . . . . . . . . . . . . . . . 309
EJB サーバー (CB) 環境における他の例の説明 . . . . . . . . . . . . . . . . 309

**付録C. Enterprise Bean での XML の使用 (CB のみ) . . . . . . . . . . . . . . 311**
標準ヘッダーおよび EJB JAR タグの作成 312
入力ファイル・タグおよび出力ファイル・タグの作成. . . . . . . . . . . . . . 312
エンティティー bean タグの作成 . . . . 313
セッション bean タグの作成 . . . . . . 314
すべての Enterprise Bean によって使用されるタグの作成 . . . . . . . . . . . 315

**付録D. EJB 仕様に対する機能拡張 . . . 319**
アクセス bean . . . . . . . . . . . . 319
Enterprise Bean 間の関連 . . . . . . . 320
Enterprise Bean における継承 . . . . . . 320

**特記事項 . . . . . . . . . . . . . . 323**
商標およびサービス・マーク . . . . . . 326

**索引 . . . . . . . . . . . . . . . . 329**

1. EJB 環境のコンポーネント . . . . . . 1
2. 分散トランザクションの例 . . . . . 10
3. エンティティー bean のコンポーネント 19
4. セッション bean のコンポーネント 21
5. 配置されたエンティティー bean の主なコンポーネント . . . . . . . . . . 25
6. EJB アプリケーションの概念図. . . . 26
7. コード例: EJB サーバー (AE) 用の AccountBeanFinderHelper インターフェース . . . . . . . . . . . . . . . 39
8. コード例: EJB サーバー (CB) 用に生成された AccountFinderHelper クラス . . 52
9. コード例: EJB サーバー (CB) 用に完成された AccountFinderHelper クラス . . 52
10. jetace ツールの初期ウィンドウ . . . . 55
11. jetace ツールの「Basic (基本)」ページ 58
12. jetace ツールの「Entity (エンティティー)」ページ . . . . . . . . . . 60
13. jetace ツールの「Session (セッション)」ページ . . . . . . . . . . . . . 62
14. jetace ツールの「Transactions (トランザクション)」ページ . . . . . . . . . 64
15. jetace ツールの「Security (セキュリティー)」ページ . . . . . . . . . . 66
16. jetace ツールの「Environment (環境)」ページ. . . . . . . . . . . . . . 67
17. jetace ツールの「Dependencies (依存関係)」ページ . . . . . . . . . . . 69
18. コード例: AccountBean クラス 120
19. コード例: AccountBean クラスの変数 121
20. コード例: AccountBean クラスのビジネス・メソッド . . . . . . . . . 123
21. コード例: AccountBean クラスの ejbCreate メソッドおよび ejbPostCreate メソッド . . . . . . . . . . . . 126
22. コード例: AccountBean クラスでの EntityBean インターフェースのインプリメント. . . . . . . . . . . . . 129
23. コード例: AccountHome ホーム・インターフェース . . . . . . . . . 131
24. コード例: findLargeAccounts メソッド 132
25. コード例: Account リモート・インターフェース. . . . . . . . . . . . 134
26. コード例: ItemKey 1 次キー・クラス 136
27. コード例: TransferBean クラス 141
28. コード例: TransferBean クラスのビジネス・メソッド . . . . . . . . . 143
29. コード例: TransferBean クラスの ejbCreate メソッドでの InitialContext オブジェクトの作成 . . . . . . . . 146
30. コード例: getProviderURL メソッド 147
31. コード例: TransferBean クラスの ejbCreate メソッドでの AccountHome オブジェクトの作成 . . . . . . . 148
32. コード例: Enterprise Bean の環境名コンテキストの検索. . . . . . . . . 149
33. コード例: TransferBean クラスでの SessionBean インターフェースのインプリメント. . . . . . . . . . . . 150
34. コード例: TransferHome ホーム・インターフェース . . . . . . . . . 152
35. コード例: Transfer リモート・インターフェース. . . . . . . . . . . . 153
36. コード例: Java アプリケーション TransferApplication の import ステートメント. . . . . . . . . . . . . 172
37. コード例: InitialContext オブジェクトの作成 . . . . . . . . . . . . 176
38. コード例: EJBHome オブジェクトの作成 . . . . . . . . . . . . . 178
39. コード例: EJB オブジェクトの作成 179
40. コード例: セッション bean の EJB オブジェクト参照のリフレッシュ . . . 180
41. コード例: セッション EJB オブジェクトの除去. . . . . . . . . . . . 182
42. コード例: EJB クライアントでのトランザクションの管理 . . . . . . . . 184
43. コード例: ORB の初期化 (アクセス bean を使用している場合) . . . . . 186
44. コード例: InitialContext オブジェクトの作成 (アクセス bean を使用していない場合) . . . . . . . . . . . . 186

© Copyright IBM Corp. 1999, 2001

45. コード例: sessionCurrent オブジェクトの作成および使用 . . . . . . . . . 187
46. コード例: CreateAccount サーブレットにアクセスするために使用する create.html ファイルのコンテンツ. . . . 190
47. CreateAccount サーブレットの初期フォームおよび出力. . . . . . . . . . 191
48. コード例: CreateAccount クラス 192
49. コード例: CreateAccount クラスのインスタンス変数 . . . . . . . . . . 193
50. コード例: CreateAccount サーブレットの init メソッド . . . . . . . . . 195
51. コード例: CreateAccount サーブレットの doGet メソッド . . . . . . . . . 197
52. コード例: doGet メソッドでの Enterprise Bean の作成 . . . . . . . 198
53. コード例: doGet メソッドでのユーザー応答の決定 . . . . . . . . . . . 199
54. コード例: doGet メソッドでのユーザーへの応答. . . . . . . . . . . . . 200
55. コード例: AccountBMBean クラス 203
56. コード例: AccountBMBean クラスのインスタンス変数. . . . . . . . . . 205
57. コード例: AccountBMBean クラスの ejbCreate メソッド. . . . . . . . . 208
58. コード例: AccountBMBean クラスの ejbFindByPrimaryKey メソッド. . . . . 210
59. コード例: AccountBMBean クラスの ejbFindLargeAccounts メソッド. . . . . 211
60. コード例: AccountBMHome ホーム・インターフェース. . . . . . . . . . . 215
61. コード例: AccountBM リモート・インターフェース . . . . . . . . . . 218
62. コード例: setEntityContext メソッドでの JDBC ドライバーのロードおよび登録 . 221
63. コード例: AccountBMBean クラスの checkConnection メソッドおよび makeConnection メソッド . . . . . . 222
64. コード例: AccountBMBean クラスの dropConnection メソッド. . . . . . . 223
65. コード例: setEntityContext メソッドにおけるデータ・ソース bean インスタンスへの EJB オブジェクト参照の取得 (DataSource を使用するように再作成したもの) . . . . . . . . . . . . . 224
66. コード例: AccountBMBean クラスの checkConnection メソッドおよび makeConnection メソッド (DataSource を使用するように再作成したもの) . . . 225
67. コード例: AccountBMBean クラスの dropConnection メソッド (DataSource を使用するように再作成したもの) . . . 226
68. コード例: ejbCreate メソッドでの SQL 更新呼び出しの構成および実行 . . . 227
69. コード例: ejbLoad メソッドでの ResultSet オブジェクトの操作 . . . . 228
70. コード例: トランザクション・コンテキストをカプセル化するオブジェクトの取得 . . . . . . . . . . . . . . . 230
71. コードの例: DistributedException クラスのコンストラクター . . . . . . . . 235
72. コードの例: DistributedException クラスを拡張する例外内のコンストラクター . 238
73. コードの例: DistributedExceptionEnabled インターフェースをインプリメントする例外クラスの構造 . . . . . . . . . 239
74. コードの例: DistributedExceptionEnabled インターフェースをインプリメントする例外クラスのコンストラクター . . . 240
75. コードの例: DistributedExceptionEnabled インターフェースでのメソッドのインプリメンテーション . . . . . . . . . 242
76. コードの例: DistributedExceptionEnabled インターフェースをインプリメントする例外のテスト . . . . . . . . . . 243
77. コードの例: チェーンへの例外の追加 244
78. コードの例: チェーンからの例外の抽出 245
79. コードの例: ターゲットを定めることができるコマンド用のインターフェースの構造 . . . . . . . . . . . . . . 247
80. コードの例: ターゲットを定め、補正が可能なコマンド用のインターフェースの構造 . . . . . . . . . . . . . . 247
81. コードの例: コマンド・インターフェースのインプリメンテーション・クラスの構造 . . . . . . . . . . . . . . 248
82. コードの例: コマンド・ターゲット・エンティティー bean の構造 . . . . . 249
83. コードの例: ModifyCheckingAccountCmd インターフェース . . . . . . . . . 252

84. コードの例: ModifyCheckingAccountCmdImpl クラスの構造 . . . 253
85. コードの例: ModifyCheckingAccountCmdImpl クラスの変数 . . . 254
86. コードの例: ModifyCheckingAccountCmdImpl クラスのコンストラクター . . . 255
87. コードの例: ModifyCheckingAccountCmdImpl クラスのコマンド固有のメソッド . . . 256
88. コードの例: ModifyCheckingAccountCmdImpl クラスの Command インターフェースからのメソッド . . . 257
89. コードの例: ModifyCheckingAccountCmdImpl クラスの TargetableCommand インターフェースからのメソッド . . . 258
90. コードの例: ModifyCheckingAccountCmdImpl クラスの CompensableCommand インターフェースからのメソッド . . . 259
91. コードの例: ModifyCheckingAccountCompensatorCmd クラスの変数およびコンストラクター . 260
92. コードの例: ModifyCheckingAccountCompensatorCmd クラスのメソッド . . . 261
93. コードの例: ModifyCheckingAccountCmd コマンドの使用法 . . . 262
94. コードの例: ModifyCheckingAccountCompensator コマンドの使用法 . . . 263
95. コードの例: CheckingAccount エンティティー bean のリモート・インターフェース、およびコマンド・ターゲット . . 264
96. コードの例: CheckingAccount エンティティー bean の bean クラス、およびコマンド・ターゲット . . . 265
97. コードの例: TargetPolicyDefault クラス 267
98. コードの例: CommandTarget を使ったターゲットの識別 . . . 268
99. コードの例: CommandTargetName を使ったターゲットの識別 . . . 269
100. コードの例: 外部アプリケーションでのターゲットへのコマンドのマッピング . 270
101. コードの例: カスタム・ターゲット・ポリシーの作成 . . . 271
102. コードの例: カスタム・ターゲット・ポリシーの使用法 . . . 272
103. コードの例: ターゲット用のクライアント側アダプターの構造 . . . 273
104. コードの例: クライアント側のアダプターのインスタンス化 . . . 273
105. コードの例: executeCommand メソッドのクライアント側のインプリメンテーション . . . 275
106. コードの例: サーブレットでのコマンドの実行 . . . 276
107. 英語メッセージ・カタログ内の 3 つのエレメント . . . 280
108. ドイツ語メッセージ・カタログ内の 3 つのエレメント . . . 280
109. コード例: LocalizableTextFormatter オブジェクトの作成およびオブジェクトでの値の設定 . . . 288
110. コード例: プログラムによるロケールの設定 . . . 289
111. コード例: LocalizableTextFormatter オブジェクトのフォーマット . . . 290
112. 変数サブストリングを持つメッセージ・カタログ項目 . . . 291
113. コード例: 変数サブストリングを持つメッセージのフォーマット . . . 292
114. 2 つの変数サブストリングを持つメッセージ・カタログ項目 . . . 293
115. コード例: ローカライズ可能変数サブストリングを持つメッセージのフォーマット . . . 294
116. コード例: LocalizableTextDateTimeArgument クラスの構造 . . . 297
117. コード例: LocalizableTextDateTimeArgument クラス内のメソッド . . . 298
118. コード例: LocalizableTextDateTimeArgument クラス内の format メソッド . . . 299
119. フォーマット Enterprise Bean の配置 302

120. 配置したフォーマット Enterprise Bean の削除 . . . . . . . . . . . . . . 302
121. コード例: 標準ヘッダーおよび EJB JAR タグ . . . . . . . . . . . . . 312
122. コード例: 入力ファイル・タグおよび出力ファイル・タグ . . . . . . . . . 312
123. コード例: エンティティー bean 固有のタグ . . . . . . . . . . . . . 314
124. コード例: セッション bean 固有のタグ 315
125. コード例: すべての Enterprise Bean に使用されるタグ . . . . . . . . . 316
126. コード例: メソッド固有のタグ 317

# 表


1. 本書で使用する規則 . . . . . . . . xv
2. FinderHelperBase クラスのメソッド 50
3. FinderHelperBase メソッドのサフィックス . . . . . . . . . . . . . . 53
4. トランザクション・コンテキストに対する Enterprise Bean のトランザクション属性の影響 . . . . . . . . . . . 163
5. EJB サーバー (AE) で使用できる例 309
6. EJB サーバー (CB) で使用できる例 309


© Copyright IBM Corp. 1999, 2001 

# 本書について

本書では、WebSphere™ Application Server プログラミング環境における、 Sun Microsystems Enterprise JavaBeans™ 仕様に合わせた Enterprise Bean の開発について説明しています。また、Enterprise Bean にアクセスすることができる EJB クライアントの開発についても説明しています。

## 本書の対象読者

本書は、WebSphere Application Server での Enterprise Bean および EJB クライアントのプログラミングに関する概要を必要とする開発者およびシステム設計者を対象としています。本書では、プログラマーが、オブジェクト指向プログラミング、分散プログラミング、および Web ベースのプログラミングに精通していることを前提としています。また、Sun Microsystems Java™ プログラム言語に関する知識も必要です。

## 本書の構成

本書は、以下のように構成されています。

- 1ページの『第1章 EJB プログラミング環境のアーキテクチャーの概要』では、WebSphere Application Server の EJB サーバー環境のマクロ的な概要について説明します。
- 17ページの『第2章 Enterprise Bean の概要』では、Enterprise Bean に関連する主な概念について説明します。
- 35ページの『第3章 EJB サーバー (AE) 環境での Enterprise Bean の開発用および配置用ツール』では、EJB サーバー (AE) 環境に含まれるツールのセットアップ方法および使用方法について説明します。また、その環境で Enterprise Bean を開発および配置する際の主要なステップについても説明します。EJB サーバー (AE) は、WebSphere Application Server アドバンスド版で使用可能な EJB サーバーのインプリメンテーションです。
- 43ページの『第4章 EJB サーバー (CB) 環境での Enterprise Bean の開発用および配置用ツール』では、EJB サーバー (CB) 環境に含まれるツールのセットアップ方法および使用方法について説明します。また、その環境で Enterprise Bean を開発および配置する際の主要なステップについても説明し

© Copyright IBM Corp. 1999, 2001 

ます。EJB サーバー (CB) は、WebSphere Application Server エンタープライズ版の一部としての Component Broker で使用可能な EJB サーバーのインプリメンテーションです。

- 117ページの『第5章 Enterprise Bean の開発』では、コンテナー管理のパーシスタンス (CMP) を持つエンティティー bean およびセッション bean の開発方法について説明します。また、その後の配置のために Enterprise Bean をパッケージ化する方法についても説明します。
- 159ページの『第6章 Enterprise Bean でのトランザクションおよびセキュリティーの使用可能化』では、適切なデプロイメント・ディスクリプター属性を使用することによって Enterprise Bean でトランザクションを使用可能にする方法について説明します。
- 169ページの『第7章 EJB クライアントの開発』では、EJB クライアントが Enterprise Bean を使用するために必要とする基本的なコードについて説明します。本章では、Enterprise Bean、Java アプリケーション、および Enterprise Bean を使用する Java サーブレットに関する一般的な問題についても説明します。
- 189ページの『第8章 Enterprise Bean を使用するサーブレットの開発』では、Enterprise Bean にアクセスするサーブレットで必要な基本的コードについて説明します。
- 201ページの『第9章 Enterprise Bean に関する高度なプログラミングの概念』では、bean 管理のパーシスタンスを持つ単純なエンティティー bean を開発する方法について説明するとともに、Enterprise Bean 自体のトランザクションを管理する Enterprise Bean に必要な基本的コードについて説明します。
- 303ページの『付録A. EJB 仕様バージョン 1.1 の変更内容』では、EJB 仕様のバージョン 1.1 における新しいフィーチャーまたは変更されたフィーチャーを紹介し、EJB 仕様のバージョン 1.0 に合わせて作成された Enterprise Beans の移行問題について説明します。
- 307ページの『付録B. WebSphere Application Server に提供されているコード例』では、本書を通じて使用する主な例について説明するとともに、さまざまな版の WebSphere Application Server で配布される追加の例についても示します。
- 311ページの『付録C. Enterprise Bean での XML の使用 (CB のみ)』では、extensible markup language (XML) について説明します。XML は、EJB サーバー (CB) 環境 で Enterprise Bean とともに使用するデプロイメント・ディスクリプターを作成する際に使用することができます。

- 319ページの『付録D. EJB 仕様に対する機能拡張』には、EJB 仕様に対する WebSphere Application Server に固有な拡張が示されています。これらの拡張機能の使用は、VisualAge for Java だけでサポートされます。

## 関連情報

本書で説明しているトピックに関する詳細については、以下の資料を参照してください。

- *WebSphere Application Server 概説*
- *WebSphere ビジネス構築のソリューション*
- Component Broker「*問題判別ガイド*」
- Component Broker「*システム管理ガイド*」
- *Component Broker リリース・ノート*

## 本書で使用されている規則

本書では、以下の書体およびキー操作の規則を使用しています。

*表 1. 本書で使用する規則*

| 規則 | 意味 |
| --- | --- |
| **太字** | コマンド名を示します。グラフィカル・ユーザー・インターフェース (GUI) に関しては、太字は、メニュー、メニュー項目、ラベル、およびボタンも示します。 |
| モノスペース | コマンド・プロンプトで入力しなければならないテキスト、およびコマンド、関数、ならびにリソース定義属性やそれらの値など、示されているとおりに使用しなければならない値を示します。モノスペースは、画面上のテキストおよびコード例も示します。 |
| *斜体* | 指定しなければならない変数値を示します (例 : *fileName* にファイルの名前を指定します)。斜体は、強調および資料の表題も示します。 |
| Ctrl-*x* | ここで、*x* はキーの名前であり、制御文字のシーケンスを示します。たとえば、<Ctrl-c> は、<Ctrl> キーを押した状態で <c> キーを押すことを意味します。 |
| Return | Return、Enter、または左矢印のラベルが付いたキーを指します。 |
| `%` | **root** 特権を必要としないコマンドに対する UNIX コマンド・シェル・プロンプトを表します。 |
| `#` | **root** 特権が必要なコマンドに対する UNIX コマンド・シェル・プロンプトを表します。 |
| `C:¥>` | Windows NT® のコマンド・プロンプトを示します。 |

*表 1. 本書で使用する規則 (続き)*

| 規則 | 意味 |
| --- | --- |
| **>** | メニューの記述に使用している場合は、一連のメニューの選択を示します。たとえば、「**「File (ファイル)」 -> 「New (新規)」**とクリックします」とある場合は、「**「File (ファイル)」**メニューから**「New (新規)」**コマンドをクリックする」ことを意味します。 |
| | ツリー表示の記述に使用している場合は、一連のフォルダーまたはオブジェクトの展開を示します。たとえば、「**「Management Zones (管理ゾーン)」 -> 「Sample Cell and Work Group Zone (サンプル・セルおよびワークグループ・ゾーン)」 -> 「Configuration (構成)」を展開します**」とある場合は、以下を意味します。<br>1. 「Management Zones (管理ゾーン)」フォルダーを展開する<br>2. 「Sample Cell and Work Group Zone (サンプル・セルおよびワークグループ・ゾーン)」という名前の管理ゾーンを展開する<br>3. 「Configurations (構成)」フォルダーを展開する<br>**注:** 表示内のオブジェクトの横にプラス記号 (**+**) がある場合は、そのオブジェクトを展開することができます。オブジェクトを展開すると、プラス記号はマイナス記号 (**-**) に変わります。 |
| **+** | ツリー構造を展開して、オブジェクトをさらに表示させます。展開するには、任意のオブジェクトの横にあるプラス記号 (**+**) をクリックします。 |
| **-** | ツリー構造の分岐を縮小し、その分岐に含まれるオブジェクトを表示から除きます。ツリー構造の分岐を縮小するには、その分岐の先頭にあるオブジェクトの横のマイナス記号 (**-**) をクリックします。 |
| コマンドの入力 | コマンドを「入力する」または「発行する」ように示されている場合は、コマンドを入力してから Return を押します。たとえば、「**ls** コマンドを入力します」という手順は、コマンド・プロンプトで **ls** と入力してから Return を押すことを意味します。 |
| [ ] | 構文記述でオプション項目を囲みます。 |
| { } | 構文記述において、項目を選択しなければならないリストを囲みます。 |
| \| | 構文記述において、中括弧 (**{ }**) で囲まれた選択項目のリストにある項目を分離します。 |
| ... | 構文記述にある省略記号は、先行する項目を 1 回以上繰り返すことができることを示します。例にある省略記号は、簡潔にするために例から情報が省略されていることを示します。 |
| IN | 関数の説明において、関数にデータを渡すために値が使用されるパラメーターを示します。これらのパラメーターが、変更されたデータを呼び出し側ルーチンに戻すために使用されることはありません。(ユーザーのコードに IN 宣言は入れないでください。) |

*表 1. 本書で使用する規則 (続き)*

| 規則 | 意味 |
| --- | --- |
| OUT | 関数の説明において、変更されたデータを呼び出し側ルーチンに戻すために値が使用されるパラメーターを示します。これらのパラメーターが関数にデータを渡すために使用されることはありません。(ユーザーのコードに OUT 宣言は入れないでください。) |
| INOUT | 関数の説明において、値が関数に渡され、関数によって変更されて呼び出し側ルーチンに戻されるパラメーターを示します。これらのパラメーターは、IN パラメーターおよび OUT パラメーターの両方として機能します。 (ユーザーのコードに INOUT 宣言は入れないでください。) |
| $CICS | CICS 製品をインストールした絶対パス名を示します。たとえば、Windows NT では C:¥opt¥cics、Solaris では /opt/cics です。CICS という名前の環境変数が製品のパス名に設定されている場合は、示されているとおりに例を使用することができます。その他の場合は、$CICS のすべてのインスタンスを CICS 製品のパス名で置換する必要があります。 |
| オープン・システム上の CICS | サポートされているすべての UNIX プラットフォームのための CICS 製品の総称です。 |
| TXSeries CICS | CICS for AIX、CICS for Solaris、および CICS for Windows NT 製品の総称です。 |
| CICS | オープン・システム上の CICS および CICS for Windows NT 製品の総称です。特定のバージョンのオープン・システム上の CICS 製品を指している場合は、オープン・システム上の CICS 製品間の相違点を強調しています。<br>CICS ファミリーのその他の CICS 製品は、オペレーティング・システムによって区別されます (CICS for OS/2 や、ESA、MVS、および VSE プラットフォーム用の IBM メインフレーム・ベースの CICS など)。 |

# 第1章 EJB プログラミング環境のアーキテクチャーの概要

ここ数年で、WWW によって企業が顧客に提供するサービスが変わりました。最初は、単に Web ホーム・ページがあるだけで十分でした。その後、各社はアクティブな Web サイトを配置し、顧客が製品やサービスを注文できるようにし始めました。今日では、これらの方法で Web を使用することに加えて、Web ベースのシステムと他の業務システムを統合する必要があります。 IBM® WebSphere Application Server は、特に Enterprise Bean のサポートとともに、この統合を実現するモデルおよびツールを提供します。

## EJB 環境のコンポーネント

Sun Microsystems Enterprise JavaBeans (EJB) 仕様の IBM のインプリメンテーションを利用することで、 WebSphere Application Server アドバンスド版および WebSphere Application Server エンタープライズ版のユーザーは、 Web ベースのシステムを他の業務システムと統合することができます。 WebSphere EJB サーバーおよびそれに関連するコンポーネントによってこれらのインプリメンテーションが提供されます。これを図1 に示します。

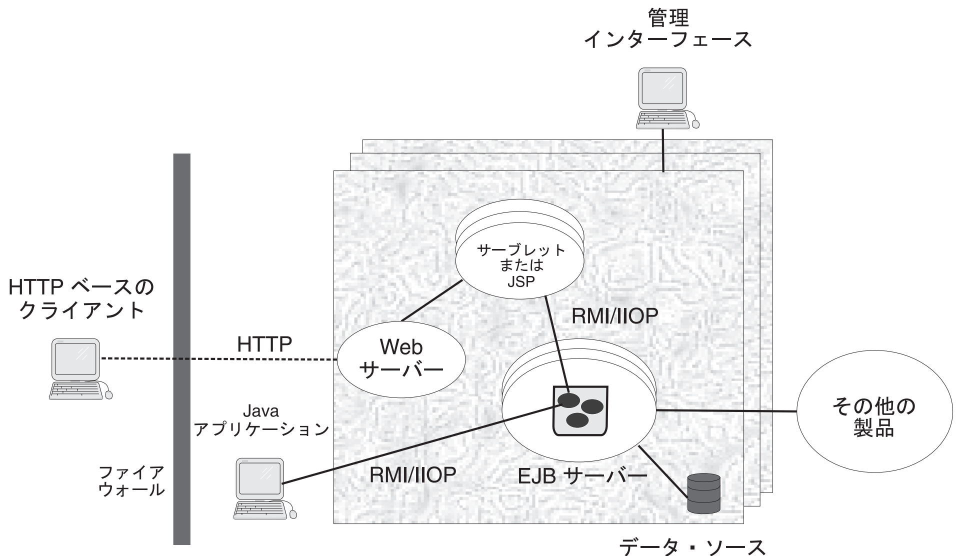



図 1. EJB 環境のコンポーネント

© Copyright IBM Corp. 1999, 2001

WebSphere EJB サーバー環境は、以下のコンポーネントを含みます。これらのコンポーネントに関する詳細については、該当するセクションで説明します。

- *EJB サーバー* − WebSphere EJB サーバーは、1 つ以上の *Enterprise Bean* を含みます。この Enterprise Bean は、EJB クライアントによって使用および共用されるビジネス・ロジックおよびデータをカプセル化します。 EJB サーバーにインストールされた Enterprise Bean は、サーバーと直接通信しません。代わりに、*EJB コンテナー* が Enterprise Bean と EJB サーバーの間のインターフェースを提供するとともに、スレッド処理、トランザクションのサポート、データの格納と取得の管理などの多くの基本的なサービスを提供します。詳細については、『EJB サーバー』を参照してください。
- データ・ソース − Enterprise Bean には、セッション bean とエンティティー bean の 2 種類があります。前者は、存続期間の短いクライアント固有のタスクおよびオブジェクトをカプセル化し、後者は、*永続* データをカプセル化します。 EJB サーバーは、この永続データをデータ・ソースに保管し、そこから取得します。データ・ソースはデータベースにも別のアプリケーションにもすることができ、場合によってはファイルでも構いません。詳細については、12ページの『データ・ソース』を参照してください。
- *EJB クライアント* − EJB クライアントには、一般に 2 つの種類があります。
  - *HTTP* ベースのクライアント。 HTTP (Hypertext Transfer Protocol) によって Java サーブレットまたは JavaServer Pages™ (JSP) のいずれかを使用して、EJB サーバーと対話します。
  - *Java* アプリケーション。IIOP 上の Java RMI (RMI/IIOP) を使用することによって、EJB サーバーと直接対話します。

  詳細については、13ページの『EJB クライアント』を参照してください。
- *管理インターフェース* − 管理インターフェースによって、EJB サーバー環境を管理することができます。詳細については、15ページの『管理インターフェース』を参照してください。

## EJB サーバー

EJB サーバーは、WebSphere Application Server の第 3 層アーキテクチャーのアプリケーション・サーバー層であり、クライアント層 (Java サーブレット、アプレット、アプリケーション、および JSP) をリソース管理層 (データ・ソース) と接続します。 WebSphere Application Server は、2 つのタイプの EJB サーバーを含みます。WAS アドバンスド版を購入されたお客さまには、これらの EJB サーバーの 1 つのみが提供されます。WAS エンタープライズ版を購入されたお客さまには、両方が提供されます。EJB サーバーを総称する場合

は、本書では *EJB サーバー* という用語を使用します。本書で特にいずれか一方を指す必要がある場合は、以下の用語を使用します。

- *EJB サーバー (AE)* − WAS アドバンスド版に付属する EJB サーバーです。(WAS アドバンスド版は WAS エンタープライズ版の一部として使用できるので、この EJB サーバーは WAS エンタープライズ版でも使用することができます。)
- *EJB サーバー (CB)* − WAS エンタープライズ版にのみ付属する EJB サーバーであり、Component Broker (CB) の一部です。

EJB サーバーは、EJB サーバー・ランタイム、EJB コンテナー、および Enterprise Bean の 3 つのコンポーネントを持っています。 EJB コンテナーは、Enterprise Bean を下位の EJB サーバーから分離し、bean とコンテナーの間の標準的なアプリケーション・プログラミング・インターフェース (API) を提供します。 EJB 仕様がこの API を定義します。

EJB サーバー (CB) は、エンティティー・コンテナーおよびセッション・コンテナーの 2 つの標準的なタイプのコンテナーを含みます。名前に示されているように、これらのコンテナーは、それぞれエンティティー bean およびセッション bean を扱うように特に最適化されています。 EJB サーバー (AE) は、エンティティー bean とセッション bean の両方をサポートする 1 つの標準のコンテナーを持ちます。

EJB サーバーとコンテナー・コンポーネントを一緒に使用すると、その中に配置された Enterprise Bean に関する下記のサービスにアクセスできるようになります。

- Enterprise Bean を配置するツール。 bean を配置すると、配置ツールは、事前に配置されている bean を構成するインターフェースをインプリメントするいくつかのクラスを作成します。さらに、配置ツールは、Java ORB、スタブ、およびスケルトン・クラスを生成し、リモート・メソッド呼び出しを可能にします。エンティティー bean の場合は、ツールは、bean の 永続データを格納するデータ・ソースと bean の間の対話を処理するための永続化機能およびファインダーの各クラスも生成します。 Enterprise Bean を配置するには、前もって開発者が *EJB モジュール* および関連する*デプロイメント・ディスクリプター* を作成しておかなければなりません。デプロイメント・ディスクリプターは、モジュール内の各 Enterprise Bean に関する情報、および bean の処理方法についてのコンテナーへの指示を提供します。配置に関する詳細については、25ページの『EJB モジュールの配置』を参照してください。

- セキュリティー・サービス。EJB サーバー環境にあるリソースにアクセスする必要があるプリンシパルに対する認証および許可を処理します。詳細については、『セキュリティー・サービス』を参照してください。
- ワークロード管理サービス。リソースが効率的に使用されます。詳細については、7ページの『ワークロード管理サービス』を参照してください。
- パーシスタンス・サービス。エンティティー bean とそのデータ・ソースの間の対話を処理し、永続データが適切に管理されるようにします。詳細については、7ページの『パーシスタンス・サービス』を参照してください。
- ネーム解決サービス。デプロイメント・ディスクリプターで定義された bean の名前をネーム・スペースにエクスポートします。 EJB サーバーは、Java Naming and Directory Interface™ (JNDI) を使用してネーム解決サービスをインプリメントしています。詳細については、8ページの『ネーム解決サービス』を参照してください。
- トランザクション・サービス。bean のデプロイメント・ディスクリプターにトランザクション属性をインプリメントします。詳細については、9ページの『トランザクション・サービス』を参照してください。

## セキュリティー・サービス

企業のコンピューター処理がある場所に集中するいくつかの強力なメインフレームのみによって行われていたときには、許可されたユーザーしかコンピューターのサービスや情報にアクセスしないようにするのは非常に容易でした。分散コンピューティング・システムでは、ユーザー、アプリケーション・サーバー、およびリソース管理プログラムが世界中に分散する可能性もあるので、コンピューター・リソースのセキュリティーを確保するのはずっと複雑になっています。それでもやはり、基盤にある問題は基本的に同じです。

### 認証および許可

よいセキュリティー・サービスは、認証および許可の 2 つの主な機能を提供します。

*認証* 処理は、*プリンシパル* (ユーザーまたはコンピューターのプロセス) が最初にコンピューター・リソースにアクセスしようとするときに行われます。その時点では、セキュリティー・サービスはプリンシパルを検査し、プリンシパルが名乗るとおりの相手であるかどうかを確認します。通常、ユーザーが人間であれば、ユーザー ID およびパスワードを入力することによって自身を証明します。プロセスは、通常、暗号化された鍵を提示します。パスワードまたは鍵が有効な場合は、セキュリティー・サービスは、ユーザーに*トークン* または*チケット* を発行します。これらによって、プリンシパルが識別され、プリンシパルが認証されたことが示されます。

プリンシパルは、認証されると、セキュリティー・サービスによって保護されたコンピューター・システムの境界内にあるリソースの使用を試みることができます。しかし、プリンシパルは、使用を許可されている場合にのみ、特定のコンピューター・リソースを使用することができます。*許可* は、認証されたプリンシパルがリソースの使用を要求し、そのリソースを使用する権利がユーザーに付与されているかどうかをセキュリティー・サービスが判別するときに行われます。通常、許可は、リソースの使用を許可するプリンシパル (またはプリンシパルのグループ) を定義する、アクセス制御リスト (ACL) をリソースと関連付けることによって行われます。プリンシパルが許可されると、リソースにアクセスすることができます。

分散コンピューティング環境では、プリンシパルおよびリソースは、それぞれ身元が名乗るとおりであることが証明されるまでは、互いの身元を疑わなければなりません。このようにする必要があるのは、プリンシパルが身元を偽ってリソースにアクセスしようとしたり、リソースがトロイの木馬であり、プリンシパルから有用な情報を得ようとしたりする可能性があるためです。この問題を解決するために、*信頼される第三者* として機能し、プリンシパルおよびリソースを認証してエンティティーが互いに身元を証明できるようにするセキュリティー・サーバーがセキュリティー・サービスに含まれています。このセキュリティー・プロトコルは、*相互認証* として知られています。

### EJB サーバー環境でのセキュリティー・サーバーの使用

2 つの EJB サーバー環境におけるセキュリティー・サービスには、いくつかの類似点があります。両方の EJB サーバー環境において、セキュリティー・サービスは、デプロイメント・ディスクリプターで定義される *アクセス制御* および*実行識別* セキュリティー属性を*使用しません*。しかし、*実行モード* 属性は、ユーザー ID をセキュリティー・コンテキストにマップするための基礎として使用されます。この属性に関する詳細については、22ページの『デプロイメント・ディスクリプター』を参照してください。

以下のセクションでは、2 つのセキュリティー・サービスの主要な相違点について説明します。

***EJB サーバー (AE) 環境でのセキュリティー:*** EJB サーバー (AE) 環境では、セキュリティー・サービスの主なコンポーネントは、セキュリティー Enterprise Bean を含む EJB サーバーです。システム管理者がセキュリティー・サービスを管理する場合は、セキュリティー EJB サーバー内でセキュリティー bean を操作します。

EJB クライアントが認証されると、操作する Enterprise Bean に対するメソッドを呼び出すことができます。メソッド呼び出しに関連するプリンシパルがメ

ソッドの呼び出しに必要なアクセス権を持つ場合は、メソッドが正常に呼び出されます。これらのアクセス権は、アプリケーション・レベル (管理者が定義する一連の Web とオブジェクト・リソース) およびメソッド・グループ・レベル (管理者が定義する一連の Java インターフェースとメソッドの組) で設定することができます。アプリケーションは、複数のメソッド・グループを含むことができます。

一般に、メソッドを呼び出すプリンシパルは、複数の Web サーバーおよび EJB サーバーにわたる呼び出しに関連します (この関連は、*代行* と呼ばれます)。この方法でメソッド呼び出しを代行すると、EJB クライアントのユーザーは、一度認証を受ける必要があるだけです。 HTTP cookie を使用して、ユーザーの認証情報を複数の Web サーバーにわたって伝搬させることができます。これらの cookie の存続時間はブラウザー・セッションの存続時間に等しく、提供されている logout メソッドによって、ユーザーの完了時にこれらの cookie が破壊されます。

EJB サーバー (AE) 環境でのセキュリティー管理に関する詳細については、WebSphere InfoCenter および WebSphere 管理コンソールで使用可能なオンライン・ヘルプを参照してください。

***EJB サーバー (CB) 環境でのセキュリティー:*** EJB サーバー (CB) 環境では、ネットワーク内のすべての Component Broker ネーム・サーバーおよびアプリケーション・サーバーについてセキュリティーを確保しなければなりません。各サーバー・ホストでネーム・サーバーのセキュリティーを確保すると、そのサーバーにあるシステム・オブジェクト (Component Broker ネーム・スペースで使用される名前コンテキストを含む) に対する無許可アクセスが防止されます。アプリケーション・サーバーのセキュリティーを確保すると、そのサーバーにあるアプリケーションのビジネス・オブジェクトに対する無許可アクセスが防止されます。

ネーム・サーバーおよびアプリケーション・サーバーのセキュリティーを確保するには、以下を行う必要があります。

- 分散コンピューティング環境 (DCE) をインストールおよび構成し、サーバーに認証サービスを提供する。これによって、サーバー間のアクセスに対するセキュリティーが確保されます。
- クライアントおよびサーバーのキー・リングを構成し、Java ベースの SSL クライアントに認証サービスを提供する。
- アプリケーション・サービスにあるビジネス・オブジェクトのアクセスに対する許可を構成する。

- 代行ポリシーを作成し、アプリケーション・サーバーが要求側クライアント・プリンシパルを他のサーバーに渡すことができるようにする。
- 認証マッピングを構成して、第 3 層システムへのアクセスを提供する。
- クライアントとアプリケーション・サーバーの間で流れるメッセージを保護するために使用する保護の品質を構成する。

これらの各作業に関する詳細については、Component Broker「*システム管理ガイド*」を参照してください。

## ワークロード管理サービス

ワークロード管理サービスによって複数の EJB サーバーをサーバー・グループ にグループ化することによって、EJB サーバー環境のスケーラビリティーが向上します。この場合、クライアントは、これらのサーバーが単一の EJB サーバーであるかのようにこれらのサーバー・グループにアクセスし、ワークロード管理サービスは、サーバー・グループ内の EJB サーバーにわたって作業負荷を均等に分散させます。 1 つの EJB サーバーは、1 つのサーバー・グループのみに属することができます。

サーバー・グループの作成は、EJB サーバー (AE) 環境用の WebSphere 管理コンソール内、および EJB サーバー (CB) 環境用のシステム管理エンド・ユーザー・インターフェース内で処理される管理用タスクです。ワークロード管理に関する詳細については、 WebSphere InfoCenter および該当する管理インターフェースのオンライン・ヘルプを参照してください。

## パーシスタンス・サービス

Enterprise Bean には、セッション bean とエンティティー bean の 2 種類があります。セッション bean は、特定のクライアントに関連する一時データをカプセル化します。エンティティー bean は、データ・ソースに格納される永続データをカプセル化します。詳細については、 17ページの『第2章 Enterprise Bean の概要』を参照してください。

パーシスタンス・サービスによって、エンティティー bean に関連するデータが、データ・ソース内の対応するデータと正しく確実に同期されます。この処理を行うために、パーシスタンス・サービスは、トランザクション・サービスとともに機能して、適切な時点でデータ・ソースに対するデータの挿入、更新、抽出、および除去を行います。

エンティティー bean には、コンテナー管理のパーシスタンス (CMP) を持つものと bean 管理のパーシスタンス (BMP) を持つものの 2 種類があります。CMP を持つエンティティー bean では、パーシスタンス・サービスが、永続デ

ータの管理に必要なほぼすべての処理を行います。 BMP を持つエンティティー bean では、bean 自体が、永続データの管理に必要なほとんどの処理を行います。

EJB サーバー (AE) 環境では、パーシスタンス・サービスは、以下のコンポーネントを使用して処理を行います。

- Java データベース・コネクティビティー (JDBC™) API。リレーショナル・データベースへの共通インターフェースをエンティティー bean に提供します。
- Java トランザクション・サポート。 11ページの『EJB サーバー環境でのトランザクションの使用』で説明します。 EJB サーバーは、永続データが必ず適切なトランザクション・コンテキスト内で処理されることを保証します。

EJB サーバー (CB) 環境では、パーシスタンス・サービスは、以下のコンポーネントを使用して処理を行います。

- X/Open XA インターフェース。リレーショナル・データベースへの標準インターフェースをエンティティー bean に提供します。
- Object Management Group (OMG) の Object Transaction Service (OTS)。これについては、11ページの『EJB サーバー環境でのトランザクションの使用』で説明します。

## ネーム解決サービス

オブジェクト指向分散コンピューティング環境では、クライアントには、オブジェクトを検索して識別し、クライアント、オブジェクト、およびリソースが同じマシンにあるように見せるためのメカニズムがなければなりません。ネーム解決サービスは、このメカニズムを提供します。EJB サーバー環境では、JNDI を使用して実際のネーム解決サービスを隠蔽し、ネーム解決サービスへの共通インターフェースを提供します。

JNDI は、ネーム解決およびディレクトリー機能を Java アプリケーションに提供しますが、API は、ネーム解決およびディレクトリー・サービスの特定のインプリメンテーションには依存しません。このインプリメンテーションの独立性によって、異なるネーム解決およびディレクトリー・サービスを JNDI API によるアクセスに使用することができます。したがって、Java アプリケーションは、LDAP (Lightweight Directory Access Protocol)、ドメイン・ネーム・サービス (DNS)、DCE セル・ディレクトリー・サービス (CDS) などの多くの既存のネーム解決およびディレクトリー・サービスを使用することができます。

JNDI は、Java のオブジェクト・モデルを使用して Java アプリケーション用に設計されています。 JNDI を使用すると、Java アプリケーションは、任意の Java オブジェクト・タイプの名前付きオブジェクトを格納および取得することができます。また、JNDI は、属性とオブジェクトの関連付けや、属性を使用したオブジェクトの検索などの標準のディレクトリー操作を実行するためのメソッドも提供します。

EJB サーバー環境では、デプロイメント・ディスクリプターを使用して Enterprise Bean の JNDI 名を指定します。 EJB サーバーを開始すると、これらの名前が JNDI に登録されます。

## トランザクション・サービス

*トランザクション* とは、1 つの整合した状態から別の状態にデータを変換する一連の操作のことです。この一連の操作は分割不可能な作業単位であり、コンテキストによっては、トランザクションが*作業論理単位* (LUW) と呼ばれる場合もあります。トランザクションは、分散システム・プログラミング用のツールであり、障害からの回復を容易にします。

トランザクションは、以下の *ACID 特性* を提供します。

- *アトミシティ (Atomicity)* : トランザクションの変更はアトミックです。つまり、トランザクションの部分となるすべての操作が行われるか行われないかのいずれかです。
- *整合性 (Consistency)* : トランザクションは、整合状態間でデータを移動します。
- *分離 (Isolation)* : トランザクションを並列に実行することができる場合でも、あるトランザクションで進行中の作業は別のトランザクションから不可視です。トランザクションは、逐次実行されているように見えます。
- *耐障害性 (Durability)* : トランザクションが正常に完了した後には、そのトランザクションによる変更は、その後の障害の影響を受けません。

例として、ある口座から別の口座に送金するトランザクションを考えてみます。このような送金には、ある口座から資金を引き出し、別の口座に預金する必要があります。ある口座から資金を引き出し、別の口座に預金する処理は、1 つの*アトミック* ・トランザクションを構成する 2 つの部分です。つまり、両方が完了しない場合には、いずれも行ってはなりません。 1 つの口座に対して同時に複数の要求を処理する場合は、互いに*分離* して、ある一時点で単一のトランザクションのみが口座を変更できるようにしなければなりません。送金の直後に銀行の中央コンピューターに障害が発生した場合であっても、システムが再度利用可能になったときには、正しい残高が示されなければなりませ

ん。つまり、変更には*耐障害性* が必要です。*整合性* がアプリケーションの機能であることに注意してください。つまり、ある口座から別の口座に送金する場合に、アプリケーションは、ある口座から減算する金額と同じ金額を別の口座に加算しなければなりません。

トランザクションは、コミットまたはロールバックの 2 つの方法のいずれかで完了することができます。トランザクションが正常に完了することをコミットと呼びます。トランザクションが失敗することを*ロールバック* と呼びます。ロールバックしたトランザクションによって行われたデータの変更は、すべて変更前の状態に完全に戻されます。この送金の例では、ある口座から資金を引き出したが、障害が発生して別の口座にその資金を預金することができない場合は、最初の口座に対する変更がすべて変更前の状態に完全に戻されます。次に任意の送信元が口座残高を照会するときには、正しい残高が示されなければなりません。

**分散トランザクションおよび 2 フェーズ・コミット処理**

*分散トランザクション* は、複数のプロセス (多くの場合はいくつかのマシン) で実行されるトランザクションです。各プロセスがトランザクションに参加します。これを図2 に示します。ここで、それぞれの長円形は異なるマシンで実行される作業を示し、それぞれの矢印は リモート・メソッド呼び出し (RMI) を示します。

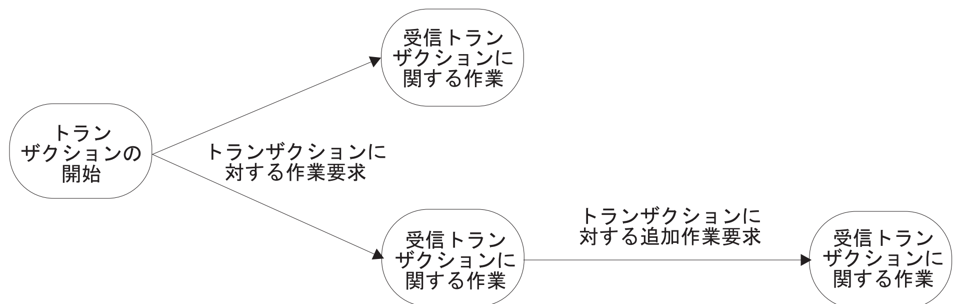



*図 2. 分散トランザクションの例*

分散トランザクションは、ローカル・トランザクションと同様に、ACID 特性を厳守しなければなりません。しかし、分散トランザクションでは、障害がどのプロセスでも発生する可能性があり、そのような障害が発生した場合には、各プロセスがトランザクションに代わって既に完了している処理を元に戻さなければならないので、これらの特性の維持は非常に複雑です。

分散トランザクション処理システムは、2 つの機能を使用することによって、分散トランザクションでの ACID 特性を維持します。

- *回復可能プロセス* : 回復可能プロセスは、障害が発生した場合に以前の状態を復元することができるプロセスです。
- *コミット・プロトコル* : コミット・プロトコルによって、複数のプロセスが、トランザクションのコミットまたはロールバック (中断) を調整することができます。最も一般的なコミット・プロトコルは、EJB サーバーでも使用されている 2 フェーズ・コミット・プロトコルです。

トランザクション状態情報は、すべての回復可能プロセスによって格納されなければなりません。しかし、アプリケーション・データを管理するプロセス (リソース管理プログラムなど) に限っては、データに対する変更の記述を格納しなければなりません。分散トランザクションに関係するすべてのプロセスが回復可能である必要はありません。一般に、クライアントはリソース管理プログラムと直接対話しないため、回復可能ではありません。回復可能でないプロセスは、*一時* プロセスと呼びます。

*2 フェーズ・コミット・プロトコル* には、その名前が示すように、準備フェーズと解決フェーズの 2 段階が必要です。各トランザクションで、1 つのプロセスがコーディネーターとして機能します。*コーディネーター* は、トランザクションの他の参加者の活動を監督し、結果の整合性を確保します。

*準備フェーズ* では、コーディネーターは、トランザクションの各プロセスにメッセージを送信し、各プロセスにコミットの準備を要求します。プロセスが準備を行うと、トランザクションをコミットし、その作業を永続的に記録することができることが保証されます。コミットできることが保証された後は、一方的にトランザクションをロールバックすることはできなくなります。プロセスが準備を行うことができない (つまり、トランザクションをコミットできることを保証できない) 場合は、トランザクションをロールバックしなければなりません。

*解決フェーズ* では、コーディネーターが応答を集計します。すべての参加者がコミットの準備を行っている場合は、トランザクションをコミットします。それ以外の場合は、トランザクションをロールバックします。いずれの場合も、コーディネーターは、すべての参加者に結果を通知します。コミットの場合は、参加者は、コミットしたことを確認します。

### EJB サーバー環境でのトランザクションの使用

Enterprise Bean トランザクション・モデルは、OMG OTS バージョン 1.1 にほとんどの点で対応します。トランザクション可能な Enterprise Bean インスタンスは、OTS TransactionalObject インターフェースのオブジェクトに対応します。ただし、Enterprise Bean トランザクション・モデルは、トランザクションのネストをサポートしていません。

EJB サーバー環境では、トランザクションは、トランザクション・サービスの 3 つの主要なコンポーネントを処理します。

- トランザクション管理プログラム・インターフェース。これによって、EJB サーバーが、bean に指定されたトランザクション属性に基づいて Enterprise Bean 内でトランザクション境界を制御することができます。
- インターフェース UserTransaction。これによって、Enterprise Bean または EJB クライアントがトランザクションを管理することができます。コンテナーは、ネーム解決サービスによって、Enterprise Bean および EJB クライアントに対してこのインターフェースを使用可能にします。
- X/Open XA インターフェースによる調整。これによって、トランザクション・リソース管理プログラム (データベースなど) が、外部のトランザクション管理プログラムによって制御されるトランザクションに参加できるようになります。

ほとんどの目的では、Enterprise Bean の開発者は、トランザクション管理に関する作業をコンテナーに代行させることができます。開発者がこの代行を行うには、トランザクションのデプロイメント・ディスクリプター属性を設定します。これらの属性および値については、 160ページの『デプロイメント・ディスクリプターのトランザクション属性の設定』で説明します。

場合によっては、Enterprise Bean の開発者が bean レベルでトランザクションを管理したり、EJB クライアントにトランザクションを管理させたりする必要があります。この方法に関する詳細については、 228ページの『bean 管理トランザクションの使用』を参照してください。

## データ・ソース

エンティティー bean は、回復可能データ・ソースに永続的に格納しなければならない永続データを含みます。多くの場合、EJB 仕様ではエンティティー bean に関連する永続データを格納する場所としてデータベースを挙げていますが、オペレーティング・システムのファイルや他のアプリケーションなどの他のデータ・ソースを使用することも可能になっています。

エンティティー bean とデータ・ソースの間の対話をコンテナーに処理させる場合は、コンテナーがサポートしているデータ・ソースを使用しなければなりません。

- EJB サーバー (AE) は、DB2®、Oracle、Sybase、および InstantDB をサポートしています。
- EJB サーバー (CB) は、DB2、Oracle、CICS®、IMS™、および MQSeries® をサポートしています。

追加のコードを作成して BMP エンティティー bean とデータ・ソースの間の対話を処理する必要がある場合は、ユーザーの要件に合い、パーシスタンス・サービスと互換性のある任意のデータ・ソースを使用することができます。詳細については、 201ページの『BMP を持つエンティティー bean の開発』を参照してください。

## EJB クライアント

EJB クライアントは、Java アプリケーション、Java サーブレット、Java アプレットとサーブレットの組み合わせ、または JSP ファイルのいずれかの形式をとることができます。 EJB サーバー (CB) の場合は、Java アプレットを使用して、Enterprise Bean と直接対話することができます。 EJB サーバー (AE) の場合は、Java アプレットは、サーブレットとの組み合わせでのみ使用することができます。

Enterprise Bean のアクセスおよび操作に必要な EJB クライアントのコードは、各種の Java EJB クライアントの間で非常に類似しています。 EJB クライアントの開発者は、以下の点について考慮しなければなりません。

- *ネーム解決および通信* − Java EJB クライアントは、HTTP または RMI のいずれかを使用して Enterprise Bean と通信しなければなりません。幸い、JNDI が EJB クライアントとネーム解決サービスの間の対話を隠蔽するため、EJB クライアントと Enterprise Bean の間で通信できるようにするために必要なコーディングには、ほとんど差はありません。
  – Java アプリケーションは、RMI/IIOP を使用して Enterprise Bean と通信します。
  – Java サーブレットおよび JSP ファイルは、HTTP を使用して Enterprise Bean と通信します。 EJB サーバーとともにサーブレットを使用するには、Web サーバーをインストールして、EJB サーバー環境のマシンで構成しなければなりません。詳細については、 15ページの『Web サーバー』を参照してください。
- *スレッド化* − Java クライアントは、クライアントが実行する必要があるタスクに応じて、単一スレッドにすることもマルチスレッドにすることもできます。セッション bean によって提供されるサービスを使用する各クライアント・スレッドは、その bean の別個のスレッドを作成するか検索して、スレッドが完了するまでその bean への参照を保持しなければなりません。複数のクライアント・スレッドは、同じエンティティー bean にアクセスすることができます。

- セキュリティー
  - HTTP を介して EJB サーバー (AE) にアクセスする EJB クライアント (サーブレットや JSP ファイルなど) には、以下の 2 つの層のセキュリティーが課されます。
    1. URL (Universal Resource Locator) セキュリティー。このセキュリティーは、セキュリティー・サービスとともに、Web サーバーに接続された WebSphere Application Server セキュリティー・プラグインによって強制されます。
    2. Enterprise Bean セキュリティー。このセキュリティーは、セキュリティー・サービスとともに機能するサーバーで強制されます。

    HTTP ベースの EJB クライアントのユーザーが Enterprise Bean にアクセスしようとすると、 (WebSphere Server プラグインを使用する) Web サーバーは、そのユーザーを認証します。この認証は、ユーザー ID およびパスワードを求める要求のフォームで行うことも、証明書を交換した後にセキュア・ソケット・レイヤー (SSL) セッションを確立するフォームで透過的に行うこともできます。

    認証ポリシーは、追加のオプションとしてのセキュア・チャネル制約によって制御されます。セキュア・チャネル制約が必要な場合は、SSL セッションは、認証の最終段階として確立しなければなりません。それ以外の場合は、SSL はオプションです。
  - EJB サーバー (CB) にアクセスする EJB クライアント、および RMI を使用して EJB サーバー (AE) にアクセスする EJB クライアント (Java アプリケーションなど) には、2 番目のセキュリティー層のみが課されます。 HTTP ベースの EJB クライアントのように、これらの EJB クライアントはセキュリティー・サービスで認証しなければなりません。

    詳細については、 4ページの『セキュリティー・サービス』を参照してください。
- トランザクション ― 両方の種類の Java クライアントで、JTA インターフェースによってトランザクション・サービスを使用してトランザクションを管理することができます。トランザクション管理に必要なコードは、2 種類のクライアントのいずれでも同じです。トランザクションおよび Java トランザクション・サービスに関する一般情報については、9ページの『トランザクション・サービス』を参照してください。Java EJB クライアントでのトランザクションの管理に関する詳細については、182ページの『EJB クライアントでのトランザクションの管理』を参照してください。

EJB サーバー (CB) 環境では、 Enterprise Bean は、Microsoft® ActiveX®、CORBA ベースの Java、および (ある程度までは) CORBA ベースの C++ を

使用する EJB クライアントからアクセスすることもできます。詳細については、184ページの『EJB サーバー (CB) 固有の EJB クライアントに関する追加情報』を参照してください。

**注:** EJB サーバー (AE) 環境では、Enterprise Bean への ActiveX および CORBA ベースのアクセスはサポートされていません。

## Web サーバー

EJB サーバーの機能にアクセスするには、Java サーブレットおよび JSP ファイルが Web サーバーにアクセスしなければなりません。 Web サーバーによって、Web クライアントと EJB サーバーの間の通信が可能になります。 EJB サーバー、Web サーバー、および Java サーブレットは、それぞれ異なるマシンに置くことができます。

EJB サーバーがサポートしている Web サーバーに関する詳細については、WAS アドバンスド版の「*はじめに*」を参照してください。

## 管理インターフェース

EJB サーバー (CB) および EJB サーバー (AE) には、それぞれ独自の管理ツールがあります。

- EJB サーバー (AE) は、WebSphere 管理コンソールを使用します。このインターフェースの詳細については、 WebSphere InfoCenter および WebSphere 管理コンソールで使用可能なオンライン・ヘルプを参照してください。
- EJB サーバー (CB) は、システム管理エンド・ユーザー・インターフェース (SM EUI) を使用します。このインターフェースに関する詳細については、Component Broker「*システム管理ガイド*」を参照してください。

EJB サーバー (AE) は、**wscp** コマンド行ツールを使用して管理することもできます。詳しくは、アドバンスド版情報センターを参照してください。

# 第2章 Enterprise Bean の概要

本章では、Enterprise Bean の特性および目的について説明します。本章では、2 つの基本的な Enterprise Bean とそれらのライフ・サイクルについて説明します。また、3 層化した分散アプリケーションを作成するための Enterprise Bean の結合方法例も記載されています。

## bean の基本

Enterprise Bean は Java コンポーネントであり、その他の Enterprise Bean やその他の Java コンポーネントと結合させて、3 層の分散アプリケーションを作成することができます。 Enterprise Bean には、以下の 2 つのタイプがあります。

- エンティティー bean。これは、永続データをカプセル化してデータベースやファイル・システムなどのデータ・ソースに保管したり、そのデータを操作する関連メソッドをカプセル化したりします。ほとんどの場合、エンティティー bean には、なんらかのトランザクションを介してアクセスしなければなりません。エンティティー bean のインスタンスは一意であり、複数のユーザーがアクセスすることができます。

  たとえば、銀行口座に関する情報をエンティティー bean にカプセル化することができます。口座エンティティー bean には、口座 ID、口座のタイプ (当座預金か普通預金か)、残高、およびそれらの変数を操作するメソッドが含まれます。

- セッション bean。特定の EJB クライアントに関連する一時 (非永続) データをカプセル化します。エンティティー bean 内のデータとは異なり、セッション bean 内のデータは永続データ・ソースに保管されないため、このデータが失われても問題は発生しません。ただし、セッション bean は、エンティティー bean にアクセスするなどして、使用しているデータベース内のデータを更新することができます。また、セッション bean はトランザクションに参加することもできます。

  セッション bean のインスタンスは作成時にはすべて同じですが、セッション bean の中には半永続データを保管できるものもあります。半永続データによって、ライフ・サイクルにおける任意の時点でセッション bean が一意になります。 1 つのセッション bean は常に単一のクライアントに関連付けられるため、同時に複数の呼び出しを行おうとすると、例外が throw されます。

© Copyright IBM Corp. 1999, 2001

たとえば、2 つの銀行口座間で送金する場合の作業を、セッション bean にカプセル化することができます。このような送金セッション bean は、口座エンティティー bean の 2 つのインスタンスを (口座 ID を使用して) 検索し、一方の口座から指定の金額を引き出して、同じ金額を他方の口座に追加することができます。

## エンティティー bean

本セクションでは、エンティティー bean の基本について説明します。

### エンティティー bean の基本コンポーネント

すべてのエンティティー bean には、 19ページの図3 に示すように、以下のコンポーネントがなければなりません。

- *bean クラス* – このクラスは、エンティティー bean のデータをカプセル化します。また、データにアクセスする、開発者がインプリメントしたビジネス・メソッドも含みます。このクラスは、コンテナーがエンティティー bean インスタンスのライフ・サイクルを管理するために使用するメソッドも含みます。EJB クライアント (その他の Enterprise Bean か、サーブレットなどのユーザー・コンポーネントかに関係なく) がこのクラスのオブジェクトに直接アクセスすることはありません。代わりに、ホーム・インターフェースおよびリモート・インターフェースに関連するコンテナー生成のクラスを使用して、エンティティー bean インスタンスを操作します。
- *ホーム・インターフェース* – このインターフェースは、クライアントがエンティティー bean のインスタンスの作成、検索、および除去を行うために使用するメソッドを定義します。このインターフェースは、配置時に、コンテナーによって、一般に *EJB ホーム・クラス* と呼ばれるクラスにインプリメントされます。このため、インスタンスは *EJB ホーム・オブジェクト* と呼ばれます。
- *リモート・インターフェース* – クライアントは、エンティティー bean へのアクセスを取得するのに一度ホーム・インターフェースを使用すると、bean クラスにインプリメントされたビジネス・メソッドを間接的に呼び出すために、このインターフェースを使用します。このインターフェースは、配置時に、コンテナーによって、一般に *EJB オブジェクト・クラス* と呼ばれるクラスにインプリメントされます。このため、インスタンスは *EJB オブジェクト* と呼ばれます。
- *1 次キー* – 特定のエンティティー bean インスタンスを一意に識別する 1 つ以上の変数。基本的な Java データ型の単一の変数から成り立つ 1 次キーは、配置時に指定することができます。 *1 次キー・クラス* は、複数の変数またはより複雑ないくつかの Java データ型から成り立つ複数の 1 次キー

をカプセル化する場合に使用します。また、1 次キー・クラスには、1 次キー・オブジェクトを作成して操作するためのメソッドも含まれます。

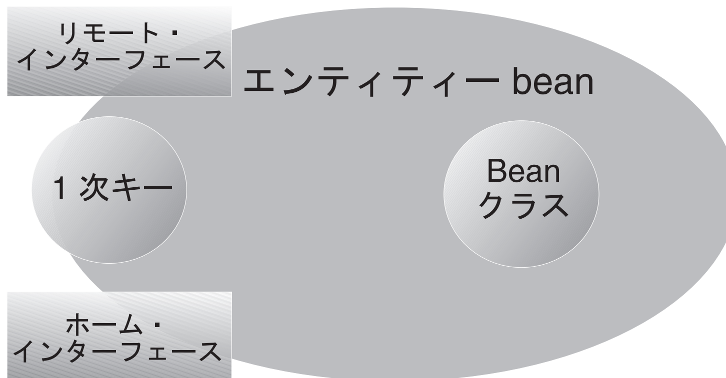



*図 3. エンティティー bean のコンポーネント*

**データのパーシスタンス**

エンティティー bean は、*永続* ビジネス・データをカプセル化して操作します。たとえば、銀行では、エンティティー bean を、顧客プロファイル、当座預金口座および普通預金口座、自動車ローン、抵当証券、および顧客のトランザクション履歴のひな型の作成に使用することができます。

このような重要なデータが失われないようにするために、エンティティー bean は、そのデータをデータベースなどのデータ・ソースに保管します。Enterprise Bean インスタンスのデータが変更されると、データ・ソース内のデータが bean データに同期されます。当然、この同期は該当のトランザクションのコンテキスト内で実行されるため、ルーターがダウンしたりサーバーに障害が発生した場合でも、永続的な変更内容は失われません。

エンティティー bean を設計する際には、このデータの同期を Enterprise Bean に操作させるか、コンテナーに操作させるかを決定しなければなりません。Enterprise Bean 自体がデータの同期を操作する Enterprise Bean は、*bean 管理のパーシスタンス* (BMP) をインプリメントするといい、コンテナーがデータの同期を操作する Enterprise Bean は、*コンテナー管理のパーシスタンス* (CMP) をインプリメントするといいます。

BMP をインプリメントする必要がない場合は、エンティティー bean が CMP を使用するように設計することをお勧めします。EJB サーバーがサポートして

いないデータ・ソースを使用する場合には、BMP を持つエンティティー bean を使用しなければなりません。 CMP を持つ Enterprise Bean のコードは、作成が簡単であり、特にデータ記憶機構にも依存しないため、EJB サーバー間で移送しやすくなります。

## セッション bean

本セクションでは、セッション bean の基本について説明します。

### セッション bean の基本コンポーネント

すべてのセッション bean には、 21ページの図4 に示すように、以下のコンポーネントがなければなりません。

- *bean クラス* − このクラスは、セッション bean に関連するデータをカプセル化します。また、このデータにアクセスする、開発者がインプリメントしたビジネス・メソッドも含みます。このクラスは、コンテナーがセッション bean インスタンスのライフ・サイクルを管理するために使用するメソッドも含みます。 EJB クライアント (その他の Enterprise Bean かユーザー・アプリケーションかに関係なく) がこのクラスのオブジェクトに直接アクセスすることはありません。代わりに、ホーム・インターフェースおよびリモート・インターフェースに関連するコンテナー生成のクラスを使用して、セッション bean を操作します。
- *ホーム・インターフェース* − このインターフェースは、クライアントがセッション bean のインスタンスの作成および除去を行うために使用するメソッドを定義します。このインターフェースは、配置時に、コンテナーによって、一般に *EJB ホーム・クラス* と呼ばれるクラスにインプリメントされます。このため、インスタンスは *EJB ホーム・オブジェクト* と呼ばれます。
- *リモート・インターフェース* − クライアントは、セッション bean へのアクセスを取得するのに一度ホーム・インターフェースを使用すると、bean クラスにインプリメントされたビジネス・メソッドを間接的に呼び出すために、このインターフェースを使用します。このインターフェースは、配置時に、コンテナーによって、一般に *EJB オブジェクト・クラス* と呼ばれるクラスにインプリメントされます。このため、インスタンスは *EJB オブジェクト* と呼ばれます。

エンティティー bean とは異なり、セッション bean には 1 次キー・クラスがありません。ユーザーは特定のセッション bean のインスタンスを検索する必要がないため、セッション bean には 1 次キー・クラスが不要です。

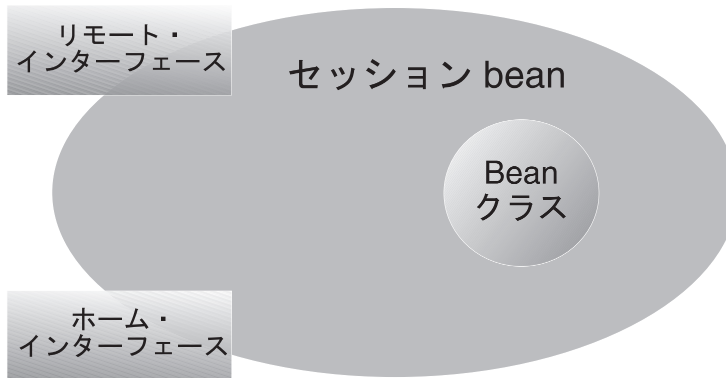



*図 4. セッション bean のコンポーネント*

**状態なしセッション bean と状態付きセッション bean**

セッション bean は、ユーザーのセッション、作業、一時的オブジェクトに関連するデータおよびメソッドをカプセル化します。セッション bean インスタンス内のデータは一時的であると定義されていますが、このデータが失われても実際に影響はありません。たとえば、銀行では、セッション bean は、送金、顧客プロファイルと新規口座の作成、および引き出しと預金に相当します。送金に関する情報を既に入力している場合に (まだコミットしていないが) サーバーに障害が発生しても、銀行口座の残高は変更されません。送金データが失われた場合にだけ、再入力することができます。

セッション bean の設計方法によって、そのデータが一時なのか永続なのかが決定します。

- セッション bean がメソッドに関する特定のデータを保守する必要がある場合は、このセッション bean は*状態付き* セッション bean と呼ばれます。セッション bean がメソッドに関するデータを保守する場合は、*会話状態* を持つといいます。 Web ベースのショッピング・カートは、状態付きセッション bean の典型的な使用法です。ショッピング・カートのユーザーがショッピング・カート内の商品を増減させるたびに、使用しているセッション bean インスタンスはカートの中身を記録しなければなりません。特定の EJB クライアントがいったん状態付きセッション bean のインスタンスを使用し始めたら、クライアントは、そのインスタンスに特定の状態が必要な限り、そのインスタンスを使用し続けなければなりません。ショッピング・カートの中身が発注される前にセッション bean インスタンスが失われた場合は、買い物客は新しいショッピング・カートをロードしなければなりません。

- セッション bean がメソッドに関する特定のデータを保守する必要がない場合は、このセッション bean は*状態なし セッション bean* と呼ばれます。138ページの『セッション bean の開発』で開発する Transfer セッション bean の例には、状態なしセッション bean の例が記載されています。状態なしセッション bean の場合は、すべてのインスタンスが同じであるため、クライアントはすべてのインスタンスを使用して任意のセッション bean のメソッドを呼び出すことができます。

## EJB モジュールの作成

Enterprise Bean の開発における最後のステップは、 EJB モジュールの作成です。 EJB モジュールは、次のものから成り立ちます。

- 1 つ以上の配置可能な Enterprise Bean。
- Extensible Markup Language (XML) ファイル内に格納される、デプロイメント・ディスクリプター。このファイルには、モジュール内の bean の構造および外部依存関係に関する情報と、アプリケーション内での bean の使い方を示すアプリケーション・アセンブリー情報が収められています。

EJB モジュールは、IBM の VisualAge for Java エンタープライズ版のような統合開発環境 (IDE) 内のツールを使用して作成することも、 Websphere 内に含まれるツールを使用して作成することもできます。詳細については、35ページの『第3章 EJB サーバー (AE) 環境での Enterprise Bean の開発用および配置用ツール』を参照してください。

EJB サーバー (CB) 環境での Enterprise Bean のパッケージに関する詳細については、 54ページの『Enterprise Bean 用の EJB JAR ファイルの作成』を参照してください。

### EJB モジュール

*EJB モジュール* は、 Enterprise Bean をアセンブルして 1 つの配置可能な単位にする場合に使用します。このファイルは、標準 Java アーカイブ・ファイル形式を使用します。 EJB モジュールには、個々の Enterprise Bean を入れることも、複数の Enterprise Bean を入れることもできます。詳細については、157ページの『EJB モジュールおよびデプロイメント・ディスクリプターの作成』を参照してください。

### デプロイメント・ディスクリプター

EJB モジュールには、 1 つ以上の配置可能な Enterprise Bean と 1 つの*デプロイメント・ディスクリプター* が入っています。デプロイメント・ディスクリプターには、モジュール内の各 bean の属性や環境設定が含まれており、コン

テナーがモジュール内のすべての bean の機能を呼び出す方法が定義されています。デプロイメント・ディスクリプター属性は、 Enterprise Bean 全体に対して、または bean 内の個々のメソッドに対して設定することができます。コンテナーは、メソッド・レベルの属性が定義されていない場合は、bean レベルの定義を使用します。メソッド・レベルの定義で使用できる属性には以下のものがあります。

デプロイメント・ディスクリプターには、エンティティー bean およびセッション bean に関する次の情報が含まれています。これらの属性は、bean でしか設定できません。属性は、bean の特定のメソッドについては設定できません。

- bean の名前、クラス、ホーム・インターフェース、リモート・インターフェース、および bean のタイプ (エンティティーまたはセッション)。
- *1 次キー・クラス* 属性 ー bean の 1 次キー・クラスを識別する。詳細については、134ページの『1 次キー・クラスの作成 (CMP を持つエンティティー)』または 218ページの『1 次キー・クラスの作成または選択 (BMP を持つエンティティー)』を参照してください。
- *パーシスタンス管理*。パーシスタンス管理を Enterprise Bean で実行するか、コンテナーで実行するかを指定する。
- *コンテナー管理フィールド* 属性 ー データ・ソースに対応するフィールドと同期させなければならない、bean クラス内の永続変数をリストする。同期させることによって、このデータが永続的で矛盾がないようにすることができます。詳細については、120ページの『変数の定義』を参照してください。
- *再入可能* 属性 ー Enterprise Bean がそれ自体に対してメソッドを呼び出せるようにするか、呼び出し側の bean に対するメソッドを呼び出す別の bean を呼び出せるようにするのかを指定する。エンティティー bean しか再入可能にすることはできません。詳細については、155ページの『Enterprise Bean でのスレッドおよび再入可能性の使用』を参照してください。
- *状態管理* 属性 ー セッション bean の会話状態を定義する。この属性は、STATEFUL か STATELESS に設定しなければなりません。これらの会話状態の意味に関する詳細については、21ページの『状態なしセッション bean と状態付きセッション bean』を参照してください。
- *タイムアウト* 属性 ー このセッション bean に関連するアイドル・タイムアウト値を秒単位で定義する (この属性は、標準デプロイメント・ディスクリプターの拡張です)。
- 環境変数の設定。
- リソース・ファクトリーなどの外部リソース、ホームなどの Enterprise Bean、およびセキュリティー役割に対する参照。

デプロイメント・ディスクリプターには、次のアプリケーション・アセンブリー情報が含まれています。

- モジュールを識別するためのアプリケーション名およびアイコン。
- クライアント・プログラムがモジュール内の bean にアクセスする際に必要となるクラス・ファイルの場所。
- *セキュリティー役割* ー アプリケーションを正常に使用するために特定のタイプのユーザーが受けるべき複数の許可を定義する。役割は、アプリケーションに対して同じアクセス権限を持っているユーザーのタイプを表します。
- *メソッド許可* ー Enterprise Bean のホーム・インターフェースおよびリモート・インターフェースの、指定されたメソッド・グループを呼び出す許可を定義する。この値は、メソッドごとに設定します。
- *トランザクション* 属性 ー コンテナーが、コンテナー管理のトランザクション区分を要求する Enterprise Bean のメソッドを呼び出す場合の、トランザクションの方法を定義する。この値は、メソッドごとに設定します。この属性の値については、 159ページの『第6章 Enterprise Bean でのトランザクションおよびセキュリティーの使用可能化』で説明します。
- *トランザクション分離レベル* 属性 ー コンテナーがどの程度までトランザクションを分離するかを定義する。この値は、メソッドごとに設定します。この属性の値については、 159ページの『第6章 Enterprise Bean でのトランザクションおよびセキュリティーの使用可能化』で説明します(この属性は、標準デプロイメント・ディスクリプターの拡張です)。
- *RunAsMode* 属性および *RunAsIdentity* 属性 ー *RunAsMode* 属性は、メソッドの呼び出しに使用する ID を定義する。特定の ID が必要な場合は、*RunAsIdentity* 属性を使用してその ID を識別します。この値は、bean ごとに設定します。*RunAsMode* 属性の値については、159ページの『第6章 Enterprise Bean でのトランザクションおよびセキュリティーの使用可能化』で説明しています (この属性は、標準デプロイメント・ディスクリプターの拡張です)。

次のバインディング属性はリポジトリーに格納されます (これは、デプロイメント・ディスクリプターの一部ではありません)。

- *JNDI ホーム名* 属性 ー JNDI (Java Naming and Directory Interface) のホーム名を指定する。この名前を使用して、EJB ホーム・オブジェクトのインスタンスが検出されます。この値は、bean ごとに設定します。このリポジトリー属性の値については、172ページの『bean の EJB オブジェクトへの参照の作成および取得』で説明します。

## EJB モジュールの配置

EJB モジュールを配置すると、配置ツールにより次のエレメントが作成または組み込まれます。

- *EJBObject* および *EJBHome* クラス (以後、EJB オブジェクトおよび EJB ホームと呼ぶ)。これらのクラスは、Enterprise Bean のホーム・インターフェースおよびリモート・インターフェースから、コンテナーによってインプリメントされます (CMP を持つエンティティー bean の場合はこの他に永続化機能クラスとファインダー・クラス)。
- スタブおよびスケルトン・ファイル。リモート・メソッド呼び出し (RMI) に必要です。

図5 に、エンティティー bean の配置を簡単に示します。

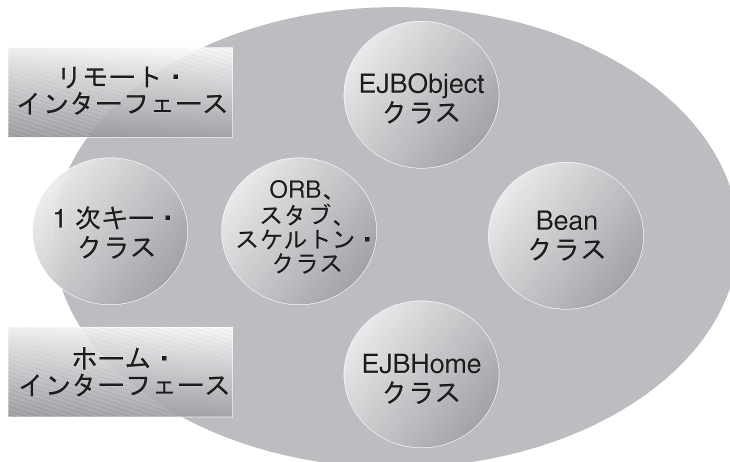



*図 5. 配置されたエンティティー bean の主なコンポーネント*

EJB モジュールは、さまざまなツールを使用して配置することができます。詳細については、35ページの『第3章 EJB サーバー (AE) 環境での Enterprise Bean の開発用および配置用ツール』または 43ページの『第4章 EJB サーバー (CB) 環境での Enterprise Bean の開発用および配置用ツール』を参照してください。

## EJB アプリケーションの開発

EJB アプリケーションを作成するには、ビジネス・データと機能をカプセル化する Enterprise Bean および EJB クライアントを作成してから、それらを適切に結合させせます。図6 は、1 つ以上のセッション bean、1 つ以上のエンティティー bean、あるいはそれら両方の bean を結合させせることによって EJB アプリケーションを作成する場合の概念図を示しています。 EJB クライアントでは個々のエンティティー bean およびセッション bean を直接使用することができますが、セッション bean はクライアントに関連付けるよう設計されており、エンティティー bean は永続データを保管するよう設計されているため、ほとんどの EJB エンティティー bean アプリケーションには、エンティティー bean にアクセスするセッション bean が含まれます。

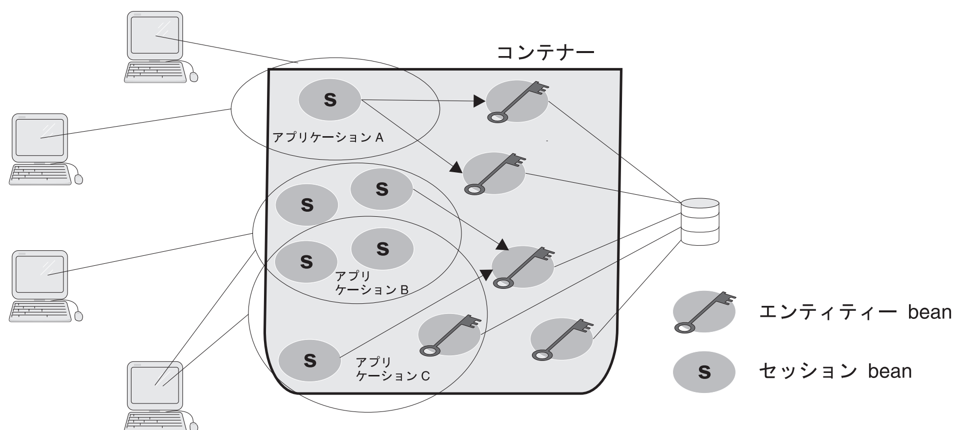



*図 6. EJB アプリケーションの概念図*

本セクションには、Enterprise Bean を結合して EJB アプリケーションを作成する場合の例を記載します。

### 例 : 銀行の場合の Enterprise Bean

銀行取引用に EJB アプリケーションを開発する場合は、以下のエンティティー bean を開発して、ビジネス・データと関連するメソッドをカプセル化することができます。

- Account bean － 顧客の当座預金口座および普通預金口座に関する情報を含むエンティティー bean。
- CarLoan bean － 自動車ローンに関する情報を含むエンティティー bean。
- Customer bean － 顧客が所有する口座や借用するローンなど、顧客に関する情報を含むエンティティー bean。

- CustomerHistory bean ー 指定の口座での顧客のトランザクションに関するレコードを含むエンティティー bean。
- Mortgage bean ー 住宅または商業用の抵当権に関する情報を含むエンティティー bean。

EJB クライアントはエンティティー bean またはセッション bean に直接アクセスすることができますが、EJB 仕様では、EJB クライアントは、特に複雑なアプリケーションにおいては、代わりにセッション bean を使用してエンティティー bean にアクセスするよう勧めています。したがって、銀行取引用に EJB を開発する場合は、クライアントの作業に相当する以下のセッション bean を作成することができます。

- LoanApprover bean ー CarLoan bean、Mortgage bean、またはその両方のインスタンスを使用してローンを承認することができるセッション bean。
- CarLoanCreator bean ー CarLoan bean の新しいインスタンスを作成するセッション bean。
- MortgageCreator bean ー Mortgage bean の新しいインスタンスを作成するセッション bean。
- Deposit bean ー Account bean の既存のインスタンスに指定額を振り込むセッション bean。
- StatementGenerator bean ー Customer および CustomerHistory エンティティー bean のインスタンスを適宜使用して、顧客の口座に関連する活動を要約するステートメントを生成するセッション bean。
- Payment bean ー CarLoan bean、Mortgage bean、またはその両方のインスタンスを使用して、顧客のローンに対する支払いを振り込むセッション bean。
- NewAccount bean ー Account bean の新しいインスタンスを作成するセッション bean。
- NewCustomer bean ー Customer bean の新しいインスタンスを作成するセッション bean。
- LoanReviewer bean ー (CarLoan bean、Mortgage bean、またはその両方のインスタンスを使用して) 顧客の未収ローンに関する情報にアクセスするセッション bean。
- Transfer bean ー Account bean の 2 つの既存のインスタンス間で指定額を送金するセッション bean。
- Withdraw bean ー Account bean の既存のインスタンスから指定額を引き出すセッション bean。

この例は、便宜上、簡略化されています。しかし、この Enterprise Bean のセットを使用することにより、該当のアプリケーション内で bean を適宜結合させて、異なるタイプのユーザーに対してさまざまな EJB アプリケーションを作成することができます。それから、1 つ以上の EJB クライアントを構築して、アプリケーションにアクセスさせることができます。

## 銀行取引 bean を使用した EJB 銀行取引アプリケーションの開発

(EJB 仕様ではなく) Sun Microsystems JavaBeans™ 仕様に従って構築された bean を使用する場合は、ボタンやテキスト・フィールドなどの事前定義のコンポーネントを結合させて、GUI アプリケーションを作成します。 Enterprise Bean を使用する場合は、銀行取引 bean などの事前定義のコンポーネントを結合させて、3 層化したアプリケーションを作成します。

たとえば、銀行取引 Enterprise Bean を使用して、以下の EJB アプリケーションを作成することができます。

- ホーム・バンキング・アプリケーション － このインターネット・アプリケーションによって、顧客は、口座間で送金したり (Transfer bean を使用)、既存の口座の資金を使用してローンの支払いを行ったり (Payment bean を使用)、自動車ローンや住宅の抵当を照会したり (CarLoanCreator bean または MortgageCreator bean を使用) することができる。
- 現金出納係アプリケーション － このイントラネット・アプリケーションによって、現金出納係は、新しい顧客の口座を作成したり (NewCustomer bean および NewAccount bean を使用)、口座間で送金したり (Transfer bean を使用)、顧客の預金と引き出しを記録したり (Withdraw bean および Deposit bean を使用) することができる。
- 貸付係アプリケーション － このイントラネット・アプリケーションによって、貸付係は、自動車ローンや住宅の抵当を作成したり承認したり (CarLoanCreator、MortgageCreator、LoanReviewer、および LoanApprover bean を使用) することができる。
- 明細作成アプリケーション － このバッチ・アプリケーションは、顧客の口座活動に関する詳細を月ごとに出力する (StatementGenerator bean を使用)。

これらの例は、銀行取引 bean で作成可能な EJB アプリケーションのサブセットにすぎません。

## Enterprise Bean インスタンスのライフ・サイクル

Enterprise Bean がコンテナーに配置されると、クライアントはその bean のインスタンスを必要に応じて作成したり使用したりすることができます。コンテナー内の Enterprise Bean のインスタンスは、定義されているライフ・サイクルを経過します。 Enterprise Bean のライフ・サイクルにおけるイベントは、EJB クライアントか、EJB サーバーのコンテナーのいずれかが開始したアクションによって実行されます。 Enterprise Bean の中には、Enterprise Bean のライフ・サイクルにおけるさまざまなイベントを処理するためのコードを、ユーザー自身が作成しなければならない場合があるため、ユーザーはこのライフ・サイクルを理解しておく必要があります。

本セクションに記載するメソッドについては、117ページの『第5章 Enterprise Bean の開発』で詳しく説明します。

### セッション bean のライフ・サイクル

本セクションでは、セッション bean インスタンスのライフ・サイクルについて説明します。状態付きセッション bean と状態なしセッション bean の違いも説明します。

#### 作成状態

セッション bean のライフ・サイクルは、bean のホーム・インターフェースで定義されている create メソッドをクライアントが呼び出した時点で始まります。このメソッドの呼び出しに応答して、コンテナーは以下を実行します。

1. セッション bean インスタンスの新しいメモリー・オブジェクトを作成する。
2. セッション bean の setSessionContext メソッドを呼び出す。 (このメソッドは、セッション・コンテキスト・インターフェースへの参照を、セッション bean インスタンスに渡します。このインターフェースは、インスタンスがコンテナー・サービスを取得したりクライアントが呼び出したメソッドの呼び出し元に関する情報を取得したりするために使用することができます。)
3. EJB クライアントが呼び出した create メソッドに対応する、セッション bean の ejbCreate メソッドを呼び出す。

#### 作動可能状態

セッション bean インスタンスが作成されると、そのライフ・サイクルの作動可能状態に進みます。この状態の間に、EJB クライアントは、リモート・インターフェースに定義されている bean のビジネス・メソッドを呼び出すことができます。この状態でのコンテナーのアクションは、メソッドをトランザクションで呼び出すか否かに応じて異なります。

- *トランザクション・メソッド呼び出し* − クライアントがトランザクション・ビジネス・メソッドを呼び出した場合は、セッション bean インスタンスがトランザクションに関連付けられます。 bean インスタンスは、いったんトランザクションに関連付けられると、そのトランザクションが完了するまで関連付けられたままになります。 (また、EJB クライアントが同じ bean インスタンスに対して別のメソッドを呼び出そうとした場合に、そのメソッドによってコンテナーが bean インスタンスを別のトランザクションに関連付けたり、トランザクションに関連付けない場合には、エラーが発生します。)

  それからコンテナーは、以下のメソッドを呼び出します。

  1. afterBegin メソッド (このメソッドが bean クラスでインプリメントされている場合)。
  2. bean のリモート・インターフェースに定義されており EJB クライアントによって呼び出されるビジネス・メソッドに相当する、bean クラス内のビジネス・メソッド。
  3. bean インスタンスの beforeCompletion メソッド (このメソッドが bean クラスでインプリメントされており、かつコンテナーがトランザクションをコミットしようとする前にコミットが要求された場合)。

  トランザクション・サービスは、それからトランザクションをコミットしようとします。この結果、このトランザクションはコミットされるかロールバックされます。トランザクションが完了すると、コンテナーは bean の afterCompletion メソッドを呼び出して、トランザクションの完了状況 (コミットまたはロールバック) を渡します。

  ロールバックが渡された場合は、状態付きセッション bean はその会話状態をトランザクションが開始される前の bean インスタンスに含まれていた値までロールバックすることができます。状態なしセッション bean は会話状態を保守しないため、ロールバックは関係ありません。
- *非トランザクション・メソッド呼び出し* − クライアントが非トランザクション・ビジネス・メソッドを呼び出した場合は、コンテナーは単に bean クラス内の対応するメソッドを呼び出すだけです。

### プール状態

コンテナーは、メモリー内に保存されている Enterprise Bean インスタンスを管理するための複雑なアルゴリズムを持っています。コンテナーは、状態付きセッション bean がメモリーに不要になったと判断すると、bean インスタンスの ejbPassivate メソッドを呼び出して、bean インスタンスを予約プールに移動させます。状態付きセッション bean インスタンスは、トランザクションに関連付けられている場合には非活動化させることができません。

クライアントが状態付きセッション bean の非活動化されたインスタンスに対してメソッドを呼び出した場合は、コンテナーは、そのインスタンスの状態を復元して bean インスタンスの ejbActivate メソッドを呼び出すことによって、インスタンスを活動化します。このメソッドが戻されると、その bean インスタンスの状態は再び作動可能状態になります。

特定のタイプの状態なしセッション bean インスタンスは、すべてそのタイプの他のインスタンスと同じであるため、状態なしセッション bean インスタンスが非活動化されたり活動化されたりすることはありません。これらのインスタンスは、除去されるまで常に作動可能状態で存在します。

#### 除去状態

セッション bean のライフ・サイクルは、bean のホーム・インターフェースおよびリモート・インターフェースで定義されている remove メソッドを EJB クライアントまたはコンテナーが呼び出した時点で終了します。このメソッドの呼び出しに応答して、コンテナーは bean インスタンスの ejbRemove メソッドを呼び出します。

bean インスタンスがトランザクションに関連付けられている間にユーザーが除去しようとすると、javax.ejb.RemoveException が throw されます。 bean インスタンスが除去された後に、そのインスタンスに対してなんらかのメソッドを呼び出そうとすると、java.rmi.NoSuchObjectException が throw されます。

EJB オブジェクトの存続時間が終了した後でも、コンテナーはインスタンスに対して暗黙的に remove メソッドを呼び出すことができます。セッション EJB オブジェクトの存続時間は、デプロイメント・ディスクリプターの *timeout* 属性で設定します。

remove メソッドの詳細については、181ページの『bean の EJB オブジェクトの除去』を参照してください。

### エンティティー bean のライフ・サイクル

本セクションでは、エンティティー bean インスタンスのライフ・サイクルについて説明します。 CMP を持つエンティティー bean と BMP を持つエンティティー bean の違いについても説明します。

#### 作成状態

エンティティー bean インスタンスのライフ・サイクルは、コンテナーがそのインスタンスを作成した時点で始まります。新しいエンティティー bean インスタンスを作成したら、コンテナーはそのインスタンスの setEntityContext メソッドを呼び出します。 (このメソッドは、エンティティー・コンテキスト・

インターフェースへの参照を、bean インスタンスに渡します。このインターフェースは、インスタンスがコンテナー・サービスを取得したり、クライアントが呼び出したメソッドの呼び出し元に関する情報を取得したりするために使用することができます。)

**プール状態**

エンティティー bean インスタンスは、作成されると、指定のエンティティー bean クラスの使用可能なインスタンスにあるプールに入れられます。インスタンスは、このプールにある限り、特定の EJB オブジェクトに関連付けられません。このプールにある同じ Enterprise Bean クラスのインスタンスは、すべて同一です。インスタンスがこのようなプール状態にある間は、コンテナーはこれを使用して bean の finder メソッドを呼び出すことができます。

**作動可能状態**

特定のエンティティー bean インスタンスをクライアントが処理する必要がある場合は、コンテナーによってこのインスタンスがプールから選出されて、クライアントが初期化した EJB オブジェクトと関連付けられます。作動可能状態の使用できるインスタンスがない場合は、エンティティー bean インスタンスがプール状態から作動可能状態に変わります。

エンティティー bean インスタンスは、以下の 2 つのイベントによってプール状態から作動可能状態に変更されます。

- クライアントが、bean のホーム・インターフェースで定義されている create メソッドを呼び出して、エンティティー bean クラスの新しい固有のエンティティー (およびデータ・ソース内の新しいレコード) を作成した場合。このメソッドが呼び出されると、コンテナーが bean インスタンスの ejbCreate メソッドおよび ejbPostCreate メソッドを呼び出し、EJB オブジェクトが bean インスタンスに関連付けられます。
- クライアントが finder メソッドを呼び出して、 (データ・ソース内の既存のレコードに関連する) エンティティー bean クラスの既存のインスタンスを操作した場合。この場合は、コンテナーが bean インスタンスの ejbActivate メソッドを呼び出して、bean インスタンスを既存の EJB オブジェクトと関連付けます。

エンティティー bean インスタンスが作動可能状態である場合、コンテナーはインスタンスの ejbLoad メソッドおよび ejbStore メソッドを呼び出して、インスタンス内のデータをデータ・ソース内の対応するデータと同期させることができます。さらに、インスタンスがこの状態にある場合には、クライアントが bean インスタンスのビジネス・メソッドを呼び出すことができます。エン

ティティー bean インスタンスのビジネス・メソッドを適切なトランザクション (または非トランザクション) 方式で処理するために必要な対話は、すべてコンテナーによって操作されます。

コンテナーは、作動可能状態のエンティティー bean インスタンスが不要になったと判断すると、そのインスタンスをプール状態に変更します。以下のイベントのいずれかによって、このようなプール状態への変更が行われます。

- コンテナーが ejbPassivate メソッドを呼び出した場合。
- EJB クライアントが、EJB オブジェクトまたは EJB ホーム・オブジェクトに対して remove メソッドを呼び出した場合。これらのメソッドのどちらかを呼び出すと、使用しているエンティティーがデータ・ソースから永久的に除去されます。

### 除去状態

エンティティー bean インスタンスのライフ・サイクルは、コンテナーがプール状態のエンティティー bean インスタンスに対して unsetEntityContext メソッドを呼び出した時点で終了します。エンティティー bean インスタンスの除去と、データがデータ・ソースに保管されている使用中のエンティティーの除去は異なります。前者は単に初期化されていないオブジェクトを除去することを示し、後者はデータ・ソースからデータを除去することを示します。

remove メソッドの詳細については、181ページの『bean の EJB オブジェクトの除去』を参照してください。

# 第3章 EJB サーバー (AE) 環境での Enterprise Bean の開発用および配置用ツール

EJB サーバー (AE) 環境での Enterprise Bean の開発および配置には、以下の 2 つの基本的アプローチがあります。

- IBM VisualAge™ for Java エンタープライズ版などの利用可能な統合環境開発 (IDE) の 1 つを使用することができます。 IDE ツールは、Enterprise Bean のコードの重要な部分を自動的に生成し、Enterprise Bean のパッケージ化およびテストのための統合ツールを含みます。VisualAge for Java は、EJB サーバー (AE) 環境の推奨開発ツールです。 VisualAge for Java の使用に関する詳細については、『VisualAge for Java の使用』を参照してください。
- Java ソフトウェア開発キット (SDK) および Application Server アドバンスド版で利用可能なツールを使用することができます。詳細については、 36ページの『EJB サーバー (AE) ツールでの Enterprise Bean の開発および配置』を参照してください。

**注:** EJB サーバー (AE) 環境用の Enterprise Bean の配置および使用は、Microsoft Windows NT® オペレーティング・システム、 IBM AIX® オペレーティング・システム、または Sun Microsystems Solaris オペレーティング・システムで行わなければなりません。

EJB サーバー (CB) 環境での Enterprise Bean の開発に関する詳細については、43ページの『第4章 EJB サーバー (CB) 環境での Enterprise Bean の開発用および配置用ツール』を参照してください。

## VisualAge for Java の使用

VisualAge for Java で Enterprise Bean を開発する前に、EJB 開発環境をセットアップしなければなりません。このセットアップ作業は、一度しか行う必要はありません。このセットアップ手順によって、VisualAge for Java は Enterprise Bean の開発に必要なすべてのクラスおよびインターフェースをインポートします。

Enterprise Bean を生成したら、以下の一般的な手順に従って開発を完了します。

1. Enterprise Bean クラスをインプリメントする。

© Copyright IBM Corp. 1999, 2001

2. bean クラス内の対応するメソッドを適切なインターフェースにプロモートすることによって、bean のホームおよびリモート・インターフェースに必要な抽象メソッドを作成する。
3. エンティティー bean に対して、以下を行う。
   a. 適切なメニュー項目を使用することによって、ホーム・インターフェースに追加の finder メソッドを作成する。
   b. ファインダー・ヘルパー・インターフェースを作成する (必要な場合)。
4. EJB モジュールおよび対応するデプロイメント・ディスクリプターを作成する。
5. bean の配置コードを生成する。

VisualAge for Java は、完全な WebSphere Application Server 実行時環境と、Enterprise Bean をテストするためのテスト・クライアントを生成するメカニズムを含みます。 VisualAge for Java での Enterprise Bean の開発に関する詳細については、VisualAge for Java の資料を参照してください。

## EJB サーバー (AE) ツールでの Enterprise Bean の開発および配置

IDE を*使用せずに* Enterprise Bean を開発することにした場合は、最低限、次のツールが必要です。

- ASCII テキスト・エディター。(Enterprise Bean の開発をサポートしていない Java 開発ツールを使用することもできます。)
- SDK Java コンパイラー (**javac**) および Java アーカイブ・ツール (**jar**)。
- WebSphere アプリケーション・アセンブリー・ツールおよび WebSphere 管理コンソール。

本セクションでは、これらのツールを使用して Enterprise Bean を開発するための手順について説明します。Enterprise Bean の開発には、以下の作業が必要です。

1. EJB サーバー (AE) 環境で Enterprise Bean を開発、配置、および実行するための前提条件ソフトウェアをインストールおよび構成していることを確認する。詳細については、37ページの『EJB サーバー (AE) 用のソフトウェアのインストールおよび構成』を参照してください。
2. EJB サーバー (AE) 環境の異なるコンポーネントに必要な CLASSPATH 環境変数を設定する。詳細については、38ページの『EJB サーバー (AE) 環境での CLASSPATH 環境変数の設定』を参照してください。

3. Enterprise Bean のコンポーネントを作成およびコンパイルする。詳細については、38ページの『Enterprise Bean のコンポーネントの作成』を参照してください。
4. (*CMP を持つエンティティー bean のみ*) 特殊化された finder メソッド (findByPrimaryKey メソッド以外のもの) を含む CMP を持つ各エンティティー bean 用のファインダー・ヘルパー・インターフェースを作成する。詳細については、39ページの『EJB サーバー (AE) でのファインダー・ロジックの作成』を参照してください。
5. アプリケーション・アセンブリー・ツールを使用して EJB モジュールおよび対応するデプロイメント・ディスクリプターを作成する。詳細については、 39ページの『EJB モジュールの作成』を参照してください。
6. (*エンティティー bean のみ*) データベース・スキーマを作成して、エンティティー bean の永続データを格納できるようにする。詳細については、41ページの『エンティティー bean が使用するデータベースの作成』を参照してください。
7. アプリケーション・アセンブリー・ツールまたは WebSphere 管理コンソールを使用して EJB モジュールを配置する。詳細については、WebSphere InfoCenter および WebSphere 管理コンソールで使用可能なオンライン・ヘルプを参照してください。
8. EJB モジュールを EJB サーバー (AE) にインストールし、 WebSphere 管理コンソールを使用してサーバーを開始する。

## EJB サーバー (AE) 用のソフトウェアのインストールおよび構成

EJB サーバー (AE) で Enterprise Bean および EJB クライアントを開発する前に、以下の前提条件ソフトウェアをインストールおよび構成していなければなりません。

- WebSphere Application Server アドバンスド版
- コンテナー管理のパーシスタンス (CMP) を持つエンティティー bean が使用するための、以下の 1 つ以上のデータベース。
  – DB2
  – Oracle
  – Sybase
  – Informix
  – Microsoft SQL Server
  – InstantDB
- Java ソフトウェア開発キット (SDK)

この環境をセットアップするための適切な製品のバージョン番号および手順の詳細については、 WebSphere InfoCenter を参照してください。

## EJB サーバー (AE) 環境での CLASSPATH 環境変数の設定

Enterprise Bean を開発する場合には、 SDK に含まれる classes.zip ファイルに加えて、次の WebSphere JAR ファイルを CLASSPATH 環境変数に付加しなければなりません。

- ejs.jar
- ujc.jar
- *otherDeployedBean*.jar (Enterprise Bean が別の Enterprise Bean を使用する場合)。これは、この Enterprise Bean が使用している Enterprise Bean が収められている配置済みの JAR ファイルです。

EJB クライアントを開発および実行する場合には、次の WebSphere JAR ファイルを CLASSPATH 環境変数に付加しなければなりません。

- ejs.jar
- ujc.jar
- servlet.jar (サーブレットである EJB クライアントが必要とする)
- *otherDeployedBean*.jar。これは、この EJB クライアントが使用している Enterprise Bean が収められている配置済みの JAR ファイルです。

## Enterprise Bean のコンポーネントの作成

Enterprise Bean の開発をサポートしていない ASCII テキスト・エディターまたは Java 開発ツールを使用する場合は、作成しようとしている Enterprise Bean を構成する各コンポーネントを作成しなければなりません。これらのコンポーネントは、117ページの『第5章 Enterprise Bean の開発』で説明している要件に適合していなければなりません。

セッション bean を手作業で開発するには、bean クラス、bean のホーム・インターフェース、および bean のリモート・インターフェースを作成しなければなりません。エンティティー bean を手作業で開発するには、bean クラス、bean の 1 次キー・クラス、bean のホーム・インターフェース、bean のリモート・インターフェース、および (必要な場合は) bean の finderHelper インターフェースを作成しなければなりません。

これらのコンポーネントを適切にコーディングしたら、Java コンパイラーを使用して、対応する Java クラス・ファイルを作成します。たとえば、サンプル

Account bean のコンポーネントは特定のディレクトリーに格納されるので、次のコマンドを実行して bean コンポーネントをコンパイルすることができます。

```
C:¥MYBEANS¥COM¥IBM¥EJS¥DOC¥ACCOUNT> javac *.java
```

このコマンドは、Account bean が使用するパッケージがすべて CLASSPATH 環境変数に含まれることを前提としています。

## EJB サーバー (AE) でのファインダー・ロジックの作成

EJB サーバー (AE) では、CMP を持つエンティティー bean のホーム・インターフェースに含まれる finder メソッド (findByPrimaryKey メソッドを除く) ごとに以下のファインダー・ロジックが必要です。

- ロジックは、*Name* BeanFinderHelper という名前のパブリック・インターフェースで定義しなければなりません。ここで、*Name* は Enterprise Bean の名前です (AccountBeanFinderHelper など)。
- ロジックは、*findMethodName*WhereClause という名前の String 変数に入っていなければなりません。ここで、*findMethodName* は finder メソッドの名前です。 String 定数は、任意の数の疑問符 (?) を含んでいても、疑問符を含まなくても構いません。この疑問符は、そのメソッドが呼び出されたときの finder メソッドの引き数の値で左から右に置換されます。

**注:** ロジックを *findMethodName*QueryString という String 定数にカプセル化することは禁止されました。

132ページの図24 に示す findLargeAccounts メソッドを定義する場合は、 図7 に示す AccountBeanFinderHelper インターフェースも作成しなければなりません。

```
...
public interface AccountBeanFinderHelper{
     String findLargeAccountsWhereClause = "balance > ?";
}
```

*図 7. コード例: EJB サーバー (AE) 用の AccountBeanFinderHelper インターフェース*

## EJB モジュールの作成

WebSphere Application Server アプリケーション・アセンブリー・ツールを使用して、 EJB モジュールを作成できます。 EJB モジュールには、1 つ以上の

Enterprise Bean を入れることができます。このツールは、ユーザーが指定した情報に基づいて、モジュールに必要なデプロイメント・ディスクリプターを自動的に作成します。

**注:** アプリケーション・アセンブリー・ツールを使用する場合は、前もって WebSphere Common Configuration Model (WCCM) MetaObject Facility (MOF) の JAR ファイルを CLASSPATH 環境変数に追加しておかなければなりません。

### アプリケーション・アセンブリー・ツールの使用

EJB モジュールおよび対応するデプロイメント・ディスクリプターを作成するには、アプリケーション・アセンブリー・ツールの「Create an EJB JAR (EJB JAR の作成)」ウィザードを使用します。このウィザードでは、モジュールに入れる各 Enterprise Bean について次の情報を入力します。

- Enterprise Bean クラス、ホーム・インターフェース・クラス、およびリモート・インターフェース・クラス。
- bean のタイプ (エンティティーまたはセッション)、および関連する属性 (パーシスタンス管理タイプ、エンティティー bean の 1 次キー・クラスなど)。
- Enterprise Bean に関連付ける環境変数。
- 別の Enterprise Bean のホーム・インターフェースへの参照およびリソース・ファクトリーへの参照。
- Enterprise Bean のセキュリティー役割への参照。

また、このウィザードでは、モジュールそのものに関する次のアプリケーション・アセンブリー情報も入力します。

- クライアント・プログラムがモジュール内の bean にアクセスする際に必要なクラス・ファイルの場所や、モジュールに関連付けるアイコンなど、EJB モジュールの一般プロパティー。
- モジュールに収める配置可能な Enterprise Bean。
- モジュール内のリソースにアクセスする際に使用するセキュリティー役割。
- Enterprise Bean のメソッドのトランザクション属性。

bean 情報およびモジュール情報の両方を使用して、デプロイメント・ディスクリプターを作成します。アプリケーション・アセンブリー・ツールの使い方の詳細については、 WebSphere InfoCenter およびオンライン・ヘルプを参照してください。

## エンティティー bean が使用するデータベースの作成

コンテナー管理のパーシスタンス *(CMP)* を持つエンティティー bean の場合は、サポートされているデータベースのいずれかに bean の永続データを格納しなければなりません。アプリケーション・アセンブリー・ツールは、 CMP エンティティー bean のデータベース・テーブルを作成するための SQL コードを自動的に生成します。ツールは、データベース・スキーマとテーブルに ejb.*beanName*beantbl という名前を付けます。ここで、*beanName* は Enterprise Bean の名前です (たとえば、ejb.accountbeantbl)。 CMP エンティティー bean に複雑なデータベース・マッピングが必要な場合は、 VisualAge for Java を使用してデータベース・テーブルのコードを生成することをお勧めします。実行時に、WebSphere 管理コンソールは、生成された SQL コードを実行してデータベース・テーブルを作成するかどうかを尋ねるプロンプトを表示します。

*bean 管理のパーシスタンス (BMP)* を持つエンティティー bean の場合は、データベース・ツールを使用してデータベースおよびデータベース・テーブルを作成することも、既存のデータベースおよびデータベース・テーブルを使用することもできます。 BMP を持つエンティティー bean はデータベースとの対話を処理するので、任意のデータベースまたはデータベース・テーブル名を使用することができます。

データベースおよびデータベース・テーブルの作成に関する詳細については、該当するデータベースの資料、および WebSphere 管理コンソールのオンライン・ヘルプを参照してください。

# 第4章 EJB サーバー (CB) 環境での Enterprise Bean の開発用および配置用ツール

以下では、EJB サーバー (CB) 環境で Enterprise Bean を開発および配置するための基本的なアプローチについて説明します。

- Java ソフトウェア開発キット (SDK) および WebSphere Application Server エンタープライズ版で利用可能なツールを使用することができます。詳細については、44ページの『EJB サーバー (CB) ツールによる Enterprise Bean の開発および配置』を参照してください。
- IBM VisualAge for Java などの利用可能ないずれかの統合開発環境 (IDE) を使用することができます。 IDE ツールは、Enterprise Bean のコードの重要な部分を自動的に生成し、Enterprise Bean のパッケージ化およびテストのための統合ツールを含みます。詳細については、35ページの『VisualAge for Java の使用』を参照してください。
- **PAOToEJB** ツールを使用することによって、既存の CICS または情報管理システム (IMS) アプリケーションから Enterprise Bean を作成することができます。アプリケーションは、このツールを使用する前に手続き型アダプター・オブジェクト (PAO) にマップしなければなりません。詳細については、108ページの『既存の CICS または IMS アプリケーションからの Enterprise Bean の作成』を参照してください。
- **mqaaejb** ツールを使用することによって、IBM MQSeries と通信する Enterprise Bean を作成することができます。詳細については、109ページの『MQSeries と通信する Enterprise Bean の作成』を参照してください。

EJB サーバー (CB) 環境で Enterprise Bean を開発する前に、112ページの『EJB サーバー (CB) 環境での制約事項』に記載されている開発上の制約事項を参照してください。

**注:** EJB サーバー (CB) 環境用の Enterprise Bean の配置および使用は、Microsoft Windows NT または Windows 2000 オペレーティング・システム、 IBM AIX オペレーティング・システム、または Sun Solaris オペレーティング・システムで行わなければなりません。

EJB サーバー (AE) 環境での Enterprise Bean の開発および配置について、詳しくは、35ページの『第3章 EJB サーバー (AE) 環境での Enterprise Bean の開発用および配置用ツール』を参照してください。

© Copyright IBM Corp. 1999, 2001

## EJB サーバー (CB) ツールによる Enterprise Bean の開発および配置

EJB サーバー (CB) 用の Enterprise Bean を開発および配置するには、以下のツールが必要です。

- ASCII テキスト・エディター。(Enterprise Bean の開発をサポートしていない Java 開発ツールを使用することもできます。)
- SDK Java コンパイラー (**javac**) および Java アーカイブ・ツール (**jar**)。
- WebSphere Application Server エンタープライズ版では、以下のツールを使用することができます。
  - **jetace**。1 つまたは複数の Enterprise Bean について EJB JAR ファイルを作成または更新することができます。これには、Enterprise Bean のデプロイメント・ディスクリプターの作成も含みます。この記述子は、Enterprise Bean を適切に管理する方法を EJB サーバーに指定します。

    **jetace** では、 EJB 仕様のバージョン 1.0 と互換性のある JAR ファイルしか作成できません。バージョン 1.1 と互換性のある JAR ファイルを使用する必要がある場合は、 40ページの『アプリケーション・アセンブリー・ツールの使用』を参照してください。
  - Object Builder。Enterprise Bean を配置するための推奨ツールです。このツールの使用については、本書では説明しません。Object Builder を使用して Enterprise Bean を配置する方法の詳細については、Component Broker の「*アプリケーション開発ツール・ガイド*」を参照してください。
  - **cbejb**。Object Builder とともに機能して、EJB サーバー (CB) が Enterprise Bean の管理に必要とするファイルを作成およびコンパイルします。 **cbejb** ツールは、EJB JAR ファイルの内部を参照し、EJB ホーム、EJB オブジェクト・クラス、およびデプロイメント・ディスクリプターを検討します。 **cbejb** ツールは、必要な配置ライブラリー・ファイルを作成するために Object Builder が使用するモデルを生成します。この処理の出力は、一連のサーバー側およびクライアント側の JAR ファイルおよびライブラリー・ファイルです。
  - **CBDeployJar**。Enterprise Bean の配置を自動化します。 **CBDeployJar** ツールでは、 EJB 仕様のバージョン 1.0 またはバージョン 1.1 のいずれかと互換性のある JAR ファイルを配置することができます。 **cbejb** ツールを実行してファイルを配置し、 CMP を持つ Enterprise Bean のデータベース・テーブル・マッピングを生成し、配置したファイルをコンパイルし、EJB サーバーを構成および開始します。また、JNDI ネーム・スペースにバージョン 1.1 と互換性がある Enterprise Bean の参照を登録します。

– **CBDeployEar**。このツールでは、Java™ 2 Enterprise Edition (J2EE™) Enterprise Archive (EAR) ファイルに格納されている JAR ファイルから Enterprise Bean を配置することができます。 **CBDeployEar** ツールは、EAR ファイルから JAR ファイルを抜き出して、その JAR ファイル上で **CBDeployJar** ツールを実行します。

– **ejbbind**。 Enterprise Bean の JNDI (Java Naming and Directory Interface) ホーム名 (デプロイメント・ディスクリプターにある) を EJB サーバー (CB) のファクトリーにバインドします。このツールを、 AIX、Windows NT、Windows 2000、および Solaris プラットフォームで稼働しているサーバーで使用してはなりません。

– **appbind**。Enterprise Bean の配置機能により、アプリケーション固有の命名コンテキストを作成し、それを選択したファクトリー・ファインダーと関連付けられるようにします。その結果、このファクトリー・ファインダーにより EJB ホームの lookup オペレーションが解決されます。このツールは、AIX、Windows NT、Windows 2000 および Solaris プラットフォームでのみ使用可能で、これらのプラットフォームのいずれかにインストールされているサーバーにのみ適用されます。

この節では、EJB サーバー (CB) ツールを使用して Enterprise Bean を開発および配置するために行わなければならない手順について説明します。以下の作業が必要です。

1. EJB サーバー (CB) で Enterprise Bean を開発および配置するための前提条件ソフトウェアがあることを確認する。詳細については、46ページの『EJB サーバー (CB) の前提条件ソフトウェア』を参照してください。
2. EJB サーバー (CB) 環境の異なるコンポーネントに必要な CLASSPATH 環境変数を設定する。詳細については、47ページの『EJB サーバー (CB) 環境での CLASSPATH 環境変数の設定』を参照してください。
3. Enterprise Bean のコンポーネントを作成およびコンパイルする。詳細については、48ページの『Enterprise Bean のコンポーネントの作成』を参照してください。
4. 特殊化された finder メソッド (findByPrimaryKey メソッド以外のもの) を含む CMP を持つ各エンティティー bean 用のファインダー・ヘルパー・クラスを作成する。詳細については、48ページの『EJB サーバー (CB) でのファインダー・ロジックの作成』を参照してください。
5. **jetace** ツールを使用して EJB JAR ファイルを作成し、Enterprise Bean を入れる。詳細については、54ページの『Enterprise Bean 用の EJB JAR ファイルの作成』を参照してください。
6. 次のいずれか 1 つを行って、Enterprise Bean を配置する。

- JAR ファイルから Enterprise Bean を自動的に配置するには、**CBDeployJar** ツールを使用します。詳細については、70ページの『CBDeployJar ツールを使った Enterprise Bean の配置』を参照してください。
- J2EE EAR ファイルから Enterprise Bean を自動的に配置するには、**CBDeployEar** ツールを使用します。詳細については、72ページの『CBDeployEar ツールを使った Enterprise Bean の配置』を参照してください。
- JAR ファイルから Enterprise Bean を手作業で配置するには、次の手順に従います。

  a. **cbejb** コマンドを使用して Enterprise Bean を配置する。詳細については、74ページの『cbejb ツールを使った Enterprise Bean の配置』を参照してください。

  b. Object Builder を使用して、Enterprise Bean が使用するデータ・オブジェクト (DO) のインプリメンテーションを構築する。詳細については、82ページの『CMP エンティティー bean 配置中のデータ・オブジェクトの構築』を参照してください。

  c. 配置した Enterprise Bean をインストールし、その EJB サーバー (CB) を構成する。詳細については、96ページの『Enterprise Bean のインストールおよびその EJB サーバー (CB) の構成』を参照してください。

  d. EJB サーバー (CB) を開始する。詳細については、Component Broker「*システム管理ガイド*」を参照してください。

  e. **ejbbind** ツールを使用して、Enterprise Bean の JNDI 名を JNDI ネーム・スペースにバインドする (このステップは、AIX、Windows NT、Windows 2000 または Solaris プラットフォームでは必要ありません)。詳細については、97ページの『JNDI ネーム・スペースへの Enterprise Bean の JNDI 名のバインド』を参照してください。

  手作業での配置の詳細については、73ページの『手作業による Enterprise Bean の配置』を参照してください。

## EJB サーバー (CB) の前提条件ソフトウェア

**注:** *PAO のみ* とマークされた項目は、いずれも、**PAOToEJB** ツールを使用していて CICS または IMS 関連のサポートを必要とする場合にのみ必要なものです。

EJB サーバー (CB) 環境で提供されるツールは、ユーザー自身で構成しなければなりません。ただし、ツールを構成する前に、Enterprise Application Server に含まれる以下の前提条件ソフトウェア製品がインストールおよび構成されていることを確認しなければなりません。

- CB サーバー
- CB ツール (Object Builder、VisualAge Component 開発ツールキット、サンプル、サーバー SDK、および (*PAO のみ*) CICS と IMS アプリケーション・アダプター SDK)
- (*PAO のみ*) CICS/IMS アプリケーション・ランタイム
- (*PAO のみ*) CICS/IMS アプリケーション・クライアント

## EJB サーバー (CB) 環境での CLASSPATH 環境変数の設定

以下にリストした作業のいずれかを行う場合は、 Java 開発キットに含まれる classes.zip ファイルが CLASSPATH 環境変数に組み込まれていることを確認してください。さらに、以下のファイルが CLASSPATH 環境変数によって識別され、関連作業が実行できるようになっていることを確認しなければなりません。

- Enterprise Bean または EJB クライアントの開発: 追加のファイルはありません。
- EJB JAR ファイルの配置:
  - somojor.zip
  - 配置する EJB JAR ファイルおよび依存するすべての JAR または ZIP ファイル
- EJB サーバー (CB) を実行し、*beanName* という名前の Enterprise Bean を管理する。以下の JAR ファイルは、CLASSPATH 環境変数に自動的に追加されます。
  - *beanName* S.jar
  - *beanName* S.jar の作成に使用する EJB JAR ファイル、およびそれが依存するすべての JAR または ZIP ファイル
- Java EJB クライアントを実行して、*beanName* という名前の Enterprise Bean を使用する。
  - *beanName* C.jar
  - somojor.zip

- *beanName* という名前の別の Enterprise Bean をアクセスする *clientBeanName* という名前の Enterprise Bean をクライアントとして含む EJB サーバー (CB) を実行する。以下の JAR ファイルは、CLASSPATH 環境変数に自動的に追加されます。
  - *clientBeanName* S.jar
  - *clientBeanName* S.jar の作成に使用する EJB JAR ファイル、およびそれが依存するすべての JAR または ZIP ファイル
  - *beanName* C.jar

## Enterprise Bean のコンポーネントの作成

Enterprise Bean の開発をサポートしていない ASCII テキスト・エディターまたは Java 開発ツールを使用する場合は、作成しようとしている Enterprise Bean を構成する各コンポーネントを作成しなければなりません。これらのコンポーネントは、EJB 仕様の要件に適合していなければなりません。これらのコンポーネントについては、117ページの『第5章 Enterprise Bean の開発』で説明しています。

セッション bean を手作業で開発するには、bean クラス、bean のホーム・インターフェース、および bean のリモート・インターフェースを作成しなければなりません。エンティティー bean を手作業で開発するには、bean クラス、bean の 1 次キー・クラス、bean のホーム・インターフェース、および bean のリモート・インターフェースを作成しなければなりません。

これらのコンポーネントを適切にコーディングしたら、Java コンパイラーを使用して、対応する Java クラス・ファイルを作成します。たとえば、Account bean の例のコンポーネントは特定のディレクトリーに格納されるので、以下のコマンドを出して bean コンポーネントをコンパイルすることができます。

```
C:¥MYBEANS¥COM¥IBM¥EJS¥DOC¥ACCOUNT> javac *.java
```

このコマンドは、Account bean が使用するパッケージがすべて CLASSPATH 環境変数に含まれることを前提としています。

## EJB サーバー (CB) でのファインダー・ロジックの作成

EJB サーバー (CB) では、ファインダー・ロジックはファインダー・ヘルパー・クラスに含まれます。 Enterprise Bean の配置機能は、ファインダー・ヘルパー・クラスをインプリメントしてから Enterprise Bean を配置し、**cbejb** ツールの -finderhelper オプションでクラスの名前を指定しなければなりません。

ホーム・インターフェースの特殊化された finder メソッド (findByPrimaryKey を除く) ごとに、ファインダー・ヘルパー・クラスには、同じ名前とパラメーター・タイプを持つ対応するメソッドがなければなりません。 EJB クライアントが特殊化された finder メソッドを呼び出す場合は、Enterprise Bean のホーム・インターフェースをインプリメントする生成済みの CB ホームが、対応するファインダー・ヘルパー・メソッドを呼び出して、EJB クライアントへの戻り値を決定します。

また、ファインダー・ヘルパー・クラスには、型 com.ibm.IManagedClient.IHome の単一の引き数をとるコンストラクターもなければなりません。 CB ホームがファインダー・ヘルパー・クラスをインスタンス化するときに、CB ホームは、自分自身への参照をファインダー・ヘルパーのコンストラクターに渡します。これによって、ファインダー・ヘルパーは、ファインダー・ヘルパー・メソッドのインプリメンテーション内で CB ホームに対するメソッドを呼び出すことができます。これは、ファインダー・ヘルパーが、コンストラクターに渡される IHome オブジェクトを限定して CB ホームに対する照会サービスを呼び出すことができるので、CB ホームが IQueryableIterableHome である場合に特に有用です。

エンティティー bean のコンテナー管理のフィールドの名前は、ビジネス・オブジェクト (BO) インターフェース、CB キー・クラス、および CB コピー・ヘルパー・クラスでは同じ名前のインターフェース定義言語 (IDL) 属性に下線 (_) を付加してマップされます。これらの名前は、DO インターフェースの IDL 属性に正確にマップされます。たとえば、AccountBean クラスにおいて、*accountId* 変数は、BO インターフェース、CB キー・クラス、および CB コピー・ヘルパー・クラスでは *accountId_* にマップされますが、DO インターフェースでは *accountId* にマップされます。

この名前変更は、Component Broker の照会サービスを使用してインプリメントされたファインダー・ヘルパー・クラスに関係するものですが、エンティティー bean のリモート・インターフェースは、BO インターフェースを介して公開しなければならない *accountId* という名前 (場合によっては異なる型) のプロパティーを持つこともできるため、この名前変更が必要です。このような場合には、BO 属性 *accountId* に対する照会はオブジェクト・スペースで行われ、BO 属性 *accountId_* に対する照会は下位のデータ・ソースに対して直接行われます (こちらがより効率的です)。

ホーム・インターフェースの特殊化された finder メソッドが単一のエンティティー bean を戻す場合は、ファインダー・ヘルパー・クラスの対応するメソッドは java.lang.Object 型を戻さなければなりません。呼び出し時に、ファインダー・ヘルパー・メソッドは、EJB オブジェクト、 CB キー・オブジェクト、エ

ンティティー bean の 1 次キー・オブジェクト、または CB 管理下オブジェクト・フレームワーク (MOFW) オブジェクトを戻すことができます。ファインダー・ヘルパー・メソッドが CB オブジェクトまたは 1 次キー・オブジェクトを戻す場合は、CB ホームは、対応する EJB オブジェクトを決定して EJB クライアントに戻します。

ホーム・インターフェースの特殊化された finder メソッドが java.util.Enumeration 型を戻す場合は、対応するファインダー・ヘルパー・メソッドも java.util.Enumeration を戻さなければなりません。呼び出し時に、ファインダー・ヘルパー・メソッドは、 EJB オブジェクトの Enumeration、CB キー・オブジェクト、CB MOFW オブジェクト、 Enterprise Bean の 1 次キー・オブジェクト、またはこれらの 4 つのうちの 1 つまたは複数の異種混合を戻すことができます。これを受けて、CB ホームは、EJB クライアントに戻された対応する EJB オブジェクトを含む直列化された Enumeration オブジェクトを構成します。

ホーム・インターフェースの特殊化された finder メソッドが java.util.Collection 型を戻す場合は、対応するファインダー・ヘルパー・メソッドも java.util.Collection を戻さなければなりません。呼び出し時に、ファインダー・ヘルパー・メソッドは、 EJB オブジェクトの Collection、CB キー・オブジェクト、CB MOFW オブジェクト、 Enterprise Bean の 1 次キー・オブジェクト、またはこれらの 4 つのうちの 1 つまたは複数の異種混合を戻すことができます。これを受けて、CB ホームは、対応する EJB オブジェクトを含む直列化された Collection オブジェクトを構成し、 EJB クライアントに戻します。

オプションの基本クラス (com.ibm.ejb.cb.runtime.FinderHelperBase) は、ファインダー・ヘルパー・クラスの開発を支援するために、EJB サーバー (CB) 環境で提供されています。このクラスは Component Broker の照会サービスをカプセル化するので、配置機能が CB 固有のコードを作成する必要は一切ありません。 FinderHelperBase 基本クラスには、 表2 にリストされるメソッドが入っています。これらのメソッドは一般に、オブジェクト指向構造化照会言語 (OOSQL) 述部をパラメーターとして取り、その照会の条件を満たすオブジェクト、またはオブジェクトの Enumeration あるいは Collection を戻します。

表 2. *FinderHelperBase クラスのメソッド*

| メソッド | パラメーター | 戻り型 | 注 |
|---|---|---|---|
| evaluate | OOSQL where 文節 | Enumeration | 必要なオブジェクトを即時にインスタンス化 |
| extendedEvaluate | 完全な OOSQL ステートメント | Enumeration | 必要なオブジェクトを即時にインスタンス化 |

表 2. *FinderHelperBase クラスのメソッド (続き)*

| | | | |
|---|---|---|---|
| lazyEvaluate | OOSQL where 文節 | Enumeration | 必要なオブジェクトを必要に応じてインスタンス化 |
| extendedLazyEvaluate | 完全な OOSQL ステートメント | Enumeration | 必要なオブジェクトを必要に応じてインスタンス化 |
| singleEvaluate | OOSQL where 文節 | オブジェクト | 見つからない場合に ObjectNotFoundException を throw |
| extendedSingleEvaluate | 完全な OOSQL ステートメント | オブジェクト | 見つからない場合に ObjectNotFoundException を throw |
| evaluateCollection | OOSQL where 文節 | Collection | 必要なオブジェクトを即時にインスタンス化 |
| extendedEvaluateCollection | 完全な OOSQL ステートメント | Collection | 必要なオブジェクトを即時にインスタンス化 |
| lazyEvaluateCollection | OOSQL where 文節 | Collection | 必要なオブジェクトを必要に応じてインスタンス化 |
| extendedLazyEvaluateCollection | 完全な OOSQL ステートメント | Collection | 必要なオブジェクトを必要に応じてインスタンス化 |

エラーが発生した場合には、これらのメソッドはすべて javax.ejb.FinderException を throw します。ファインダー・ヘルパー・クラスは、この例外を捕そくする必要はありません。代わりに、クラスが EJB クライアントにこの例外を渡します。

また、**com.ibm.ejb.cb.emit.cb.FinderHelperGenerator** という名前のクラス (developEJB.jar ファイルに含まれます) も、配置機能によるファインダー・ヘルパー・クラスの開発を支援するために提供されています。このユーティリティーは、エンティティー bean のホーム・インターフェースの名前を受け取り、com.ibm.ejb.cb.runtime.FinderHelperBase を拡張するクラスおよびホーム・インターフェースの特殊化された finder メソッドごとのスケルトン・メソッドを含むクラスを含む Java ソース・ファイルを生成します。さらに、各ファインダー・ヘルパー・メソッドには、 50ページの表2 にリストされている適切な FinderHelperBase メソッドを呼び出すためのコールが入っています。

**ejbfhgen** という **FinderHelperGenerator** ユーティリティーを使用することで、配置機能では、ファインダー・ヘルパー・クラスを簡単にインプリメントすることができます。バッチ・ファイルを使用して、ユーティリティーを実行

することができます。たとえば、AccountHome の例のインターフェース用にファインダー・ヘルパー・クラスを生成するには、以下のコマンドを入力します。

```
# ejbfhgen com.ibm.ejs.doc.account.AccountHome
```

このコマンドによって、図8 に示すファインダー・ヘルパー・クラスが生成されます。

```
...
public class AccountFinderHelper extends FinderHelperBase {
     ...
     AccountFinderHelper(IManagedClient.IHome iHome) {
          ...
     }
     public Enumeration findLargeAccounts(float amount) {
          return evaluate("replace with appropriate code");
     }
 }
```

*図 8. コード例: EJB サーバー (CB) 用に生成された AccountFinderHelper クラス*

ヘルパー・クラスを配置済み Enterprise Bean で使用できるようにするには、evaluate の呼び出しのパラメーターを少し編集します。たとえば、AccountFinderHelper クラスの場合は、 図9 に示すように、 `"replace with appropriate code"` ストリングが `"balance_>" + amount` に置き換わります。生成されるファインダー・ヘルパー・クラスは、**cbejb** ツールの -queryable オプションを使用することによって照会可能なホームを持つように配置された Enterprise Bean とともにのみ使用することができます。

```
...
public class AccountFinderHelper extends FinderHelperBase {
     ...
     AccountFinderHelper(IManagedClient.IHome iHome) {
          ...
     }
     public Enumeration findLargeAccounts(float amount) {
          return evaluate("balance_>" + amount);
     }
 }
```

*図 9. コード例: EJB サーバー (CB) 用に完成された AccountFinderHelper クラス*

## VisualAge for Java スタイルの finder-helper インターフェースの使用

VisualAge for Java の finder-helper インターフェース (39ページの『EJB サーバー (AE) でのファインダー・ロジックの作成』を参照) は、 表3 に示すように、 FinderHelperBase メソッドにマップするサフィックスをサポートしています。

*表 3. FinderHelperBase メソッドのサフィックス*

| サフィックス | 戻り型 | メソッド |
|---|---|---|
| CBWhereClause | Enumeration | evaluate |
| CBQueryString | Enumeration | extendedEvaluate |
| CBWhereClause | Collection | evaluateCollection |
| CBQueryString | Collection | extendedEvaluateCollection |
| CBWhereClause | オブジェクト | singleEvaluate |
| CBQueryString | オブジェクト | extendedSingleEvaluate |
| CBLazyWhereClause | Enumeration | lazyEvaluate |
| CBLazyQueryString | Enumeration | extendedLazyEvaluate |
| CBLazyWhereClause | Collection | lazyEvaluateCollection |
| CBLazyQueryString | Collection | extendedLazyEvaluateCollection |

VisualAge for Java は、finder-helper インターフェースで指定されている CBWhereClause、CBQueryString、CBLazyWhereClause、または CBLazyQueryString を使って EJB JAR ファイルを CB にインポートしたときに、 CB finder-helper クラスを自動的に作成します。

また、VisualAge for Java スタイルの finder-helper インターフェースを 2 つ目のパラメーターとして **ejbfhgen** ユーティリティーに渡すことによって、 CB finder-helper インターフェースを手作業で作成することもできます。たとえば、次のコマンドを実行したとします。

```
ejbfhgen com.ibm.ejs.doc.account.AccountHome
     com.ibm.ejs.doc.account.AccountBeanFinderHelper
```

VisualAge for Java スタイルの finder-helper インターフェースを入力として指定してこのコマンドを呼び出した場合には、 `"replace with appropriate code"` ストリングを戻す代わりに OOSQL ステートメントを埋め込んで、コードをコンパイルします。 OOSQL ストリングがすべて収められている VisualAge for Java スタイルの finder-helper インターフェースを渡す場合には、コードを手作業で編集する必要はありません。配置機能では、コンパイルした CB finder-helper クラスを EJB JAR ファイルに追加する必要がありま

す。または、そのクラスは、-serverdep パラメーターと一緒に **cbejb** ツールを使用して、別の JAR ファイルにパッケージすることもできます。

### 逐次列挙型の使用

evaluate メソッドで戻される列挙型は、*一括* 列挙型と呼ばれます。照会に合致する Enterprise Bean 参照はすべてメモリーに読み込まれ、サーバーからクライアントに渡される前に列挙型に保管されます。照会により戻される参照数が多い場合、配置機能は、*逐次* 列挙型を使用することができます。つまり、クライアントが列挙型で nextElement メソッドを呼び出す場合にのみ、さらに Enterprise Bean 参照を増分的に取り出します。

逐次列挙型を使用する場合は、FinderHelper の evaluate メソッド呼び出しを lazyEvaluate メソッド呼び出しに変更してください。トランザクションは、ホームのファインダー・メソッドを呼び出す前に、既に開始されていなければなりません。呼び出し側は、トランザクションが完了した後、列挙型で nextElement メソッドを呼び出してはいけません。

構成時に、システム管理エンド・ユーザー・インターフェースを使用して、逐次列挙型の設定を使用可能にしなければなりません。100ページの『逐次列挙型を使用可能にするシステム管理の構成』を参照してください。

## Enterprise Bean 用の EJB JAR ファイルの作成

bean コンポーネントを構築したあとで、次に、それらのコンポーネントを EJB JAR ファイルにパッケージします。WebSphere Application Server の **jetace** ツールは、1 つまたは複数の Enterprise Bean 用の EJB JAR ファイルを作成し、各 Enterprise Bean のためのデプロイメント・ディスクリプター・ファイルを生成するために使用することができます。結果として得られる EJB JAR ファイルは、各 Enterprise Bean のクラス・ファイルとデプロイメント・ディスクリプター、および EJB に準拠するマニフェスト・ファイルを含みます。

**注: jetace** ツールでは、 EJB 仕様のバージョン 1.0 と互換性がある JAR ファイルしか作成できません。バージョン 1.1 と互換性のある JAR ファイルを作成する必要がある場合は、アプリケーション・アセンブリー・ツールを使用してください。 40ページの『アプリケーション・アセンブリー・ツールの使用』を参照してください。

1 つまたは複数の Enterprise Bean 用に EJB JAR ファイルを作成する前に、次のうちの *1 つ* を行わなければなりません。

- 各 Enterprise Bean のすべてのコンポーネントを単一のディレクトリーに置きます。

- Java アーカイブ・ツール (**jar**) を使用することによって、各 Enterprise Bean のクラス・ファイルおよびインターフェース・ファイルを含む標準の JAR ファイルを作成します。以下のコマンドを Account bean のフル・パッケージ名のルート・ディレクトリーから実行すると、デフォルトのマニフェスト・ファイルを持つファイル AccountIn.jar を作成することができます。

```
C:¥MYBEANS> jar cfv AccountIn.jar com¥ibm¥ejs¥doc¥account¥*.class
```

- WinZip® などのツールを使用することによって、各 Enterprise Bean のクラス・ファイルおよびインターフェース・ファイルを含む標準の ZIP ファイルを作成します。

### jetace ツールの実行

**jetace** ツールを実行するには、コマンド行で **jetace** と入力します。図10 に示すウィンドウが表示されます。

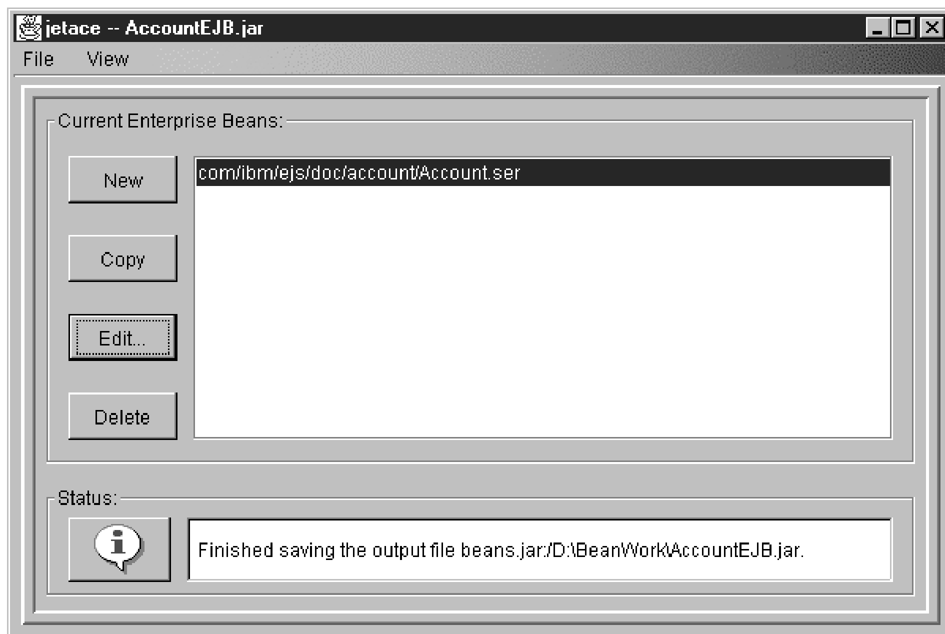



*図 10. jetace ツールの初期ウィンドウ*

**jetace** ツールで EJB JAR ファイルを生成するには、以下を行います。

1. **「File (ファイル)」 -> 「Load (ロード)」**と選択して、JAR ファイル、ZIP ファイル、あるいは 1 つまたは複数の Enterprise Bean を含むディレクトリーを選択する。ファイルまたはディレクトリーを取得するには、**「Browse (ブラウズ)」**ボタンを使用する。

**注:** 現行ディレクトリーを入力元として指定するには、ブラウザー・ウィンドウの**「File Name (ファイル名)」**フィールドに = (等号) を入力して**「Open (オープン)」**をクリックします。

新規 EJB JAR ファイルを作成する場合には、**「New (新規)」**をクリックする。これにより、デプロイメント・ディスクリプターのデフォルト名 (たとえば、UNAMED_BEAN_1.ser) が**「Current Enterprise Beans (現行 Enterprise Bean)」**リスト・ボックスに表示されます。(**jetace** GUI の残りのタブ付きページに現れるこの名前は、各タブ付きページの最上部にある**「Deployed Name (配置された名前)」**フィールドを編集することによって、編集できます。このフィールドについては 58ページの『Enterprise Bean コンポーネントおよび JNDI ホーム名の指定』で説明します。)

既存の EJB JAR ファイルを編集する場合には、55ページの図10 に示すように、EJB JAR ファイル内にある各 Enterprise Bean のデプロイメント・ディスクリプターの名前が**「Current Enterprise Beans (現行 Enterprise Bean)」**リスト・ボックスに表示されます。

- リストされた Enterprise Bean を結果の EJB JAR ファイルに組み込みたくない場合には、その Enterprise Bean のデプロイメント・ディスクリプターを強調表示させて、**「Delete (削除)」**をクリックする。このアクションにより、リスト・ボックスからそのデプロイメント・ディスクリプターが除去されます。
- Enterprise Bean の複製を作成したい場合には、そのデプロイメント・ディスクリプターを強調表示させて、**「Copy (コピー)」**をクリックするこのアクションにより、新しいデフォルト・デプロイメント・ディスクリプターがリスト・ボックスに追加されます。コピーされる bean のデプロイメント・ディスクリプターに似た Enterprise Bean のデプロイメント・ディスクリプターを作成したい場合には、コピーが便利です。その後で、新しいデプロイメント・ディスクリプターを編集する必要があります。

2. 新しいデプロイメント・ディスクリプターを作成したり既存のデプロイメント・ディスクリプターを編集したりするためには、そのデプロイメント・ディスクリプターを強調表示させて、**「Edit (編集)」**ボタンをクリックする。このアクションにより、**「Basic (基本)」**ページが表示される。このページで、デプロイメント・ディスクリプター・ファイル、Enterprise Bean クラス、ホーム・インターフェース、およびリモート・インターフェースの名前を設定または確認し、 Enterprise Bean の JNDI 名を指定する。詳細については、58ページの『Enterprise Bean コンポーネントおよび JNDI ホーム名の指定』を参照してください。

3. エンティティー bean またはセッション bean に対するエンタープライズ bean のデプロイメント・ディスクリプターをおのおの**「Entity (エンティティー)」**ページまたは、**「Session (セッション)」**ページで設定する。エンティティー bean に対するデプロイメント・ディスクリプター属性の設定に関する詳細については、59ページの『エンティティー bean 固有の属性の設定』を参照してください。セッション bean に対するデプロイメント・ディスクリプター属性の設定に関する詳細については、61ページの『セッション bean 固有の属性の設定』を参照してください。
4. **「Transactions (トランザクション)」**ページで、Enterprise Bean のデプロイメント・ディスクリプターに対するトランザクション属性を設定する。詳細については、63ページの『トランザクション属性の設定』を参照してください。
5. **「Security (セキュリティー)」**ページで、Enterprise Bean のデプロイメント・ディスクリプターに対するセキュリティー属性を設定する。詳細については、65ページの『セキュリティー属性の設定』を参照してください。
6. **「Environment (環境)」**ページで、Enterprise Bean に関連付ける環境変数を設定する。詳細については、67ページの『Enterprise Bean 用の環境変数の設定』を参照してください。
7. **「Dependencies (依存関係)」**ページで、Enterprise Bean に関連付けるクラスの依存関係を設定する。詳細については、68ページの『Enterprise Bean のクラス依存関係の設定』を参照してください。
8. 各 Enterprise Bean のための適切なデプロイメント・ディスクリプター属性を設定してから、**「File (ファイル)」 -> 「Save As (別名保管)」**とクリックして、EJB JAR ファイルを作成する。(必要な場合は、JAR ファイルの代わりに ZIP ファイルを作成することもできます。)

**jetace** ツールを使用すると、XML 版の Enterprise Bean のデプロイメント・ディスクリプターの読み取りや生成も行うことができます。 XML ファイルを読み取るには、**「File (ファイル)」 -> 「Read XML (XML の読み取り)」**と選択します。既存の Enterprise Bean から XML ファイルを生成するには、(出力 EJB JAR ファイルを保管した後に)**「File (ファイル)」 -> 「Read XML (XML の書き込み)」**と選択します。

**jetace** ツールは、コマンド行から実行して EJB JAR ファイルを作成することもできます。このコマンドの構文を以下に示します。ここで、*xmlFile* は Enterprise Bean のデプロイメント・ディスクリプターを含む XML ファイルの名前です。

```
% jetace -f xmlFile
```

このコマンドに必要な XML ファイルの構文に関する詳細については、311ページの『付録C. Enterprise Bean での XML の使用 (CB のみ)』を参照してください。

**Enterprise Bean コンポーネントおよび JNDI ホーム名の指定**

**「Basic (基本)」**ページは、デプロイメント・ディスクリプター・ファイルの絶対パス名の設定、Enterprise Bean クラス、ホーム・インターフェース、ならびにリモート・インターフェースの Java パッケージ名の設定、および Enterprise Bean の JNDI ホーム名の設定に使用します。このページを図11 に示します。このページにアクセスするには、**「Basic (基本)」**タブを選択します。

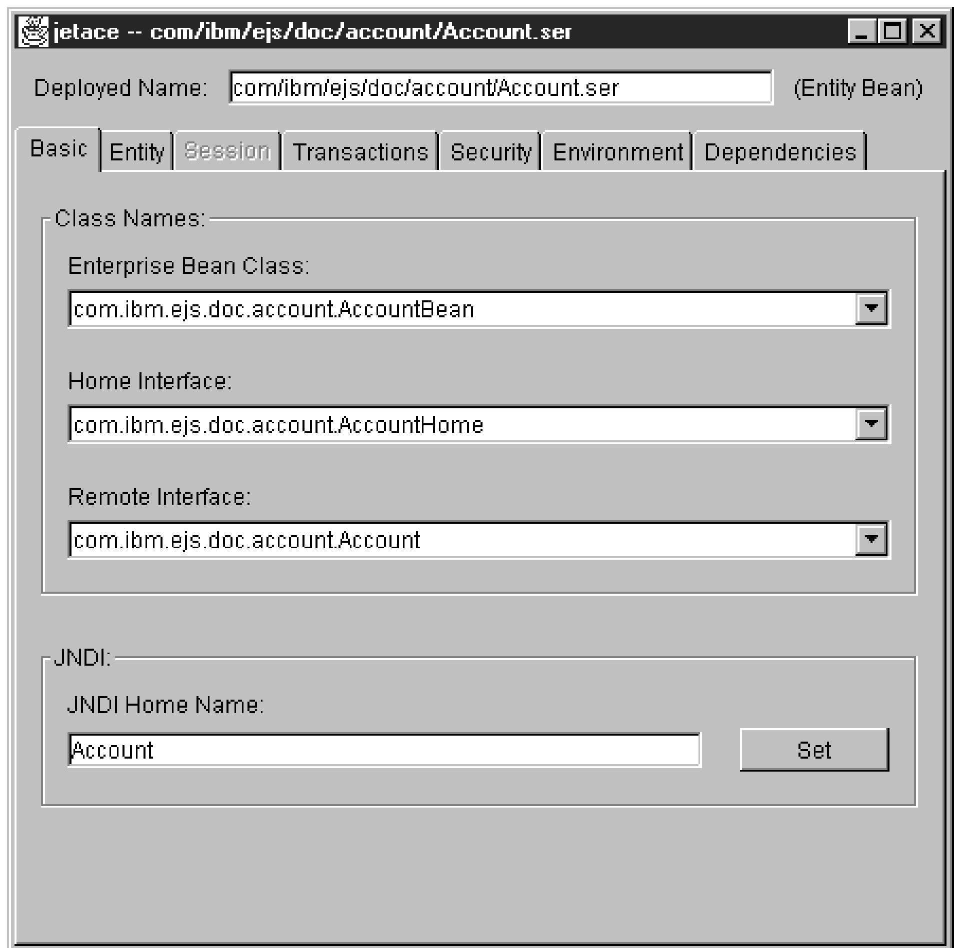



*図 11. jetace ツールの「Basic (基本)」ページ*

「Basic (基本)」ページで、以下のフィールドに対して値を選択するか確定しなければなりません。

- **「Deployed Name (配置された名前)」** − 作成するデプロイメント・ディスクリプター・ファイルのパス名。このディレクトリー名は、 Enterprise Bean クラスのフル・パッケージ名に一致させるようにしてください。Account bean のフルの名前は `com/ibm/ejs/doc/account/Account.ser` です。
- **「Enterprise Bean Class (Enterprise Bean クラス)」** − bean クラスのフル・パッケージ名を指定します。 Account bean のフルの名前は、`com.ibm.ejs.doc.account.AccountBean` です。
- **「Home Interface (ホーム・インターフェース)」** − bean のホーム・インターフェースのフル・パッケージ名を指定します。 Account bean のフルの名前は、`com.ibm.ejs.doc.account.AccountHome` です。
- **「Remote Interface (リモート・インターフェース)」** − bean のリモート・インターフェースのフル・パッケージ名を指定します。Account bean のフルの名前は、`com.ibm.ejs.doc.account.Account` です。
- **「JNDI Home Name (JNDI ホーム名)」** − bean のホーム・インターフェースの JNDI ホーム名を指定します。これは、Enterprise Bean のホーム・インターフェースの登録に使用する名前なので、EJB クライアントがホーム・インターフェースを検索するときにはこの名前を指定しなければなりません。 Account bean の場合は、JNDI ホーム名は `Account` です。

### エンティティー bean 固有の属性の設定

特にエンティティー bean に関連するデプロイメント・ディスクリプター属性を設定するには、**jetace** ツールの**「Entity (エンティティー)」**タブをクリックして、60ページの図12 に示す**「Entity (エンティティー)」**ページを表示させます。このタブは、**jetace** の初期ウィンドウで強調表示させた Enterprise Bean がセッション bean である場合は使用不可になっています。

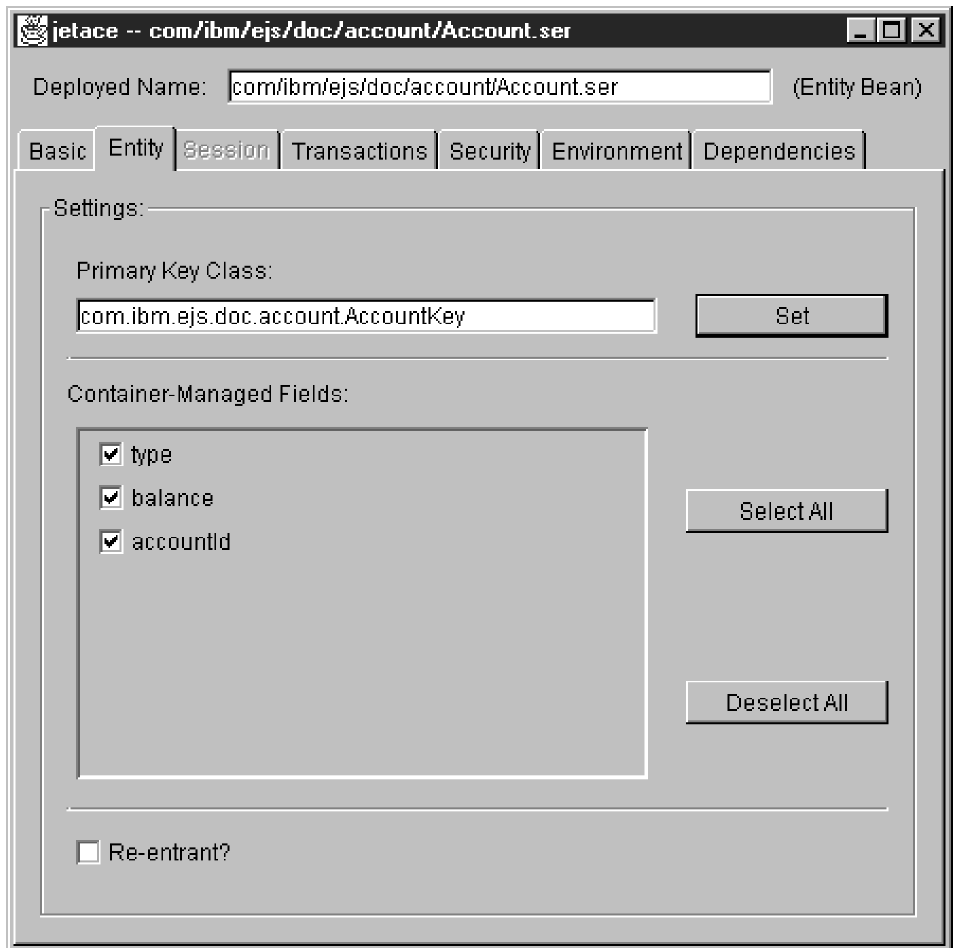



図 12. jetace ツールの「Entity (エンティティー)」ページ

**「Entity (エンティティー)」**ページで、以下のフィールドに対して値を選択するか確定しなければなりません。

- **「Primary Key Class (1 次キー・クラス)」** － bean の 1 次キー・クラスのフル・パッケージ名を指定します。Account bean の例の場合は、フルの名前は com.ibm.ejs.doc.account.AccountKey です。
- **「Container-Managed Fields (コンテナー管理フィールド)」** － コンテナーがパーシスタンス管理を処理する必要がある bean クラスの変数に対するチェック・ボックスにチェック・マークを付けます。これは、CMP を持つエンティティー bean の場合にのみ必要であり、BMP を持つエンティティー

bean の場合に設定してはなりません。 Account bean の場合は、type、balance、および accountId の各変数がコンテナー管理なので、各ボックスにチェック・マークを付けます。

- 「**Re-entrant (再入可能)**」 ー bean が再入可能な場合はこのチェック・ボックスにチェック・マークを付けます。デフォルトでは、エンティティー bean は再入可能ではありません。再入不能なエンティティー bean のインスタンスがトランザクション・コンテキストでクライアント要求を実行していて、同じトランザクション・コンテキストを使用して別の要求を受け取ると、 EJB コンテナーは、2 番目の要求に対して java.rmi.RemoteException 例外を throw します。コンテナーは、別の bean のリーガル・ループバック呼び出しと別のクライアントまたはクライアント・スレッドのイリーガルな同時呼び出しを区別できないため、クライアントが、再入可能 bean の同時呼び出しが行われないように注意していなければなりません。 Account bean の例は、再入可能ではありません。

### セッション bean 固有の属性の設定

特にセッション bean に関連するデプロイメント・ディスクリプター属性を設定するには、**jetace** ツールの「**Session (セッション)**」タブをクリックして、62ページの図13 に示す「**Session (セッション)**」ページを表示させます。このタブは、**jetace** の初期ウィンドウで強調表示させた Enterprise Bean がエンティティー bean である場合は使用不可になっています。

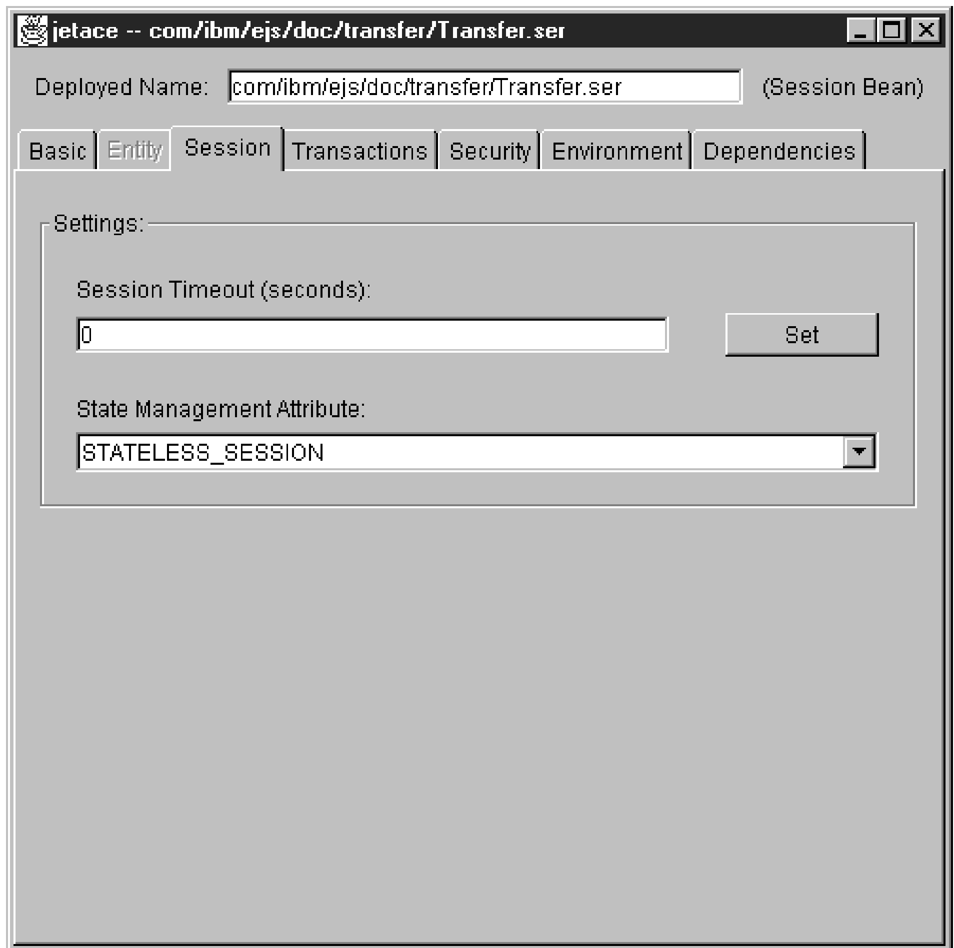



*図 13. jetace ツールの「Session (セッション)」ページ*

**「Session (セッション)」**ページで、以下のフィールドに対して値を選択するか確定しなければなりません。

- **Session Timeout (seconds) (セッション・タイムアウト (秒))** －この bean に対するアイドル・タイムアウト値を秒単位で指定します。 `0` (ゼロ) は、最大許容タイムアウト期間が経過したあとでアイドル bean インスタンスがタイムアウトになることを示します。 Transfer bean の場合は、この値は `0` のままにして、デフォルトのタイムアウトを使用することを指定します。

  **注:** EJB サーバー (CB) 環境では、この属性を使用しません。

- **「State Management Attribute (状態管理属性)」** － bean が状態付きであるか状態なしであるかを指定します。 Transfer bean の例は STATELESS_SESSION です。詳細については、 21ページの『状態なしセッション bean と状態付きセッション bean』を参照してください。

### トランザクション属性の設定

**「Transactions (トランザクション)」**ページは、 Enterprise Bean のすべてのメソッドおよび Enterprise Bean の個々のメソッドについて、トランザクション属性およびトランザクション分離レベル属性を設定するために使用します。個々のメソッドに属性を設定する場合は、その属性によって、Enterprise Bean 全体に設定されたデフォルトの属性値が上書きされます。

**注:** EJB サーバー (CB) では、トランザクション属性は、bean 全体にのみ設定することができます。bean 内の個々のメソッドについてトランザクション属性を設定することはできません。

**「Transactions (トランザクション)」**ページにアクセスするには、**jetace** ツールで**「Transactions (トランザクション)」**タブをクリックします。 64ページの図14 に、このページの例を示します。

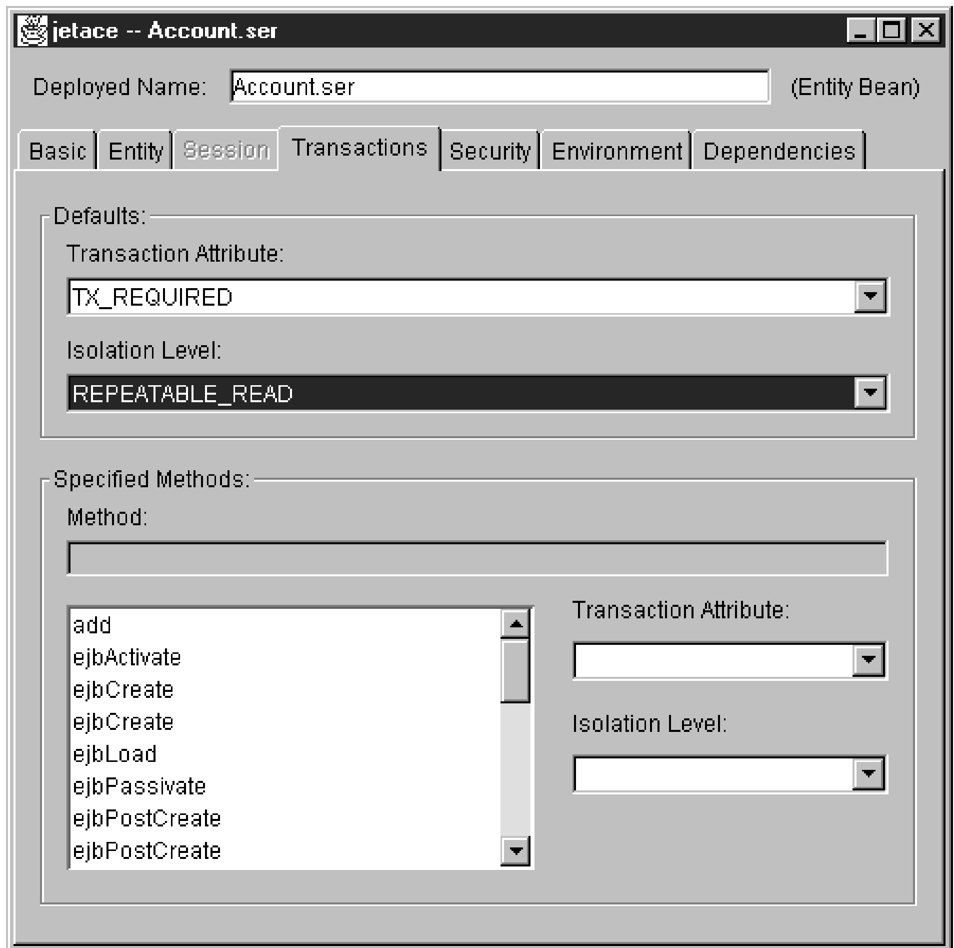



*図 14. jetace ツールの「Transactions (トランザクション)」ページ*

**「Transactions (トランザクション)」**ページでは、**「Defaults (デフォルト)」**グループ・ボックスにある以下のフィールドで値を選択するか確定しなければなりません。

- **「Transaction Attribute (トランザクション属性)」** － トランザクション属性の値を設定します。この属性の値については、159ページの『第6章 Enterprise Bean でのトランザクションおよびセキュリティーの使用可能化』で説明します。Account bean の場合は、呼び出し時にこの bean のメソッドを既存のトランザクションと関連付けなければならないため、値 `TX_MANDATORY` を使用します。結果として、Transfer bean は、新しいトランザクションを開始する値か既存のトランザクションに渡す値を使用しなければなりません。

- **「Isolation Level (分離レベル)」** － トランザクション分離レベル属性の値を設定します。この属性の値については、159ページの『第6章 Enterprise Bean でのトランザクションおよびセキュリティーの使用可能化』で説明します。Account bean の場合は、値 `REPEATABLE_READ` が使用されます。

必要であれば、該当するメソッドを強調表示させ、**「Specified Methods (指定されたメソッド)」**グループ・ボックスの一方または両方の属性を設定することによって、個々のメソッドに対してこれらの属性を設定することもできます。

### セキュリティー属性の設定

**「Security (セキュリティー)」**ページは、 Enterprise Bean のすべてのメソッドおよび Enterprise Bean の個々のメソッドについて、セキュリティー属性を設定するために使用します。個々のメソッドに属性を設定する場合は、その属性によって、Enterprise Bean 全体に設定されたデフォルトの属性値が上書きされます。

**「Security (セキュリティー)」**ページにアクセスするには、**jetace** ツールで**「Security (セキュリティー)」**タブをクリックします。66ページの図15 に、このページの例を示します。

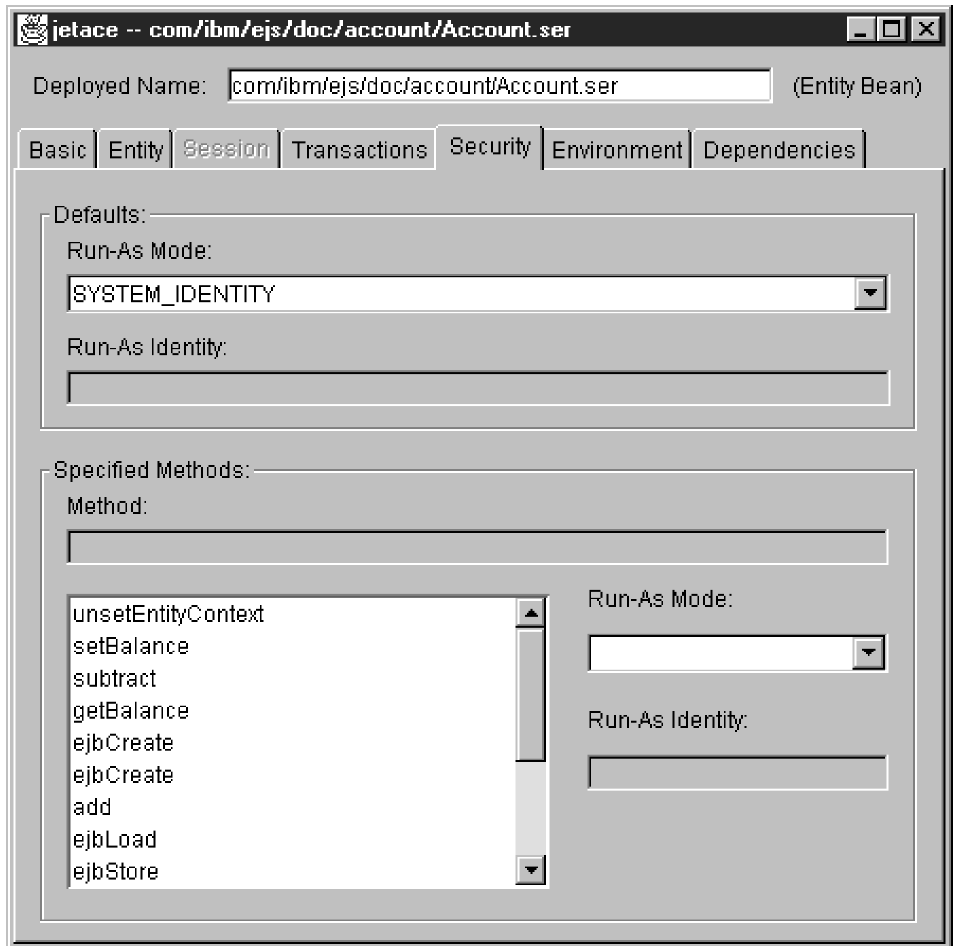



*図 15. jetace ツールの「Security (セキュリティー)」ページ*

**「Security (セキュリティー)」**ページでは、**「Defaults (デフォルト)」**グループ・ボックスにある**「Run-As Mode (実行モード)」**フィールドで値を選択するか確定しなければなりません。このフィールドは、167ページの『デプロイメント・ディスクリプターのセキュリティー属性の設定』で説明する値のいずれかに設定しなければなりません。*実行識別* 属性は、EJB サーバー (CB 環境) では使用されないため、**jetace** ツールで対応するフィールドに値を設定することはできません。

必要であれば、該当するメソッドを強調表示させ、**「Specified Methods (指定されたメソッド)」**グループ・ボックスの属性を設定することによって、個々のメソッドに対して*実行モード* 属性を設定することもできます。

### Enterprise Bean 用の環境変数の設定

**「Environment (環境)」**ページは、環境変数 (および対応する値) を Enterprise Bean と関連付けるために使用します。**「Environment (環境)」**ページにアクセスするには、**jetace** ツールで**「Environment (環境)」**タブをクリックします。図16 に、このページの例を示します。

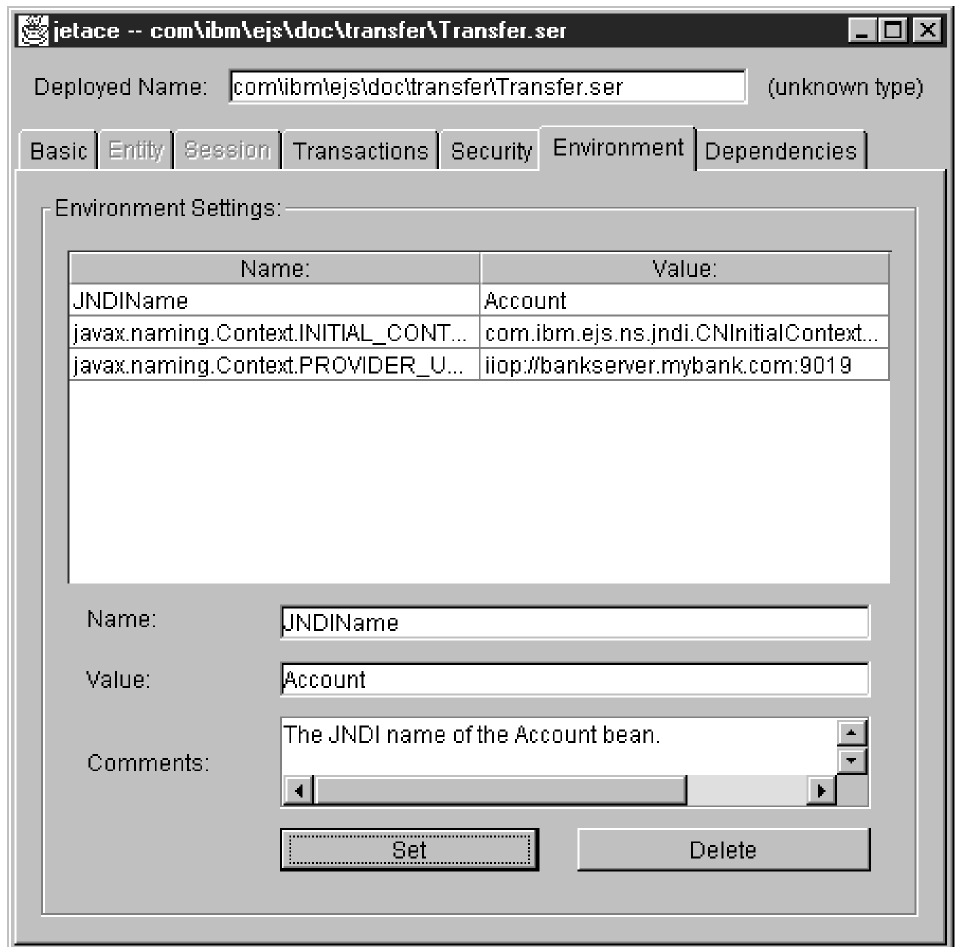



*図 16. jetace ツールの「Environment (環境)」ページ*

環境変数に値を設定するには、**「Name (名前)」**フィールドに環境変数名を指定し、**「Value (値)」**フィールドに環境変数値を指定します。必要な場合は、**「Comment (注釈)」**フィールドを使用して、環境変数の詳細を記述することができます。値を設定するには、**「Set (設定)」**ボタンを使用します。環境変

数を削除するには、**「Environment Settings (環境設定)」**ウィンドウで変数を強調表示させ、**「Delete (削除)」**ボタンをクリックします。

Transfer bean の例では、以下の環境変数が必要です。

- JNDIName - Account bean の JNDI 名。この bean は Transfer bean によってアクセスされます。詳細については、58ページの図11 を参照してください。
- javax.naming.Context.INITIAL_CONTEXT_FACTORY ー Transfer bean が Account bean の JNDI 名を検索するために使用する初期コンテキスト・ファクトリーの名前。
- javax.naming.Context.PROVIDER_URL ー Transfer bean が Account bean の JNDI 名を検索するために使用するネーム解決サービスのロケーション。

Transfer bean がこれらの環境変数を使用する方法に関する詳細については、144ページの『ejbCreate メソッドのインプリメント』を参照してください。

### Enterprise Bean のクラス依存関係の設定

**「Dependencies (依存関係)」**ページは、Enterprise Bean が依存するクラスを指定するために使用します。**「Dependencies (依存関係)」**ページにアクセスするには、**jetace** ツールで**「Dependencies (依存関係)」**タブをクリックします。69ページの図17 に、このページの例を示します。

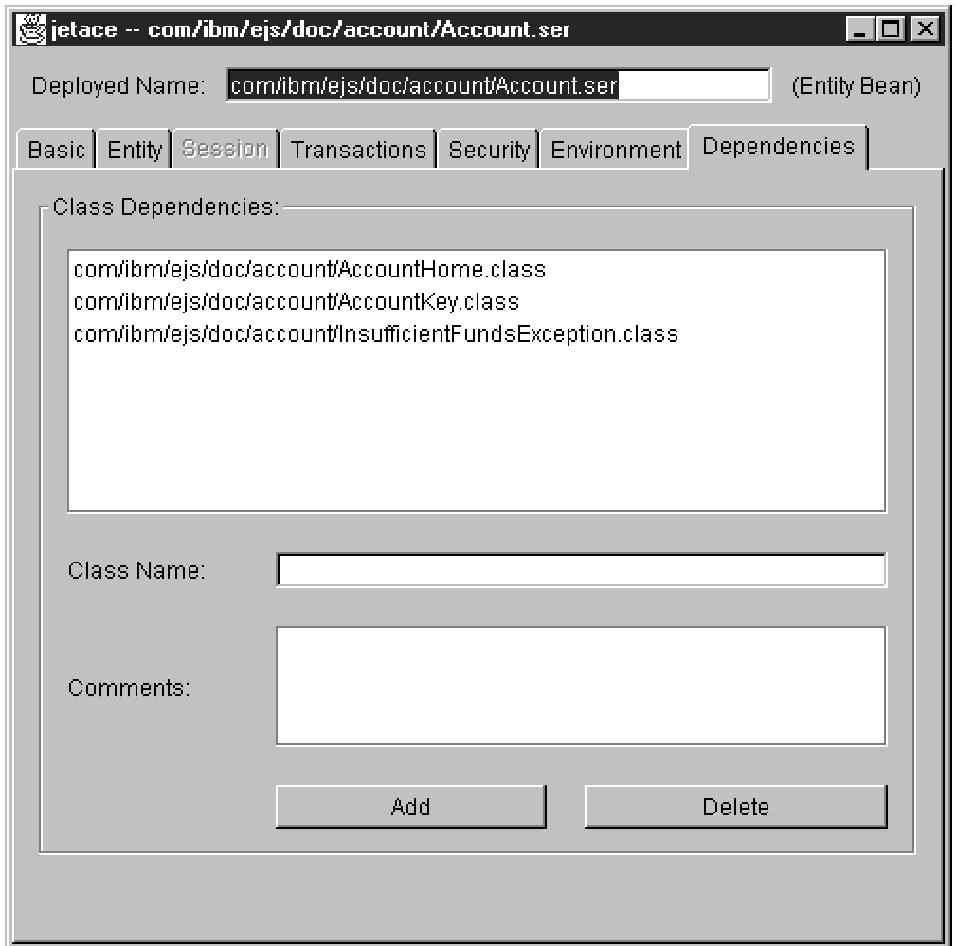



*図 17. jetace ツールの「Dependencies (依存関係)」ページ*

通常、**jetace** ツールは、クラスの依存関係を自動的に検出してここに設定します。 Enterprise Bean が他のクラス依存関係を必要とする場合は、**「Classname (クラス名)」**フィールドに完全修飾 Java クラス名を入力することによって、ここで設定しなければなりません。必要な場合は、**「Comment (注釈)」**フィールドを使用して、依存関係の詳細を記述することができます。値を設定するには、**「Add (追加)」**ボタンを使用します。依存関係を除去するには、**「Class Dependencies (クラス依存関係)」**ウィンドウで強調表示させ、**「Delete (削除)」**ボタンをクリックします。

Account bean の例の場合は、**jetace** ツールは、図17 に示すように依存関係を設定します。

## CBDeployJar ツールを使った Enterprise Bean の配置

**CBDeployJar** ツールは、 Enterprise Bean の配置に関連する作業を自動化します。このツールで実行できることは次のとおりです。

- JAR ファイルから Enterprise Bean を配置する
- Enterprise Bean が JAR ファイルから配置されているかどうかを確認する
- JAR ファイルに関連付けられた Enterprise Bean の配置を解除する

**CBDeployJar** ツールは、 EJB 仕様のバージョン 1.0 およびバージョン 1.1 のどちらも互換性のある JAR ファイル上で実行できます。このツールで配置できる Enterprise Bean のタイプは次のとおりです。

- セッション bean
- BMP を持つエンティティー bean
- トップダウン・マッピングを使用する、または VisualAge for Java のマッピング情報を持つ、 CMP を持つエンティティー bean

このツールでは、meet-in-the-middle マッピングを使用していて、 VisualAge for Java を使って作成されなかった CMP を持つエンティティー bean は配置できません。これらの Enterprise Bean は手作業で配置してください (73ページの『手作業による Enterprise Bean の配置』を参照してください)。

JAR ファイルから Enterprise Bean を配置する場合に、 **CBDeployJar** ツールが実行する作業は次のとおりです。

1. JAR ファイルが EJB 仕様のバージョン 1.1 と互換性がある場合は、バージョン 1.1 デプロイメント・ディスクリプターの XML を構文解析し、バージョン 1.0 スタイルの逐次化したデプロイメント・ディスクリプターを使って新しい JAR ファイルを生成します (これが必要なのは、他の Component Broker EJB ツールはバージョン 1.0 の JAR ファイルしか使用しないためです)。また、すべての EJB 1.1 デプロイメント・ディスクリプター環境変数を JDNI ネーム・スペースに java:comp/env/*environVarName* で登録します。ここで、*environVarName* は環境変数の名前です。
2. ユーザー指定のオプションを使用して、 JAR ファイル上で **cbejb** ツールを実行します。
3. ユーザー指定のオプションを使用して、プラットフォームに対する **make** コマンドを実行します。
4. CMP を持つエンティティー bean のパーシスタンス・フィールドをデータベース・テーブルにマップします。
5. アプリケーション・ファミリーを Component Broker システム管理にロードする一連の **wscmd** コマンドを実行することによって、 Component Broker

EJB サーバーを構成および開始します。新しい管理ゾーン、構成、および EJB サーバーを作成し、配置済みの Enterprise Bean を EJB サーバー上で構成して、 EJB サーバーを開始します。

6. EJB 仕様のバージョン 1.1 に合わせて作成された Enterprise Bean の場合は、これらの bean の参照を JNDI ネーム・スペースの java:comp/env/ejb 下の適切な場所に登録します (これは、Enterprise Bean 間でネーミングの衝突を避けるために必要です)。

**CBDeployJar** コマンドの構文は次のとおりです。

```
CBDeployJar ejb-jarFile hostname [-cbejb options] [-make options] [-noTables]
     [-prepJarOnly] [-cbejbOnly] [-makeOnly]
CBDeployJar ejb-jarFile hostname  -isDeployed
CBDeployJar ejb-jarFile  hostname  -undeploy
```

ここで、

- *ejb-jarFile* － JAR ファイルの名前 (必須)。これは、最初のパラメーターでなければなりません。
- *hostname* － Enterprise Bean を配置するマシンの完全修飾ホスト名 (必須)。これは、2 番目のパラメーターでなければなりません。
- -cbejb *options* － **CBDeployJar** ツールで配置プロセスの一部として実行される、 **cbejb** コマンドの必要なオプションを指定します。このフラグが設定されていない場合は、コマンドのデフォルト・オプションが使用されます。 **cbejb** コマンド行オプションの完全なリストについては、 74ページの『cbejb ツールを使った Enterprise Bean の配置』を参照してください。

**注:** -cbejb フラグおよび -make フラグに渡すオプションにスペースが入っている場合は、二重引用符 ("") を使用します。

- -make *options* － **CBDeployJar** ツールで配置プロセスの一部として実行される、プラットフォームに対する **make** コマンドの必要なオプションを指定します。このフラグが設定されていない場合は、コマンドのデフォルト・オプションが使用されます。 **make** オプションの完全なリストについては、お手持ちのコンパイラーの資料を参照してください。
- -noTables － **CBDeployJar** ツールが、 CMP を持つエンティティー bean 内のパーシスタンス・フィールドに対するテーブルを作成することを禁止します。 DB2 以外のデータベースでサポートされている CMP を持つエンティティー bean を使用している場合は、このフラグを指定してください (**CBDeployJar** ツールは、DB2 データベース・テーブルしか作成しません)。
- -prepJarOnly － バージョン 1.1 対応の JAR ファイルをバージョン 1.0 形式に変換した後 (ステップ 1)、プロセスを停止します。

- -cbejbOnly ー **cbejb** ツールを実行した後 (ステップ 2)、プロセスを停止します。
- -makeOnly ー **make** コマンドを実行した後 (ステップ 3)、プロセスを停止します。
- -isDeployed ー 特定の JAR ファイルが配置されているかどうかを検証します。このオプションは、*ejb-jarFile* パラメーターおよび *hostname* パラメーターと一緒の場合しか指定できません。
- -undeploy ー **CBDeployJar** ツールを使ってすでに配置済みの JAR ファイルの配置を解除します。このオプションは、*ejb-jarFile* パラメーターおよび *hostname* パラメーターと一緒の場合しか指定できません。 -undeploy オプションは、**cbejb** コマンドおよび **make** コマンドで生成されたファイルをすべて除去し、 CMP を持つエンティティー bean に対するデータ・オブジェクトのインプリメンテーションを削除し、サーバー構成および関連する情報を削除し、 EJB サーバーを停止し、JNDI ネーム・スペースから EJB 1.1 対応の Enterprise Bean の参照を削除します。

次に、**CBDeployJar** コマンドの使用例を示します。

```
CBDeployJar EJBsavingsAccount.jar test.netbank.ibm.com
CBDeployJar EJBcalculator.jar trident.ibm.com -make IVB_COMBINE_SOURCE=0
CBDeployJar EJBportfolio.jar bringup.ibm.com -cbejb "-dbname Investors"
CBDeployJar EJBhello.jar tasmania.ibm.com -noTables
CBDeployJar EJBtest.jar trip.ibm.com -isDeployed
CBDeployJar EJBtest.jar trip.ibm.com -undeploy
```

## CBDeployEar ツールを使った Enterprise Bean の配置

**CBDeployEar** ツールは、 J2EE EAR ファイルにカプセル化された JAR ファイルから Enterprise Bean を自動的に配置します。このツールは、指定された EAR ファイルから JAR ファイルを抜き出して、その JAR ファイル上で **CBDeployJar** ツールを実行して Enterprise Bean を配置します。

**CBDeployEar** コマンドの構文は次のとおりです。

```
CBDeployEar earFile hostname [-cbejb options] [-make options] [-noTables]
[-prepJarOnly] [-cbejbOnly] [-makeOnly] [-bindEJBRefs]
CBDeployEar earFile hostname  -isDeployed
CBDeployEar earFile  hostname  -undeploy
```

ここで、

- *earFile* ー J2EE EAR ファイルの名前 (必須)。これは、最初のパラメーターでなければなりません。
- *hostname* ー Enterprise Bean を配置するマシンの完全修飾ホスト名 (必須)。これは、2 番目のパラメーターでなければなりません。

- -cbejb *options* ー **CBDeployEar** ツールが **CBDeployJar** ツールを呼び出したときに実行される、**cbejb** コマンドのオプションを指定します。詳細については、70ページの『CBDeployJar ツールを使った Enterprise Bean の配置』を参照してください。
- -make *options* ー **CBDeployEar** ツールが **CBDeployJar** ツールを呼び出したときに実行される、**make** コマンドのオプションを指定します。詳細については、70ページの『CBDeployJar ツールを使った Enterprise Bean の配置』を参照してください。
- -noTables ー ツールが、CMP を持つエンティティー bean 内のパーシスタンス・フィールドに対するテーブルを作成することを禁止します。 DB2 以外のデータベースでサポートされている CMP を持つエンティティー bean を使用している場合は、このフラグを指定してください。
- -prepJarOnly ー バージョン 1.1 対応の JAR ファイルをバージョン 1.0 形式に変換した後、プロセスを停止します。
- -cbejbOnly ー **cbejb** ツールを実行した後、プロセスを停止します。
- -makeOnly ー **make** コマンドを実行した後、プロセスを停止します。
- -bindEJBRefs ー EJB 1.1 対応の Enterprise Bean の参照を JNDI ネーム・スペースにバインドします。このオプションは、デフォルトで指定されます。ただし、**cbejb** ステップおよび **make** ステップを飛ばし、 **CBDeployEar** ツールを実行するときに JNDI バインディングを実行した方が便利な場合もあります。たとえば、最初に **CBDeployEar** ツールを実行したときに EJB サーバーが開始されていなかった場合には、ツールを再度実行する時間を節約することができます。
- -isDeployed ー 特定の EAR ファイルから JAR ファイルが配置されているかどうかを検証します。このオプションは、*earFile* パラメーターおよび *hostname* パラメーターと一緒の場合しか指定できません。
- -undeploy ー **CBDeployEar** ツールを使って EAR ファイルからすでに配置済みの JAR ファイルの配置を解除します。このオプションは、*earFile* パラメーターおよび *hostname* パラメーターと一緒の場合しか指定できません。

次に、**CBDeployEar** コマンドの使用例を示します。

```
CBDeployEar EJB11Big3.ear greenland.ibm.com
CBDeployEar EJB11Big3.ear greenland.ibm.com -bindEJBRefs
CBDeployEar EJB11Big3.ear greenland.ibm.com -isDeployed
CBDeployEar EJB11Big3.ear greenland.ibm.com -undeploy
```

## 手作業による Enterprise Bean の配置

ファイルの作成に使用したツールとは無関係に、どのようなタイプの Enterprise Bean タイプが収められた JAR ファイルでも手作業を配置すること

ができます。次のステップは、Enterprise Bean を Component Broker EJB サーバー上に手作業で配置する場合に実行しなければならない作業をまとめたものです。

1. **cbejb** コマンドを使用して Enterprise Bean を配置する。
2. Object Builder を使用して、Enterprise Bean が使用するデータ・オブジェクト (DO) のインプリメンテーションを構築する (このステップは、配置プロセスの一部です)。
3. 配置した Enterprise Bean をインストールし、その EJB サーバー (CB) を構成する。
4. Component Broker「*システム管理ガイド*」で説明されているように、 EJB サーバー (CB) を開始する。
5. **ejbbind** ツールを使用して、Enterprise Bean の JNDI 名を JNDI ネーム・スペースにバインドする (このステップは、AIX、Windows NT、Windows 2000、または Solaris プラットフォームでは必要ありません)。

この節では、ステップ 1、2、3、および 5 の実行方法について説明します。

### cbejb ツールを使った Enterprise Bean の配置

配置中には、配置する JAR ファイルが EJB JAR ファイルから生成されます。**cbejb** ツールを使用して、EJB サーバー (CB) 環境に Enterprise Bean を配置します。配置する JAR ファイルは、EJB サーバーが必要とするクラスを含みます。また、**cbejb** ツールは、EJB サーバー (CB) への Enterprise Bean のインストール中に使用されるデータ定義言語 (DDL) ファイルも生成します。

開発した (**cbejb** を実行した) マシンと異なるマシンで Enterprise Bean を使用する場合は、Component Broker「*システム管理ガイド*」にあるアプリケーションのインストールの指針に従ってください。 Enterprise Bean とともにコピーする必要がある追加のファイル (他の JAR ファイルなど) を Enterprise Bean が使用する場合は、アプリケーション (ファミリーではありません) のプロパティー・ノートブックでそれらのファイルを指定します。

**注: cbejb** ツールでは、 EJB 仕様のバージョン 1.0 と互換性がある JAR ファイルしか配置できません。バージョン 1.1 対応の JAR ファイルを手作業で配置するには、まず -prepJarOnly オプションを指定して **CBDeployJar** ツールを実行してから、 JAR ファイルをバージョン 1.0 形式に変換しなければなりません。詳細については、70ページの『CBDeployJar ツールを使った Enterprise Bean の配置』を参照してください。

**cbejb** ツールの構文は以下のとおりです。

```
cbejb ejb-jarFile [-rsp responseFile][-ob projDir] [-nm] [-ng] [-nc] [-cc]
[-bean beanNames] [-platform [NT | AIX | OS390 | Solaris | HP]]
[-guisg] [-usecurdopo] [-nousraction] [-dllname DLLName beanName]
[-polymorphichome [beanNames]]  [-queryable [beanNames]]
[-dbname DBName [beanName]]
[-cacheddb2v52 | -cacheddb2v61 | -db2v61 |-oracle | -informix |
-jdbcaa [beanNames]]
[-hod | -eci | -appc | -exci | -otma | -ccf [beanNames]]
[-family familyName [beanNames]]
[-finderhelper finderHelperClassName [beanNames]]
[-usewstringindo [beanNames]]  [-workloadmanaged [beanNames]]
[-clientdep deployed-jarFile [beanNames]]
[-serverdep deployed-jarFile [beanNames]]
[-sentinel [JavaPrimitiveObjectType=]sentinelValue [beanNames[+CMFieldNames]]
[-strbehavior [strip | corba] [beanNames[+CMFieldNames]]
```

*ejb-jarFile* パラメーターは必須です。このパラメーターは最初の引き数でなければならず、有効な EJB JAR ファイルを指定しなければなりません。 -ob オプションを使用する場合は、コマンド行の 2 番目に置かなければなりません。その他のオプションは、任意の順序で指定することができます。 *beanNames* 引き数は、1 つまたは複数の完全修飾 Enterprise Bean 名をコロン (:) で区切ったリストです (たとえば com.ibm.ejs.doc.transfer.Transfer:com.ibm.ejs.doc.account.Account)。 Enterprise Bean 名には、bean のリモート・インターフェース名またはデプロイメント・ディスクリプターの名前のいずれかを指定します。特定のオプションで *beanNames* 引き数を指定しない場合は、そのオプションの効果は、オプションが有効な EJB JAR ファイルにあるすべての Enterprise Bean に適用されます。

**注:** *ejb-jarFile* 変数および 2 つの *deployed-jarFile* 変数によって指定された JAR ファイルの相対ファイル名は、互いに異なっていなければなりません。相対ファイル名が同じでパスが異なる JAR ファイル名は無効です。

残りのコマンド・パラメーターはオプションで、任意の順序で指定できます。説明のために、オプションは機能によって以下の 3 つの総称的なカテゴリーにグループ化することができます。

- 配置オプション。コードの生成およびコンパイルを管理します。
- 記憶域オプション。永続記憶域を管理します。
- 実行オプション。実行時環境を管理します。

-rsp オプションは、上記のカテゴリーには属しません。このオプションを使用して、その他のオプションのいくつかまたはすべて、および (*ejb-jarFile* パラメーターを除く) それらの値が入っているファイルを作成することができま

す。それから、このファイルを **cbejb** コマンドに実行依頼することができます。これにより、共通の設定値が保管できるので、コマンドの発行がより簡単になります。

- 配置オプション
  - -ob *projDir* ー 生成ファイルを格納するプロジェクト・ディレクトリーの相対パスまたは絶対パスを指定します。このオプションが指定されない場合は、現行作業ディレクトリーがプロジェクト・ディレクトリーとして使用されます。
  - コンパイル修飾子 ー デフォルトでは、**cbejb** ツールは、EJB JAR ファイルに含まれる Enterprise Bean ごとに以下のことを行います。
    1. XML を生成およびインポートします。
    2. コードの生成 ー EJB JAR ファイルに含まれる Enterprise Bean ごとに DDL ファイル、makefile、およびその他のソース・ファイルを作成します。これらのファイルは、指定されたプロジェクト・ディレクトリーに置かれます。
    3. コンパイルおよびリンク ー 生成された makefile を起動して、アプリケーションをコンパイルします。各アプリケーション・ファイルは、指定されたプロジェクト・ディレクトリーに置かれます。ダイナミック・リンク・ライブラリー (DLL) のリンク中に、シンボルが重複していることを意味する警告が大量に表示されますが、これらの警告は実害がないので無視して構いません。

    以下のコマンド・オプションにより、デフォルトのコンパイルの振る舞いが変更されます。

    - -nm - XML プロセス・ステップを抑制します。
    - -ng - コード生成ステップを抑制します。
    - -nc - コンパイルおよびリンク・ステップを抑制します。
    - -cc - 生成した makefile を起動して、ソース・ファイル以外のファイルを除去することで、既にコンパイルおよびリンク済みのコードを除去します。このオプションは、以下のいずれかのように、オプションを組み合わせて指定する場合には使用しなければなりません。
      - -ng -nc
      - -nm -ng -nc
  - -bean *beanNames* ー 配置する EJB JAR ファイル内の Enterprise Bean を識別します。デフォルトでは、EJB JAR ファイル内のすべての Enterprise

Bean が配置されます。複数の Enterprise Bean を配置するには、bean 名を **:** (コロン) で区切ってください。たとえば、`Account:Transfer` のように指定します。

- -platform ー コードを生成するプラットフォームを指定します。これは、Object Builder ツールにも配置プラットフォームを設定しますが、作成の制約事項を表示したり、生成したり、適用したりすることはありません。これらについては、**「Platform (プラットフォーム)」**メニューでの選択により、手作業で設定しなければなりません。
- -guisg - Object Builder グラフィカル・ユーザー・インターフェース (GUI) を表示するようにツールに指示して、ツールがコマンド行ではなくユーザー・インターフェースからオプションを受け入れられるようにします。
- -usecurdopo - Object Builder インターフェースを立ち上げてマッピングを作成するのではなく、既存のモデルのデータ・オブジェクトと永続オブジェクトの間の現行マッピングを使用するようにツールに指示します。このオプションは、既に十分に使用できるマッピングが存在する bean を再配置する場合に使用します。配置は自動的に行われます。

  最初に CMP エンティティー bean を配置するときは、このオプションを使用 *しない*でください。そうすれば、このツールによりデータ・オブジェクトと永続オブジェクトの間のデフォルト・マッピングが作成され、-guisg オプションを指定している場合は、 Object Builder インターフェースを立ち上げます。
- -nousraction ー データ・オブジェクトと永続オブジェクトの間にマッピングを作成した後は、コマンド行の情報だけを使用するようにツールに指示します。そうしなければ、-guisg オプションも指定している場合に、ツールから次のアクションについてのプロンプトが出されます。
- -polymorphichome ー ポリモアフィック・ホーム・インターフェースを使用する bean を指定します。
- -queryable ー 照会可能 CB ホーム・オブジェクトを生成します。このオプションは、永続データをリレーショナル・データベースに格納する CMP を持つエンティティー bean に対してのみ使用することができます。このオプションは、ファインダー・ヘルパー・クラス (CMP エンティティー bean で finder メソッドをインプリメントするために使用します) が CB 照会サービスを使用する場合に使用しなければなりません。このオプションは、エンティティー bean が CICS または IMS を使用して永続データを格納する場合には*使用*できません。

  デフォルトでは、Enterprise Bean の CB ホームのインターフェース定義言語 (IDL) インターフェースは、IManagedClient::IHome クラスを拡張

し、ホーム・インプリメンテーションは IManagedAdvancedServer::ISpecializedHome クラスを拡張します。照会可能ホームの IDL インターフェースは、IManagedAdvancedClient::IQueryableIterableHome クラスを拡張し、ホーム・インプリメンテーションは IManagedAdvancedServer::ISpecializedQueryableIterableHome クラスを拡張します。

さらに、生成される BO インターフェースには照会可能とマークが付けられます。照会可能ホームについては、EJB クライアント・プログラミング・モデルは変更されません。しかし、共通オブジェクト・リクエスト・ブローカー・アーキテクチャー (CORBA) EJB クライアントは、EJB ホームを IManagedAdvancedClient::IQueryableIterableHome オブジェクトとして扱うことができます。

照会可能ホームに関する詳細については、「*上級プログラミング・ガイド*」を参照してください。

- 記憶域オプション
  - -dbname *DBName* － CMP を持つ bean のデータベース名を指定します。
  - Database choices － コンテナー管理 bean の永続記憶域のデフォルト・データベースは、SQL が組み込まれた DB2 バージョン 5.2 です。このデフォルトは、以下のオプションを使用して上書きすることができます。
    - -cacheddb2v52 － CMP を持つエンティティー bean が永続データの格納にキャッシュ・サービスと共に使用される DB2 バージョン 5.2 を必要とすることを示します。
    - -cacheddb2v61 － CMP を持つエンティティー bean が永続データの格納にキャッシュ・サービスと共に使用される DB2 バージョン 6.1 を必要とすることを示します。
    - -db2v61 － CMP を持つエンティティー bean が永続データの格納に組み込まれた SQL と共に使用される DB2 バージョン 6.1 を必要とすることを示します。
    - -oracle － CMP を持つエンティティー bean が永続データの格納に Oracle を必要とすることを示します。このオプションを指定する場合は、-queryable オプションも使用しなければなりません。
    - -informix － CMP を持つエンティティー bean が永続データの格納に Informix を必要とすることを示します。指定されたトランザクションは、CB サーバーから複数の Informix データベースにアクセスできません。 1 つのトランザクションで 2 つの Informix データベースにア

クセスするには、異なる CB サーバーからそれぞれアクセスしなければなりません。このオプションを指定する場合は、-queryable オプションも使用しなければなりません。

- -jdbcaa ー BMP を持つエンティティー bean が永続データの格納に JDBC を必要とすることを示します。このオプションを使用して、bean インプリメンテーションでトランザクション・サービスに接続できるようにして、 bean が分散トランザクションを結合することができます。このオプションを使用しない、BMP を持つ bean は、インプリメンテーション依存の方法でトランザクションを処理します。

– -hod ー CMP を持つエンティティー bean が永続データの格納に Host-on Demand (HOD) を必要とすることを示します。これらの bean は、セッション・サービスを使用します。このオプションは、**PAOToEJB** ツールから生成された Enterprise Bean には*使用*できません。

– -eci ー CMP を持つエンティティー bean が永続データの格納に外部呼び出しインターフェース (ECI) を必要とすることを示します。これらの bean は、セッション・サービスを使用します。このオプションは、**PAOToEJB** ツールから生成された Enterprise Bean には*使用*できません。

– -appc ー CMP を持つエンティティー bean が永続データの格納に拡張プログラム間通信機能 (APPC) を必要とすることを示します。これらの bean は、トランザクション・サービスを使用します。このオプションは、**PAOToEJB** ツールから生成された Enterprise Bean には*使用*できません。

– -exci ー CMP を持つエンティティー bean が永続データの格納に EXCI を使用することを示します。これらの bean は、トランザクション・サービスを使用します。このオプションは、**PAOToEJB** ツールから生成された Enterprise Bean には*使用*できません。

– -otma ー CMP を持つエンティティー bean が永続データの格納に OTMA を必要とすることを示します。これらの bean は、トランザクション・サービスを使用します。このオプションは、**PAOToEJB** ツールから生成された Enterprise Bean には*使用*できません。

– -ccf ー CMP を持つエンティティー bean が、共通のコネクター・フレームワーク (CCF) バック・エンドである SAP インターフェースを使用することを示します。これらの bean は、トランザクション・サービスを使用します。

• 実行オプション

- -family *familyName* ー 生成するアプリケーション・ファミリー名を指定します。デフォルトでは、この名前は、EJB JAR ファイルの名前に語 Family を付加したものに設定されます。このオプションは、値が固有である限り何度でも指定できます。
- -finderHelper *finderHelperClassName remoteInterface* ー CMP を持つエンティティー bean のファインダー・ヘルパー・クラス名 (*finderHelperClassName*) およびリモート・インターフェース名 (*remoteInterface*) を指定します。指定しない場合は、ファインダー・ヘルパー・クラスを配置機能では提供しないものと想定されます。このオプションは、値が固有である限り何度でも指定できます。 ファインダー・ヘルパー・クラスに関する詳細については、131ページの『finder メソッドの定義』を参照してください。
- -usewstringindo ー エンティティー bean のコンテナー管理フィールドを DO の (string 型ではなく) wstring IDL 型にマップします。データ・ソースが 1 バイト文字データを含む場合には、string IDL 型にマップする方が適していますが、データ・ソースが 2 バイト文字データまたは Unicode 文字データを含む場合には、wstring IDL 型にマップする方が適しています。
- -workloadmanaged ー CMP エンティティー bean または状態なしセッション bean を、ワークロード管理コンテナーに構成し、またワークロード管理されるホーム・インターフェースとともに構成するように、ツールに指示します。BMP エンティティー bean または状態付きセッション bean の場合には、ワークロード管理されるホーム・インターフェースだけでその bean を構成するように、ツールに指示します。
- -clientdep *deployed-jarFile* ー 配置する Enterprise Bean を使用する EJB クライアントが必要とする依存 JAR の名前を指定します。ファイルの絶対パスを指定しなければなりません。複数のクライアント JAR ファイルを作成するには、JAR ファイルごとにこのオプションを指定しなければなりません。このオプションは、値が固有である限り何度でも指定できます。
- -serverdep *deployed-jarFile* ー 配置する Enterprise Bean を実行する EJB サーバー (CB) が必要とする依存 JAR の名前を指定します。ファイルの絶対パスを指定しなければなりません。複数の依存 JAR ファイルを作成するには、JAR ファイルごとにこのオプションを指定しなければなりません。このオプションは、配置する Enterprise Bean が必要とするクラスを含む既存の JAR ファイルを識別するために使用することもできます。これを行う場合は、EJB サーバーの CLASSPATH 環境変数が自動的に更

新され、指定されたこの JAR ファイルが組み込まれます。このオプションは、値が固有である限り何度でも指定できます。

– -sentinel *sentinelValue* ー 配置された bean の Java タイプ・フィールドまたはコンテナー管理フィールドに値を指定します。Java タイプの値を設定している場合は、= (等号) 記号の前後にスペースを入れないでください。
– -strbehavior ー 配置された bean のコンテナー管理ストリング・フィールドのストリングの振る舞いをツールにより決定する方法を指定します。corba 値は、CORBA ストリングとしてストリングを処理することを意味します。ストリップ値はストリング内の末尾スペースを除去するようにツールに指示します。

セッション bean または BMP を持つエンティティー bean の場合は、コード生成処理は、追加のユーザー介入なしに実行されます。CMP を持つエンティティー bean の場合は、コマンドの実行中に Object Builder GUI が表示されるので、エンティティー bean の永続データを管理する DO インプリメンテーションを作成しなければなりません。詳細については、82ページの『CMP エンティティー bean 配置中のデータ・オブジェクトの構築』を参照してください。

**cbejb** ツールは、 XML (Extensible Markup Language) ファイルを生成して Object Builder にインポートすることによって Enterprise Bean を配置します。XML のインポートに失敗した場合は、プロジェクト・ディレクトリーにある import_model.log ファイルで、Object Builder によって生成されるエラー・メッセージを参照することができます。

CLASSPATH 環境変数が長すぎる場合は、**cbejb** コマンド・ファイルは失敗します。この場合は、不要なファイルをすべて除外して CLASSPATH を短くしてください。

**cbejb** ツールは、Account という名前の Enterprise Bean を含む EJB JAR ファイルのために以下のファイルを生成します。

- AccountS.jar および (*Windows NT* および *Windows 2000*) AccountS.dll または (*AIX* または *Solaris*) libAccountS.so ー この Enterprise Bean が入っている EJB サーバー (CB) が必要とするファイル。 AccountS.jar ファイルには、Account EJB JAR ファイルから生成されたコードが入っています。AccountS.dll および libAccountS.so ファイルには、必要な C++ クラスが入っています。

  (*Windows NT* および *Windows 2000*) EJB サーバー (CB) で Account Enterprise Bean を実行するには、サーバーの CLASSPATH 環境変数に

AccountS.jar ファイルを定義して、サーバーの PATH 環境変数に AccountS.dll ファイルを定義しなければなりません。一般には、配置する Enterprise Bean を EJB サーバー (CB) にインストールするときにシステム管理エンド・ユーザー・インターフェース (SM EUI) がこれらの環境変数を設定します。

(*AIX* または *Solaris*) EJB サーバー (CB) で Account Enterprise Bean を実行するには、サーバーの CLASSPATH 環境変数に AccountS.jar ファイルを定義して、サーバーの LD_LIBRARY_PATH 環境変数に libAccountS.so ファイルを定義しなければなりません。一般には、配置する Enterprise Bean を EJB サーバー (CB) にインストールするときに SM EUI がこれらの環境変数を設定します。

- AccountC.jar − EJB クライアントが必要とするファイルで、他の Enterprise Bean にアクセスする Enterprise Bean を含みます。この JAR ファイルは、Enterprise Bean インプリメンテーション・クラスを除き、元の EJB JAR ファイルに含まれるものをすべて含みます。Account Enterprise Bean を使用するには、Java EJB クライアントの CLASSPATH 環境変数に AccountC.jar および IBM Java ORB を定義していなければなりません。
- (*PAO のみ*) *paotoejbName*.jar − このファイルは、**PAOToEJB** ツールによって作成され、Enterprise Bean 内の既存の手続き型アダプター・オブジェクト (PAO) をラップするために使用されます。
- EJBAccountFamily.DDL − このファイルは、SM EUI によって使用されるデータベースを更新するために、Account ファミリーを EJB サーバー (CB) にインストールするときに使用されます。この名前は、EJB JAR ファイル名に Family.DDL というストリングが付加されたものです。

## CMP エンティティー bean 配置中のデータ・オブジェクトの構築

CMP を持つエンティティー bean を配置する場合は、Component Broker の Object Builder を使用して、DO インプリメンテーションを作成しなければなりません。この DO インプリメンテーションは、エンティティー bean の永続データを管理します。

DO インプリメンテーションを構築するには、83ページの『データ・ソースへのコンテナー管理フィールドのマッピング指針』で説明しているように、エンティティー bean のコンテナー管理フィールドを適切なデータ・ソースにマップします。その後、以下のいずれかを行います。

- 既存の DB2、Informix、または Oracle データベースを使用して bean の永続データを格納する。詳細については、86ページの『既存の DB2 または Oracle データ・ソースを使用した永続データの格納』を参照してください。

- 既存の CICS または IMS アプリケーションを使用して bean の永続データを格納する。詳細については、89ページの『既存の CICS または IMS アプリケーションを使用した永続データの格納』を参照してください。
- 新規の DB2、Informix、または Oracle データベースを使用して bean の永続データを格納する。詳細については、93ページの『永続データ保管用の新しい DB2 または Oracle データベースの定義』を参照してください。

***データ・ソースへのコンテナー管理フィールドのマッピング指針:*** **cbejb** ツールで Enterprise Bean を配置すると、Component Broker DO IDL インターフェースが作成されます。このインターフェースの IDL 属性は、エンティティー bean のコンテナー管理フィールドに対応します。 Object Builder を使用して DO インプリメンテーションを定義し、データ・ソースにあるデータ・タイプに対応する永続オブジェクト (PO) または手続き型アダプター・オブジェクト (PAO) の属性に DO 属性をマップしなければなりません。

本節では、**cbejb** ツールがエンティティー bean のコンテナー管理フィールドを DO IDL 属性にマップする方法、および Enterprise Bean の配置機能が DO IDL 属性をエンティティー bean のデータ・ソースにマップする方法について説明します。これらの指針は、既存のデータ・ソースを使用している場合でも (meet-in-the-middle 配置として知られています) 新しいデータ・ソースを定義している場合でも (トップダウン配置として知られています) 適用されます。

- EJBObject または EJBHome 変数 － EJBObject または EJBHome インターフェース・マップを Object IDL 型にインプリメントするクラスのオブジェクト。実行時には、この DO 属性は、EJBObject または EJBHome オブジェクトの CORBA プロキシーを含みます。 CB EJB ランタイムは、 (bean のコンテナー管理フィールドに格納される) EJBObject または EJBHome オブジェクトと (C++ DO に格納される) CORBA::Object 属性の間の変換を自動的に行います。同じタイプのコンテナー管理フィールドをもつコンテナー管理 bean を配置することができます。たとえば、リンク・リスト・インプリメンテーションは、そのリストの各ノードに次のノードへの参照があるコンテナー管理 bean です。コンテナー管理フィールドに巡回参照をもつこともできます。たとえば、コンテナー管理 Bean A に、Bean B タイプのコンテナー管理フィールドがあり、そこに、さらに Bean A タイプのコンテナー管理フィールドがあるというものです。Object Builder で DO から PO へのマッピングを定義する場合は、CORBA::Object からデータ・ソースへの定義済みの Component Broker マッピングを使用するか、C++ の DO から PO へのマッピング・ヘルパーを (標準の Component Broker の方法で) インプリメントして、C++ プロキシーに対してメソッドを呼び出し、永続データを取

得することもできます。C++ の DO から PO へのマッピングの作成に関する詳細については、Component Broker「プログラミング・ガイド」を参照してください。

**注:** Component Broker ではエンティティー bean のコンテナー管理フィールドを EJBObject または EJBHome オブジェクトにすることができますが、Enterprise JavaBeans 1.0 仕様では認められていません。

- 1 次キー変数 ー Enterprise Bean の 1 次キー変数を、DB2、Informix、または Oracle データベースで SQL 型の long varchar にマップしてはなりません。代わりに、varchar または char 型のいずれかを使用して、長さを適切に設定してください。
- java.lang.String 変数 ー このクラスのオブジェクトは、**cbejb** ツール (73ページの『手作業による Enterprise Bean の配置』参照) を使用してエンティティー bean を配置するときに使用されたコマンド行オプションに応じて、型 string または wstring の DO IDL 属性にマップされます。デフォルトでは、型 java.lang.String の変数は型 string の DO IDL 属性にマップされます。しかし、**cbejb** ツールの -usewstringindo オプションを使用すると、java.lang.String 変数を型 wstring の DO IDL 属性にマップすることができます。 (bean の String フィールドの一部を IDL の string 型にマップし、他のものを IDL の wstring 型にマップすることはサポートされていません。) データ・ソースが 1 バイト文字データを含む場合には、string IDL 型にマップする方が適していますが、データ・ソースが 2 バイト文字データまたは Unicode 文字データを含む場合には、wstring IDL 型にマップする方が適しています。
- java.io.Serializable 変数 ー このインターフェースをインプリメントするクラスのオブジェクトは、型 ByteString の DO IDL 属性にマップされます (この型は、IManagedClient.idl ファイルで定義された octet のシーケンスに対する型定義です)。 EJB サーバー (CB) は、 (エンティティー bean のコンテナー管理フィールドに格納された) 逐次化可能なオブジェクトを、逐次化形式のオブジェクトを含む octet の C++ シーケンスに自動的に変換します。Component Broker のデフォルトの DO から PO へのマッピングを ByteString に使用して、逐次化されたオブジェクトをデータ・ソースに直接格納します。

  C++ の DO から PO へのマッピング・ヘルパーをインプリメントして、言語間オブジェクト・モデル (IOM) によって C++ の ByteString を Java インプリメンテーションに渡すようにしていない限り、C++ の DO インプリメンテーション内部からの ByteString に含まれる逐次化された Java オブジェクトを操作することはできません。したがって、トップダウンで Enterprise Bean を開発している場合に、逐次化された Java オブジェクトをデータ・ソ

ースに格納しない場合は、型 Serializable のコンテナー管理フィールドを定義しないようにしてください。代わりに、Serializable 変数を非永続変数として、基本型のコンテナー管理フィールドを定義し、Serializable 変数の状態を取得して、Enterprise Bean の ejbLoad および ejbStore メソッドで Serializable 変数と基本型変数の間の変換を行ってください。

- 配列変数 ー これらの変数は、個々の型を DO IDL 属性にマップするのと同じ方法で、対応する型の DO IDL シーケンスにマップされます。たとえば、java.lang.String クラスの配列は、型 string のシーケンスである DO IDL 属性にマップされます (**cbejb** ツールの -usewstringindo オプションを使用した場合は、型 wstring のシーケンスとなります)。 EJB サーバー (CB) は、(エンティティー bean のコンテナー管理フィールドに格納された) 配列と、(DO に格納された) C++ シーケンスの間の変換を自動的に行います。データ・ソース内のシーケンス全体を丸ごと格納することも、C++ の DO から PO へのマッピング・ヘルパーを (標準の Component Broker の方法で) 作成してシーケンスについて繰り返し、個々の要素を個別にデータ・ソースに格納することもできます。 C++ の DO から PO へのマッピングの作成に関する詳細については、Component Broker「プログラミング・ガイド」を参照してください。
- Date/Time フィールド ー **cbejb** ツールは、他の Serializable フィールドから別々に java.util.Date タイプのコンテナー管理フィールドおよびサブクラス (java.sql.Date、java.sql.Time、java.sql.Timestamp のみ) をマップします。使用するマッピング規則は以下のとおりです。
  - java.util.Date: ISO 形式のタイム・スタンプ・ストリング (*yyyy-mm-dd-hh.mm.ss.mmmmmm*)
  - java.sql.Date: ISO 形式の日付ストリング (*yyyy-mm-dd*)
  - java.sql.Time: ISO 形式のタイム・ストリング (*hh.mm.ss*)
  - java.sql.Timestamp: ISO 形式のタイム・スタンプ・ストリング (*yyyy-mm-dd-hh.mm.ss.mmmmmm*)

  つまり、上記のタイプのコンテナー管理フィールドのいずれかを、 ISO 形式のストリングを入力値として取ることができるストリングまたはデータベース固有の日付または時刻フィールドのいずれかにマップする必要があります。 (たとえば、DB2 および Oracle は両方とも Date/Time/Timestamp 列タイプに、 ISO ストリングを入力値として取ることができます。) 配置機能で Date/Time コンテナー管理フィールドを上記のタイプ以外のものにマップするように選択する場合は、特定のデータ・マッピング・コードを DO インプリメンテーションに記述する必要があります。マッピング・コードは、ISO 形式のストリングとバックエンド固有のタイプの間で相互に変換できなければなりません。

java.sql.Timestamp クラスの精度はナノ秒単位であるのに対して、ISO タイム・スタンプ形式の精度はマイクロ秒単位です。したがって、Timestamp CMP フィールドがマップされると、 (ナノ秒単位が最も近いマイクロ秒単位に丸められるので) 精度は落ちます。 java.sql.Timestamp クラスを bean の 1 次キーの属性の 1 つとして使用する場合は、この点に特に留意する必要があります。

java.sql.Date を ISO Date 形式にマッピングする場合、時刻フィールドの値は無視されます。同様に、 java.sql.Time を ISO Time にマッピングする場合、日付フィールドの値は無視されます。

**注:** DB2 の場合のみ: 既存のデータベースの日付または時刻の出力が ISO 以外の形式である場合、配置機能は、 ″DATETIME ISO″ オプションを使用して、DB2 パッケージを再バインドしなければなりません。

***既存の DB2 または Oracle データ・ソースを使用した永続データの格納:*** 既存の DB2 または Oracle データベースを使用して CMP エンティティー bean の永続データを格納するには、以下の手順を行います。最終的な結果はデータベース・スキーマ内の項目に対応する属性を持つ PO です。

1. Object Builder を開始すると、「Open Project (プロジェクトを開く)」ダイアログが表示されます。 Enterprise Bean のプロジェクト・ディレクトリーの位置を選択し、**「Finish (完了)」**をクリックする。
2. 既存のリレーショナル・データベース・スキーマをインポートするために、**「DBA-Defined Schemas (DBA 定義のスキーマ)」**をクリックして、適切なデータベース・タイプを右クリックする。
   a. ポップアップ・メニューで、**「Import (インポート)」**および**「SQL」**をクリックする。
   b. **「Import SQL (SQL のインポート)」**ダイアログ・ボックスで、**「Find (検索)」**をクリックし、該当する SQL ファイルを表示させる。
   c. 該当する SQL ファイルをダブルクリックする。
   d. **「Database Name (データベース名)」**テキスト・フィールドの名前を、`Database` から実際のデータベース名に変更する。
   e. 適切なデータベース・タイプを選択して、**「Finish (完了)」**をクリックする。
3. データベース・スキーマから永続オブジェクト (PO) を作成するために、**「DBA-Defined Schemas (DBA 定義のスキーマ)」**を展開し、該当するグループを展開する。

a. 該当するスキーマを強調表示させ、右クリックしてポップアップ・メニューを表示させる。**「Add (追加)」 -> 「Persistent Object (永続オブジェクト)」**とクリックする。

b. **「Names and Attributes (名前および属性)」**ダイアログ・ボックスで、デフォルトを受け入れて**「Finish (完了)」**をクリックする。

4. 以下のように、DO インプリメンテーションを作成する。

a. **「User-Defined DOs (ユーザー定義の DO)」**を展開して、**「DO File (DO ファイル)」**(たとえば、CBAccountDO) を展開し、**「DO Interface (DO インターフェース)」**(たとえば、com_ibm_ejs_doc_account_AccountDO) を展開し、**「DO Implementation (DO インプリメンテーション)」**を選択する。

b. **「DO Implementation (DO インプリメンテーション)」**ポップアップ・メニューで、**「Properties (プロパティー)」**を選択する。

c. **「Name and Platform (名前およびプラットフォーム)」**ページで、**「Deployment Platform (配置プラットフォーム)」**(たとえば、NT、AIX、または Solaris) を選択して、**「Next (次へ)」**をクリックする。

d. **「Behavior (振る舞い)」**ページで、以下のように適切な内容を選択して**「Next (次へ)」**をクリックする。

- *DB2 の場合は*、**「Environment (環境)」**には「BOIM with any Key (任意のキーと BOIM)」を選択し、**「Form of Persistent Behavior and Implementation (永続の振る舞いとインプリメンテーションの形式)」**には「Embedded SQL (組み込み SQL)」を選択し、**「Data Access Pattern (データ・アクセス・パターン)」**には「Delegating (代行)」を選択し、**「Handle for Storing Pointers (ポインター格納用の処理)」**には「Home name (ホーム名)」および「key (キー)」を選択する。
- *Oracle の場合は*、**「Environment (環境)」**には「BOIM with any Key (任意のキーと BOIM)」を選択し、**「Form of Persistent Behavior and Implementation (永続の振る舞いとインプリメンテーションの形式)」**には「Oracle Caching service (Oracle のキャッシュ・サービス)」を選択し、**「Data Access Pattern (データ・アクセス・パターン)」**には「Delegating (代行)」を選択し、**「Handle for Storing Pointers (ポインター格納用の処理)」**には「Home name (ホーム名)」および「key (キー)」を選択する。

e. **「Implementation Inheritance (インプリメンテーションの継承)」**ページで、以下のように親クラスに関して適切な選択を行って、**「Next (次へ)」**をクリックする。

- *DB2* の場合は、IRDBIMExtLocalToServer::IDataObject を選択する。
- *Oracle* の場合は、IRDBIMExtLocalToServer::ICachingServiceDataObject を選択する。

f. **「Attributes (属性)」**、**「Methods (メソッド)」**、および**「Key and Copy Helper (キーおよびコピー・ヘルパー)」**各のページで、**「Next (次へ)」**をクリックして、それぞれのデフォルトを受け入れる。

g. **「Associated Persistent Objects (関連する永続オブジェクト)」**ページで、**「Add Another (もう 1 つ追加)」**をクリックする。インスタンス名のデフォルト (iPO) を受け入れ、正しいタイプを選択する。**「Next (次へ)」**をクリックする。

h. **「Attribute Mapping (属性のマッピング)」**ページで、エンティティー bean のコンテナー管理フィールドをデータベース・スキーマの対応する項目にマップする。 Object Builder が、対応する永続オブジェクト属性を識別することのできる、データ・オブジェクト属性のデフォルト・マッピングを作成します。通常、デフォルト・マッピングは 1 次キー変数を除くすべての場合に適切ですが、 1 次キー変数は、long varchar 型ではなく、varchar 型または char 型にマップしなければなりません。詳細については、83ページの『データ・ソースへのコンテナー管理フィールドのマッピング指針』を参照してください。属性のマッピングが完了したら、**「Finish (完了)」**をクリックする。

i. *Oracle* のみ。CMP を持つエンティティー bean を Oracle データベースにマッピングする場合は、**「Container Definition (コンテナー定義)」**フォルダーを展開して EJB コンテナーを右クリックする。ポップアップ・メニューで、**「Properties (プロパティー)」**をクリックする。ウィザードで、**「Data Access Pattern (データ・アクセス・パターン)」**が表示されるまで**「Next (次へ)」**をクリックする。このページが表示されたら、**「Cache Service (キャッシュ・サービス)」**チェック・ボックスにチェック・マークを付けて**「Finish (完了)」**をクリックする。

j. **「File (ファイル)」 -> 「Exit (終了)」**とクリックして Object Builder を終了する。変更内容を保管するかどうかを確認するプロンプトが表示された場合は、保管する。

k. **「Database (データベース)」**テキスト・フィールドで指定したデータベースを作成し、**「Schema File (スキーマ・ファイル)」**テキスト・フィールドで指定した SQL ファイルを使用して、データベース・テーブルを作成する。 SQL ファイルによるデータベースおよびデータベース・テーブルの作成に関する詳細については、DB2 または Oracle の資

料を参照してください。 SQL ファイルは、以下のディレクトリーにあります。ここで、*projDir* は、**cbejb** ツールによって作成されたプロジェクト・ディレクトリーです。

- *Windows NT* および *Windows 2000* では、*projDir*¥Working¥NT
- *AIX* では、*projDir* /Working/AIX
- *Solaris* では、*projDir* /Working/Solaris

***既存の CICS または IMS アプリケーションを使用した永続データの格納:*** 永続オブジェクト・アダプター (PAO) 格納用に CICS または IMS を使用するには、以下の手順に従ってください。永続保管に (PAO によって) CICS または IMS アプリケーションを使用する場合は、アプリケーション・データのみが使用されます。CICS または IMS アプリケーションのメソッドは、プッシュダウン・メソッドであり、データの格納およびロードではなく、アプリケーション固有のロジックを実行します。

CMP を持つエンティティー bean を既存の CICS または IMS アプリケーションにマップするには、以下の前提条件が満たされていなければなりません。

- エンティティー bean を HOD または ECI ベースのアプリケーションにマップする場合は、その bean のトランザクション属性を TX_MANDATORY に設定しなければなりません。 APPC ベースのアプリケーションにマップする場合は、トランザクション属性を TX_MANDATORY または TX_REQUIRED のいずれかに設定しなければなりません。
- 既存の CICS または IMS アプリケーションは、手続き型アダプター・オブジェクト (PAO) として表さなければなりません。 PAO の作成に関する詳細については、「*手続き型アプリケーション・アダプター開発ガイド*」を参照してください。
- PAO クラス・ファイルを CLASSPATH 環境変数で指定しなければなりません。
- エンティティー bean は、すべての Enterprise Bean ロジックをインプリメントしなければなりません。そのうえで、Enterprise Bean のコンテナー管理フィールドを PAO にマップする必要があります。PAO のプッシュダウン・メソッドを Enterprise Bean から使用することはできません。 (PAO プッシュダウン・メソッドは、 108ページの『既存の CICS または IMS アプリケーションからの Enterprise Bean の作成』で説明するように **PAOToEJB** を使用して生成された、CMP を持つエンティティー bean から使用することができます。)
- **cbejb** ツールを以下のように実行しなければなりません。ここで、*ejb-jarFile* は、エンティティー bean を含む EJB JAR ファイルです。

```
# cbejb ejb-jarFile [-hod | -eci | -appc [beanNames]]
```

**cbejb** ツールの構文に関する詳細については、73ページの『手作業による Enterprise Bean の配置』を参照してください。

前提条件を満たしたら、Object Builder を使用して、エンティティー bean および CICS または IMS アプリケーションの間のマッピングを作成します。

1. Object Builder を開始すると、「Open Project (プロジェクトを開く)」ダイアログが表示されます。 Enterprise Bean のプロジェクト・ディレクトリーの位置を選択し、**「Finish (完了)」**をクリックする。
2. メインメニューから、**「Platform (プラットフォーム)」**、**「Target (ターゲット)」**の順に選択する。390 プラットフォームのチェック・マークを解除する。
3. **「User-Defined PA Schemas (ユーザー定義の PA スキーマ)」**をクリックして、選択項目を右クリックする。
4. ポップアップ・メニューから、**「Import (インポート)」**、**「bean」**の順にクリックする。**「Import Bean (bean のインポート)」**ダイアログ・ボックスで、PAO bean のクラス名を入力して**「Next (次へ)」**をクリックする。
5. 適切なコネクター・タイプを選択して**「Next (次へ)」**をクリックする。
6. **「Properties (プロパティー)」**リストから 1 次キー属性名を選択する。
7. **「>>」**をクリックしてその 1 次キーを**「Key Attributes (キー属性)」**リストに移動させ、**「Finish (完了)」**をクリックする。
8. *HOD* および *ECI* のみについて、MO および HomeMO の両方に対して以下を行う。
   a. **「Tasks and Object (タスクおよびオブジェクト)」**パネルで、**「User-Defined Business Objects (ユーザー定義のビジネス・オブジェクト)」**を展開し、オブジェクトを展開して、そのオブジェクトの BO を展開する。 MO ファイルのポップアップ・メニューから、**「Properties (プロパティー)」**をクリックする。
   b. **「Service to use (使用するサービス)」**プロパティーを、「`Transaction Service` (トランザクション・サービス)」から「`Session Service` (セッション・サービス)」に変更する。
9. 以下のように、DO インプリメンテーションを作成する。
   a. **「Tasks and Object (タスクおよびオブジェクト)」**パネルで、**「User-Defined DOs (ユーザー定義の DO)」**を展開し、メニューから**「DO File (DO ファイル)」**を展開して、**「DO Interface (DO インターフェース)」**をクリックする。

b. **「DO Interface (DO インターフェース)」**ポップアップ・メニューで、**「Add Implementation (インプリメンテーションの追加)」**を選択する。
c. **「Behavior (振る舞い)」**ページで、**「Environment (環境)」**には「BOIM with any Key (任意のキーと BOIM)」を選択し、**「Form of Persistent Behavior and Implementation (永続の振る舞いとインプリメンテーションの形式)」**には「Procedural Adapters (手続き型アダプター)」を選択し、**「Data Access Pattern (データ・アクセス・パターン)」**には「Delegating (代行)」を選択し、**「Handle for Storing Pointer (ポインター格納用の処理)」**には「Default (デフォルト)」を選択する。**「Next (次へ)」**をクリックする。
d. **「Implementation Inheritance (インプリメンテーションの継承)」**ページ、**「Attributes (属性)」**ページ、**「Methods (メソッド)」**ページ、および**「Key and Copy Helper (キーおよびコピー・ヘルパー)」**ページで**「Next (次へ)」**をクリックする。
e. **「Associated Persistent Object (関連する永続オブジェクト)」**ページで、**「Add Another (もう 1 つ追加)」**をクリックし、以前に作成した PO が選択されていることを確認してから、**「Next (次へ)」**をクリックする。
f. **「Attribute Mapping (属性のマッピング)」**ページで、エンティティー bean のコンテナー管理フィールドと既存の PAO の項目との対応関係を指定する。この指定を行うには、 (エンティティー bean のコンテナー管理フィールドに一致する) DO の属性と (既存の PAO に一致する) PO の属性の間のマッピングを定義する。**「Attributes (属性)」**リストには、bean のコンテナー管理フィールドそれぞれに対応する DO 属性があります。

   **「Attributes (属性)」**リストの DO 属性ごとに、その属性を右クリックして、メニューから**「Primitive (プリミティブ)」**をクリックする。**「Persistent Object Attribute (永続オブジェクト属性)」**ドロップダウン・メニューから、DO 属性に対応する PO 属性 (属性データベース・スキーマからの項目) を選択する。詳細については、83ページの『データ・ソースへのコンテナー管理フィールドのマッピング指針』を参照してください。すべてのコンテナー管理フィールドを処理したら、**「Next (次へ)」**をクリックする。
g. **「Methods Mapping (メソッド・マッピング)」**ページで、**「Special Framework Methods (特別なフレームワーク・メソッド)」**のリストにあるメソッドごとに、**「Add Mapping (マッピングの追加)」**を右クリックする。**「Persistent Object Method (永続オブジェクト・メソ**

**ッド)」**ドロップダウン・メニューから、選択した DO メソッドと同じ名前の PO メソッドを選択する。使用可能なマッピングに追加のメソッドがある場合は、メソッドを、同様の名前の付いたメソッドにマップする。たとえば、update は update() にマップします。すべてのメソッドを処理したら、**「Finish (完了)」**をクリックする。

h. **「Container Definition (コンテナー定義)」**フォルダーを展開して EJB コンテナーを右クリックする。ポップアップ・メニューで、**「Properties (プロパティー)」**をクリックする。ウィザードで、**「Data Access Pattern (データ・アクセス・パターン)」**が表示されるまで**「Next (次へ)」**をクリックする。

i. **「Data Access Pattern (データ・アクセス・パターン)」**ページで、以下の項目のいずれかを選択して**「Next (次へ)」**をクリックする。

- *HOD または ECI の場合は*、「PAA セッション・サービスの使用」を選択する。
- *APPC の場合は*、「PAA トランザクション・サービスの使用」を選択する。

j. **「Service Details (サービスの詳細)」**ページで、以下を行って**「Next (次へ)」**をクリックする。

- *HOD または ECI の場合は*、**「Behavior for Methods Called Outside a Transaction (トランザクション外部で呼び出されたメソッドの振る舞い)」**には「Throw an exception and abandon the call (例外を throw して呼び出しを中止)」を選択する。接続名 (MY_PAA_Connection など) を定義して、**「Type of connection (接続のタイプ)」**に、それぞれ「Host on Demand (要求時ホスト)」または「ECI connection (ECI 接続)」を選択する。
- *APPC の場合は*、TX_MANDATORY トランザクション属性を持つ Enterprise Bean に「Throw an exception and abandon the call (例外を throw して呼び出しを中止)」を選択するか、TX_REQUIRED トランザクション属性を持つ Enterprise Bean に「Start a new transaction and complete the call (新規トランザクションを開始して呼び出しを完了)」を選択する。

k. **「Business Object (ビジネス・オブジェクト)」**には「Caching (キャッシュ)」を選択する。

l. **「Data Object (データ・オブジェクト)」**には「Delegating (代行)」を選択する。

m. **「Finish (完了)」**をクリックする。

10. **「File (ファイル)」 -> 「Exit (終了)」**とクリックして Object Builder を終了する。変更内容を保管するかどうかを確認するプロンプトが表示された場合は、保管する。

***永続データ保管用の新しい DB2 または Oracle データベースの定義:*** トップダウン開発アプローチを使用して Enterprise Bean を開発する場合は、以下の 3 段階で Enterprise Bean を配置しなければなりません。

1. データベース・スキーマを定義し、CMP を持つエンティティー bean のコンテナー管理フィールドをそのデータベース・スキーマにマップし、コードを生成してこのマッピングをカプセル化する。詳細については、『データベース・スキーマのマッピング』を参照してください。
2. データベースおよびデータベース・テーブルを作成する。詳細については、『データベースおよびデータベース・テーブルの作成』を参照してください。
3. フェーズ 1 で生成したコードをコンパイルする。データベースおよびデータベース・テーブルが存在しない場合は、コンパイルは失敗します。

***データベース・スキーマのマッピング:*** エンティティー bean をデータベースにマップする方法を定義したら、コード生成後にコンパイルしないように、-nc とともに **cbejb** ツールを実行してマッピングを作成します。たとえば、EJBAccount.jar という名前の EJB JAR ファイルに格納された Account bean のためのマッピングを作成するには、以下のコマンドを入力します。

```
# cbejb EJBAccount.jar -nc -queryable [-oracle | -cacheddb2]
```

**注:** 永続データの格納に使用するデータベースが Oracle または DB2 のいずれかである場合は、それら用のオプションも指定しなければなりません。

***データベースおよびデータベース・テーブルの作成:*** Object Builder の GUI を使用してデータベースおよびデータベース・テーブルを作成するには、以下の手順に従ってください。

1. Object Builder を開始すると、「Open Project (プロジェクトを開く)」ダイアログが表示されます。 Enterprise Bean のプロジェクト・ディレクトリーの位置を選択し、**「Finish (完了)」**をクリックする。
2. 以下のように、DO インプリメンテーションを作成する。
   a. **「User-Defined DOs (ユーザー定義の DO)」**を展開して、メニューから**「DO File (DO ファイル)」**を展開し、**「DO Interface (DO インターフェース)」**をクリックする。
   b. **「DO Interface (DO インターフェース)」**ポップアップ・メニューで、**「Add Implementation (インプリメンテーションの追加)」**を選択

する。そのインプリメンテーションがすでに存在している場合には、それを選択し、ポップアップ・メニューを起動し、**「Properties (プロパティー)」**を選択することにより、インプリメンテーションを変更することができます。

c. **「Name and Platform (名前およびプラットフォーム)」**ページで、プラットフォームを選択して**「Next (次へ)」**をクリックする。

d. **「Behavior (振る舞い)」**ページで、以下のように適切な内容を選択して**「Next (次へ)」**をクリックする。

- *DB2 の場合*：**「Environment (環境)」**には「BOIM with any Key (任意のキーと BOIM)」を選択し、**「Form of Persistent Behavior and Implementation (永続の振る舞いとインプリメンテーションの形式)」**には「Embedded SQL (組み込み SQL)」を選択し、**「Data Access Pattern (データ・アクセス・パターン)」**には「Delegating (代行)」を選択し、**「Handle for Storing Pointers (ポインター格納用の処理)」**には「Home name (ホーム名)」および「key (キー)」を選択する。
- *Oracle の場合*：**「Environment (環境)」**には「BOIM with any Key (任意のキーと BOIM)」を選択し、**「Form of Persistent Behavior and Implementation (永続の振る舞いとインプリメンテーションの形式)」**には「Oracle Caching services (Oracle のキャッシュ・サービス)」を選択し、**「Data Access Pattern (データ・アクセス・パターン)」**には「Delegating (代行)」を選択し、**「Handle for Storing Pointers (ポインター格納用の処理)」**には「Home name (ホーム名)」および「key (キー)」を選択する。

e. **「Implementation Inheritance (インプリメンテーションの継承)」**ページで、以下のように親クラスに関して適切な選択を行って、**「Next (次へ)」**をクリックする。

- *DB2 の場合は*、IRDBIMExtLocalToServer::IDataObject を選択する。
- *Oracle の場合は*、RDBIMExtLocalToServer::ICachingServiceDataObject を選択する。
- *CICS または IMS PAO の場合は*、IRDBIMExtLocalToServer::IDataObject を選択する。

f. **「Attributes (属性)」**、**「Methods (メソッド)」**、および**「Key and Copy Helper (キーおよびコピー・ヘルパー)」**各のページで、**「Next (次へ)」**をクリックして、それぞれのデフォルトを受け入れる。

g. **「Associated Persistent Objects (関連する永続オブジェクト)」**ページで、**「Add Another (もう 1 つ追加)」**をクリックする。インスタンス名のデフォルト (iPO) を受け入れ、正しいタイプを選択する。**「Next (次へ)」**をクリックする。

h. **「Attribute Mapping (属性のマッピング)」**ページで、エンティティー bean のコンテナー管理フィールドをデータベース・スキーマの対応する項目にマップする。通常、デフォルト・マッピングは 1 次キー変数を除くすべての場合に適切ですが、 1 次キー変数は、long varchar 型ではなく、varchar 型または char 型にマップしなければなりません。Object Builder が、対応する永続オブジェクト属性を識別することのできる、データ・オブジェクト属性のデフォルト・マッピングを作成します。詳細については、83ページの『データ・ソースへのコンテナー管理フィールドのマッピング指針』を参照してください。属性のマッピングが完了したら、**「Finish (完了)」**をクリックする。

i. *Oracle* のみ。CMP を持つエンティティー bean を Oracle データベースにマッピングする場合は、**「Container Definition (コンテナー定義)」**フォルダーを展開して EJB コンテナーを右クリックする。ポップアップ・メニューで、**「Properties (プロパティー)」**をクリックする。ウィザードで、**「Data Access Pattern (データ・アクセス・パターン)」**が表示されるまで**「Next (次へ)」**をクリックする。このページが表示されたら、**「Cache Service (キャッシュ・サービス)」**チェック・ボックスにチェック・マークを付けて**「Finish (完了)」**をクリックする。

j. **「File (ファイル)」 -> 「Exit (終了)」**とクリックして Object Builder を終了する。変更内容を保管するかどうかを確認するプロンプトが表示された場合は、保管する。

k. **「Database (データベース)」**テキスト・フィールドで指定したデータベースを作成し、**「Schema File (スキーマ・ファイル)」**テキスト・フィールドで指定した SQL ファイルを使用して、データベース・テーブルを作成する。 SQL ファイルによるデータベースおよびデータベース・テーブルの作成に関する詳細については、DB2 または Oracle の資料を参照してください。 SQL ファイルは、以下のディレクトリーにあります。ここで、*projDir* は、**cbejb** ツールによって作成されたプロジェクト・ディレクトリーです。

- *Windows NT* では、*projDir*¥Working¥NT
- *AIX* では、*projDir* /Working/AIX
- *Solaris* では、*projDir* /Working/Solaris

***生成されたコードのコンパイル:*** データベースおよびデータベース・テーブルを両方とも作成したら、以下のコマンドを使用して Enterprise Bean のコードをコンパイルします。

- **Windows NT の場合**

  > cd *projDir*¥Working¥NT

  > nmake -f all.mak
- **AIX の場合**

  # cd *projDir*/Working/AIX

  # make -f all.mak
- **Solaris の場合。**

  # cd *projDir*/Working/Solaris

  # make -f all.mak

### Enterprise Bean のインストールおよびその EJB サーバー (CB) の構成

Enterprise Bean をインストールし、その EJB サーバー (CB) を構成するには、以下の手順に従ってください。

1. (*DB2 を使用し、CMP を持つエンティティー bean のみ*) バインド・ファイルを使用して、Enterprise Bean をデータベースにバインドする。このバインド・ファイルは、**cbejb** を使用する副次作用として Object Builder によって生成されるものです (たとえば、db2 bind AccountTblPO.bnd)。
2. SM EUI を使用して、**cbejb** によって生成されたアプリケーションをインストールする。一般に、このインストールは、Object Builder によって生成された Component Broker アプリケーションのインストールと同じです。
   a. ホスト・イメージにアプリケーションをロードする。
   b. アプリケーションを構成に追加する。
   c. EJB アプリケーションをサーバー・グループまたはサーバーに関連付ける。(サーバー・グループまたはサーバーが存在しない場合は、作成しなければなりません。)
   d. (*CMP を持つエンティティー bean のみ*) エンティティー bean のデータ・ソース (DB2、Oracle、CICS、または IMS PAA) を EJB アプリケーションに関連付ける。
      - DB2: DB2 サービス (iDB2IMServices) を EJB サーバーに関連付ける。
      - Oracle: Oracle サービス (iOAAServices) を EJB サーバーに関連付ける。

- CICS または IMS PAA: PAA サービス (iPAAServices) を EJB サーバーに関連付ける。

e. ホストで EJB サーバー (CB) を構成する。

f. クライアントおよびサーバーの両方について、ORB 要求タイムアウトを 300 秒に設定する。

g. EJB サーバーのために Java 仮想マシン (JVM) プロパティーを設定する必要がある場合は、JVM プロパティーを編集する。この操作は、サーバー・イメージではなく、サーバー・モデルで行ってください。たとえば、Enterprise Bean が JNDI 検索を実行して別の Enterprise Bean にアクセスする場合は、Enterprise Bean のホストとして機能するサーバーは、JNDI プロパティーの値を含むように JVM プロパティーが設定されていなければなりません。

h. EJB サーバー構成を活動化する。

i. EJB サーバーを開始する。

### JNDI ネーム・スペースへの Enterprise Bean の JNDI 名のバインド

**注:** この節は、AIX、Windows NT、Windows 2000、および Solaris プラットフォームで稼働しているサーバーには適用されません。

Enterprise Bean の JNDI ホーム名は、22ページの『デプロイメント・ディスクリプター』で説明するデプロイメント・ディスクリプター内で定義されます。この名前は、EJB クライアント (他の Enterprise Bean を含む) によって、Enterprise Bean のホーム・インターフェースを検索するために使用されます。

**ejbbind** ツールは、Component Broker のネーム・スペースで、Enterprise Bean の EJBHome インターフェースをインプリメントする CB ホームを検索します。また、Enterprise Bean のデプロイメント・ディスクリプターで指定された JNDI ホーム名を使用してホーム名をネーム・スペースに再バインドします。このバインドによって、EJB クライアントは、bean のデプロイメント・ディスクリプターで指定された JNDI 名を使用して EJB ホームを検索することができます。 Enterprise Bean は、bean が配置されたマシンとは異なるマシンにバインドすることができます。

Component Broker ネーム・スペースのサブツリーは JNDI 名のバインド先であり、**ejbbind** ツールとともにコマンド行オプションを使用して制御することができます。名前のバインド方法 (選択するサブツリー) は、EJB クライアントが Enterprise Bean の EJB ホームの検索に使用しなければならない JNDI 名

に影響を及ぼすだけでなく、Enterprise Bean の EJB ホームの可視性にも影響を及ぼします。特に、JNDI 名は、以下のいずれかの方法でバインドすることができます。

- JNDI 名をローカル・ルートにバインドすることができます。このバインド方法では、EJB クライアントは、Enterprise Bean のデプロイメント・ディスクリプターにある JNDI 名を使用します。この方法では、同じネーム・サーバー (同じブートストラップ・ホスト) を使用する EJB クライアントへの EJB ホーム からの可視性が制限され、ツリー内の他の名前と重複する可能性があります。
- JNDI 名をホスト名ツリー (host/resources/factories/EJBHomes にあります) にバインドすることができます。このバインド方法では、EJB クライアントは、bean のデプロイメント・ディスクリプターで指定された JNDI 名に接頭部としてストリング `host/resources/factories/EJBHomes` を付加しなければなりません。この方法では、ツリー内の他の名前との重複が最小化されますが、同じネーム・サーバーを使用するクライアントへの Enterprise Bean ホームの可視性が制限されます。
- JNDI 名をワークグループ名ツリー (workgroup/resources/factories/EJBHomes にあります) にバインドすることができます。このバインド方法では、EJB クライアントは、Enterprise Bean のデプロイメント・ディスクリプターで指定された JNDI 名に接頭部としてストリング `workgroup/resources/factories/EJBHomes` を付加しなければならず、同じ優先ワークグループに属するネーム・サーバーを使用するすべての EJB クライアントに対してこの EJB ホームが可視になります。
- JNDI 名をセル名ツリー (cell/resources/factories/EJBHomes にあります) にバインドすることができます。このバインド方法では、EJB クライアントは、bean のデプロイメント・ディスクリプターで指定された JNDI 名に接頭部としてストリング `cell/resources/factories/EJBHomes` を付加しなければならず、EJB ホームがセル全体にわたって可視になります。

**ejbbind** ツールを実行する前に、以下を行います。

- **cbejb** ツールを使用して、Component Broker 用の Enterprise Bean を配置する。詳細については、73ページの『手作業による Enterprise Bean の配置』を参照してください。
- **cbejb** ツールが生成した Component Broker アプリケーションをインストールし、SM EUI を使用して特定の EJB サーバー (CB) で構成する。詳細については、96ページの『Enterprise Bean のインストールおよびその EJB サーバー (CB) の構成』を参照してください。

- CBConnector サービスおよびネーム・サーバーを開始する (まだ実行されていない場合)。詳細については、Component Broker「*システム管理ガイド*」を参照してください。
- アプリケーションを実行する EJB サーバー (CB) を含む構成を活動化する。
- ネーム・サーバーを実行するマシンの IP アドレス (ブートストラップ・ホスト名) およびポート番号 (ブートストラップ・ポート) を決定する。

以下の構文で、**ejbbind** コマンドを起動します。

```
ejbbind ejb-jarFile [beanParm] [-f]
[-BindLocalRoot ] [-BindHost]  [-BindWorkgroup] [-BindCell] [-BindAllTrees]
[-ORBInitialHost hostName] [-ORBInitialPort portNumber]
[-u] [-UnbindLocalRoot] [-UnbindHost] [-UnbindWorkgroup] [-UnbindCell]
[-UnbindAllTrees]
```

*ejb-jarFile* は、バインドまたはアンバインドを行う Enterprise Bean を含む EJB JAR ファイルの完全修飾パス名です。オプションの *beanParm* 引き数は、EJB JAR ファイル内の単一の Enterprise Bean をバインドするために使用します。この bean は、完全修飾名 (com.ibm.ejs.doc.account.Account など。ここで、Account は bean 名です) または Enterprise Bean のデプロイメント・ディスクリプター・ファイルから拡張子 .ser を除いたものを指定することによって識別することができます。 Enterprise Bean が EJB JAR ファイルに複数のデプロイメント・ディスクリプターを持つ場合は、Enterprise Bean 名ではなく、デプロイメント・ディスクリプター・ファイル名を指定しなければなりません。

オプションを指定しない場合は、ローカル・ホストおよびポート 900 をブートストラップ・ホスト (ネーム・サーバー) に使用して、JNDI 名がローカル・ルートのネーム・ツリーにバインドされます。

その他のオプションを以下に示します。

- -f － JNDI 名が既にネーム・スペースにバインドされている場合でも、強制的にバインドします。このオプションは、アンバインド・コマンド・オプションを指定する場合は無効です。
- -BindLocalRoot － JNDI 名をローカル・ルートのネーム・ツリーにバインドします。
- -BindHost － JNDI 名をホスト名ツリーにバインドします。
- -BindWorkgroup － JNDI 名をワークグループ名ツリーにバインドします。
- -BindCell － JNDI 名をセル名ツリーにバインドします。

- -BindAllTrees － JNDI 名をホスト、ワークグループ、およびセル名ツリーにバインドします。
- -ORBInitialHost *hostName* － ブートストラップ・ホストを識別します (デフォルトはローカル・ホストです)。
- -ORBInitiallPort *portNumber* － ブートストラップ・ポートを識別します (デフォルトはポート 900 です)。
- -u － JNDI 名をアンバインドします。このオプションは、バインド・コマンド・オプションとともに指定する場合は無効です。
- -UnbindLocalRoot － JNDI 名をローカル・ルート (root) のネーム・ツリーからアンバインドします。
- -UnbindHost － JNDI 名をホスト名ツリーからアンバインドします。
- -UnbindWorkgroup － JNDI 名をワークグループ名ツリーからアンバインドします。
- -UnbindCell － JNDI 名をセル名ツリーからアンバインドします。
- -UnbindAllTrees － JNDI 名をホスト、ワークグループ、およびセル名ツリーからアンバインドします。

コマンドが成功した場合は、以下のようなメッセージが出されます。

```
名前 AccountHome は、CB ホームにバインドされました。
```

以下の場合は、**ejbbind** ツールを再度実行しなければなりません。

- Enterprise Bean の JNDI 名を変更する。 **jetace** ツールを使用して、JNDI 名を変更することができます。詳細については、39ページの『EJB モジュールの作成』を参照してください。
- Component Broker を再構成する。この場合は、この構成によってサービスが提供されるすべての Enterprise Bean を再バインドしなければなりません。
- 異なる EJB サーバー (CB) または異なるマシンに Enterprise Bean を移動する。

## 逐次列挙型を使用可能にするシステム管理の構成

逐次列挙型 (48ページの『EJB サーバー (CB) でのファインダー・ロジックの作成』を参照) を使用可能にするには、以下のステップに従います。

1. システム管理エンド・ユーザー・インターフェース (SM EUI) から「View (ビュー)」メニューに進み、「View Level to Control (制御するビュー・レベル)」を設定します。
2. **「Host Image (ホスト・イメージ)」**を展開します。
3. ホスト名を展開します。

4. **「Server Image (サーバー・イメージ)」**を展開します。
5. サーバー名を展開します。
6. **「Container Image (コンテナー・イメージ)」**を展開します。
7. **「iIteratorSysObjsNoPRef」**を右マウス・ボタン・クリックします。ポップアップ・メニューから**「Properties (プロパティー)」**を選択します。以下のようにプロパティーを変更します。
   - **「Default transaction policy (デフォルト・トランザクション・ポリシー)」**を「throwException」に変更します。
   - **「Memory management policy (メモリー管理ポリシー)」**を「passivate at end of transaction (トランザクション終了時に起動)」に変更します。

トランザクション・ポリシーは、呼び出し側が確実にトランザクションを開始できるようにします。メモリー管理ポリシーは、トランザクションの完了時に逐次列挙型が確実に処理されるようにします。

## CBConnector のライフ・サイクル・サービスの使用による EJB ホームの解決

**注:** この節は、AIX、Windows NT、Windows 2000、および Solaris プラットフォームで稼働しているサーバーにのみ適用されます。

EJB クライアントで、単純な JNDI lookup を実行する場合は、名前と特定の EJB ホーム・インスタンスの間で 1 対 1 のマッピングが行われます。分散環境で、このモデルに制限を設けることができます。このような環境で、たとえば、同じタイプの Enterprise Bean をサポートする EJB ホームが多数存在する場合があります。このホームの特定のインスタンスを要求する場合は、アプリケーションを必要としない方法を採ることをお勧めします。さらに、システムに変更があっても、 EJB ホームの別のインスタンスを指定するのに、アプリケーションを変更したり、再配置したりする必要がないことが重要です。CBConnector ライフ・サイクル・サービスは、アプリケーションが、分散環境の特定の有効範囲内にあるホームを要求できるが、その環境の厳密な構成の指定から分離された間接性および抽象性のレベルを提示するものです。ライフ・サイクル・ファクトリー・ファインダーについて、詳しくは、「*上級プログラミング・ガイド*」のライフ・サイクルの節を参照してください。

CBConnector を使用して、 JNDI コンテキストをライフ・サイクル・サービスのファクトリー・ファインダーと関連付けることができます。その結果、関連付けられたファクトリー・ファインダーを使用して、コンテキストからの EJB ホームの lookup オペレーションを解決します。このようなコンテキストは、

EJB アプリケーションの配置機能がこれらのアプリケーションのクライアントに透過的な方法でファクトリー・ファインダー機能を最大限利用できるようにします。

ファクトリー・ファインダーを使って EJB ホームの lookup を解決するには、アプリケーションの配置機能により、 CBConnector 提供のデフォルト・ファクトリー・ファインダーに関連付けられた定義済みのデフォルト・アプリケーションのコンテキストを使用するか、アプリケーション固有のコンテキストを作成し、それらを任意に指定されたファクトリー・ファインダーと関連付ける **appbind** ツールを使用することができます。それぞれの方法について、詳しくは、『デフォルトのコンテキストとファインダーの関連付け』および 104ページの『アプリケーション固有のコンテキストおよび appbind ツール』を参照してください。

**注:** デフォルトのアプリケーション・コンテキストおよびアプリケーション固有のコンテキストを使用すると、 JNDI 名と EJB ホーム・インスタンスの単純な 1 対 1 対応のマッピングを行う **ejbbind** ツールは必要ではなくなります。クライアントは、 **appbind** ツールにより生成されるデフォルトの初期コンテキスト・ファクトリーまたはアプリケーション固有のコンテキスト・ファクトリーのいずれかを使用しなければなりません。

## デフォルトのコンテキストとファインダーの関連付け

CBConnector に作成され、ファクトリーの検出時に特定の有効範囲にある位置をそれぞれ検索するデフォルトのファクトリー・ファインダーが複数あります。 EJB アプリケーションが CBConnector サーバーに配置されている場合、アプリケーションの EJB ホームは、アプリケーションの EJB jar ファイルに入っているデプロイメント・ディスクリプターで指定されている EJB ホームの名前を使用して、ライフ・サイクル・リポジトリーでバインドされます。ファクトリー・ファインダーは、特定の検索規則の有効範囲内で任意の EJB ホームを検出することができます。

EJB クライアントは、CBConnector に標準装備されている特定のデフォルト・ファクトリー・ファインダーを、そのファクトリー・ファインダーに対応する初期コンテキスト・ファクトリーを使用することで簡単に使用することができます。コンテキスト・ファクトリーで戻される初期コンテキストは、それに対応するファクトリー・ファインダーを使用して、 EJB ホームの lookup 要求を解決します。

以下の初期コンテキスト・ファクトリーからコンテキストが戻されます。

1. com.ibm.ejb.cb.runtime.CBCtxFactoryHostDefault

2. com.ibm.ejb.cb.runtime.CBCtxFactoryHostWidenedDefault
3. com.ibm.ejb.cb.runtime.CBCtxFactoryHostServerDefault
4. com.ibm.ejb.cb.runtime.CBCtxFactoryHostServerWidenedDefault
5. com.ibm.ejb.cb.runtime.CBCtxFactoryWorkGroupDefault
6. com.ibm.ejb.cb.runtime.CBCtxFactoryWorkGroupWidenedDefault
7. com.ibm.ejb.cb.runtime.CBCtxFactoryWorkGroupServerDefault
8. com.ibm.ejb.cb.runtime.CBCtxFactoryWorkGroupServerWidenedDefault
9. com.ibm.ejb.cb.runtime.CBCtxFactoryCellDefault
10. com.ibm.ejb.cb.runtime.CBCtxFactoryCellServerDefault
11. com.ibm.ejb.cb.runtime.CBCtxFactoryCellServerWidenedDefault

上記のコンテキストにより、以下の対応するファクトリー・ファインダーを使って EJB ホームの lookup オペレーションを解決します。

1. host/resources/factory-finders/host-scope
2. host/resources/factory-finders/host-scope-widened
3. host/resources/factory-finders/*server* -server-scope
4. host/resources/factory-finders/*server* -server-scope-widened
5. workgroup/resources/factory-finders/workgroup-scope
6. workgroup/resources/factory-finders/workgroup-scope-widened
7. workgroup/resources/factory-finders/*server* -server-scope
8. workgroup/resources/factory-finders/*server* -server-scope-widened
9. cell/resources/factory-finders/host-scope
10. cell/resources/factory-finders/*server* -server-scope
11. cell/resources/factory-finders/*server* -server-scope-widened

サーバー・ベースのコンテキスト・ファクトリーを使用できるのは、CBConnector サーバーとして稼働しているクライアントのみです。この場合の *server* は CBConnector サーバーの名前です。

デフォルトのコンテキスト・ファクトリーを使用できるのは、完全修飾された EJB ホームの lookup を発行するクライアント・アプリケーションのみです。クライアントがサブコンテキストを走査してから、部分修飾された EJB ホームの lookup を実行する場合、 **appbind** ツールを実行して、ホームのサブコンテキストをもつアプリケーション固有のコンテキストを作成し、アプリケー

ション固有の初期コンテキスト・ファクトリーを生成しなければなりません。詳細については、『アプリケーション固有のコンテキストおよび appbind ツール』を参照してください。

## アプリケーション固有のコンテキストおよび appbind ツール

CBConnector 提供のデフォルト・ファクトリー・ファインダーを使用して、EJB ホームを探し出す場合、 CBConnector は、アプリケーション・コンテキストとデフォルト・ファクトリー・ファインダーの間のデフォルト・マッピングを提供します (詳しくは、102ページの『デフォルトのコンテキストとファインダーの関連付け』を参照してください)。柔軟性が強化され、Enterprise Bean の配置機能により、オプションで EJB ホームのサブコンテキストを指定したアプリケーション固有のコンテキストを作成し、それを任意のファクトリー・ファインダーと関連付けることができます。ファクトリー・ファインダーの関連付けは、必要に応じて後で変更することができます。クライアントを実際のコンテキスト名から分離させるには、Enterprise Bean の配置機能により、**appbind** ツールを使用して、アプリケーション固有のコンテキストに対する初期コンテキスト・ファクトリーを生成します。

**appbind** ツールは、配置機能により、アプリケーション固有の命名コンテキストを作成し、それを選択したファクトリー・ファインダーと関連付けるようにします。その結果、そのファクトリー・ファインダーにより lookup オペレーションが解決されます。これらのアプリケーション固有のコンテキストは、EJB クライアントの初期 JNDI コンテキストとなるように設計されています。その結果、 EJB ホームにおける JNDI lookup 呼び出しは、関連付けられたファクトリー・ファインダーにより透過的に解決されます。 **appbind** ツールにより、ユーザーはこのようなアプリケーション固有のコンテキストを作成、変更および削除することができます。アプリケーションの EJB ホーム・インスタンスは、実際はアプリケーション固有のコンテキストの下にはバインドされていないことに注意してください。代わりに、ライフ・サイクル・リポジトリーにバインドされています。関連付けられたファクトリー・ファインダーにより、そのために定義されたライフ・サイクル規則を使用して、 EJB ホーム lookup を解決します。

**appbind** ツールを使用すれば、 EJB 仕様のバージョン 1.1 に合わせて作成された Enterprise Bean のネーミングの衝突を避けることもできます。このツールを使用して、JNDI 名は同じだが、配置するときの初期コンテキスト・ファクトリーがネーム・スペース内の別の場所にある Enterprise Bean について別個の JNDI ネーム・スペースを作成することができます。これにより、これらの bean 間でネーミングが衝突することがなくなります。

すべてのアプリケーション固有のコンテキストには、以下のコンテキスト名の語幹の 1 つがなければなりません。

- host/applications/initial-contexts
- workgroup/applications/initial-contexts
- cell/applications/initial-contexts

コンテキストの作成時に、ホスト、ワークグループ、またはセルのうちのどの有効範囲を指定するのかによって決まります。

デフォルトでは、ファクトリー・ファインダー host/resources/factory-finders/host-scope-widened は、 **appbind** ツールで作成されたアプリケーション固有のコンテキストと関連付けられます。ただし、別のファクトリー・ファインダーを指定しても構いません。そのファクトリー・ファインダーは、他のデフォルト・ファクトリー・ファインダーの 1 つで、System Management を使用した管理機能で作成したものや、作成したアプリケーション・プログラムで作成したものです。詳しくは、「*上級プログラミング・ガイド*」のライフ・サイクルの節を参照してください。

アプリケーション固有のコンテキストの下に、 EJB ホーム名のサブコンテキストをオプションで作成することができます。たとえば、ホームの名前が com/mycom/myapp/MyHome である場合、サブコンテキスト com/mycom/myapp を作成することができます。このようなサブコンテキストを作成することにより、クライアントに対する透過性がさらに高まります。これにより、クライアントは、アプリケーション固有のコンテキストから EJB ホーム名のリーフ・コンポーネント以外のものに対応する任意のサブコンテキストに至るまで、JNDI ネーム・スペースを走査することができます。アプリケーション固有のコンテキストに関連付けられたファクトリー・ファインダーを使用して、これらのサブコンテキストの EJB ホームの lookup オペレーションも解決します。**appbind** ツールにより、指定された EJB JAR ファイル内のデプロイメント・ディスクリプターのホーム名ごとにサブコンテキストを作成することができます。

**appbind** ツールにより、オプションで、作成されるアプリケーション固有のコンテキストの初期コンテキスト・ファクトリーに対して Java ソース・ファイルを作成することができます。この初期コンテキスト・ファクトリーは、クライアントで初期コンテキスト・ファクトリーとして使用することができます。**appbind** ツールにより、ユーザーは ORB 初期設定で使用するデフォルトのブートストラップ・ホストを上書きすることもできます。

以下の構文で **appbind** ツールを起動します。

```
appbind [-u] -name  contextName [-sc jarFileName] [-host | -workgroup | -cell]
[-factoryfinder factoryFinderPath]
[-genctxfactory  factoryClassName [-o  targetDir]]
[-boothost  bootstrapHostUrl]
```

バインドまたはアンバインドされるコンテキストは、必須の -name オプションで指定されます。ここで、*contextName* は、バインドまたはアンバインドする JNDI アプリケーション固有のコンテキスト名です。すべてのアプリケーション・コンテキスト名は、以下のコンテキスト名の語幹の 1 つと関係があります。

- host/applications/initial-contexts
- workgroup/applications/initial-contexts
- cell/applications/initial-contexts

ホスト、ワークグループ、またはセルのうちのどの有効範囲を指定したのかによって決まります。 (後述の -host、-workgroup、および -cell オプションを参照してください。)

-u オプションが指定されていない場合は、バインド・オペレーションが実行されます。これが指定されている場合は、アンバインド・オペレーションが実行されます。バインド・オペレーションが既存のコンテキストで実行される場合、現行のファクトリー・ファインダーの関連付けが追加されるか、あるいは置き換えられます。このコンテキストは、既にファクトリー・ファインダーが関連付けられているコンテキストの子にも親にもなれません。

その他のオプションを以下に示します。

- -u ー このフラグは、アンバインド・オペレーションを実行する場合に使用します。アンバインド・オペレーションは、 -name オプションおよび -sc オプション (指定されている場合) で指定されたコンテキストをアンバインドします。 -sc オプションが指定されている場合は、 JAR のデプロイメント・ディスクリプターの JNDI ホーム名に対応するサブコンテキストだけが除去されます。 -sc オプションが使用されていない場合は、 -name オプションで指定されたコンテキストおよびそのすべてのサブコンテキストがアンバインドされます。名前のツリーの管理可能な状態を保持するために、コンテキストまたはサブコンテキストがアンバインドされると、親コンテキストは、コンテキスト名の語幹 (既述の -name オプションを参照) に達するまで、あるいは非ヌルの親が検出されるまで繰り返しアンバインドされます。
- -sc ー このオプションは、サブコンテキストを指定する場合に使用します。ここで、*jarFileName* ファイルは、 EJB ホーム名をもつデプロイメント・ディスクリプターが入っている EJB JAR ファイルの名前です。リーフ名コンポーネント以外の EJB ホーム名は、それぞれサブコンテキスト名として処

理されます。たとえば、ホームの名前が com/mycom/myapp/MyHome である場合、サブコンテキスト名は com/mycom/myapp です。

バインド処理の場合、 -name フラグで指定されたアプリケーション固有のコンテキストの下にサブコンテキスト名が作成されます。アンバインド処理の場合、アンバインドされるコンテキストは JAR ファイルで識別されるサブコンテキスト名に限定されます。バインド処理またはアンバインド処理のいずれの場合も、その他のサブコンテキストには影響はありません。

- -host、-workgroup、-cell ― これらのフラグは、バインドまたはアンバインドされるアプリケーション・コンテキストの有効範囲を制御します。前述の -name フラグの節で説明しているように、有効範囲にはそれぞれ対応するコンテキスト名の語幹があります。コンテキストごとに、 -host、-workgroup、および -cell フラグは、ホスト、ワークグループ、またはセルの有効範囲をそれぞれ指定します。デフォルトの有効範囲は、ホストの有効範囲です。バインド・オペレーションまたはアンバインド・オペレーションごとに、有効範囲を 1 つだけ指定できます。
- -factoryfinder ― このオプションは、バインドされるアプリケーション固有のコンテキストと関連付けるファクトリー・ファインダーを指定する場合に使用します。ここで、*factoryFinderPath* はファクトリー・ファインダーの名前です。デフォルト・ファクトリー・ファインダーは、host/resources/factory-finders/host-scope-widened です。

  このオプションは、アンバインド・オペレーションには適用されません。
- -genctxfactory ― 通常、アプリケーション固有のコンテキストがバインドされるときに、アプリケーション固有のコンテキストに対する初期コンテキスト・ファクトリーがあるほうが望ましいです。このオプションは、 **appbind** ツールに初期コンテキスト・ファクトリーの Java ソース・ファイルを作成するように指示します。ここで、 *factoryClassName* は、コンテキスト・ファクトリーの完全修飾されたクラス名です。パッケージ接頭部サブディレクトリーはすべて必要に応じて作成されます。ソース・ファイルがすでに存在する場合は、置き換えられます。ファイルおよびそのファイルに入っているサブディレクトリーは、 -o オプションで指定されたディレクトリーまたは現行ディレクトリー (デフォルト) に対応して作成されます。

  このオプションは、アンバインド・オペレーションには適用されません。
- -o ― このオプションを使用して、初期コンテキスト・ファクトリー・ファイルのターゲット・ディレクトリーを指定します (-genctxfactory オプションを参照)。ここで *targetDir* はディレクトリー・パス (パッケージ接頭部ディレクトリーは含みません)。デフォルト・ターゲット・ディレクトリーは現行ディレクトリーです。

  このオプションは、アンバインド・オペレーションには適用されません。

-o オプションを使用する場合は、-genctxfactory フラグを使用する必要があります。

- -boothost ー このオプションを使用して、ORB 初期設定で使用されるデフォルトのホストとポートを上書きします。ここで、*bootstrapHostUrl* はブートストラップ・ホストの URL です。ブートストラップ・ホスト の URL の形式は以下のとおりです。

  ```
  iiop:// hostName [: portNumber]
  ```

## 既存の CICS または IMS アプリケーションからの Enterprise Bean の作成

**PAOToEJB** ツールを使用することによって、既存の CICS または IMS アプリケーションから Enterprise Bean を作成することができます。アプリケーションは、Enterprise Bean を作成する前に PAO にマップしなければなりません。 PAO の作成に関する詳細については、Component Broker の資料である「*手続き型アプリケーション・アダプター開発ガイド*」および VisualAge for Java エンタープライズ版の資料を参照してください。

**PAOToEJB** ツールは、この章で説明するその他のツールとは独立して実行されます。 PAO クラスから Enterprise Bean を作成するには、以下を行います。

1. PAO クラス・ファイルが存在するディレクトリーに変更する。
2. PAO クラス・ファイルのディレクトリー (クラスを含む JAR ファイル) を CLASSPATH 環境変数に追加する。
3. 以下の構文で **PAOToEJB** コマンドを起動する。

   ```
   PAOToEJB -name [ejbName] paoClass -hod |  -eci | -appc
   ```

   *ejbName* 引き数はオプションであり、Enterprise Bean の名前 (Account など) を指定します。この名前を指定しない場合は、Enterprise Bean には、PAO クラスの短縮名を使用して名前が付けられます。*paoClass* 引き数は必須であり、 .class 拡張子を除いた PAO クラスの完全修飾 Java 名を指定します。PAO クラスは、常に com.ibm.ivj.eab.paa.EntityProceduralAdapterObject のサブクラスです。以下のオプションのいずれかも指定しなければなりません。

   - -hod ー PAO クラスが Host On-Demand (HOD) 用であることを示します。 HOD は、ブラウザー・ベースの 3270 Telnet 接続です。
   - -eci ー PAO クラスが外部呼び出しインターフェース (ECI) 用であることを示します。ECI は、リモート・プロシージャー呼び出し (RPC) 式のインターフェースを CICS に提供する所有プロトコルです。

- -appc － PAO クラスが拡張プログラム間通信機能 (APPC) 用であることを示します。この通信機能は、LU 6.2 通信用のシステム・ネットワーク体系 (SNA) です。

**注:** HOD または ECI for CICS を使用し、CMP を持つエンティティー bean にアクセスする EJB クライアントまたは IMS アプリケーションは、これらのエンティティー bean のメソッドを呼び出す前にトランザクションを開始しなければなりません。これが必要な理由は、これらのタイプのエンティティー bean が TX_MANDATORY トランザクション属性を使用しなければならないためです。

4. *paoClass* が Java パッケージの一部である場合は、対応するディレクトリー構造を作成し、生成された Java ファイルをそのディレクトリーに移動しなければならない。
5. 新しく作成された Enterprise Bean の Java ソース・ファイルを以下のようにコンパイルする。

   ```
   javac ejbName*.java
   ```
6. Enterprise Bean のコンパイル済みクラス・コンポーネントを JAR または ZIP ファイルに置き、**jetace** ツールを使用して、bean に対する EJB JAR ファイルを作成する (39ページの『EJB モジュールの作成』で説明しています)。
7. **cbejb** ツールを使用して、EJB JAR ファイルを配置する (73ページの『手作業による Enterprise Bean の配置』で説明しています)。

## MQSeries と通信する Enterprise Bean の作成

Component Broker には、MQSeries メッセージを送受信する BO を開発するためのツールが含まれます。また、分散トランザクション内で MQSeries 待ち行列にアクセスすることができます。EJB サーバー (CB) は、この MQSeries サポート上に構築され、これによって、 MQSeries ベースの BO をラップする Enterprise Bean を生成することができます。

MQSeries EJB サポートによって、EJB クライアント・アプリケーションは、EJB クライアント・インターフェースを介して MQSeries と間接的に対話することができます。ここで説明している MQSeries BO 用の Component Broker サポートおよび EJB サポートは、両方とも、Object Builder によって生成された DO インプリメンテーションの変更を必要とします。サポートされているこれらの 2 つの方法の主な相違点は、Component Broker MQSeries ベースの BO を構築する場合には Object Builder を介して MQSeries メッセージの内容が指

定されるのに対し、EJB サポートでは MQSeries メッセージの内容を Java のプロパティー・ファイルで指定する必要があることです。

Component Broker での MQSeries サポートに関する詳細については、「*MQSeries アプリケーション・アダプター開発ガイド*」を参照してください。

**mqaaejb** ツールは、MQSeries Application Adaptor ベースの Component Broker BO をラップするセッション bean を生成します。結果として得られるセッション bean インプリメンテーションは、EJB サーバー (CB) に固有のものであり、他の EJB サーバーで使用することはできません。生成されたセッション bean を配置するには、**cbejb** ツールを使用します。 **mqaaejb** ツールは、他の EJB サーバー (CB) ツールとは独立して実行されます。

特定の MQSeries 待ち行列に対するセッション bean を作成するには、以下を行います。

1. 以下の項目を含む Java プロパティー・ファイルを作成します。
   - メッセージ・タイプ指定 – プロパティー名は messageType でなければならず、その値は Inbound、Outbound、または InOut のいずれかでなければなりません。 InOut を選択する場合は、単一の Enterprise Bean ではなく、Enterprise Bean の組を作成して、インバウンドおよびアウトバウンドのメッセージ待ち行列の組に対応させます。この指定の例を示します。

     ```
     messageType=Inbound
     ```
   - メッセージ・フィールド指定 – 各メッセージ・フィールドについて、プロパティー名はフィールド名であり、プロパティー値はフィールド型です。この指定の例を示します。

     ```
     bankName=java.lang.String
     ```

     ```
     accountNumber=int
     ```

   **注:** タイプ指定における Java クラス名は、完全修飾パッケージ名でなければなりません。
2. 以下の構文で **mqaaejb** コマンドを実行します。

   ```
   # mqaaejb -f propertiesFile -n baseBeanName [-p packageName]
     [-i existingInboundBOInterfaceName]
     [-o existingOutboundBOInterfaceName]
     [-c existingOutboundCopyName
   ```

   -f および -n オプションは必須です。 *propertiesFile* は、1 のステップ で作成したプロパティー・ファイルの名前を指定します。*baseBeanName* 引き数には、生成される 1 つまたは複数の Enterprise Bean のベース名を指定

します。たとえば、ベース名 が Account であり、プロパティー・ファイルで、それがインバウンドおよびアウトバウンド・メッセージの両方のメッセージ用であることを指定している場合、 **mqaaejb** コマンドは、セッション bean、関連インターフェース、および以下の名前の生成物を生成します。

AccountInboundBean

AccountEJBObject

AccountInboundEJBHome

AccountOutboundBean

AccountOutboundEJBObject

AccountOutboundEJBHome

AccountMsgTemplate

-p オプションは、Enterprise Bean のパッケージ名を指定します。このオプションを指定しない場合は、パッケージ名はデフォルトで `mytest.ejb.mqaa` になります。

-i オプションまたは -o オプションおよび -c オプションが指定されていない場合、 **mqaaejb** コマンドは、**cbejb** コマンドをマークして、後で **cbejb** コマンドが bean 上で実行されたときに、セッション bean に必要な補助メッセージ BO を生成します。 (「*MQSeries アプリケーション・アダプター開発ガイド*」で説明している手順に従って) MQSeries Application Adaptor ベースの BO の作成およびテストを既に完了している場合は、それらをセッション bean でラップするだけで十分です。**mqaaejb** コマンドに、これらの BO の名前および Copy オブジェクトを指定できます。**mqaaejb** コマンドは、次に、指定された BO を使用するセッション bean を作成します。これらのオブジェクト名は完全修飾された名前でなければなりません。たとえば、以下のようになります。

```
mqaaejb -f mymsg.properties -n Account -i TextMessage::TMInbound ¥
   -o TextMessage::TMOutbound -c TextMessageCopy::TMOutboundCopy
```

それでも、既存の BO とは別に基本 bean 名には -n オプションを指定しなければなりません。プロパティー・ファイルも指定しなければなりません。このファイルで指定されたメッセージ・フォーマットは、既存の BO と互換性がなければなりません。 IDL C++/Java 結合文書を参照して、 BO 内の C++ フィールド・タイプとプロパティー・ファイルの Java タイプの間に妥当なマッピングを設定することができます。

**mqaaejb** コマンドが正常に終了すると、作業ディレクトリーに以下の項目が生成されます。

- パッケージ名に対応するサブディレクトリーに Enterprise Bean を構成する Java ソース・ファイル (および対応するコンパイル済みクラス・ファイル)。
- Enterprise Bean を構成する Java ソース・ファイルおよびコンパイル済みファイルを含む JAR ファイル。
- Enterprise Bean のデプロイメント・ディスクリプターを含む XML ファイル。

3. 以下のように **jetace** ツールを実行して、Enterprise Bean に対する EJB JAR ファイルを生成します。

```
# jetace -f beanName.xml
```

4. **cbejb** ツールを実行して、EJB JAR ファイルに Enterprise Bean を配置します。詳細については、73ページの『手作業による Enterprise Bean の配置』を参照してください。 **cbejb** コマンドが完了すると、既存の BO を使用していない限り、「*MQSeries アプリケーション・アダプター開発ガイド*」のステップに従って、DO インプリメンテーションを変更する必要がある可能性があります。

## EJB サーバー (CB) 環境での制約事項

EJB サーバー (CB) 環境用の Enterprise Bean を開発する場合には、以下の制約事項が適用されます。

- Java ソース・ファイルだけでなくクラス・ファイルも入っている EJB JAR ファイルを配置する場合、または JAR ファイルに Java ソース・コードを組み込む際に JAR の依存関係がある場合には、配置は I/O 例外 ″Could not compile″ を発行して異常終了することがあります。これは、javac コンパイラーが、JAR ファイル内の .java ファイルに関して、古い .class ファイルを更新しようとしたことが原因です。これを避けるためには、ファイルを VisualAge for Java 内から JAR ファイルにインポートするときに「export .java files (.java ファイルをエクスポートする)」チェック・ボックスがチェックされていないことを確認するか、JAR ファイルを作成するときに .java ファイルを追加しないでください。
- Enterprise Bean では、非修飾インターフェースおよび例外名が重複していてはなりません。たとえば、com.ibm.ejs.doc.bank.Account という名前のパッケージで com.ibm.ejs.doc.account.Account インターフェースを再利用することはできません。この制約事項が必要な理由は、EJB サーバー (CB) ツールが非修飾名のみを使用する Enterprise Bean サポート・ファイルを生成するためです。

- エンティティー bean のコンテナー管理フィールドは、CORBA IDL ファイルで使用できる有効なものでなければなりません。特に、変数名には ISO Latin-1 文字セットを使用しなければなりません。下線文字 (_) で始めることはできません。ドル記号 ($) を含めることはできません。また、CORBA のキーワードは使用できません。大文字と小文字の違いを除いて同名の変数も許されていません。 (たとえば、変数 *accountId* と *AccountId* を同じクラスで使用することはできません。) CORBA IDL に関する詳細については、CORBA プログラミング説明書を参照してください。

  また、エンティティー bean 内のコンテナー管理フィールドは、有効な Java 型でなければなりませんが、型 ejb.javax.Handle にしたり、型 EJBObject または EJBHome の配列にしたりすることはできません。
- ユーザー定義インターフェースおよび例外クラスの名前では下線 (_) を使用しないようにしてください。
- リモート・インターフェースのメソッド名は、Component Broker 管理下オブジェクト Framework のメソッド名 (つまり、IManagedServer::IManagedObjectWithCachedDataObject、CosStream::Streamable、CosLifeCycle::LifeCycleObject、および CosObjectIdentity::IdentifiableObject インターフェースのメソッド) に一致していてはなりません。 Managed Object Framework に関する詳細については、Component Broker「プログラミング・ガイド」を参照してください。さらに、プロパティー名またはメソッド名の末尾に下線 (_) を使用してはなりません。この制約事項によって、コンテナー管理フィールドに対応する BO インターフェースの照会可能属性と名前が重複するのを避けることができます。
- javax.ejb.EJBContext インターフェースの getUserTransaction メソッド (これは、SessionContext インターフェースによって継承されたものです) は、型 javax.jts.UserTransaction ではなく型 javax.transaction.UserTransaction のオブジェクトを戻します。これは、1.0 バージョンの EJB 仕様からは逸脱していますが、1.1 バージョンの EJB 仕様では、getUserTransaction メソッドが型 javax.transaction.UserTransaction のオブジェクトを戻すことを要求していて、型 javax.jts.UserTransaction のオプションを戻す要件が除去されています。
- javax.ejb.SessionSynchronization インターフェースはサポートされていません。
- Java データベース・コネクティビティー (JDBC) を使用してデータベースにアクセスする BMP を持つエンティティー bean は、環境が XA 対応の JDBC をサポートしていないため、分散トランザクションで実行することができません。

- BMP エンティティー bean の 1 次キー・クラスの変数はパブリックでなければなりません。
- *実行識別* および*アクセス制御* デプロイメント・ディスクリプター属性は、使用されません。
- (javax.ejb.EJBObject インターフェースから) Enterprise Bean のリモート・インターフェースによって継承される remove メソッドは、Enterprise Bean の対応する ejbRemove() メソッドが javax.ejb.RemoveException 例外を throw する場合であっても、この例外を throw してはなりません。この制約事項が必要な理由は、remove メソッドと CORBA の CosLifeCycle::LifeCycleObject::remove メソッドの間で名前が重複するためです。後者は、すべての Component Broker 管理下オブジェクトによって継承されます。
- Enterprise Bean への単一スレッド・アクセスは、bean のトランザクション属性が TX_NOT_SUPPORTED または TX_BEAN_MANAGED のいずれかに設定されている場合にのみ強制されます。他の Enterprise Bean については、異なるトランザクションからのアクセスは逐次化されますが、同じトランザクションで実行されている異なるスレッドからのアクセスの逐次化は強制されません。 TX_NOT_SUPPORTED または TX_BEAN_MANAGED トランザクション属性を設定して配置された Enterprise Bean に対する不正なコールバックを行うと、java.rmi.RemoteException 例外が EJB クライアントに throw されます。
- セッション bean のタイムアウト属性はサポートされません。
- トランザクション属性は、bean 全体にのみ設定することができます。bean 内の個々のメソッドについてトランザクション属性を設定することはできません。
- 状態付きセッション bean のトランザクション属性値が TX_BEAN_MANAGED の場合は、トランザクションを開始するメソッドは、トランザクションの完了 (トランザクションのコミットまたはロールバック) も行わなければなりません。つまり、EJB サーバー (CB) 環境で使用する場合は、トランザクションが状態付きセッション bean の複数のメソッドにわたることはできません。
- TX_MANDATORY トランザクション属性値は、HOD または ECI を使用して CICS または IMS アプリケーションにアクセスする、コンテナー管理のパーシスタンス (CMP) を持つエンティティー bean で使用しなければなりません。結果として、これらのエンティティー bean にアクセスする EJB クライアントは、クライアント開始の 1 フェーズ・コミット・トランザクション (CB セッション・サービス) 内で実行しなければなりません。

- TX_NOT_SUPPORTED トランザクション属性値は、CMP を持つエンティティー bean ではサポートされていません。これは、これらの bean をトランザクション内でアクセスしなければならないためです。
- TX_REQUIRES_NEW トランザクション属性は、 EJB バージョン 1.0 形式の JAR ファイルではサポートされていません。 EJB 1.1 形式の JAR ファイルでは、TX_REQUIRES_NEW トランザクション属性は TX_REQUIRED と解釈されます。
- EJB 1.1 形式の JAR ファイルでは、 TX_NEVER トランザクション属性は TX_NOT_SUPPORTED と解釈されます。
- TX_SUPPORTS トランザクション属性は TX_MANDATORY と解釈されます。
- トランザクション分離レベル属性はサポートされません。
- com.ibm.ejb.cb.runtime.CBCtxFactory コンテキスト・ファクトリーを使用している場合、デフォルトの初期コンテキスト・ファクトリー (102ページの『デフォルトのコンテキストとファインダーの関連付け』を参照)、または **appbind** ツールで生成されたアプリケーション固有の初期コンテキスト・ファクトリー (104ページの『アプリケーション固有のコンテキストおよび appbind ツール』を参照)、 javax.naming.Context.list および javax.naming.Context.listBindings メソッドが javax.naming.NamingEnumeration オブジェクトに戻せる要素数は、いずれも最大 1000 個までになります。
- C++ CORBA ベースの EJB クライアントはサポートされていません。

# 第5章 Enterprise Bean の開発

本章では、最も一般的なタイプの Enterprise Bean を開発およびパッケージ化するために必要な基本的な作業について説明します。特に本章では、コンテナー管理のパーシスタンス (CMP) を使用する状態なしセッション bean およびエンティティー bean の作成について詳しく説明します。状態なしセッション bean の説明には、状態付き bean に関する重要な情報も記載されています。bean 管理のパーシスタンス (BMP) を使用するエンティティー bean の開発については、 201ページの『BMP を持つエンティティー bean の開発』を参照してください。

本章の説明はすべてを網羅しているわけではありませんが、基本的な Enterprise Bean の開発に必要な情報は記載されています。より複雑な Enterprise Bean を開発する場合の詳細については、Enterprise Bean の開発に関する市販の資料を参照してください。本章で説明する Enterprise Bean の例、およびそれを使用する Java アプリケーションとサーブレットの例は、307ページの『本書に記載する例の説明』に記載されています。

本章では、Enterprise Bean の主要な各コンポーネントを構築する際の要件について説明します。市販されている統合開発環境 (IDE) (IBM の VisualAge for Java など) のいずれも*使用しない* 場合は、これらの各コンポーネントを (Java 開発キット および WebSphere のツールを使用して) 手作業で構築しなければなりません。 Enterprise Bean を手作業で開発するのは、IDE で開発するよりも困難でエラーが多く発生します。したがって、より手軽な IDE を使用するようにしてください。

**注:** EJB サーバー (CB) 環境では、Enterprise Bean 内の非修飾インターフェースと例外名を重複させてはなりません。たとえば、com.ibm.ejs.doc.account.Account インターフェースを、com.ibm.ejs.doc.bank.Account という名前のパッケージで再利用することはできません。これは、EJB サーバー (CB) ツールが、非修飾名のみを使用して Enterprise Bean のサポート・ファイルを生成するためです。

© Copyright IBM Corp. 1999, 2001 

## CMP を持つエンティティー bean の開発

CMP を持つエンティティー bean では、コンテナーは、エンティティー bean とデータ・ソースの間の対話を操作します。 BMP を持つエンティティー bean では、エンティティー bean には、エンティティー bean とデータ・ソースとの対話に必要なコードがすべて入っていなければなりません。このため、CMP を持つエンティティー bean の開発は、BMP を持つエンティティー bean の開発よりも単純です。

本セクションでは、CMP を持つエンティティー bean の開発について検証します。 BMP を持つエンティティー bean にも本セクションの情報の多くを利用できますが、これら 2 つのタイプには大きな違いがあります。 BMP を持つエンティティー bean の開発に必要な作業については、 201ページの『BMP を持つエンティティー bean の開発』を参照してください。

すべてのエンティティー bean には、以下の基本的なパーツがなければなりません。

- Enterprise Bean クラス。詳細については、『Enterprise Bean クラスの作成 (CMP を持つエンティティー)』を参照してください。
- Enterprise Bean のホーム・インターフェース。詳細については、129ページの『ホーム・インターフェースの作成 (CMP を持つエンティティー)』を参照してください。
- Enterprise Bean のリモート・インターフェース。詳細については、133ページの『リモート・インターフェースの作成 (CMP を持つエンティティー)』を参照してください。
- Enterprise Bean の 1 次キー・クラス。詳細については、134ページの『1 次キー・クラスの作成 (CMP を持つエンティティー)』を参照してください。

### Enterprise Bean クラスの作成 (CMP を持つエンティティー)

CMP エンティティー bean において、bean クラスは、Enterprise Bean のビジネス・メソッドの定義とインプリメント、Enterprise Bean のインスタンスの作成に使用するメソッドの定義およびインプリメント、およびインスタンスのライフサイクルにおいて重要なイベントを Enterprise Bean のインスタンスに通知するためにコンテナーが使用するメソッドのインプリメントを行います。 Enterprise Bean クライアントが bean クラスに直接アクセスすることはありません。代わりに、ホーム・インターフェースおよびリモート・インターフェースをインプリメントするクラスを使用して、bean クラスに定義されているメソッドを間接的に呼び出します。

規則では、Enterprise Bean の名前は、*Name*Bean になります。ここで、*Name* は、ユーザーが Enterprise Bean に割り当てた名前です。たとえば、Account Enterprise Bean の場合は、Enterprise Bean クラスの名前は AccountBean になります。

CMP を持つすべてのエンティティー bean は、以下の要件を満たしていなければなりません。

- パブリックでなければならず、抽象クラスであってはならない。また、javax.ejb.EntityBean インターフェースをインプリメントしなければならない。詳細については、126ページの『EntityBean インターフェースのインプリメント』を参照してください。
- Enterprise Bean 関連の永続データに対応するインスタンス変数を定義しなければならない。詳細については、120ページの『変数の定義』を参照してください。
- Enterprise Bean 関連のデータのアクセスおよび操作に使用するビジネス・メソッドをインプリメントしなければならない。詳細については、122ページの『ビジネス・メソッドのインプリメント』を参照してください。
- Enterprise Bean をインスタンス化する方法ごとに ejbCreate メソッドを定義およびインプリメントしなければならない。 ejbCreate メソッドごとに、対応する ejbPostCreate メソッドを定義しなければならない。詳細については、124ページの『ejbCreate メソッドおよび ejbPostCreate メソッドのインプリメント』を参照してください。

**注:**

Enterprise Bean クラスは Enterprise Bean のリモート・インターフェースをインプリメントすることができますが、これはお勧めしません。 Enterprise Bean クラスがリモート・インターフェースをインプリメントすると、この 変数をメソッドの引き数として誤って渡してしまう可能性があります。

インターフェース内のメソッドのシグニチャーが異なっていてもメソッドの名前が同じである場合には、 Java-IDL マッピングの仕様のために、Enterprise Bean クラスは、2 つの異なるインターフェースを実装することができません。 Enterprise Bean を配置するときにエラーが発生することがあります。

図18 に、Account Enterprise Bean の場合の Enterprise Bean の主要な部分を示します。 (強調したいコードは太字になっています。) 以降のセクションでは、これらの部分について詳細に説明します。

```
...
import java.util.Properties;
import javax.ejb.*;
import java.lang.*;
public class AccountBean implements EntityBean {
     // Set instance variables here
     ...
     // Implement methods here
     ...
}
```

*図 18. コード例: AccountBean クラス*

## 変数の定義

エンティティー bean クラスには、永続インスタンス変数と非永続インスタンス変数を両方とも含めることができます。ただし、静的変数は、最終値 (つまり定数) でない場合には、Enterprise Bean でサポートされません。静的変数がサポートされない理由として、これらの変数が Enterprise Bean のインスタンスにわたって必ずしも一定ではないという点が挙げられます。

コンテナー管理フィールドは永続変数であり、データベースに保管されます。コンテナー管理フィールドはパブリックでなければなりません。

非永続変数は一時的な変数であり、データベースに保管されません。非永続変数の使用には注意が必要です。また、メソッドの呼び出し間の EJB クライアントの状態を保守するためにこの変数を使用してはなりません。これは、トランザクション外でのメソッド呼び出しの間に非永続変数が同じ状態を維持することが保証されないためです。つまり、その他の EJB クライアントがこれらの変数を変更できるため、エンティティー bean が非活動化されたときに変数が失われることがあります。

**注:** EJB サーバー (CB) 環境では、エンティティー bean のコンテナー管理フィールドは、CORBA IDL ファイルで使用できる有効なものでなければなりません。特に、変数名には ISO Latin-1 文字セットを使用しなければなりません。下線文字 (_) で始めることはできません。ドル記号 ($) を含めることはできません。また、CORBA のキーワードは使用できません。変数名は、大文字と小文字は区別されません。(たとえば、同じクラス内で変

数 *accountId* と *AccountId* を使用することはできません。)CORBA IDL に関する詳細については、CORBA のプログラミング説明書を参照してください。

また、エンティティー bean 内のコンテナー管理フィールドは、有効な Java 型でなければなりませんが、型 javax.ejb.Handle にしたり、型 EJBObject または EJBHome の配列にしたりすることはできません。

AccountBean クラスには、以下の 3 つのコンテナー管理フィールドが含まれています (図19 を参照)。

- *accountId*。口座に関連する口座 ID を識別する。
- *type*。口座の種類 (普通預金 (1) または当座預金 (2)) を識別する。
- *balance*。その口座の現在の残高を識別する。

```
...
public class AccountBean implements EntityBean {
     private EntityContext entityContext = null;
     private ListResourceBundle bundle =
          ResourceBundle.getBundle(
                    "com.ibm.ejs.doc.account.AccountResourceBundle");
     public long accountId = 0;
     public int type = 1;
     public float balance = 0.0f;
     ...
}
```

*図 19. コード例: AccountBean クラスの変数*

デプロイメント・ディスクリプターは、CMP を持つエンティティー bean でコンテナー管理フィールドを識別するために使用します。 CMP を持つエンティティー bean では、コンテナー管理フィールドを、それぞれ ejbCreate メソッドによって初期化しなければなりません (124ページの『ejbCreate メソッドおよび ejbPostCreate メソッドのインプリメント』を参照)。

コンテナー管理フィールドのサブセットは、Enterprise Bean の各インスタンスに関連する 1 次キー・クラスを定義するために使用します。 134ページの『1 次キー・クラスの作成 (CMP を持つエンティティー)』に示すように、*accountId* 変数は、Account Enterprise Bean の 1 次キーを定義します。

AccountBean クラスには、以下の 2 つの非持続性変数が含まれています。

- *entityContext*。Account Enterprise Bean の各インスタンスのエンティティー・コンテキストを識別する。エンティティー・コンテキストを使用して、bean インスタンスに現在関連付けられている EJB オブジェクトへの参照を取得

したり、その EJB オブジェクトに関連付けられている 1 次キー・オブジェクトを取得したりすることができます。

- *bundle*。Account bean によって使用されるロケール固有のオブジェクトを含む、リソース・バンドル・クラス (com.ibm.ejs.doc.account.AccountResourceBundle) をカプセル化する。

**ビジネス・メソッドのインプリメント**

エンティティー bean クラスのビジネス・メソッドは、クラス内にカプセル化されたデータを操作する方法を定義します。 Enterprise Bean クラスにインプリメントされたビジネス・メソッドは、EJB クライアントが直接呼び出すことはできません。代わりに、EJB クライアントは、Enterprise Bean のインスタンスに関連する EJB オブジェクトを使用して、Enterprise Bean のリモート・インターフェースに定義されている対応するメソッドを呼び出します。これにより、Enterprise Bean のインスタンスにある対応するメソッドがコンテナーによって呼び出されます。

したがって、Enterprise Bean クラスにインプリメントされているビジネス・メソッドごとに、対応するビジネス・メソッドを、Enterprise Bean のリモート・インターフェースに定義しなければなりません。 Enterprise Bean のリモート・インターフェースは、Enterprise Bean の配置時にコンテナーによって EJB オブジェクト・クラスにインプリメントされます。

123ページの図20 に、AccountBean クラスのビジネス・メソッドを示します。これらのメソッドを使用して、指定の金額を口座残高に追加して新しい残高を戻したり (add)、口座の現時点の残高を戻したり (getBalance)、口座の残高を設定したり (setBalance)、指定の金額を口座残高から引き出して新しい残高を戻したり (subtract) します。

subtract メソッドは、口座の残高以上の金額をクライアントが口座から引き出そうとした場合に、ユーザー定義の例外 com.ibm.ejs.doc.account.InsufficientFundsException を throw します。 134ページの図25 に示すように、Account bean のリモート・インターフェースにある subtract メソッドもこの例外を throw することになっています。 Enterprise Bean のユーザー定義の例外クラスは、任意の他のユーザー定義の例外クラスと同様に作成します。例外 InsufficientFundsException のメッセージの内容は、*bundle* オブジェクトで getMessage メソッドを呼び出すことによって、AccountResourceBundle クラス・ファイルから取得します。

**注:** Enterprise Bean のコンテナーが Enterprise Bean のビジネス・メソッドからシステム例外を受け取り、そのメソッドがコンテナー管理トランザクション内で実行されている場合、コンテナーは、クライアントに例外を渡す

前にトランザクションをロールバックします。ただし、ビジネス・メソッドがアプリケーション例外を throw している場合は、アプリケーションが setRollbackOnly 関数を呼び出していない限り、トランザクションはロールバックされません (コミットされます)。 setRollbackOnly 関数を呼び出している場合は、トランザクションがロールバックされてから、例外が再び throw されます。

**注:** EJB サーバー (CB) 環境では、ユーザー定義のインターフェースや例外クラスの名前に下線 (_) を使用することはできません。

```
...
public class AccountBean implements EntityBean {
     ...
     public long accountId = 0;
     public int type = 1;
     public float balance = 0.0f;
     ...
     public float add(float amount) {
          balance += amount;
          return balance;
     }
     ...
     public float getBalance() {
          return balance;
     }
     ...
     public void setBalance(float amount) {
          balance = amount;
     }
     ...
     public float subtract(float amount) throws InsufficientFundsException {
          if(balance < amount) {
               throw new InsufficientFundsException(
                    bundle.getMessage("insufficientFunds"));
          }
          balance -= amount;
          return balance;
     }
     ...
}
```

*図 20. コード例: AccountBean クラスのビジネス・メソッド*

## エンティティー bean の標準アプリケーション例外

EJB 仕様のバージョン 1.1 では、Enterprise Bean で使用する標準のアプリケーション例外がいくつか定義されています。これらの例外はすべて、javax.ejb.EJBException クラスのサブクラスです。コンテナー管理パーシスタン

スと bean 管理パーシスタンスの両方を持つエンティティー bean については、 EJB 仕様では次のアプリケーション例外が定義されています。

- javax.ejb.CreateException
- javax.ejb.DuplicateKeyException
- javax.ejb.RemoveException
- javax.ejb.FinderException
- javax.ejb.ObjectNotFoundException

アプリケーション・プログラマーは、汎用の EJBException クラスや、提供されているサブクラス例外の 1 つを使用することも、また、この例外ファミリーをサブクラス化して独自の例外を定義することもできます。これらの例外はすべて、javax.ejb.RuntimeException クラスから継承したものなので、 throws 文節で明示的に宣言する必要はありません。

各例外の詳細については、次のセクションを参照してください。

- CreateException および DuplicateKeyException (CreateException クラスのサブクラス) については、『ejbCreate メソッドおよび ejbPostCreate メソッドのインプリメント』を参照してください。
- javax.ejb.RemoveException については、126ページの『EntityBean インターフェースのインプリメント』を参照してください。
- FinderException および ObjectNotFoundException (FinderException クラスのサブクラス) については、 131ページの『finder メソッドの定義』を参照してください。

**注:** EJB 仕様のバージョン 1.0 では、 java.rmi.RemoteException クラスを使用してアプリケーション固有の例外を捕そくしていました。 EJBException クラスとそのサブクラスは、バージョン 1.1 から新たに使用し始めたものです。したがって、RemoteException クラスは使用せずに、より正確な例外クラスを使用してください。 RemoteException クラスを使用している古いアプリケーションは今までどおり実行できますが、バージョン 1.1 準拠の Enterprise Bean は新しい例外クラスを使用しなければなりません。

### ejbCreate メソッドおよび ejbPostCreate メソッドのインプリメント

ユーザーは、Enterprise Bean の新しいインスタンスの作成方法ごとに、ejbCreate メソッドを定義してインプリメントしなければなりません。また、ejbCreate メソッドごとに、対応する ejbPostCreate メソッドも定義しなければなりません。 ejbCreate メソッドおよび ejbPostCreate メソッドは、それぞれホーム・インターフェースの create メソッドに相当します。

bean クラスのビジネス・メソッドと同様、ejbCreate メソッドおよび ejbPostCreate メソッドは、クライアントが直接呼び出すことはできません。代わりに、クライアントは、EJB ホーム・オブジェクトを使用して Enterprise Bean のホーム・インターフェースの create メソッドを呼び出します。これにより、ejbCreate メソッドおよび ejbPostCreate メソッドが順にコンテナーによって呼び出されます。 ejbCreate メソッドおよび ejbPostCreate メソッドの実行に成功した場合は、EJB オブジェクトが作成され、そのオブジェクトに関連する永続データがデータ・ソースに挿入されます。

CMP を持つエンティティー bean の場合は、コンテナーは、ejbCreate メソッドを呼び出してから ejbPostCreate メソッドを呼び出すまでの間にエンティティー bean とデータ・ソースとの間に必要な対話を操作します。 BMP を持つエンティティー bean の場合は、この対話を直接的に操作するためのコードが ejbCreate メソッドになければなりません。BMP を持つエンティティー bean に関する詳細については、201ページの『BMP を持つエンティティー bean の開発』を参照してください。

CMP を持つエンティティー bean にあるすべての ejbCreate メソッドは、以下の要件を満たしていなければなりません。

- パブリックであり、1 次キーと同じ型を戻さなければならない。実際の戻り値はヌルでなければなりません。
- その引き数が、Java リモート・メソッド呼び出し (RMI) に有効でなければならない。詳細については、155ページの『java.io.Serializable インターフェースおよび java.rmi.Remote インターフェース』を参照してください。
- Enterprise Bean インスタンスのコンテナー管理フィールドを初期化しなければならない。コンテナーは、これらの変数の値を抽出して、ejbCreate メソッドが戻った後にデータ・ソースに書き込みます。

ejbPostCreate メソッドはすべてパブリックで、戻り値なし (void) でなければなりません。また、対応する ejbCreate メソッドと同じ引き数を持っていなければなりません。

必要に応じて、 ejbCreate メソッドと ejbPostCreate メソッドのどちらでも、javax.ejb.EJBException 例外、または作成関連のサブクラス、CreateException 例外、または DuplicateKeyException 例外のうちの 1 つを throw することができます。 DuplicateKeyException クラスは CreateException クラスのサブクラスです。 java.rmi.RemoteException 例外の throw は行わないでください。詳細については、123ページの『エンティティー bean の標準アプリケーション例外』を参照してください。

図21 に、AccountBean クラスの例に必要な ejbCreate メソッドと ejbPostCreate メソッドを、2 セットずつ示します。 ejbCreate メソッドおよび ejbPostCreate メソッドの最初の各セットはラッパーであり、メソッドの次のセットを呼び出して、*type* 変数を 1 (普通預金口座を示す) に、*balance* 変数を 0 (0 ドル) に設定します。

```
...
public class AccountBean implements EntityBean {
     ...
     public long accountId = 0;
     public int type = 1;
     public float balance = 0.0f;
     ...
     public Integer ejbCreate(AccountKey key) {
          ejbCreate(key, 1, 0.0f);
     }
     ...
     public Integer ejbCreate(AccountKey key, int type, float initialBalance)
          throws EJBException {
          accountId = key.accountId;
          type = type;
          balance = initialBalance;
     }
     ...
     public void ejbPostCreate(AccountKey key)
          throws EJBException {
          ejbPostCreate(key, 1, 0);
     }
     ...
     public void ejbPostCreate(AccountKey key, int type, float initialBalance) { }
     ...
}
```

*図 21. コード例: AccountBean クラスの ejbCreate メソッドおよび ejbPostCreate メソッド*

## EntityBean インターフェースのインプリメント

各エンティティー bean クラスは、javax.ejb.EntityBean インターフェースから継承されたメソッドをインプリメントしなければなりません。コンテナーは、これらのメソッドを呼び出して、インスタンスのライフ・サイクルにおける重要なイベントを bean インスタンスに通知します (詳細については、 31ページの『エンティティー bean のライフ・サイクル』を参照してください)。これらのメソッドは、すべてパブリックで戻り値なし (void) でなければなりません。 javax.ejb.EJBException 例外、または ejbRemove メソッドの場合は javax.ejb.RemoveException 例外を throw することができます。

java.rmi.RemoteException 例外の throw は行わないでください。詳細については、123ページの『エンティティー bean の標準アプリケーション例外』を参照してください。

- ejbActivate。このメソッドは、コンテナーがインスタンス・プールからエンティティー bean を選択して特定の既存の EJB オブジェクトにそのインスタンスを割り当てる場合に、コンテナーによって呼び出されます。このメソッドは、Enterprise Bean インスタンスを活動化する場合に実行させるコードをすべて含んでいなければなりません。
- ejbLoad。このメソッドは、エンティティー bean のコンテナー管理フィールドを、データ・ソース内の対応するデータと同期させるためにコンテナーによって呼び出されます。 (つまり、データ・ソース内のフィールドの値が、対応する Enterprise Bean インスタンスのコンテナー管理フィールドにロードされます。) このメソッドは、Enterprise Bean インスタンスをデータ・ソース内の関連するデータと同期させる場合に実行するコードをすべて含んでいなければなりません。
- ejbPassivate。このメソッドは、コンテナーがエンティティー bean インスタンスの EJB オブジェクトとの関連付けを解除して、インスタンス・プールに Enterprise Bean インスタンスを入れる場合に、コンテナーによって呼び出されます。このメソッドは、Enterprise Bean インスタンスを「非活動化」する場合に実行させるコードをすべて含んでいなければなりません。
- ejbRemove。このメソッドは、Enterprise Bean のホーム・インターフェースによって javax.ejb.EJBHome インターフェースから継承された remove メソッドをクライアントが呼び出す場合に、コンテナーによって呼び出されます。このメソッドは、Enterprise Bean インスタンスをコンテナーから除去 (および関連データをデータ・ソースから除去) する前に実行させるすべてのコードを含んでいなければなりません。このメソッドは、Enterprise Bean インスタンスを除去できない場合に javax.ejb.RemoveException 例外を throw することができます。
- setEntityContext。このメソッドは、javax.ejb.EntityContext インターフェースへの参照を Enterprise Bean インスタンスに渡すためにコンテナーによって呼び出されます。 Enterprise Bean インスタンスがそのライフ・サイクルの任意の時点でこのコンテキストを使用する必要がある場合は、この値を保管するための変数が Enterprise Bean クラスに入っていなければなりません。このメソッドは、コンテキストへの参照を保管するために必要なすべてのコードを含んでいなければなりません。
- ejbStore。このメソッドは、コンテナーがデータ・ソース内のデータを Enterprise Bean インスタンスのコンテナー管理フィールドの値に同期させる必要がある場合に、コンテナーによって呼び出されます。 (つまり、

Enterprise Bean インスタンスの変数値がデータ・ソースにコピーされ、直前の値が上書きされます。) このメソッドは、データ・ソース内のデータを Enterprise Bean インスタンス内の対応する値で上書きする場合に実行させるすべてのコードを含んでいなければなりません。

- unsetEntityContext。このメソッドは、Enterprise Bean インスタンスの除去前に Enterprise Bean インスタンスに関連するすべてのリソースを解放するために、コンテナーによって呼び出されます。これは、Enterprise Bean インスタンスを除去する前に呼び出される最後のメソッドです。

CMP を持つエンティティー bean では、コンテナーは、これらのメソッドに必要なデータ・ソースとの対話を操作します。 BMP を持つエンティティー bean では、これらのメソッドが、必要なデータ・ソースとの対話を直接操作します。 BMP を持つエンティティー bean に関する詳細については、 201ページの『第9章 Enterprise Bean に関する高度なプログラミングの概念』を参照してください。

これらのメソッドは、以下のように使用することができます。

- 監査コードまたはデバッグ・コードを含めることができる。
- bean インスタンスが使用する追加のリソース (メインフレームへの SNA 接続など) を割り振ったり割り振り解除したりするためのコードを含めることができる。

129ページの図22 に示すように、AccountBean クラスでは、setEntityContext メソッドおよび unsetEntityContext メソッドを除くすべての上記のメソッドが空になっています。これは、これらのメソッドに関連する特定のライフ・サイクルにおける状態に対して、bean が必要とする追加のアクションがないためです。 setEntityContext メソッドおよび unsetEntityContext メソッドは、*entityContext* 変数の値を設定するために慣習的に使用されます。

```
...
public class AccountBean implements EntityBean {
     private EntityContext entityContext = null;
     ...
     public void ejbActivate() throws EJBException { }
     ...
     public void ejbLoad () throws EJBException { }
     ...
     public void ejbPassivate() throws EJBException { }
     ...
     public void ejbRemove() throws EJBException { }
     ...
     public void ejbStore () throws EJBException { }
     ...
     public void setEntityContext(EntityContext ctx) throws EJBException {
          entityContext = ctx;
     }
     ...
     public void unsetEntityContext() throws EJBException {
          entityContext = null;
     }
}
```

*図 22. コード例: AccountBean クラスでの EntityBean インターフェースのインプリメント*

## ホーム・インターフェースの作成 (CMP を持つエンティティー)

エンティティー bean のホーム・インターフェースは、bean の新しいインスタンスの作成、既存のインスタンスの検出および除去、およびインスタンスに関するメタデータの取得を行うためにクライアントが使用するメソッドを定義します。ホーム・インターフェースは、Enterprise Bean 開発者によって定義され、Enterprise Bean の配置時にコンテナーによって作成済みの EJB ホーム・クラスにインプリメントされます。

コンテナーによって、ホーム・インターフェースは、JNDI (Java Naming and Directory Interface) を介して Enterprise Bean にアクセスできるようになります。 JNDI は特定のネーム解決およびディレクトリー・サービスとは独立しており、Java ベースのアプリケーションを標準的な方法で任意のネーム解決およびディレクトリー・サービスにアクセスさせることができます。

規則では、ホーム・インターフェースの名前は *Name* Home になります。ここで、*Name* は、ユーザーが Enterprise Bean に割り当てた名前です。たとえば、Account Enterprise Bean のホーム・インターフェースの名前は、AccountHome になります。

すべてのホーム・インターフェースは、以下の要件を満たしていなければなりません。

- javax.ejb.EJBHome インターフェースを拡張しなければならない。ホーム・インターフェースは、javax.ejb.EJBHome インターフェースからいくつかのメソッドを継承します。これらのメソッドに関する詳細については、154ページの『javax.ejb.EJBHome インターフェース』を参照してください。
- インターフェースの各メソッドが、EJB オブジェクト・クラスの ejbCreate メソッドと ejbPostCreate メソッドのセットに相当する create メソッドか、あるいは finder メソッドでなければならない。詳細については、131ページの『create メソッドの定義』および 131ページの『finder メソッドの定義』を参照してください。
- ホーム・インターフェースに定義された各メソッドのパラメーターおよび戻り値が、Java RMI に有効でなければならない。詳細については、155ページの『java.io.Serializable インターフェースおよび java.rmi.Remote インターフェース』を参照してください。さらに、各メソッドの throws 文節に、java.rmi.RemoteException 例外クラスが含まれていなければならない。

131ページの図23 に、Account bean の例における、ホーム・インターフェース (AccountHome) の定義に関連する部分を示します。このインターフェースは、2 つの抽象 create メソッドを定義しています。最初のメソッドは、関連する AccountKey オブジェクトを使用して Account オブジェクトを作成します。2 番目のメソッドは、関連する AccountKey オブジェクトを使用し、口座タイプと初期残高を指定して、Account オブジェクトを作成します。このインターフェースは、必要な findByPrimaryKey メソッドおよび findLargeAccounts メソッドを定義し、残高が指定金額以上の口座の集合を戻します。

```
...
import java.rmi.*;
import java.util.*;
import javax.ejb.*;
public interface  AccountHome extends EJBHome {
     ...
     Account create (AccountKey id) throws CreateException, RemoteException;
     ...
     Account create(AccountKey id, int type, float initialBalance)
          throws CreateException, RemoteException;
     ...
     Account findByPrimaryKey (AccountKey id)
          RemoteException, FinderException;
     ...
Enumeration findLargeAccounts(float amount)
           throws RemoteException, FinderException;
}
```

*図 23. コード例: AccountHome ホーム・インターフェース*

### create メソッドの定義

create メソッドは、クライアントが Enterprise Bean インスタンスを作成して、そのインスタンスの関連データをデータ・ソースに挿入するために使用します。すべての create メソッドには create という名前が付いていなければなりません。また、Enterprise Bean クラスの対応する ejbCreate メソッドと同じ数と型の引き数を持っていなければなりません。(ejbCreate メソッド自体に、対応する ejbPostCreate メソッドがなければなりません。)

すべての create メソッドは、以下の要件を満たしていなければなりません。

- create という名前が付いていなければならない。
- Enterprise Bean のリモート・インターフェースの型を戻されなければならない。たとえば、AccountHome インターフェースの create メソッドの場合に戻される型は、Account です (図23 を参照)。
- java.rmi.RemoteException 例外、javax.ejb.CreateException 例外、および対応する ejbCreate メソッドと ejbPostCreate メソッドの throws 文節で定義されているすべてのアプリケーション例外を含む、throws 文節を持っていなければならない。

### finder メソッドの定義

finder メソッドは、1 つ以上の既存のエンティティー EJB オブジェクトを検出するために使用します。すべての finder メソッドの名前は find*Name* でなければなりません。ここで、*Name* には、finder メソッドの詳細な目的を記述します。

少なくとも、各ホーム・インターフェースには、findByPrimaryKey メソッドを定義しなければなりません。このメソッドによって、クライアントは、1 次キーだけを使用して EJB オブジェクトを見付けることができます。findByPrimaryKey メソッドは唯一の引き数として bean の 1 次キー・クラスを持っており、bean のリモート・インターフェースの型を戻します。

その他のすべての finder メソッドは、以下の要件を満たしていなければなりません。

- Enterprise Bean のリモート・インターフェースの型、 java.util.Enumeration インターフェースの型、または java.util.Collection インターフェースの型 (finder メソッドが複数の EJB オブジェクトまたは 1 つの EJB コレクションを戻すことができる場合) を戻さなければならない。
- java.rmi.RemoteException および javax.ejb.FinderException 例外クラスを含む throws 文節 を持っていなければならない。

すべてのエンティティー bean にはデフォルトの finder メソッドがなければなりませんが、必要であればさらに finder メソッドを作成することができます。たとえば、図24 に示すように、Account bean のホーム・インターフェースは、残高が指定金額以上の口座をカプセル化するオブジェクトを検索するための findLargeAccounts メソッドを定義しています。この finder メソッドは複数の EJB オブジェクトを戻すと予想されるため、戻される型は Enumeration になります。

```
Enumeration findLargeAccounts(float amount)
          throws RemoteException, FinderException;
```

*図 24. コード例: findLargeAccounts メソッド*

EJB サーバーは、findByPrimaryKey メソッドをインプリメントすることができます。 Enterprise Bean の配置時には、コンテナーによって、適切な Enterprise Bean インスタンスをデータベースで検索するために必要なコードが生成されます。

ただし、ユーザーがホーム・インターフェースで定義した追加の finder メソッドについては、Enterprise Bean 配置機能によって、関連するファインダー・ロジックがその finder メソッドに関連付けならなければなりません。このロジックは、配置時に、finder メソッドのインプリメントに必要なコードを生成するために、EJB サーバーによって使用されます。

EJB 仕様ではファインダー・ロジックの形式が定義されていないため、使用している EJB サーバーに合わせて形式を変更することができます。ファインダ

ー・ロジックの作成に関する詳細については、39ページの『EJB サーバー (AE) でのファインダー・ロジックの作成』または 48ページの『EJB サーバー (CB) でのファインダー・ロジックの作成』を参照してください。

## リモート・インターフェースの作成 (CMP を持つエンティティー)

エンティティー bean のリモート・インターフェースは、bean クラスで使用可能なビジネス・メソッドへのアクセスを提供します。また、bean インスタンスに関連する EJB オブジェクトを除去したり、bean インスタンスのホーム・インターフェース、オブジェクト・ハンドル、および 1 次キーを取得するメソッドも提供します。リモート・インターフェースは、Enterprise Bean 開発者によって定義され、Enterprise Bean の配置時にコンテナーによって作成済みの EJB リモート・クラスにインプリメントされます。

規則では、リモート・インターフェースの名前は *Name* になります。ここで、*Name* は、ユーザーが Enterprise Bean に割り当てた名前です。たとえば、Account Enterprise Bean のリモート・インターフェースの名前は、Account になります。

すべてのリモート・インターフェースは、以下の要件を満たしていなければなりません。

- javax.ejb.EJBObject インターフェースを拡張しなければならない。 Enterprise Bean のリモート・インターフェースは、javax.ejb.EJBObject インターフェースからいくつかのメソッドを継承します。これらのメソッドに関する詳細については、154ページの『javax.ejb.EJBObject から継承されたメソッド』を参照してください。
- Enterprise Bean クラスにインプリメントされているビジネス・メソッドごとに、対応するビジネス・メソッドを定義しなければならない。
- このインターフェースに定義された各メソッドのパラメーターおよび戻り値が、Java RMI に有効でなければならない。詳細については、155ページの『java.io.Serializable インターフェースおよび java.rmi.Remote インターフェース』を参照してください。
- 各メソッドの throws 文節に、java.rmi.RemoteException 例外クラスが含まれていなければならない。

**注:** EJB サーバー (CB) 環境では、Component Broker Managed Object Framework 内のメソッド名 (つまり、IManagedServer::IManagedObjectWithCachedDataObject、CosStream::Streamable、CosLifeCycle::LifeCycleObject、および CosObjectIdentity::IdentifiableObject インターフェースのメソッド)

と一致するメソッド名をリモート・インターフェースで使用してはなりません。 Managed Object Framework に関する詳細については、Component Broker「プログラミング・ガイド」を参照してください。また、プロパティー名およびメソッド名の末尾に下線 (_) を使用してはなりません。これは、コンテナー管理フィールドに対応するビジネス・オブジェクト・インターフェースの照会可能な属性と、名前が重複するのを避けるためです。

図25 に、Account Enterprise Bean の例における、リモート・インターフェース (Account) の定義に関連する部分を示します。このインターフェースは、AccountBean クラスにインプリメントされているビジネス・メソッドと正確に一致する口座残高を表示および操作するための 4 つのメソッドを定義します。

リモート・インターフェースのビジネス・メソッドはすべて、java.rmi.RemoteException 例外クラスを throw します。 subtract メソッドも、ユーザー定義の例外 com.ibm.ejs.doc.account.InsufficientFundsException を throw します。これは、bean クラス内の対応するメソッドがこの例外を throw するためです。さらに、このメソッドを呼び出すすべてのクライアントは、例外を処理するか、または throw によって例外を渡すかのどちらかを行わなければなりません。

```
...
import java.rmi.*;
import javax.ejb.*;
public interface Account extends EJBObject
{
     ...
     float add(float amount) throws RemoteException;
     ...
     float getBalance() throws RemoteException;
     ...
     void setBalance(float amount) throws RemoteException;
     ...
     float subtract(float amount) throws InsufficientFundsException,
          RemoteException;
}
```

*図 25. コード例: Account リモート・インターフェース*

## 1 次キー・クラスの作成 (CMP を持つエンティティー)

コンテナー内の各エンティティー EJB オブジェクトは、それぞれ固有の ID を持っています。この ID は、オブジェクトのホーム・インターフェースの名

前とその 1 次キーとの組み合わせを使用して定義されます。1 次キーは、オブジェクトの作成時にオブジェクトに割り当てられます。 2 つの EJB オブジェクトが同じ ID を持つ場合は、同一であると見なされます。

1 次キーを指定する方法は 2 通りあります。

- 単純 1 次キー。エンティティー bean クラスの単一のフィールドにマップし、プリミティブな Java データ型 (integer 型や long 型など) から成り立ちます。単純 1 次キーは、デプロイメント・ディスクリプターで指定します。
- 複合 1 次キー。エンティティー bean クラスの複数のフィールド (またはプリミティブな Java データ型から作成されたデータ構造) にマップします。*1 次キー・クラス* でカプセル化しなければなりません。複雑な Enterprise Bean は、1 次キーを表すインスタンス変数を複数持つ複合 1 次キーを持つ傾向にあります。

1 次キー・クラスを使用して、EJB オブジェクトの 1 次キーを管理します。規則では、1 次キー・クラスの名前は *Name* Key になります。ここで、*Name* は、Enterprise Bean の名前です。たとえば、Account Enterprise Bean の 1 次キー・クラスの名前は AccountKey になります。

1 次キー・クラスは、以下の要件を満たしていなければなりません。

- パブリックであり、逐次化が可能でなければならない。詳細については、155ページの『java.io.Serializable インターフェースおよび java.rmi.Remote インターフェース』を参照してください。
- そのインスタンス変数がパブリックであり、変数名が、Enterprise Bean クラスで定義されたコンテナー管理フィールド名のサブセットと一致しなければならない。
- 少なくとも、パブリックのデフォルト・コンストラクターがなければならない。

**注:** EJB サーバー (AE) 環境の場合、CMP エンティティー bean の 1 次キー・クラスでは、java.lang.Object クラスから継承した equals メソッドと hashCode メソッドを上書きする必要があります。

136ページの図26 に、Enterprise Bean 例である Item の複合 1 次キー・クラスを示します。実際は、このクラスはストリング変数 *productId* および *vendorId* のラッパーの役割を果たします。 ItemKey クラスの hashCode メソッドは、*productId* 変数の値を使用して一時的なストリング・オブジェクトを作成した後で、java.lang.String クラス内の対応する hashCode メソッドを呼び出します。デフォルトのコンストラクターの他に、ItemKey クラスでは、1 次キー変数の値を指定のストリングに設定するコンストラクターも定義しています。

```
...
import java.io.*;
// Composite primary key class
public class ItemKey implements java.io.Serializable {

     public String productId;
     public String vendorId;
     // Constructors
     public ItemKey() { };
     public ItemKey(String productId, String vendorId) {
          this.productId = productId;
          this.vendorId = vendorId;
     }

     public String getProductId() {
          return productId;
     }
     public String getVendorId() {
            return vendorId;
      }
 ...
     // EJB server (AE)-specific method
      public boolean equals(Object other) {
              if (other instanceof ItemKey) {
                   return (productId.equals(((ItemKey)
                              other).productId)
                         && vendorId.equals(((ItemKey)
                              other).vendorId));
               }
               else
               return false;
       }
     ...
     // EJB server (AE)-specific method
     public int hashCode() {
                return (new productId.hashCode());
       }
 }
```

*図 26. コード例: ItemKey 1 次キー・クラス*

1 次キー・クラスを使用して、未知の 1 次キーを前もってカプセル化しておくこともできます。たとえば、エンティティー bean を複数の永続データのストアに使用する予定の場合、それぞれの bean ごとに異なる 1 次キー構造が必要となります。エンティティー bean の 1 次キーの型は、エンティティー・オブジェクトを格納するベースとなるデータベースで使用する 1 次キーの型から派生します。そのため、Enterprise Bean の開発者が必ずしも型を認識しておく必要はありません。

未知の 1 次キーを指定するには、次のようにします。

- findByPrimaryKey クラスの引き数を java.lang.Object として宣言する。
- ejbCreate メソッドの戻り値を java.lang.Object として宣言する。
- デプロイメント・ディスクリプターで、1 次キー・クラスを型 java.lang.Object の 1 次キー・クラスとして指定する。

1 次キーの選択を配置まで据え置く場合には、クライアント・アプリケーションは、 1 次キーの型の認識を利用するメソッドを使用できません。さらに、戻り型は配置時に決まるため、アプリケーションは、1 次キーの型を戻すメソッド (EntityContext.getPrimaryKey メソッドなど) に依存できるとも限りません。

## データベースとの対話

**注:** このセクションは、アドバンスド版の EJB 環境にしか適用されません。Component Broker には、キャッシングを制御する独自の手段があります。詳細については、Component Broker「*上級プログラミング・ガイド*」を参照してください。

次に、Enterprise Bean とデータベース・アクセスの一般情報およびヒントを紹介します。

- 必ずしも必要なことではありませんが、データ・ソースに使用している Enterprise Bean または bean のコンテナーのいずれかで、データ・ソースのユーザー ID およびパスワードを指定することは良い習慣です。
- コンテナーは、Option A キャッシングと Option C キャッシングをサポートしています。 Option A キャッシングを使用している場合、永続ストア内のデータを更新できるのは、 Enterprise Bean コンテナーをホストしているアプリケーション・サーバーだけです。このため、Option A キャッシングは次のものと互換性がありません。
  - ワークロード管理下のサーバー (複製のクラスターなど)
  - 複数のアプリケーションと共有しているデータが格納されたデータベース

  デフォルトのキャッシング・オプションは C です (異なるサーバーに存在する可能性がある複数のエンティティー bean のインスタンスは、データベース内の bean 状態を更新することができます)。デフォルトのキャッシング・オプションを Option C から Option A に変更するには、エンティティー bean を作成するときに管理コンソールで「exclusive persistent store (排他的永続ストア)」を選択します。

共用データベース・アクセスは Option C キャッシングに対応します。 Option A キャッシングおよび Option C キャッシングは、それぞれコミット・オプション A およびコミット・オプション C とも呼ばれています。

## セッション bean の開発

セッション bean の基本的な構成は、エンティティー bean に似ています。しかし、その目的は大きく異なります。

コンポーネント全体からすれば、これら 2 つのタイプの Enterprise Bean の最大の相違点の 1 つは、セッション bean には 1 次キー・クラスがなく、かつセッション bean のホーム・インターフェースが finder メソッドを定義しないということです。セッション Enterprise Bean には 1 次キーや finder メソッドは必要ありません。これは、エンティティー EJB オブジェクトは、データ・ソースに永続データを書き込んで、1 次キーを使用して一意に識別できるのに対し、セッション EJB オブジェクトは、作成されて特定のクライアントに関連付けられた後、必要に応じて除去されるためです。つまり、セッション bean のデータは永続的に保管されないため、セッション bean クラスには、データ・ソースとの間でデータの保管やロードを行うためのメソッドがありません。

すべてのセッション bean には、以下の基本的なパーツがなければなりません。

- Enterprise Bean クラス。詳細については、『Enterprise Bean クラスの作成 (セッション)』を参照してください。
- Enterprise Bean のホーム・インターフェース。詳細については、151ページの『ホーム・インターフェースの作成 (セッション)』を参照してください。
- Enterprise Bean のリモート・インターフェース。詳細については、152ページの『リモート・インターフェースの作成 (セッション)』を参照してください。

### Enterprise Bean クラスの作成 (セッション)

セッション bean クラスは、Enterprise Bean のビジネス・メソッドの定義とインプリメント、Enterprise Bean インスタンスの作成時にコンテナーが使用するメソッドのインプリメント、およびインスタンスのライフ・サイクルにおいて重要なイベントを Enterprise Bean インスタンスに通知するためにコンテナーが使用するメソッドのインプリメントを行います。規則では、Enterprise Bean の名前は、*Name*Bean になります。ここで、*Name* は、ユーザーが Enterprise Bean に割り当てた名前です。たとえば、Transfer Enterprise Bean の場合は、Enterprise Bean クラスの名前は TransferBean になります。

すべてのセッション bean クラスは、以下の要件を満たしていなければなりません。

- Enterprise Bean に関連する作業を実行するビジネス・メソッドを定義してインプリメントしなければならない。詳細については、141ページの『ビジネス・メソッドのインプリメント』を参照してください。
- Enterprise Bean クラスをインスタンス化可能にするための方法ごとに、ejbCreate メソッドを定義して実行しなければならない。詳細については、144ページの『ejbCreate メソッドのインプリメント』を参照してください。
- パブリックでなければならず、抽象クラスであってはならない。また、javax.ejb.SessionBean インターフェースをインプリメントしなければならない。詳細については、149ページの『SessionBean インターフェースのインプリメント』を参照してください。

**注:** EJB 仕様のバージョン 1.0 では、セッション bean クラスのメソッドは、アプリケーション以外の例外を示す java.rmi.RemoteException 例外を throw することができました。これは、EJB 仕様のバージョン 1.1 では使用しないでください。バージョン 1.1 準拠のセッション bean は、代わりに javax.ejb.EJBException 例外 (java.lang.RuntimeException クラスのサブクラス) または別の RuntimeException 例外を throw することになっています。javax.ejb.EJBException クラスは java.lang.RuntimeException のサブクラスなので、メソッドの throws 文節に EJBException 例外を明示的にリストする必要はありません。

セッション bean は、状態付きにすることも、状態なしにすることもできます。状態なしセッション bean には、任意の他のメソッドによって設定された変数値に依存するメソッドはありません。ただし、ejbCreate メソッドの場合は、例外的に各 bean インスタンスの初期 (同一の) 状態を設定します。状態付き Enterprise Bean には、なんらかの他のメソッドによって設定された変数値に依存するメソッドがいくつかあります。エンティティー bean の場合と同様、静的変数は、最終値でない場合にはセッション bean でサポートされません。

場合によっては、状態付きセッション bean では、それらが操作されるトランザクションのコンテキストとそれらの会話状態とを同期させる必要があります。たとえば、状態付きセッション bean は、トランザクションがロールバックされた場合には、その変数のいくつかの値をリセットしなければなりません。また、トランザクションが正常に完了した場合には、それらの変数を変更しなければなりません。

bean がその会話状態をトランザクションのコンテキストに同期させる必要がある場合は、bean クラスは javax.ejb.SessionSynchronization インターフェースをインプリメントしなければなりません。このインターフェースは、トランザク

ションが開始された時点、ほぼ終了した時点、および完全に終了した時点をセッション bean に通知するメソッドを含んでいます。 Enterprise Bean の開発者は、これらのメソッドを使用して、セッション Enterprise Bean インスタンスの状態を、実行中のトランザクションに同期させることができます。

**注:** SessionSynchronization インターフェースは、EJB サーバー (CB) 環境ではサポートされていません。

Enterprise Bean クラスは Enterprise Bean のリモート・インターフェースをインプリメントすることができますが、これはお勧めしません。 Enterprise Bean クラスがリモート・インターフェースをインプリメントすると、この 変数をメソッドの引き数として誤って渡してしまう可能性があります。

141ページの図27 に、Transfer bean の場合の Enterprise Bean の主要な部分を示します。以降のセクションでは、これらの部分について詳細に説明します。

Transfer bean は状態なし bean です。 Transfer bean の transferFunds メソッドが、getBalance メソッドによって戻される *balance* 変数の値に依存している場合は、TransferBean が状態付きになります。

```
...
import java.rmi.RemoteException;
import java.util.Properties;
import java.util.ResurceBundle;
import java.util.ListResourceBundle;
import javax.ejb.*;
import java.lang.*;
import javax.naming.*;
import com.ibm.ejs.doc.account.*;
...
public class TransferBean implements SessionBean {
     ...
     private SessionContext mySessionCtx = null;
     private InitialContext initialContext = null;
     private AccountHome accountHome = null;
     private Account fromAccount = null;
     private Account toAccount = null;
     ...
     public void ejbActivate() throws EJBException { }
     ...
     public void ejbCreate() throws EJBException {
          ...
     }
     ...
     public void ejbPassivate() throws EJBException { }
     ...
     public void ejbRemove() throws EJBException { }
     ...
     public float getBalance(long acctId) throws FinderException,
          EJBException {
          ...
     }
     ...
     public void setSessionContext(javax.ejb.SessionContext ctx)
          throws EJBException {
          ...
     }
     ...
     public void transferFunds(long fromAcctId, long toAcctId, float amount)
          throws EJBException {
          ...
     }
}
```

*図 27. コード例: TransferBean クラス*

### ビジネス・メソッドのインプリメント

セッション bean クラスのビジネス・メソッドでは、EJB クライアントが Enterprise Bean を操作する方法を定義します。Enterprise Bean クラスにインプリメントされたビジネス・メソッドは、EJB クライアントから直接呼び出すこ

とはできません。代わりに、EJB クライアントは、Enterprise Bean のインスタンスに関連する EJB オブジェクトを使用して、Enterprise Bean のリモート・インターフェースに定義されている対応するメソッドを呼び出します。これにより、Enterprise Bean のインスタンスにある対応するメソッドがコンテナーによって呼び出されます。

したがって、Enterprise Bean のリモート・インターフェースに定義されているビジネス・メソッドごとに、対応するビジネス・メソッドを、Enterprise Bean クラスにインプリメントしなければなりません。 Enterprise Bean のリモート・インターフェースは、Enterprise Bean の配置時にコンテナーによって EJBObject クラスにインプリメントされます。

143ページの図28 に、TransferBean クラスのビジネス・メソッドを示します。getBalance メソッドは、口座の残高を照会するために使用します。このメソッドは、最初に適切な Account EJB オブジェクトを見付けてから、そのオブジェクトの getBalance メソッドを呼び出します。

transferFunds メソッドは、 (2 つの Account エンティティー EJB オブジェクトにカプセル化された) 2 つの口座間で指定の金額を送金するために使用します。 findByPrimaryKey メソッドを使用して適切な Account EJB オブジェクトが検出されると、transferFunds メソッドは、一方の口座で add メソッドを呼び出し、他方の口座で subtract メソッドを呼び出します。

すべての finder メソッドと同様に、findByPrimaryKey は FinderException 例外と RemoteException 例外の両方を throw することができます。findByPrimaryKey メソッドの呼び出しの前後に try/catch ブロックを設定して、ユーザーによる無効な口座 ID の入力を処理します。セッション bean のユーザーが無効な口座 ID を入力した場合は、findByPrimaryKey メソッドが EJB オブジェクトを検出できないため、 finder メソッドによって FinderException 例外が throw されます。この例外は受け取られて、無効な口座 ID に関する情報を含む新しい FinderException 例外に変換されます。

findByPrimaryKey メソッドを呼び出すには、2 つのビジネス・メソッドが EJB ホーム・オブジェクトにアクセスできなければなりません。EJB ホーム・オブジェクトは、129ページの『ホーム・インターフェースの作成 (CMP を持つエンティティー)』で説明しているように、AccountHome インターフェースをインプリメントします。 EJB ホーム・オブジェクトの取得については、144ページの『ejbCreate メソッドのインプリメント』で説明しています。

```
public class TransferBean implements SessionBean {
    ...
   private Account fromAccount = null;
   private Account toAccount = null;
   ...
   public float getBalance(long acctId) throws FinderException, EJBException {
        AccountKey key = new AccountKey(acctId);
        try {
             fromAccount = accountHome.findByPrimaryKey(key);
        } catch(FinderException ex) {
             throw new FinderException("Account " + acctId
                        + " does not exist.");
        } catch(RemoteException ex) {
             throw new FinderException("Account " + acctId
                        + " could not be found.");
        }
        return fromAccount.getBalance();
   }
   ...
   public void transferFunds(long fromAcctId, long toAcctId, float amount)
        throws EJBException, InsufficientFundsException, FinderException {
        AccountKey fromKey = new AccountKey(fromAcctId);
        AccountKey toKey = new AccountKey(toAcctId);
        try {
             fromAccount = accountHome.findByPrimaryKey(fromKey);
        } catch(FinderException ex) {
             throw new FinderException("Account " + fromAcctId
                       + " does not exist.");
        } catch(RemoteException ex) {
             throw new FinderException("Account " + acctId
                       + " could not be found.");
        }
        try {
             toAccount = accountHome.findByPrimaryKey(toKey);
        } catch(FinderException ex) {
             throw new FinderException("Account " + toAcctId
                       + " does not exist.");
        } catch(RemoteException ex) {
             throw new FinderException("Account " + acctId
                       + " could not be found.");
        }
        try {
             toAccount.add(amount);
             fromAccount.subtract(amount);
        } catch(InsufficientFundsException ex) {
              mySessionCtx.setRollbackOnly();
              throw new InsufficientFundsException("Insufficient funds in  "
                       + fromAcctId);
        }
   }
}
```

*図 28. コード例: TransferBean クラスのビジネス・メソッド*

### ejbCreate メソッドのインプリメント

ユーザーは、Enterprise Bean をインスタンス化する方法ごとに、ejbCreate メソッドを定義してインプリメントしなければなりません。

各 ejbCreate メソッドは、Enterprise Bean のホーム・インターフェースの create メソッドに相当します (エンティティー bean の場合とは異なり、セッション bean には ejbPostCreate メソッドがないことに注意してください)。Enterprise Bean クラスのビジネス・メソッドと異なり、ejbCreate メソッドはクライアントが直接呼び出すことはできません。代わりに、クライアントは bean インスタンスのホーム・インターフェースの create メソッドを呼び出します。これにより、ejbCreate メソッドがコンテナーによって呼び出されます。ejbCreate メソッドが正常に実行されると、EJB オブジェクトが作成されます。

セッション bean の ejbCreate メソッドは、次の要件を満たしていなければなりません。

- メソッドはパブリックとして宣言しなければならない。最終または静的として宣言することはできない。
- 戻り値なし (void) でなければならない。
- 状態なしセッション bean は、戻り値なし (void) で、引き数を含まない、単一の ejbCreate メソッドしか持ってはならない。状態付きセッション bean は複数の ejbCreate メソッドを持つことができる。

throws 文節では、任意のアプリケーション例外を定義できます。javax.ejb.EJBException または別の実行時例外を使用して、アプリケーション以外の例外を示すことができます。

エンティティー bean の ejbCreate メソッドは、次の要件を満たしていなければなりません。

- メソッドはパブリックとして宣言しなければならない。最終または静的として宣言することはできない。
- エンティティー bean の 1 次キー型を戻さなければならない。
- EJB オブジェクトが必要とするすべての変数値を設定するコードを含まなければならない。

throws 文節では、任意のアプリケーション例外を定義できます。javax.ejb.EJBException または別の実行時例外を使用して、アプリケーション以外の例外を示すことができます。

146ページの図29 に、TransferBean クラスの例の場合に必要な ejbCreate メソッドを示します。 Transfer bean の ejbCreate メソッドは、Account bean のホーム・オブジェクトへの参照を取得します。この参照は、Transfer bean のビジネス・メソッドに必要です。 Enterprise Bean のホーム・インターフェースへの参照は、以下の 2 つのステップの処理で取得されます。

1. 必要なプロパティー値を設定して、InitialContext オブジェクトを構成する。 Transfer bean の例では、これらのプロパティー値は Transfer bean のデプロイメント・ディスクリプターで定義されています。
2. InitialContext オブジェクトを使用して、ホーム・オブジェクトへの参照を作成して取得する。 Transfer bean の例では、Account bean の JNDI 名は Transfer bean のデプロイメント・ディスクリプターの環境変数に保管されます。

***InitialContext オブジェクトの作成:*** コンテナーが Transfer bean の ejbCreate メソッドを呼び出すと、プロパティー変数 (*env*) が作成され、Enterprise Bean の *initialContext* オブジェクトが構成されます。このプロパティー変数には、以下の値が必要です。

- ネーム・サービスのロケーション (javax.naming.Context.PROVIDER_URL)。
- 初期コンテキスト・ファクトリーの名前 (javax.naming.Context.INITIAL_CONTEXT_FACTORY)。

これらのプロパティーの値については、172ページの『bean の EJB オブジェクトへの参照の作成および取得』で詳しく説明します。

```
...
public class TransferBean implements SessionBean {
    private static final String INITIAL_NAMING_FACTORY_SYSPROP =
            javax.naming.Context.INITIAL_CONTEXT_FACTORY;
    private static final String PROVIDER_URL_SYSPROP =
            javax.naming.Context.PROVIDER_URL;
    ...
    private String nameService = null;
    ...
    private String providerURL = null;
    ...
    private InitialContext initialContext = null;
    ...
    public void ejbCreate() throws EJBException {
        // Get the initial context
        try {
            Properties env  = System.getProperties();
            ...
            env.put( PROVIDER_URL_SYSPROP, getProviderUrl() );
            env.put( INITIAL_CONTEXT_FACTORY_SYSPROP, getNamingFactory() );
            initialContext = new InitialContext( env );
        } catch(Exception ex) {
            ...
        }
        ...
        // Look up the home interface using the JNDI name
        ...
}
```

*図 29. コード例: TransferBean クラスの ejbCreate メソッドでの InitialContext オブジェクトの作成*

Transfer bean の例では、Account bean の例と同様にロケール特定の変数のいくつかがリソース・バンドル・クラスに保管されますが、そのデプロイメント・ディスクリプターに保管されている環境変数の値にも依存します。これらの InitialContext プロパティー値は、それぞれ Transfer bean のデプロイメント・ディスクリプターに入っている環境変数から取得されます。各値の取得には、プロパティー変数に対応するプライベート get メソッドが使用されます (getNamingFactory および getProviderURL)。これらのメソッドは、Enterprise Bean の開発者自身が作成しなければなりません。 Transfer bean のデプロイメント・ディスクリプターで、以下の環境変数を適切な値に設定しなければなりません。

- javax.naming.Context.INITIAL_CONTEXT_FACTORY
- javax.naming.Context.PROVIDER_URL

(67ページの『Enterprise Bean 用の環境変数の設定』に、これらの変数を設定するために必要な **jetace** ページの例が示してあります。)

図30 に、PROVIDER_URL プロパティー値の取得に使用する getProviderURL メソッドに関連する部分を示します。javax.ejb.SessionContext 変数 (*mySessionCtx*) は、getEnvironment メソッドを呼び出して、デプロイメント・ディスクリプターにある Transfer bean の環境を取得するために使用します。getEnvironment メソッドによって戻されたオブジェクトを使用し、getProperty メソッドを呼び出して、特定の環境変数の値を取得することができます。

```
...
public class TransferBean implements SessionBean {
      private SessionContext mySessionCtx = null;
     ...
     private String getProviderURL() throws RemoteException {
          //get the provider URL property either from
          //the EJB properties or, if it isn't there
          //use "iiop:///", which causes a default to the local host
          ...
          String pr = mySessionCtx.getEnvironment().getProperty(
                    PROVIDER_URL_SYSPROP);
          if (pr == null)
               pr = "iiop:///";
          return pr;
     }
     ...
}
```

*図 30. コード例: getProviderURL メソッド*

***ホーム・オブジェクトへの参照の取得:*** Enterprise Bean にアクセスするには、ホーム・インターフェースをインプリメントしているクラスを JNDI を使用して名前で検索します。ホーム・インターフェースのメソッドが、リモート・インターフェースをインプリメントしているクラスのインスタンスへのアクセスを提供します。

InitialContext オブジェクトを構成し終わると、 ejbCreate メソッドは Account Enterprise Bean の JNDI 名を使用して JNDI 検索を実行します。PROVIDER_URL プロパティーおよび INITIAL_CONTEXT_FACTORY プロパティーと同様、この名前も、(getHomeName というプライベート・メソッドを呼び出すことによって) Transfer bean のデプロイメント・ディスクリプターに含まれる環境変数から抽出されます。lookup メソッドは java.lang.Object 型のオブジェクトを戻します。

戻されたオブジェクトは、指定の Enterprise Bean の EJB ホーム・オブジェクトへの参照を取得するために、静的メソッド javax.rmi.PortableRemoteObject.narrow を使用して限定されます。 narrow メソッ

ドのパラメーターは、限定するオブジェクトの名前、および限定後に作成されるオブジェクトのクラスです。 JNDI で Enterprise Bean を検索して限定し、EJB ホーム・オブジェクトを取得するために必要なコードの詳しい説明は、172ページの『bean の EJB オブジェクトへの参照の作成および取得』に記載されています。

```
...
public class TransferBean implements SessionBean {
     ...
     private String accountName = null;
     ...
     private InitialContext initialContext = null;
     ...
     public void ejbCreate() throws EJBException {
          // Get the initial context
          ...
          // Look up the home interface using the JNDI name
          try {
               java.lang.Object ejbHome = initialContext.lookup(accountName);
               accountHome = (AccountHome)javax.rmi.PortableRemoteObject.narrow(
                          (org.omg.CORBA.Object) ejbHome,  AccountHome.class);
          } catch (NamingException e) { // Error getting the home interface
               ...
          }
          ...
     }
     ...
}
```

*図 31. コード例: TransferBean クラスの ejbCreate メソッドでの AccountHome オブジェクトの作成*

***Enterprise Bean の環境名コンテキストの検索:*** Enterprise Bean の環境はコンテナーがインプリメントします。これを使用することで、bean のソース・コードにアクセスしたり、ソース・コードを変更する必要なく、 bean のビジネス・ロジックをカスタマイズすることができます。コンテナーは、Enterprise Bean の環境を格納する JNDI 名コンテキストのインプリメンテーションを提供します。ビジネス・メソッドは、JNDI インターフェースを使用して環境にアクセスします。デプロイメント・ディスクリプターは、Enterprise Bean が実行時に必要とする環境項目を提供します。

各 Enterprise Bean は、そのすべてのインスタンス間 (つまり、ホームが同じであるすべてのインスタンス間) で共有する独自の環境項目を定義します。環境項目は、Enterprise Bean 間では共有されません。

Enterprise Bean の環境項目は、環境名コンテキスト (またはそのサブコンテキストの 1 つ) に直接格納されます。環境名コンテキストを取得するため、Enterprise Bean のインスタンスは、引き数なしでコンストラクターを使用し

て、 InitialContext オブジェクトを作成します。そして、名前 java:comp/env で、InitialContext オブジェクトを介して環境名を検索します。

図32 の Enterprise Bean は、環境項目を検索して新しい口座番号を探すことで、口座番号を変更します。

```
public class AccountService implements SessionBean {
...
     public void changeAccountNumber(int accountNumber, ... )
          throws InvalidAccountNumberException{
               ....
               // Obtain the bean's environment naming context
               Context initialContext = new InitialContext();
               Context myEnvironment = (Context)initialContext.lookup("java:comp/env);
            ...
            // Obtain new account number from environment
            Integer newNumber = (Integer)myEnvironment.lookup("newAccountNumber");
           ... }
}
```

*図 32. コード例: Enterprise Bean の環境名コンテキストの検索*

### SessionBean インターフェースのインプリメント

各セッション bean クラスは、javax.ejb.SessionBean インターフェースから継承されたメソッドをインプリメントしなければなりません。コンテナーは、これらのメソッドを呼び出して、インスタンスのライフ・サイクルにおける重要なイベントを Enterprise Bean インスタンスに通知します。これらのメソッドはすべてパブリックで、戻り値なし (void) でなければなりません。また、これらのメソッドは、javax.ejb.EJBException を throw することができます (java.rmi.RemoteException 例外の throw は行わないでください。詳細については、139 ページを参照してください)。

- ejbActivate。このメソッドは、コンテナーがインスタンス・プールから Enterprise Bean を選択して特定の既存の EJB オブジェクトに割り当てる場合に、コンテナーによって呼び出されます。このメソッドは、Enterprise Bean インスタンスを活動化する場合に実行させるコードをすべて含んでいなければなりません。
- ejbPassivate。このメソッドは、コンテナーが Enterprise Bean インスタンスの EJB オブジェクトとの関連付けを解除して、インスタンス・プールに Enterprise Bean インスタンスを入れる場合に、コンテナーによって呼び出されます。このメソッドは、Enterprise Bean インスタンスを非活動化する場合に実行させるコードをすべて含んでいなければなりません。
- ejbRemove。このメソッドは、Enterprise Bean のホーム・インターフェースによって (javax.ejb.EJBHome インターフェースから) 継承された remove メソッドをクライアントが呼び出す場合に、コンテナーによって呼び出されま

す。このメソッドは、Enterprise Bean インスタンスをコンテナーから除去する場合に実行させるコードをすべて含んでいなければなりません。

- setSessionContext。このメソッドは、javax.ejb.SessionContext インターフェースへの参照をセッション bean インスタンスに渡すためにコンテナーによって呼び出されます。 Enterprise Bean インスタンスがそのライフ・サイクルの任意の時点でこのコンテキストを使用する必要がある場合は、この値を保管するための変数が Enterprise Bean クラスに入っていなければなりません。このメソッドは、コンテキストへの参照を保管するために必要なすべてのコードを含んでいなければなりません。

  セッション・コンテキストを使用して、状態付きセッション bean の特定のインスタンスのハンドルを取得することができます。228ページの『bean 管理トランザクションの使用』で説明するように、これを使用して、トランザクション・コンテキスト・オブジェクトへの参照を取得することもできます。

  **注:** EJB サーバー (CB) 環境では、javax.ejb.EJBContext インターフェースから継承された isCallerInRole メソッドと getCallerIdentity メソッドはサポートされません。

図33 に示すように、TransferBean クラスでは、setSessionContext メソッド以外のすべての上記のメソッドが空になっています。これは、これらのメソッドに関連する特定のライフ・サイクルにおける状態に対して、bean が必要とする追加のアクションがないためです。 setSessionContext メソッドは、慣習的に、*mySessionCtx* 変数の値を設定するために使用されます。

```
...
public class TransferBean implements SessionBean {
     private SessionContext mySessionCtx = null;
     ...
     public void ejbActivate() throws EJBException { }
     ...
     public void ejbPassivate() throws EJBException { }
     ...
     public void ejbRemove() throws EJBException { }
     ...
     public void setSessionContext(SessionContext ctx) throwEJBException {
          mySessionCtx = ctx;
     }
     ...
}
```

*図 33. コード例: TransferBean クラスでの SessionBean インターフェースのインプリメント*

## ホーム・インターフェースの作成 (セッション)

セッション bean のホーム・インターフェースは、Enterprise Bean のインスタンスの作成と除去、およびインスタンスに関するメタデータの取得を行うためにクライアントが使用するメソッドを定義します。ホーム・インターフェースは、Enterprise Bean 開発者によって定義され、Enterprise Bean の配置時にコンテナーによって作成済みの EJB ホーム・クラスにインプリメントされます。コンテナーによって、ホーム・インターフェースは JNDI を介してクライアントにアクセスできるようになります。

規則では、ホーム・インターフェースの名前は *Name* Home になります。ここで、*Name* は、ユーザーが Enterprise Bean に割り当てた名前です。たとえば、Transfer Enterprise Bean のホーム・インターフェースの名前は、TransferHome になります。

すべてのセッション bean のホーム・インターフェースは、以下の要件を満たしていなければなりません。

- javax.ejb.EJBHome インターフェースを拡張しなければならない。ホーム・インターフェースは、javax.ejb.EJBHome インターフェースからいくつかのメソッドを継承します。これらのメソッドに関する詳細については、154ページの『javax.ejb.EJBHome インターフェース』を参照してください。
- インターフェースの各メソッドが、Enterprise Bean クラスの ejbCreate メソッドに相当する create メソッドでなければならない。詳細については、144ページの『ejbCreate メソッドのインプリメント』を参照してください。エンティティー bean と異なり、セッション bean のホーム・インターフェースは finder メソッドを含みません。
- このインターフェースに定義された各メソッドのパラメーターおよび戻り値が、Java RMI に有効でなければならない。詳細については、155ページの『java.io.Serializable インターフェースおよび java.rmi.Remote インターフェース』を参照してください。さらに、各メソッドの throws 文節に、java.rmi.RemoteException 例外クラスが含まれていなければならない。

152ページの図34 に、Transfer bean の例におけるホーム・インターフェース (TransferHome) の定義に関連する部分を示します。

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

*図 34. コード例: TransferHome ホーム・インターフェース*

create メソッドは、クライアントが Enterprise Bean インスタンスを作成するために使用します。状態付きセッション bean には複数の create メソッドを含めることができますが、状態なしセッション bean には引き数を持たない create メソッドを 1 つしか含めることができません。状態なしセッション bean のこの制約により、状態なしセッション bean のすべてのインスタンスが、同じ型のすべての他のインスタンスと確実に同じになります。(たとえば、すべての Transfer bean インスタンスは、すべての他の Transfer bean インスタンスと同じになります。)

すべての create メソッドには create という名前が付いていなければなりません。また、EJB オブジェクト・クラスの対応する ejbCreate メソッドと同じ数と型の引き数を持っていなければなりません。create メソッドとその対応する ejbCreate メソッドは、必ず異なる型を戻します。

すべての create メソッドは、以下の要件を満たしていなければなりません。

- Enterprise Bean のリモート・インターフェースの型を戻されなければならない。たとえば、TransferHome インターフェースの create メソッドの場合に戻される型は、Transfer です。
- java.rmi.RemoteException 例外、javax.ejb.CreateException 例外クラス、および対応する ejbCreate メソッドの throws 文節に定義されているすべての例外を含む throws 文節を持っていなければならない。

## リモート・インターフェースの作成 (セッション)

セッション bean のリモート・インターフェースは、Enterprise Bean クラスで使用可能なビジネス・メソッドへのアクセスを提供します。また、Enterprise Bean インスタンスの除去、および Enterprise Bean のホーム・インターフェースとハンドルの取得を行うメソッドも提供します。リモート・インターフェースは、Enterprise Bean 開発者によって定義され、Enterprise Bean の配置時にコンテナーによって作成済みの EJB リモート・クラスにインプリメントされます。

規則では、リモート・インターフェースの名前は *Name* になります。ここで、*Name* は、ユーザーが Enterprise Bean に割り当てた名前です。たとえば、Transfer Enterprise Bean のリモート・インターフェースの名前は、Transfer になります。

すべてのリモート・インターフェースは、以下の要件を満たしていなければなりません。

- javax.ejb.EJBObject インターフェースを拡張しなければならない。リモート・インターフェースは、EJBObject インターフェースからいくつかのメソッドを継承します。これらのメソッドに関する詳細については、154ページの『javax.ejb.EJBObject から継承されたメソッド』を参照してください。
- Enterprise Bean クラスにインプリメントされているビジネス・メソッドごとに、対応するビジネス・メソッドを定義しなければならない。
- このインターフェースに定義された各メソッドのパラメーターおよび戻り値が、Java RMI に有効でなければならない。詳細については、155ページの『java.io.Serializable インターフェースおよび java.rmi.Remote インターフェース』を参照してください。
- 各メソッドの throws 文節に、java.rmi.RemoteException 例外クラスが含まれていなければならない。

図35 に、Transfer bean の例におけるリモート・インターフェース (Transfer) の定義に関連する部分を示します。このインターフェースは、2 つの Account bean インスタンスの間で送金したり、Account bean インスタンスの残高を取得したりするためのメソッドを定義します。

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

*図 35. コード例: Transfer リモート・インターフェース*

## 複数のタイプの Enterprise Bean に共通なインターフェースのインプリメント

Enterprise Bean では、本書で説明するインターフェースを適切な Enterprise Bean コンポーネントにインプリメントする必要があります。

### javax.ejb.EJBObject から継承されたメソッド

リモート・インターフェースは、javax.ejb.EJBObject インターフェースから以下のメソッドを継承します。これらのメソッドは、配置時にコンテナーによってインプリメントされます。

- getEJBHome。Enterprise Bean のホーム・インターフェースを戻す。
- getHandle。EJB オブジェクトのハンドルを戻す。
- getPrimaryKey。EJB オブジェクトの 1 次キーを戻す。 (セッション bean の場合は、セッション bean に 1 次キーがないため、これを使用することはできません。)
- isIdentical。この EJB オブジェクトを EJB オブジェクトの引き数と比較して、同一であるかどうかを判別する。
- remove。この EJB オブジェクトを除去する。

上記のメソッドの構文は以下のとおりです。

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

これらのメソッドは、コンテナーによって EJB オブジェクト・クラスにインプリメントされます。

### javax.ejb.EJBHome インターフェース

ホーム・インターフェースは、javax.ejb.EJBHome インターフェースから 2 つの remove メソッドと getEJBMetaData メソッドを継承します。ホーム・インターフェースに直接定義するメソッドのように、これらの継承メソッドも、配置時にコンテナーによって、作成済みの EJB ホーム・クラスにインプリメントされます。

remove メソッドは、既存の EJB オブジェクト (およびデータベース内の関連データ) を、EJB オブジェクトのハンドルまたは 1 次キーのいずれかを指定して除去するために使用します。 (*primaryKey* 変数を受け取る remove メソッドは、エンティティー bean でしか使用することができません。) getEJBMetaData

メソッドは、Enterprise Bean に関するメタデータを取得するために使用します。このメソッドは、主に開発ツールでの使用を目的としています。

上記のメソッドの構文は以下のとおりです。

```
public abstract EJBMetaData getEJBMetaData();
public abstract void remove(Handle handle);
public abstract void remove(Object primaryKey);
```

javax.ejb.EJBHome インターフェースには、ホーム・インターフェースのハンドルを取得するメソッドも含まれます。構文は次のとおりです。

```
public abstract HomeHandle getHomeHandle();
```

### java.io.Serializable インターフェースおよび java.rmi.Remote インターフェース

リモート・メソッド呼び出し (RMI) での使用を可能にするには、メソッドの引き数および戻り値を以下の型のいずれかにしなければなりません。

- 基本型。int や long など。
- java.io.Serializable を直接または間接的にインプリメントするクラスのオブジェクト。java.lang.Long など。
- java.rmi.Remote を直接または間接的にインプリメントするクラスのオブジェクト。
- 有効な型またはオブジェクトの配列。

ユーザーが無効なパラメーターを使用しようとすると、java.rmi.RemoteException 例外が throw されます。以下の一般的な型は無効 なので注意してください。

- Serializable と Remote の両方を直接または間接的にインプリメントするクラスのオブジェクト。
- Remote を直接または間接的にインプリメントするが、RemoteException または RemoteException から継承された例外を throw しないメソッドを含むクラスのオブジェクト。

## Enterprise Bean でのスレッドおよび再入可能性の使用

Enterprise Bean には、新しいスレッドを開始するコードを含めてはなりません (また、synchronized というキーワードを使用してメソッドを定義することもできません)。セッション bean は再入可能にはできません。つまり、呼び出し側の bean でメソッドを起動する別の bean を呼び出すことはできません。エン

ティティー bean は再入可能にすることができますが、再入可能なエンティティー bean の構築はお勧めしません。また、本書でも説明していません。

EJB サーバー (AE) は、すべての Enterprise Bean に単一スレッド・アクセスを強制します。不正にコールバックを行うと、java.rmi.RemoteException 例外が EJB クライアントに対して throw されます。

EJB サーバー (CB) は、Enterprise Bean のトランザクションの属性が TX_NOT_SUPPORTED か TX_BEAN_MANAGED に設定されている場合に限り、これらの bean に単一スレッド・アクセスを強制します。その他の Enterprise Bean については、異なるトランザクションからのアクセスは逐次化されますが、同じトランザクションで実行されている異なるスレッドからのアクセスは逐次化されません。 Enterprise Bean がトランザクション属性値の TX_NOT_SUPPORTED または TX_BEAN_MANAGED を指定して配置された場合は、不正なコールバックを行うと、RemoteException 例外が EJB クライアントに throw されます。

## Enterprise Bean の EJB モジュールの作成

Enterprise Bean を配置できるようにするには、2 つの作業を行わなければなりません。

- bean のコンポーネントを同じ Java パッケージの一部にする。詳細については、157ページの『bean コンポーネントを Java パッケージの一部にする』を参照してください。
- EJB モジュールおよび関連するデプロイメント・ディスクリプター (AE のみ) を作成する。詳細については、157ページの『EJB モジュールおよびデプロイメント・ディスクリプターの作成』を参照してください。

Enterprise Bean を IDE で開発する場合は、これらの作業は、使用するツール内で処理されます。 Enterprise Bean を IDE で開発しない場合は、Java ソフトウェア開発キット (SDK) および WebSphere Application Server に含まれるツールを使用して、これらの作業をそれぞれ処理しなければなりません。

- EJB サーバー (AE) プログラミング環境で EJB モジュールを作成する場合に使用するツールの詳細については、 35ページの『第3章 EJB サーバー (AE) 環境での Enterprise Bean の開発用および配置用ツール』 を参照してください。
- EJB サーバー (CB) プログラミング環境での bean のパッケージ化に使用するツールに関する詳細については、43ページの『第4章 EJB サーバー (CB) 環境での Enterprise Bean の開発用および配置用ツール』を参照してください。

## bean コンポーネントを Java パッケージの一部にする

Enterprise Bean をどのように Java パッケージに割り振るかは、ユーザー自身が決定します。Java パッケージには、1 つ以上の Enterprise Bean を含めることができます。Account および Transfer bean の例は、別個のパッケージに保管されています。Account bean を構成するすべての Java ソース・ファイルには、以下の package ステートメントが入っています。

```
package com.ibm.ejs.doc.account;
```

Transfer bean を構成するすべての Java ソース・ファイルには、以下の package ステートメントが入っています。

```
package com.ibm.ejs.doc.transfer;
```

## EJB モジュールおよびデプロイメント・ディスクリプターの作成

EJB モジュールには、1 つ以上の配置可能な Enterprise Bean が入っています。また、各 Enterprise Bean に関する情報、およびモジュール内のすべての Enterprise Bean の処理方法についてのコンテナーへの指示を提供するデプロイメント・ディスクリプターも入っています。デプロイメント・ディスクリプターは、XML ファイルに格納されます。

EJB モジュールの作成時に、モジュールに含める各 Enterprise Bean のファイルを指定します。これらのファイルには、次のものが含まれます。

- Enterprise Bean の各コンポーネントに関連するクラス・ファイル。
- Enterprise Bean に関連する追加のクラスおよびファイル。ユーザー定義の例外クラス、プロパティー・ファイル、リソース・バンドル・クラスなど。

また、他の Enterprise Bean の参照、リソース・ファクトリー、セキュリティー役割など、 bean に関するその他の情報も指定します。モジュールに含める Enterprise Bean を定義し終えたら、そのモジュール全体に適用するアプリケーション・アセンブリー指示を指定します。 bean 情報およびモジュール情報の両方を使用して、デプロイメント・ディスクリプターを作成します。デプロイメント・ディスクリプターの設定と属性のリストについては、 22ページの『デプロイメント・ディスクリプター』を参照してください。

# 第6章 Enterprise Bean でのトランザクションおよびセキュリティーの使用可能化

本章では、適切なデプロイメント・ディスクリプター属性を設定することによって Enterprise Bean でトランザクションおよびセキュリティーを使用可能にする方法について説明します。

- トランザクションについては、セッション bean は、コンテナー管理トランザクションを使用することも、bean 管理トランザクションをインプリメントすることもできます。エンティティー bean は、コンテナー管理トランザクションを使用しなければなりません。コンテナー管理トランザクションを使用可能にするには、トランザクション属性を TX_BEAN_MANAGED 以外の任意の値に設定し、トランザクション分離レベル属性を設定しなければなりません。bean 管理トランザクションを使用可能にするには、トランザクション属性を TX_BEAN_MANAGED に設定し、トランザクション分離レベル属性を設定しなければなりません。詳細については、160ページの『デプロイメント・ディスクリプターのトランザクション属性の設定』を参照してください。

  セッション bean にそれ自体のトランザクションを管理させる場合は、228ページの『bean 管理トランザクションの使用』で説明するようにトランザクションの境界を明示的に分離するコードを作成しなければなりません。

  EJB クライアントにそれ自体のトランザクションを管理させる場合は、182ページの『EJB クライアントでのトランザクションの管理』で説明するように、クライアントが管理を行うように明示的にコーディングしなければなりません。

- セキュリティーについては、EJB サーバー環境では実行モード 属性が使用されます。この属性の有効な値に関する詳細については、167ページの『デプロイメント・ディスクリプターのセキュリティー属性の設定』を参照してください。

これらの属性は、他のデプロイメント・ディスクリプター属性のように、EJB サーバー (AE) または EJB サーバー (CB) のいずれかで使用可能なツールを使用して設定します。詳細については、35ページの『第3章 EJB サーバー (AE) 環境での Enterprise Bean の開発用および配置用ツール』または 43ページの『第4章 EJB サーバー (CB) 環境での Enterprise Bean の開発用および配置用ツール』を参照してください。

© Copyright IBM Corp. 1999, 2001

## デプロイメント・ディスクリプターのトランザクション属性の設定

EJB 仕様では、Enterprise Bean によって操作されるデータのトランザクションでの整合性を強制するアプリケーションの作成について定めています。しかし、分散トランザクションをサポートする他の仕様とは異なり、EJB 仕様では、Enterprise Bean および EJB クライアントの開発者がトランザクションを使用するための特別なコードを作成する必要はありません。この場合、コンテナーは、EJB モジュールに関連付けられた 2 つのデプロイメント・ディスクリプター属性に基づいてトランザクションを管理するため、Enterprise Bean および EJB アプリケーションの開発者は、それらのアプリケーションのビジネス・ロジックを扱う必要はないのです。

Enterprise Bean の開発者は、トランザクションを明示的に管理する Enterprise Bean および EJB アプリケーションを特に設計することができます。詳細については、228ページの『bean 管理トランザクションの使用』を参照してください。

ほとんどの条件において、トランザクション管理は、Enterprise Bean 内で処理することができるので、EJB クライアントの開発者にとってこの作業は不要です。しかし、必要な場合は、EJB クライアントはトランザクションを操作することができます。詳細については、182ページの『EJB クライアントでのトランザクションの管理』を参照してください。

2 つの属性で、Enterprise Bean をトランザクションの観点から管理する方法が決まります。

- *トランザクション* 属性は、コンテナーがメソッドを呼び出す際のトランザクションの形態を定義します。この属性は、標準デプロイメント・ディスクリプターの一部です。 『トランザクション属性の設定』では、この属性の有効な値と、それらの意味について説明します。
- *トランザクション分離レベル* 属性は、コンテナーがトランザクションを相互に分離する方法を定義します。この属性は、標準デプロイメント・ディスクリプターの拡張です。 164ページの『トランザクション分離レベル属性の設定』では、この属性の有効な値と、それらの意味について説明します。

### トランザクション属性の設定

トランザクション属性は、コンテナーが Enterprise Bean のメソッドを呼び出す際のトランザクションの形態を定義します。この属性は、bean 内の個々のメソッドに対して設定されます。

**注:** EJB サーバー (CB) は、個々の Enterprise Bean メソッドへのトランザクション属性の設定をサポートしていません。トランザクション属性は、bean 全体に対してのみ設定することができます。

以下に、トランザクションの厳密性の高いものから低いものの順に、この属性の有効な値を示します。

**TX_BEAN_MANAGED**

トランザクションの境界画定を bean クラスが直接処理することをコンテナーに通知します。この属性値は、セッション bean に対してのみ指定することができ、個々の bean メソッドに指定することはできません。この属性値をインプリメントするためのセッション bean の設計に関する詳細については、228ページの『bean 管理トランザクションの使用』を参照してください。

EJB サーバー (CB) 環境において、状態付きセッション bean がこの属性値を持つ場合は、トランザクションを開始するメソッドは、トランザクションの完了 (コミットまたはロールバック) も行わなければなりません。つまり、EJB サーバー (CB) 環境で使用する場合は、トランザクションを状態付きセッション bean の複数のメソッドに引き渡すことはできません。

**TX_MANDATORY**

常にクライアントに関連付けられたトランザクション・コンテキスト内で bean メソッドを呼び出すようにコンテナーに指定します。クライアントがトランザクション・コンテキストなしで bean メソッドを呼び出そうとすると、コンテナーは、javax.jts.TransactionRequiredException 例外をクライアントに throw します。トランザクション・コンテキストは、Enterprise Bean メソッドがアクセスするすべての EJB オブジェクトまたはリソースに渡されます。

これらのエンティティー bean にアクセスする EJB クライアントは、既存のトランザクション内で処理を行わなければなりません。他の Enterprise Bean の場合は、Enterprise Bean または bean メソッドは、TX_BEAN_MANAGED 値をインプリメントするか TX_REQUIRED または TX_REQUIRES_NEW 値を使用しなければなりません。非 Enterprise Bean EJB クライアントの場合は、クライアントは、182ページの『EJB クライアントでのトランザクションの管理』で説明するように、javax.transaction.UserTransaction インターフェースを使用してトランザクションを起動しなければなりません。

EJB サーバー (CB) 環境では、この属性値は、Host On-Demand (HOD) または外部呼び出しインターフェース (ECI) を使用して CICS または

IMS アプリケーションにアクセスする、コンテナー管理のパーシスタンス (CMP) を持つエンティティー bean で使用しなければなりません。

**TX_REQUIRED**

トランザクション・コンテキスト内で bean メソッドを呼び出すようにコンテナーに指定します。クライアントがトランザクション・コンテキスト内から bean メソッドを呼び出した場合は、コンテナーは、そのクライアント・トランザクション・コンテキスト内で bean メソッドを呼び出します。クライアントがトランザクション・コンテキスト外から bean メソッドを呼び出した場合は、コンテナーは、新しいトランザクション・コンテキストを作成してそのコンテキスト内から bean メソッドを呼び出します。トランザクション・コンテキストは、この bean メソッドが使用するすべての Enterprise Bean オブジェクトまたはリソースに渡されます。

**TX_REQUIRES_NEW**

クライアントがトランザクション・コンテキスト内でメソッドを呼び出すかトランザクション・コンテキスト外でメソッドを呼び出すかに無関係に、常に新しいトランザクション・コンテキスト内で bean メソッドを呼び出すようにコンテナーに指定します。トランザクション・コンテキストは、この bean メソッドが使用するすべての Enterprise Bean オブジェクトまたはリソースに渡されます。

EJB サーバー (CB) は、EJB 仕様のバージョン 1.0 に合わせて作成された Enterprise Bean のこの属性値をサポートしていません。 EJB 仕様のバージョン 1.1 に合わせて作成された bean では、TX_REQUIRES_NEW 属性は TX_REQUIRED と解釈されます。

**TX_SUPPORTS**

クライアントがトランザクション内で bean メソッドを呼び出した場合に、トランザクション・コンテキスト内で bean メソッドを呼び出すようにコンテナーに指定します。クライアントがトランザクション・コンテキストなしで bean メソッドを呼び出した場合は、コンテナーは、トランザクション・コンテキストなしで bean メソッドを呼び出します。トランザクション・コンテキストは、この bean メソッドが使用するすべての Enterprise Bean オブジェクトまたはリソースに渡されます。

EJB サーバー (CB) 環境では、CMP を持つエンティティー bean にはトランザクション内でアクセスしなければなりません。 CMP を持つエンティティー bean がこのトランザクション属性を使用する場合は、EJB クライアントは、トランザクションを開始してからそのエンティティー bean に対するメソッドを呼び出します。

**TX_NOT_SUPPORTED**

トランザクション・コンテキストなしで bean メソッドを呼び出すようにコンテナーに指定します。クライアントがトランザクション・コンテキスト内から bean メソッドを呼び出した場合は、コンテナーは、トランザクションと現行スレッドの間の関連付けを中断してから、Enterprise Bean インスタンスに対するメソッドを呼び出します。その後、メソッド呼び出しから戻ると、コンテナーは中断した関連付けを再開します。中断されたトランザクション・コンテキストは、この bean メソッドが使用するいずれの Enterprise Bean オブジェクトやリソースにも渡されません。

EJB サーバー (CB) 環境では、CMP を持つエンティティー bean にはトランザクション内でアクセスしなければなりません。したがって、この属性値は、EJB サーバー (CB) 環境の CMP を持つエンティティー bean ではサポートされません。

**TX_NEVER**

トランザクション・コンテキストなしで bean メソッドを呼び出すようにコンテナーに指定します。

- クライアントがトランザクション・コンテキスト内から bean メソッドを呼び出した場合、コンテナーは java.rmi.RemoteException 例外を throw します。
- クライアントがトランザクション・コンテキスト外から bean メソッドを呼び出した場合、コンテナーは TX_NOT_SUPPORTED トランザクション属性が設定されている場合と同じように振る舞います。クライアントは、トランザクション・コンテキストなしでメソッドを呼び出さなければなりません。

EJB サーバー (CB) 環境で、TX_NEVER 属性は TX_NOT_SUPPORTED と解釈されます。したがって、クライアントがトランザクション・コンテキスト内から bean メソッドを呼び出した場合は、例外は throw されません。

*表 4. トランザクション・コンテキストに対する Enterprise Bean のトランザクション属性の影響*

| トランザクション属性 | クライアントのトランザクション・コンテキスト | bean のトランザクション・コンテキスト |
| --- | --- | --- |
| TX_MANDATORY | トランザクションなし | 許可されていない |
| | クライアント・トランザクション | クライアント・トランザクション |

表 4. トランザクション・コンテキストに対する Enterprise Bean のトランザクション属性の影響 (続き)

| トランザクション属性 | クライアントのトランザクション・コンテキスト | bean のトランザクション・コンテキスト |
|---|---|---|
| TX_REQUIRES_NEW | トランザクションなし | 新しいトランザクション |
| | クライアント・トランザクション | 新しいトランザクション |
| TX_REQUIRED | トランザクションなし | 新しいトランザクション |
| | クライアント・トランザクション | クライアント・トランザクション |
| TX_SUPPORTS | トランザクションなし | トランザクションなし |
| | クライアント・トランザクション | クライアント・トランザクション |
| TX_NOT_SUPPORTED | トランザクションなし | トランザクションなし |
| | クライアント・トランザクション | トランザクションなし |
| TX_NEVER | トランザクションなし | トランザクションなし |
| | トランザクションなし | トランザクションなし |

エンティティー bean のデプロイメント・ディスクリプターを設定している場合は、 getter メソッドを ″Read-Only″ メソッドとして指定してパフォーマンスを向上させることができます。トランザクション作業単位に ″Read-Only″ 指定メソッドしか入っていない場合、エンティティー bean の状態同期はストアを呼び出しません。

## トランザクション分離レベル属性の設定

**注:** EJB サーバー (CB) は、トランザクション分離レベル属性をサポートしていません。

トランザクション分離レベルは、あるトランザクションを別のトランザクションから分離する強さのレベルを決定するものです。この属性は、bean 内の個々のメソッドに対して設定されます。しかし、トランザクション・コンテキスト内では、最初のメソッド呼び出しに関連付けられた分離レベルが、そのトランザクション内で呼び出される他のすべてのメソッドに対して要求される分離レベルになります。最初のメソッドの分離レベルとは異なる分離レベルでメソッドが呼び出されると、java.rmi.RemoteException 例外が throw されます。

以下に、分離のレベルの降順に、この属性に有効な値を示します。

**TRANSACTION_SERIALIZABLE**

このレベルでは、以下の種類の読み取りがすべて禁止されます。

- *ダーティー読み取り*。これは、2 番目のトランザクションによるコミットされていない変更を含むデータベースの行をトランザクションが読み取るものです。
- *繰り返し不能読み取り*。これは、あるトランザクションが行を読み取った後に 2 番目のトランザクションがその行を変更すると、最初のトランザクションが再度その行を読みだしたときに異なる値が取得されるものです。
- *偽読み取り*。これは、あるトランザクションが SQL WHERE 条件を満たすすべての行を読み取ったあと、2 番目のトランザクションがその WHERE 条件を満たす行を挿入した場合に、最初のトランザクションが同じ WHERE 条件を適用すると、2 番目のトランザクションによって挿入された行が取得されるものです。

**TRANSACTION_REPEATABLE_READ**

このレベルでは、ダーティー読み取りおよび繰り返し不能読み取りが禁止されますが、偽読み取りは許可されます。

**TRANSACTION_READ_COMMITTED**

このレベルでは、ダーティー読み取りが禁止されますが、繰り返し不能読み取りおよび偽読み取りは許可されます。

**TRANSACTION_READ_UNCOMMITTED**

このレベルでは、ダーティー読み取り、繰り返し不能読み取り、および偽読み取りが許可されます。

これらの分離レベルは、Java データベース・コネクティビティー (JDBC) の java.sql.Connection インターフェースで定義された分離レベルに対応します。

コンテナーは、以下のようにトランザクション分離レベル属性を使用します。

- bean 管理のパーシスタンス (BMP) を持つセッション bean およびエンティティー bean ー bean が使用するデータベース接続ごとに、コンテナーが、各トランザクションの開始時にトランザクション分離レベルを設定します。
- コンテナー管理のパーシスタンス (CMP) を持つエンティティー bean ー コンテナーが、指定された分離レベルをインプリメントするデータベース・アクセス・コードを生成します。

これらの値で、 2 つのトランザクションが同じデータを同時に更新することを許可しているものはありません。一方のトランザクションが終了して初めて、もう一方のトランザクションは同じデータを更新することができます。これら

の値は、データ読み込みの際のロックの管理方法を決定します。ただし、整合性のリスクは、トランザクションが読み込んだ値に基づいてさらに作業を進める場合、読み取り操作から発生する可能性があります。たとえば、1 つのトランザクションがあるデータを更新していて、データの変更は終了したが更新トランザクションが終了していないうちに、 2 番目のトランザクションがそのデータの読み取りを許可された場合、読み取りトランザクションは、結果的に変更がロールバックされることも考慮して決定を行うことができます。 2 番目のトランザクションには、一時データに対して決定を行ってしまうリスクがあります。

使用する分離レベルの判断は、以下のいくつかの要因によって決まります。

- データの整合性に関するリスクの許容レベル
- 並行性およびパフォーマンスの許容レベル
- 基礎データベースでサポートされる分離レベル

整合性リスクと並行性レベルという最初の 2 つの要因には相互関係があります。整合性に関するリスクが小さくするには、並行性を低下させなければなりません。整合性に関するリスクを削減するにはロックのホールド時間を長くする必要があるためです。 1 つのデータについてロックのホールド時間を長くすると、並行して実行しているトランザクションがそのデータにアクセスする場合の待機時間も長くなります。 TRANSACTION_SERIALIZABLE 値は、データに対して同時にアクセスさせないようにして、データを保護します。逆に言えば、TRANSACTION_READ_UNCOMMITTED 値により、並行性の度合いを最大にできますが、整合性のリスクも最大にすることになります。アプリケーションにおいて、これら 2 つの要因のバランスをうまく取る必要があります。

デフォルトでは、ほとんどの開発者が、トランザクション分離レベルが TRANSACTION_SERIALIZABLE に設定された Enterprise Bean を配置します。これは、IBM VisualAge for Java エンタープライズ版およびその他の配置ツールでのデフォルト値です。また、最大のオーバーヘッドを発生させる、最も制限され、保護されたトランザクション分離レベルでもあります。中には、TRANSACTION_SERIALIZABLE が提供する分離レベルと保護を必要としないワークロードもあります。アプリケーションによっては、ベースとなるデータを更新しなかったり、並行更新も行う他のアプリケーションとは一緒に実行しなかったりする場合もあります。こうした場合には、アプリケーションをダーティー読み取り、繰り返し不能読み取り、または偽読み取りに関連付ける必要はありません。おそらく TRANSACTION_READ_UNCOMMITTED 分離レベルで十分でしょう。

トランザクション分離レベルは EJB モジュールのデプロイメント・ディスクリプターで設定するので、トランザクション分離レベルが異なる別のアプリケーションでも同じ Enterprise Bean を再使用できます。分離レベル要件を検討し、適切に調整して、パフォーマンスを向上させる必要があります。

3 番目の要因である、データベースでサポートされる分離レベルは、EJB 仕様では、トランザクション分離のうちの 4 つのレベルのうちの 1 つを要求できるようになっていますが、そのアプリケーションで使用するデータベースによっては、すべてのレベルをサポートしているとは限りません。また、データベース製品のベンダーにより実装される分離レベルは異なるため、アプリケーションの正確な振る舞いは、データベースによってすべて異なります。展開記述子のトランザクション分離属性の値を決定する場合に、サポートされるデータベースおよび分離レベルを考慮する必要があります。サポートされる分離レベルについて、詳しくは、そのデータベースの資料を参照してください。

## デプロイメント・ディスクリプターのセキュリティー属性の設定

EJB クライアントが Enterprise Bean に対するメソッドを呼び出すと、クライアント・プリンシパルのユーザー・コンテキストが CORBA の Current オブジェクトにカプセル化され、そのオブジェクトに、プリンシパルの証明書プロパティーが入れられます。 Current オブジェクトは、メソッドの完了に必要なメソッド呼び出しの参加者同士で受け渡されます。

セキュリティー・サービスは、証明書情報を使用して、さまざまなリソースに対するプリンシパルのアクセス権を判別します。適切な時点で、セキュリティー・サービスは、プリンシパルのアクセス権に基づいて、特定のリソースの使用がプリンシパルに許可されているかどうかを判別します。

メソッド呼び出しが許可されている場合、セキュリティー・サービスは、Enterprise Bean の *実行モード* 属性の値に基づき、プリンシパルの証明書プロパティーに対して次のことを行います。特定の ID が必要な場合は、*RunAsIdentity* 属性を使用してその ID を識別します。

**CLIENT_IDENTITY**
セキュリティー・サービスは、プリンシパルの証明書プロパティーを変更しません。

**SYSTEM_IDENTITY**
セキュリティー・サービスは、プリンシパルの証明書プロパティーを更新して、EJB サーバーと関連付けられた証明書プロパティーに一致させます。

**SPECIFIED_IDENTITY**

セキュリティー・サービスは、プリンシパルの証明書プロパティーを、Enterprise Bean が関連付けられたアプリケーションのものに一致させようとします。成功した場合は、セキュリティー・サービスは、プリンシパルの証明書プロパティーを更新して、アプリケーションの証明書プロパティーに一致させます。

# 第7章 EJB クライアントの開発

Enterprise Bean には、両方の EJB サーバー環境において以下のすべてのタイプの EJB クライアントからアクセスすることができます。

- Java サーブレット。 Enterprise Bean を使用する Java サーブレットの作成に関する詳細については、189ページの『第8章 Enterprise Bean を使用するサーブレットの開発』を参照してください。
- JavaServer Pages (JSP)。JSP の作成に関する詳細については、市販の資料を参照してください。
- リモート・メソッド呼び出し (RMI) を使用する Java アプリケーション。Java アプリケーションの作成に関する詳細については、市販の資料を参照してください。
- その他の Enterprise Bean。たとえば、117ページの『第5章 Enterprise Bean の開発』で説明しているように、Transfer セッション bean は Account bean に対するクライアントとして機能します。

EJB エンティティー bean にはクライアントまたはサーブレットのコードからアクセスしないようにすることをお勧めします。代わりに、EJB エンティティー bean を EJB セッション bean でラップして、そこからアクセスしてください。これにより、次の 2 つの方法でパフォーマンスが向上します。

- リモート・メソッド呼び出しの数が削減されます。クライアント・アプリケーションがエンティティー bean に直接アクセスする場合、各 getter メソッドはリモート呼び出しです。ラッピングしているセッション bean は、エンティティー bean をローカルでアクセスし、構造内のデータを収集して、値で戻します。
- EJB エンティティー bean に外部トランザクション・コンテキストを提供します。エンティティー bean は、各トランザクションの完了時に、その状態を、ベースとなるデータ・ストアと同期させます。クライアント・アプリケーションがエンティティー bean に直接アクセスする場合、各 getter メソッドは完全なトランザクションとなります。ストアおよびロード・アクションが、各メソッドの後に続きます。セッション bean で bean 全体をラッピングして、外部トランザクション・コンテキストを提供すると、エンティティー bean は、外部のセッション bean がトランザクション境界に達したときにその状態を同期します。

© Copyright IBM Corp. 1999, 2001 

本章で説明する基本的なプログラミング作業を除き、Enterprise Bean に対するクライアントである Java サーブレット、JSP、または Java アプリケーションを作成するのは、標準的なバージョンのこれらのタイプの Java プログラムを設計するのとほとんど変わりません。本章は、ユーザーが Java サーブレット、Java アプリケーション、または JSP ファイルの作成の基礎について理解していることを前提としています。

本章で使用するコードは、特記しない限り、すべて TransferApplication という名前の Java アプリケーションの例からの引用です。本書のコード例で使用可能なこの Java アプリケーションおよびその他の EJB クライアントについては、307ページの『本書に記載する例の説明』で説明しています。

以前にリストした Java ベースの EJB クライアント・タイプのいずれかで Enterprise Bean をアクセスおよび操作するには、EJB クライアントは以下を実行しなければなりません。

- ネーム解決、リモート・メソッド呼び出し (RMI)、および Enterprise Bean の対話に必要な Java パッケージをインポートする。
- JNDI (Java Naming and Directory Interface) を使用して、bean の EJB オブジェクトのインスタンスへの参照を取得する。詳細については、172ページの『bean の EJB オブジェクトへの参照の作成および取得』を参照してください。
- セッション bean を使用する場合は、無効な EJB オブジェクトを処理する。詳細については、179ページの『セッション bean の無効な EJB オブジェクトの処理』を参照してください。
- セッション EJB オブジェクトが不要になったときにそのオブジェクトを除去するか、データ・ソース内の関連データを除去しなければならないときにエンティティー EJB オブジェクトを除去する。詳細については、181ページの『bean の EJB オブジェクトの除去』を参照してください。

また、EJB クライアントは、クライアントが使用する Enterprise Bean に関連するトランザクションを操作することもできます。詳細については、182ページの『EJB クライアントでのトランザクションの管理』を参照してください。

**注:** EJB サーバー (CB) 環境では、Enterprise Bean には、Java アプレット、ActiveX クライアント、CORBA ベースの Java クライアント、および (ある程度までは) C++ CORBA クライアントもアクセスすることができます。307ページの『本書に記載する例の説明』で簡単に説明している Travel の例では、これらのタイプのクライアントのいくつかについて説明して

います。184ページの『EJB サーバー (CB) 固有の EJB クライアントに関する追加情報』に、ActiveX および CORBA ベースの Java と C++ を使用する EJB クライアントに関する追加情報が記載されています。

## 必要な Java パッケージのインポート

各 EJB クライアントに必要な Java パッケージはさまざまですが、以下のパッケージはすべての EJB クライアントに必要です。

- java.rmi ー このパッケージには、リモート・メソッド呼び出し (RMI) に必要なクラスのほとんどが入っています。
- javax.rmi ー このパッケージには、EJB オブジェクトへの参照を取得するために必要な PortableRemoteObject クラスが入っています。
- java.util - このパッケージには、Properties、Hashtable、および Enumeration などの各種 Java ユーティリティー・クラスが入っています。これらは、すべての Enterprise Bean および EJB クライアントにわたってさまざまな方法で使用されます。
- javax.ejb ー このパッケージには、EJB 仕様で定義されているクラスおよびインターフェースが入っています。
- javax.naming ー このパッケージには、JNDI (Java Naming and Directory Interface) の仕様で定義されたクラスおよびインターフェースが入っています。このパッケージは、クライアントが EJB オブジェクトへの参照を取得するために使用します。
- クライアントが対話する Enterprise Bean を含む、1 つまたは複数のパッケージ。

Java クライアントのオブジェクト・リクエスト・ブローカー (ORB) (EJB クライアントで自動的に初期化されます) は、サーバーからクライアントへのインプリメンテーション・バイトコードの動的ダウンロードをサポートしていません。このため、EJB クライアントが実行時に必要とするすべてのクラスを、クライアントの CLASSPATH 環境変数で識別されるファイルおよびディレクトリーから使用できなければなりません。 EJB クライアントが必要とする JAR ファイルに関する詳細については、38ページの『EJB サーバー (AE) 環境での CLASSPATH 環境変数の設定』または 47ページの『EJB サーバー (CB) 環境での CLASSPATH 環境変数の設定』を参照してください。クライアント・マシンに必要なファイルは、マシンに WebSphere Application Server をインストールすることでインストールすることができます。 Application Server アドバンスド版を使用している場合は、**「Developer’s Client Files (開発者のクライアント・ファイル)」**オプションを選択します。 Component Broker を使用している場合は、**「Java client (Java クライアント)」**オプションを選択します。

また、クライアント・マシンで ioser および ioserx 実行可能ファイルを利用できることを確認しておくことも必要です。これらのファイルは、通常、Java インストールの一環になっています。 Windows NT を使用している場合は、EJB クライアントが実行時に ioser.dll ライブラリー・ファイルを探し出せることを確認します。

図36 に、Java アプリケーションの例 com.ibm.ejs.doc.client.TransferApplication の import ステートメントを示します。前述の必須 Java パッケージの他に、このアプリケーションの例では com.ibm.ejs.doc.transfer パッケージもインポートされます。これは、このアプリケーションが Transfer bean と通信するためです。アプリケーションの例では、Account bean と同じパッケージに入っている InsufficientFundsException クラスもインポートされます。

```
...
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.rmi.*
...
import javax.naming.*;
import javax.ejb.*;
import javax.rmi.PortableRemoteObject;
...
import com.ibm.ejs.doc.account.InsufficientFundsException;
import com.ibm.ejs.doc.transfer.*;
...
public class TransferApplication extends Frame implements
     ActionListener, WindowListener {
     ...
}
```

*図 36. コード例: Java アプリケーション TransferApplication の import ステートメント*

## bean の EJB オブジェクトへの参照の作成および取得

bean のビジネス・メソッドを呼び出すには、クライアントはその bean の EJB オブジェクトを作成または検索しなければなりません。クライアントがこのオブジェクトを作成または検索した後は、このオブジェクトは、それ自体に対するメソッドを標準の方法で呼び出すことができます。

bean の EJB オブジェクトのインスタンスを作成または検索するには、クライアントは以下を実行しなければなりません。

1. その bean の EJB ホーム・オブジェクトを見付けて作成する。詳細については、『EJB ホーム・オブジェクトの検索および作成』を参照してください。
2. EJB ホーム・オブジェクトを使用して、bean の EJB オブジェクトのインスタンスを作成 (エンティティー bean の場合のみ) または検索する。詳細については、178ページの『EJB オブジェクトの作成』を参照してください。

TransferApplication クライアントは、Transfer EJB オブジェクトへの参照を 1 つ含みます。アプリケーションはこの参照を使用して、Transfer bean に対するすべてのメソッドを呼び出します。 Java アプリケーションでセッション bean を使用する場合は、EJB オブジェクトへの参照を、メソッドに対してローカルな関係にある変数ではなく、クラス・レベルの変数にすることをお勧めします。これによって、ユーザーの EJB クライアントは同じ EJB オブジェクトに対してメソッドを繰り返し呼び出すことができるようになるため、クライアントがセッション bean メソッドを呼び出すたびに新しいオブジェクトを作成する必要がなくなります。200ページの『スレッド化に関する問題』で説明しているように、複数のスレッドを処理するように設計しなければなりませんが、この方法はサーブレットにはお勧めしません。

## EJB ホーム・オブジェクトの検索および作成

JNDI は、EJB ホーム・オブジェクトの名前を検索するために使用します。EJB クライアントが JNDI の初期化および EJB ホーム・オブジェクトの検索に使用するプロパティーは、EJB サーバーのインプリメンテーションごとに異なります。 EJB サーバーのインプリメンテーションの間で Enterprise Bean を簡単に移送できるようにするには、環境変数のこれらのプロパティー、プロパティー・ファイル、またはリソース・バンドルを Enterprise Bean にハード・コーディングするのではなく、外部で設定するようお勧めします。

Transfer bean の例では、144ページの『ejbCreate メソッドのインプリメント』の説明のように環境変数が使用されています。TransferApplication は、com.ibm.ejs.doc.client.ClientResourceBundle.class ファイルに含まれているリソース・バンドルを使用します。

JNDI ネーム・サービスを初期化するには、EJB クライアントは、以下の JNDI プロパティーに対して適切な値を設定しなければなりません。

**javax.naming.Context.PROVIDER_URL**

このプロパティーは、EJB クライアントが使用するネーム・サーバーのホスト名およびポートを指定します。プロパティー値の形式は iiop://*hostname* :*port* でなければなりません。ここで、*hostname* はネー

ム・サーバーが実行されているマシンの IP アドレスまたはホスト名で、*port* はネーム・サーバーが listen するポート番号です。

たとえば、プロパティー値 `iiop://bankserver.mybank.com:9019` は、ポート 9019 で listen している bankserver.mybank.com という名前のホストでネーム・サーバーを検索するよう EJB クライアントに指示します。プロパティー値 `iiop://bankserver.mybank.com` は、ポート番号 900 にある bankserver.mybank.com という名前のホストでネーム・サーバーを検索するよう EJB クライアントに指示します。プロパティー値 `iiop:///` は、ポート 900 で listen しているローカル・ホストでネーム・サーバーを検索するよう EJB クライアントに指示します。プロパティー値を指定しない場合は、このプロパティーは、デフォルトでローカル・ホストのポート番号 900 になるため、 `iiop:///` を指定するのと同じになります。EJB サーバー (AE) では、ネーム・サービスで使用されるポート番号は、管理インターフェースを使用して変更することができます。

**javax.naming.Context.INITIAL_CONTEXT_FACTORY**

このプロパティーは、EJB クライアントが使用しなければならない実際のネーム・サービスを識別します。

- EJB サーバー (AE) 環境では、このプロパティーを `com.ibm.ejs.ns.jndi.CNInitialContextFactory` に設定しなければなりません。
- EJB サーバー (CB) 環境では、このプロパティーは `com.ibm.ejb.cb.runtime.CBCtxFactory`、そのサブクラスのうちの 1 つ (`com.ibm.ejb.cb.runtime.CBCtxFactoryHostDefault` など)、または **appbind** ツールによって作成された初期コンテキスト・ファクトリーに設定しなければなりません。このコンテキスト・ファクトリーを使用する場合は、javax.naming.Context.list メソッドおよび javax.naming.Context.listBindings メソッドが javax.naming.NamingEnumeration オブジェクトに戻すエレメントは最大 1000 個までになります。 **appbind** ツールの使い方の詳細については、104ページの『アプリケーション固有のコンテキストおよび appbind ツール』を参照してください。

EJB ホーム・オブジェクトは、以下の 2 ステップの処理で検索します。

1. javax.naming.InitialContext オブジェクトを作成する。詳細については、175ページの『InitialContext オブジェクトの作成』を参照してください。

2. InitialContext オブジェクトを使用して、EJB ホーム・オブジェクトを作成する。詳細については、176ページの『EJB ホーム・オブジェクトの作成』を参照してください。

### InitialContext オブジェクトの作成

176ページの図37 は、InitialContext オブジェクトの作成に必要なコードを示しています。このオブジェクトを作成するには、java.util.Properties オブジェクトを作成して Properties オブジェクトに値を追加し、そのオブジェクトを引き数として InitialContext コンストラクターに渡します。 TransferApplication では、各プロパティーの値は com.ibm.ejs.doc.client.ClientResourceBundle というリソース・バンドル・クラスから取得されます。このクラスには、TransferApplication に必要なロケール固有の変数がすべて保管されています。(このクラスには、本書の例に含まれるその他の EJB クライアントが使用する変数も保管されます。307ページの『本書に記載する例の説明』を参照してください。)

リソース・バンドル・クラスは、ResourceBundle.getBundle メソッドを呼び出すことによってインスタンス化されます。リソース・バンドル・クラス内の変数の値は、*bundle* オブジェクトに対して getString メソッドを呼び出すことによって抽出されます。

TransferApplication の createTransfer は、179ページの『セッション bean の無効な EJB オブジェクトの処理』で説明しているように、何度でも呼び出すことができます。しかし、InitialContext オブジェクトは、いったん作成されるとクライアント・セッションの間中有効になります。したがって、InitialContext オブジェクトの作成に必要なコードは、InitialContext オブジェクトがヌルかどうかを判別する if ステートメントに入れます。参照がヌルの場合は InitialContext オブジェクトが作成されます。それ以外の場合は、これ以降の EJB オブジェクトの作成で参照を再利用することができます。

```
...
public class TransferApplication extends Frame implements ActionListener,
     WindowListener {
     ...
     private InitialContext ivjInitContext = null;
     private Transfer ivjTransfer = null;
     private ResourceBundle bundle = ResourceBundle.getBundle(
               "com.ibm.ejs.doc.client.ClientResourceBundle");
     ...
     private String nameService = null;
     private String accountName = null;
     private String providerUrl = null;
     ...
     private Transfer createTransfer() {
          TransferHome transferHome = null;
          Transfer transfer = null;
     // Get the initial context
     if (ivjInitContext == null) {
          try {
               Properties properties = new Properties();
               // Get location of name service
               properties.put(javax.naming.Context.PROVIDER_URL,
                         bundle.getString("providerUrl"));
               // Get name of initial context factory
               properties.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
                         bundle.getString("nameService"));
               ...
               ivjInitContext = new InitialContext(properties);
           } catch (Exception e) { // Error getting the initial context
          ...
          }
     }
     ...
     // Look up the home interface using the JNDI name
     ...
     // Create a new Transfer object to return
     ...
     return transfer;
}
```

*図 37. コード例: InitialContext オブジェクトの作成*

## EJB ホーム・オブジェクトの作成

InitialContext オブジェクト (*ivjInitContext*) が作成されたら、アプリケーションはこれを使用して、178ページの図38 に示すように EJB ホーム・オブジェクトを作成します。これを作成するには、lookup メソッドを呼び出します。このメソッドは、以下のように Enterprise Bean の JNDI 名を String 形式で受け取り、java.lang.Object オブジェクトを戻します。

- EJB サーバー (AE。AIX、Windows NT、または Solaris プラットフォームの場合は CB) に配置された Enterprise Bean に対して JNDI lookup を実行する場合は、デプロイメント・ディスクリプターに指定された JNDI 名だけが使用されます。
- EJB サーバー (AIX、Windows NT、および Solaris 以外のプラットフォームの場合は CB) に配置された Enterprise Bean に対して JNDI lookup を実行する場合、 lookup メソッドに渡される JNDI ホーム名は、Enterprise Bean のデプロイメント・ディスクリプターに指定された JNDI 名に CB 固有の接頭部が付いたものです。この接頭部の内容は、Component Broker ネーム・スペースにおいて、システム管理者が (**ejbbind** ツールを使用して) EJB ホームをバインドした位置によって異なります。

  システム管理者が固有のブートストラップ・ホストのホスト名ツリーに EJB ホームをバインドする場合は、JNDI 名の接頭部は `host/resources/factories/EJBHomes` になります。システム管理者がワークグループ名ツリーに JNDI 名をバインドする場合は、JNDI 名の接頭部は `workgroup/resources/factories/EJBHomes` になるため、EJB クライアントが同じ優先ワークグループに属していなければなりません。システム管理者がセル名ツリーに EJB ホームをバインドする場合は、JNDI 名の接頭部は `cell/resources/factories/EJBHomes` になります。

TransferApplication の例では、Transfer bean の JNDI 名が ClientResourceBundle クラスから取得されます。

lookup メソッドによってオブジェクトが戻されたら、静的メソッド javax.rmi.PortableRemoteObject.narrow を使用して、指定の Enterprise Bean の EJB ホーム・オブジェクトを取得します。 narrow メソッドは、限定するオブジェクトと、narrow メソッドによって戻される EJB ホーム・オブジェクトのクラスの 2 つをパラメーターとして受け取ります。 javax.rmi.PortableRemoteObject.narrow メソッドによって戻されたオブジェクトは、ホーム・インターフェースに関連するクラスにキャストされます。

```
private Transfer createTransfer() {
     TransferHome transferHome = null;
     Transfer transfer = null;
     // Get the initial context
     ...
     // Look up the home interface using the JNDI name
     try {
          java.lang.Object homeObject = ivjInitContext.lookup(
                    bundle.getString("transferName"));
          transferHome = (TransferHome)javax.rmi.PortableRemoteObject.narrow(
                    (org.omg.CORBA.Object) homeObject, TransferHome.class);
     } catch (Exception e)  { // Error getting the home interface
          ...
     }
     ...
     // Create a new Transfer object to return
     ...
     return transfer;
}
```

*図 38. コード例: EJBHome オブジェクトの作成*

## EJB オブジェクトの作成

EJB ホーム・オブジェクトが作成されると、これを使用して EJB オブジェクトが作成されます。 179ページの図39 に、EJB ホーム・オブジェクトを使用して EJB オブジェクトを作成するために必要なコードを示します。 create メソッドを呼び出して EJB オブジェクトを作成する (エンティティー bean の場合のみ) か、あるいは finder メソッドを呼び出して既存の EJB オブジェクトを検索します。 Transfer bean は状態なしセッション bean であるため、デフォルトの create メソッドしか使用できません。

```
private Transfer createTransfer() {
     TransferHome transferHome = null;
     Transfer transfer = null;
     // Get the initial context
     ...
     // Look up the home interface using the JNDI name
     ...
     // Create a new Transfer object to return
     try {
          transfer = transferHome.create();
     } catch (Exception e) { // Error creating Transfer object
          ...
     }
     ...
     return transfer;
}
```

*図 39. コード例: EJB オブジェクトの作成*

## セッション bean の無効な EJB オブジェクトの処理

セッション bean は一時的なものであるため、クライアントは EJB オブジェクトを常に有効なものにするためにセッション bean に依存することはできません。 EJB サーバーが故障したり再始動されたりした場合や、セッション bean が非活動化されてタイムアウトになった場合には、セッション bean の EJB オブジェクトへの参照が無効になる可能性があります。 (エンティティー bean の EJB オブジェクトへの参照は、該当のオブジェクトが除去されない限り常に有効です。) したがって、セッション bean のクライアントには、EJB オブジェクトが無効になった状態を処理するコードがなければなりません。

EJB クライアントは、EJB オブジェクトが有効であるかどうかを判別することができます。これを行うには、メソッドが処理する必要があるすべての他の例外に加えて、java.rmi.NoSuchObjectException を受け取る try/catch ブロック内に、メソッドの呼び出しを設定します。それから、EJB クライアントは、この例外を処理するコードを呼び出すことができます。

無効な EJB オブジェクトの処理方法は、ユーザーが決定します。 TransferApplication の例では、現在使用中のオブジェクトが無効になった場合には、新しい Transfer EJB オブジェクトが作成されます。

古い EJB オブジェクトが無効になった場合に新しいものを作成するためのコードは、元の EJB オブジェクトを作成するために使用するコードと同じです。このコードについては、

172ページの『bean の EJB オブジェクトへの参照の作成および取得』で説明しています。 TransferApplication クライアントの例では、このコードは、createTransfer メソッドに含まれています。

図40 に、TransferApplication の例の、新しい EJB オブジェクトの作成に使用する getBalance メソッドのコードを示します。 getBalance メソッドにはローカル・ブール変数 *sessionGood* が含まれています。この変数は、変数 *ivjTransfer* によって参照される EJB オブジェクトが有効かどうかを指定するために使用されます。また、*sessionGood* 変数を使用して、do-while ループから抜け出る時点も判別されます。

*sessionGood* 変数は、false に初期化されます。getBalance メソッドの呼び出し時に *ivjTransfer* が無効な EJB オブジェクトを参照しないようにするためです。 *ivjTransfer* 参照が有効な場合は、TransferApplication は Transfer bean の getBalance メソッドを呼び出して、残高を戻します。*ivjTransfer* 参照が無効な場合は、NoSuchObjectException が検出され、TransferApplication の createTransfer メソッドが呼び出されて新しい Transfer EJB オブジェクト参照が作成されます。それから、*sessionGood* 変数が false に設定され、その新しい EJB オブジェクトで do-while ループが繰り返されます。ループが無限に繰り返されるのを避けるために、他の例外が throw されると *sessionGood* 変数が true に設定されます。

```
private float getBalance(long acctId) throws NumberFormatException, RemoteException,
     FinderException {
     // Assume that the reference to the Transfer session bean is no good
     ...
     boolean sessionGood = false;
     float balance = 0.0f;
     do {
     try {
               // Attempt to get a balance for the specified account
              balance = ivjTransfer.getBalance(acctId);
              sessionGood = true;
              ...
          } catch(NoSuchObjectException ex) {
              createTransfer();
              sessionGood = false;
          } catch(RemoteException ex) {
              // Server or connection problem
              ...
          } catch(NumberFormatException ex) {
              // Invalid account number
              ...
          } catch(FinderException ex) {
              // Invalid account number
              ...
          }
     } while(!sessionGood);
     return balance;
}
```

*図 40. コード例: セッション bean の EJB オブジェクト参照のリフレッシュ*

## bean の EJB オブジェクトの除去

EJB クライアントに状態付きセッション EJB オブジェクトが不要になった場合は、 EJB クライアントはそのオブジェクトを除去しなければなりません。状態付きセッション bean のインスタンスは、特定のクライアントとの類縁性があります。これらのインスタンスは、クライアントで明示的に除去するまで、またはタイムアウト時にコンテナーで除去するまで、コンテナーに残ります。同時に、コンテナーは、非アクティブな状態付きセッション bean を非活性化してディスクに入れておくことが必要になります。これには、コンテナーのオーバーヘッドが必要であり、アプリケーションのパフォーマンスに影響が及びます。その後、非活性化状態のセッション bean がアプリケーションで要求されると、コンテナーは、ディスクからその bean を復元してアクティブ化します。使用し終えたときは状態付きセッション bean を明示的に除去することによって、アプリケーションは、非活性化の必要性を削減し、コンテナーのオーバーヘッドを最小限に抑えることができます。

エンティティー EJB オブジェクトを除去するのは、エンティティー EJB オブジェクトに関連するデータ・ソース内の情報をユーザーが除去したい場合だけです。

EJB オブジェクトを除去するには、オブジェクトに対して remove メソッドを呼び出します。172ページの『bean の EJB オブジェクトへの参照の作成および取得』で説明しているように、TransferApplication は、アプリケーションの初期化時に作成された Transfer EJB オブジェクトへの参照しか含んでいません。

182ページの図41 に、killApp メソッドにおいて、TransferApplication で Transfer EJB オブジェクトの例が除去される様子を示します。TransferApplication の初期化時に Transfer EJB オブジェクトを並行して作成させるために、アプリケーションの GUI ウィンドウがクローズされる直前に、*ivjTransfer* 参照に関連する最後の EJB オブジェクトをアプリケーションが除去します。 killApp メソッドは、これ自体に対して dispose メソッドを呼び出して、ウィンドウをクローズします。

```
...
private void killApp() {
     try {
          ivjTransfer.remove();
          this.dispose();
          System.exit(0);     } catch (Throwable ivjExc) {
          ...
     }
}
```

*図 41. コード例: セッション EJB オブジェクトの除去*

## EJB クライアントでのトランザクションの管理

一般に、Enterprise Bean を設計する際に、すべてのトランザクション管理が Enterprise Bean レベルで操作されるようにすると便利です。完全 3 層の分散アプリケーションでは、このような設計は必ずしも可能で望ましいものではありません。しかし、EJB アプリケーションの中央の層にはセッション bean とエンティティー bean の 2 つのサブコンポーネントを含めることができるため、トランザクション管理を完全にアプリケーション・サーバー層内で設計する方が簡単です。もちろん、リソース管理プログラム層もトランザクションをサポートするように設計しなければなりません。

**注:** EJB クライアントは、CICS または IMS アプリケーションに Host On-Demand (HOD) または外部呼び出しインターフェース (ECI) を使用する CMP を持つエンティティー bean にアクセスする場合、これらのエンティティー bean に対してメソッドを呼び出す前に、トランザクションを開始しなければなりません。これは、これらのタイプのエンティティー bean が TX_MANDATORY トランザクション属性を使用しなければならないためです。

このような場合でも、特殊な状況においてクライアントがトランザクションを操作するように、(Enterprise Bean ではない) EJB クライアントをプログラムすることができます。トランザクションを操作するには、EJB クライアントは以下を実行しなければなりません。

1. Java Transaction Application Programming Interface (JTA) での定義に従って、JNDI を使用して javax.transaction.UserTransaction インターフェースへの参照を取得する。
2. このオブジェクト参照を使用して、以下のメソッドをすべて呼び出す。

- begin ー トランザクションを開始する。このメソッドは引き数を受け取らず、戻り値もありません (void)。
- commit ー トランザクションのコミットを試みる。トランザクションをロールバックさせる要因がなければ、このメソッドが正常に完了するとトランザクションがコミットされます。このメソッドは引き数を受け取らず、戻り値もありません (void)。
- getStatus ー 参照先のトランザクションの状況を戻す。このメソッドは引き数を受け取らず、int を戻します。参照に関連するトランザクションがない場合は、STATUS_NO_TRANSACTION を戻します。このメソッドに有効な戻り値は、以下のとおりです。
  - STATUS_ACTIVE ー トランザクション処理が進行中であることを示す。
  - STATUS_COMMITTED ー トランザクションがコミットされ、そのトランザクションへの変更が永久に有効になったことを示す。
  - STATUS_COMMITTING ー トランザクションがコミット中である (つまり、トランザクションのコミットは開始されたが処理は完了していない) ことを示す。
  - STATUS_MARKED_ROLLBACK ー ロールバックを行うトランザクションとしてマークが付けられていることを示す。
  - STATUS_NO_TRANSACTION ー 現行のトランザクション・コンテキストにトランザクションが存在しないことを示す。
  - STATUS_PREPARED ー トランザクションが準備はされたが完了していないことを示す。
  - STATUS_PREPARING ー トランザクションが準備中である (つまり、トランザクションの準備は開始されたが処理は完了していない) ことを示す。
  - STATUS_ROLLEDBACK ー トランザクションがロールバックされたことを示す。
  - STATUS_ROLLING_BACK ー トランザクションがロールバック中である (つまり、トランザクションのロールバックは開始されたが処理は完了していない) ことを示す。
  - STATUS_UNKNOWN ー トランザクションの状況が不明であることを示す。
- rollback ー 参照先のトランザクションをロールバックする。このメソッドは引き数を受け取らず、戻り値もありません (void)。

- setRollbackOnly ー 出力可能なトランザクションのみがロールバックされることを指定する。このメソッドは引き数を受け取らず、戻り値もありません (void)。
- setTransactionTimeout ー トランザクションに関連するタイムアウトを (秒単位で) 設定する。トランザクションの参加者が特にこの値を設定していない場合は、デフォルトのタイムアウトが使用されます。このメソッドは (int 型で) 秒数を受け取り、戻り値はありません (void)。

図42 の例では、EJB クライアントは、UserTransaction オブジェクトへの参照を作成してから、そのオブジェクトを使用してトランザクションのタイムアウトを設定し、トランザクションを開始してトランザクションをコミットしようとしています。 (この例のソース・コードは、本書に記載されているコード例で使用することはできません。) このクライアントは lookup の結果に対して簡単な型キャストを実行しており、その他の JNDI lookup が必要とする narrow メソッドは呼び出していません。どちらの EJB サーバー環境でも、UserTransaction インターフェースの JNDI 名は `java:comp/UserTransaction` です。

```
...
import javax.transaction.*;
...
// Use JNDI to locate the UserTransaction object
Context initialContext = new InitialContext();
UserTransaction tranContext = (
     UserTransaction)initialContext.lookup("java:comp/UserTransaction");
// Set the transaction timeout to 30 seconds
tranContext.setTransactionTimeout(30);
...
// Begin a transaction
tranContext.begin();
// Perform transaction work invoking methods on enterprise bean references
...
// Call for the transaction to commit
tranContext.commit();
```

*図 42. コード例: EJB クライアントでのトランザクションの管理*

## EJB サーバー (CB) 固有の EJB クライアントに関する追加情報

EJB サーバー (CB) 環境において EJB クライアントを開発する場合には、以下のタイプのクライアントを開発することができます。

- Microsoft ActiveX クライアント。一般情報については、『ActiveX を使用する EJB クライアント』を参照してください。
- Component Broker のセッション・サービスを使用するクライアント。一般情報については、『Component Broker のセッション・サービスを使用するクライアント』を参照してください。

これらのタイプのクライアントの開発に関する詳細については、IBM Redbook の「*IBM Component Broker Connector Overview*」(資料番号 SG24-2022-02) を参照してください。

## ActiveX を使用する EJB クライアント

JavaBeans™ 仕様に準拠したコンポーネントとして EJB クライアントを作成する場合は、JavaBeans ブリッジを使用して、EJB クライアントを ActiveX コントロールとして実行することができます。このタイプの EJB クライアントは、引き数のないコンストラクターを提供しなければなりません。また、java.io.Serializable インターフェースをインプリメントし、 (適用できる場合は) readObject と writeObject メソッドを持たなければなりません。

EJB クライアントがアプレットでもある場合は、オブジェクトの構造体の一部として JNDI の初期化を実行することはできません。代わりに、アプレットの start メソッドで JNDI の初期化を実行します。 JavaBeans ブリッジは、EJB クライアントのインスタンスを作成して、ブリッジがそれ自体を内省して、ブリッジ用の ActiveX プロキシーの作成に必要なスタブを作成することができるようにします。ユーザーは、ActiveX プロパティー・シートを介して必要なプロパティーが指定できるようになるまで、JNDI 接続を遅らせなければなりません。

## Component Broker のセッション・サービスを使用するクライアント

Component Broker は、トランザクション・サービスのほかに、CICS や IMS などのバックエンド・システムを使用できるようにする Procedural Application Adaptor (PAA) のための、セッション・サービスも提供します。JTA にはセッション・サービスがないので、JNDI を使用して EJB クライアント内のセッション・サービスへのハンドルを検索することはできません。この場合、EJB クライアントは通常の CB Java クライアントとして機能しなければなりません。

CB Java クライアントに対する通常の検索手順では、CORBA の resolve_initial_references メソッドを使用します。この場合、検索対象の CORBA オブジェクトは SessionCurrent という名前になります。

resolve_initial_references メソッドを呼び出せるようにするには、ORB は、CB 実行時環境に対して適宜初期化しておく必要があります。initialization メソッドは、CB 環境で VisualAge for Java アクセス bean を使用しているかどうかによって決まります。アクセス bean を使用している場合、ORB は手動で初期化しなければなりません。アクセス bean の ORB 初期化は、″遅延″ 方式で行われます。つまり、初期化は最初のリモート・メソッドが呼び出されて初めて実行されます。しかし、セッションはそのメソッドを呼び出す前に開始していなければならないため、ORB 初期化は手動で行う必要があります。図43 のコード例では、この初期化について示しています。

```
String[] CBargs = null;
CBargs = new String[6];
CBargs[0] = "-ORBBootstrapHost";
// substitute your bootstrap host name
CBargs[1] = "cbs3.rchland.ibm.com";
CBargs[2] = "-ORBBootstrapPort";
CBargs[3] = "900";
CBargs[4] = "-ORBClass";
CBargs[5] = "com.ibm.CORBA.iiop.ORB";
com.ibm.CBCUtil.CBSeriesGlobal.Initialize(CBargs);
```

*図 43. コード例: ORB の初期化 (アクセス bean を使用している場合)*

アクセス bean を使用していない場合、初期化コードは必要ありません。ORB は、適切なプロパティーにより、InitialContext オブジェクトの作成時に適宜初期化されます。たとえば、クライアント・コードには、図44 にある行と同様の行が常に含まれている必要があります。このコードを使用して、サービスの検出、ホーム・オブジェクトの検索、ホーム・オブジェクトの限定、およびプロキシー・オブジェクトの作成を行います (アクセス bean を使用している場合、これらの作業は自動的に実行されます)。

```
Properties properties = new Properties();
properties.put(javax.naming.Context.PROVIDER_URL, "iiop:///");
// CB Factory Name
properties.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
"com.ibm.ejb.cb.runtime.CBCtxFactory");
Context ctx = new InitialContext(properties);
```

*図 44. コード例: InitialContext オブジェクトの作成 (アクセス bean を使用していない場合)*

ORB の初期化 (自動または手動) が完了すると、 sessionCurrent オブジェクトを作成および使用するために、CB 固有の API を使用しなければなりません。187ページの図45 にあるコード例と同様のコードを組み込む必要があります。

```
org.omg.CORBA.Object orbCurrent = null;
com.ibm.ISessions.Current sessionCurrent = null;
...
orbCurrent = com.ibm.CBCUtil.CBSeriesGlobal.orb().resolve_initial_references(
"ISessions::Current");
sessionCurrent = com.ibm.ISessions.CurrentHelper.narrow(orbCurrent);
sessionCurrent.beginSession("myApp");
...
// commit
sessionCurrent.endSession(com.ibm.ISessions.EndMode.EndModeCheckPoint, true);
```

*図 45. コード例: sessionCurrent オブジェクトの作成および使用*

resolve_initial_references メソッドの使い方に関する詳細については、Component Broker の「プログラミング・ガイド」を参照してください。

# 第8章 Enterprise Bean を使用するサーブレットの開発

サーブレットは、Web サーバーの機能にアクセスできるようにする Java アプリケーションです。サーブレットを使用するには、Web サーバーが必要です。WebSphere Application Server は、一般に使用されているいくつかの Web サーバーに組み込まれます。さらに、WAS アドバンスド版と WAS エンタープライズ版の両方で、IBM HTTP Web サーバーを使用することができます。詳細については、アドバンスド版の InfoCenter を参照してください。

Java サーブレットを Enterprise Bean と組み合わせると、強力な EJB アプリケーションを作成することができます。本章では、サーブレット内で Enterprise Bean を使用する方法について説明します。 CreateAccount サーブレットの例では、Account bean の例を使用しており、本章で説明している概念を具体的に示しています。本章で扱うサーブレット例および Enterprise Bean については、307ページの『本書に記載する例の説明』で説明しています。

## 標準のサーブレット・メソッドの概要

通常、サーブレットは、ユーザーのブラウザーにある HTML フォームから呼び出されます。最初にサーブレットが呼び出されるときには、サーブレットの init メソッドが実行され、起動時に必要なすべての初期化を行います。サーブレットの最初の呼び出しに加え、その後のすべての呼び出しにおいて、doGet メソッド (または代わりに doPost メソッド) が実行されます。 doGet メソッド (または doPost メソッド) では、サーブレットは、ユーザーによって HTML フォームに提供された情報を取得し、その情報を使用して、サーバーで処理を実行したりサーバー・リソースにアクセスしたりします。

その後、サーブレットは、応答を作成してユーザーに送信します。サーブレットは、ロードされると、同時に複数のユーザー要求を処理することができます。多重要求スレッドは、doGet (または doPost) メソッドを同時に呼び出すことができるので、サーブレットはスレッド・セーフにする必要があります。

サーブレットが終了すると、そのサーブレットの destroy メソッドが実行され、必要な終了処理が行われます。

© Copyright IBM Corp. 1999, 2001

## サーブレットを組み込んだ HTML ページの作成

図46 に、CreateAccount サーブレットを呼び出すために使用する HTML ファイル (名前は create.html) を示します。HTML フォームは、新規口座の口座番号、口座の種類 (当座預金か普通預金か)、および初期残高を指定するために使用されます。要求はサーブレットの doGet メソッドに渡されます。ここで、例に示すように、サーブレットは完全な Java パッケージ名で識別されます。

```
<html>
<head>
<title>Create a new Account</title>
</head>
<body>
<h1 align="center">Create a new Account</h1>
<form method="get"
action="/servlet/com.ibm.ejs.doc.client.CreateAccount">
<table border align="center">
<!-- specify a new account number -->
<tr bgcolor="#cccccc">
<td align="right">Account Number:</td>
<td colspan="2"><input type="text" name="account" size="20"
maxlength="10">
</tr>
<!-- specify savings or checking account -->
...
<!-- specify account starting balance -->
...
<!-- submit information to servlet -->
...
<input type="submit" name ="submit" value="Create">
...
<!-- message area -->
...
</form>
</body>
</html>
```

*図 46. コード例: CreateAccount サーブレットにアクセスするために使用する create.html ファイルのコンテンツ*

サーブレットからの HTML 応答は、create.html と同じ表示になるように設計して、ユーザーが引き続き新規口座を作成できるようにします。
191ページの図47 に、ブラウザーでの create.html の表示を示します。

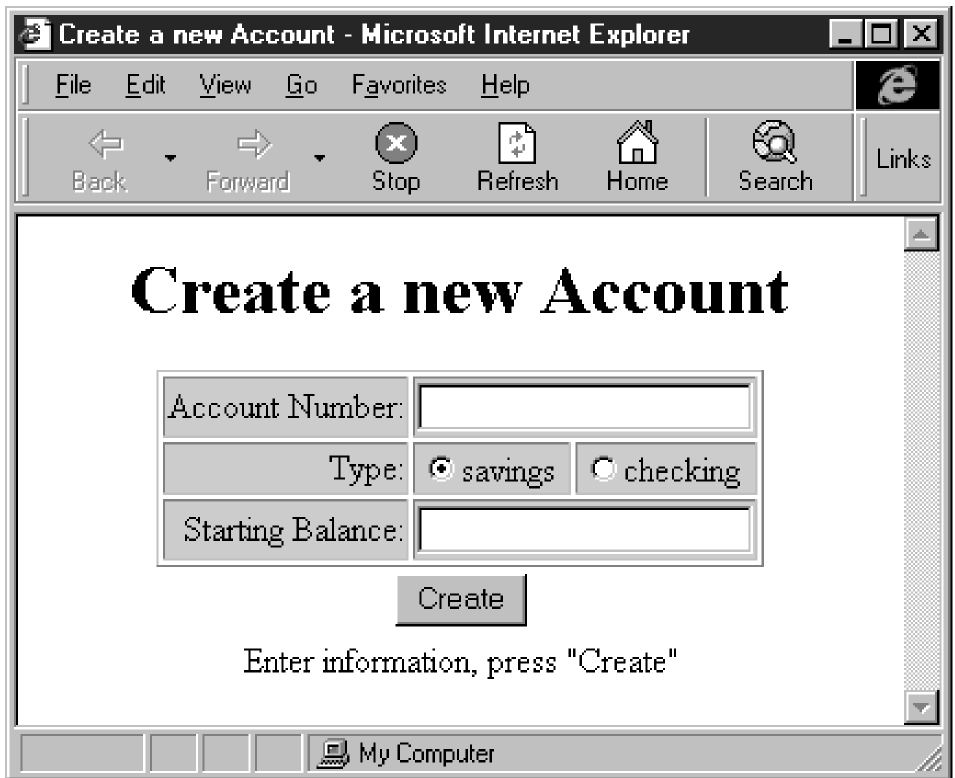



*図 47. CreateAccount サーブレットの初期フォームおよび出力*

## サーブレットの開発

本セクションでは、Enterprise Bean と対話するサーブレットに必要な基本的なコードについて説明します。192ページの図48 に、CreateAccount サーブレットを構成するコードの大枠を示します。例に示すように、CreateAccount サーブレットは、javax.servlet.http.HttpServlet クラスを拡張し、init メソッドおよび doGet メソッドをインプリメントしています。

```
package com.ibm.ejs.doc.client;
// General enterprise bean code.
import java.rmi.RemoteException;
import javax.ejb.DuplicateKeyException;
// Enterprise bean code specific to this servlet.
import com.ibm.ejs.doc.account.AccountHome;
import com.ibm.ejs.doc.account.AccountKey;
import com.ibm.ejs.doc.account.Account;
// Servlet related.
import javax.servlet.*;
import javax.servlet.http.*;
// JNDI (naming).
import javax.naming.*;  // for Context, InitialContext, NamingException
// Miscellaneous:
import java.util.*;
import java.io.*;
...
public class CreateAccount extends HttpServlet {
   // Variables
   ...
   public void init(ServletConfig config) throws ServletException {
      ...
   }
   public void doGet(HttpServletRequest req, HttpServletResponse res)
   throws ServletException, IOException {
      // --- Read and validate user input, initialize. ---
      ...
      // --- If input parameters are good, try to create account. ---
      ...
      // --- Prepare message to accompany response. ---
      ...
      // --- Prepare and send HTML response. ---
      ...
}
```

*図 48. コード例: CreateAccount クラス*

## サーブレットのインスタンス変数

193ページの図49 に、CreateAccount サーブレットで使用されているインスタンス変数を示します。*nameService*、*accountName*、および *providerUrl* の各変数は、JNDI 検索中に必要なプロパティー値を指定するために使用します。172ページの『bean の EJB オブジェクトへの参照の作成および取得』で説明しているように、これらの値は ClientResourceBundle クラスから取得します。

CreateAccount クラスは、ユーザーに応答する HTML 応答を作成するために使用するストリング定数も初期化します。 (3 つの変数しか示していませんが、実際には多くの変数があります)。CreateAccount サーブレットの init メソッド

によって、リソース・バンドルからストリングを読み取り、これらの米国英語のデフォルトを上書きして各国語の応答を提供することができます。

インスタンス変数 *accountHome* は、新しい Account bean インスタンスを作成するために、すべてのクライアント要求によって使用されます。*accountHome* 変数は、図49 に示すように、init メソッドで初期化されます。

```
...
public class CreateAccount extends HttpServlet {
     // Variables for finding the home
     private String nameService = null;
     private String accountName = null;
     private String providerURL = null;
     private ResourceBundle bundle =  ResourceBundle.getBundle(
               "com.ibm.ejs.doc.client.ClientResourceBundle");
     // Strings for HTML output - US English defaults shown.
     static String title = "Create a new Account";
     static String number  = "Account Number:";
     static String type  = "Type:";
     ...
     // Variable for accessing the enterprise bean.
     private AccountHome accountHome = null;
     ...

}
```

*図 49. コード例: CreateAccount クラスのインスタンス変数*

## サーブレットの init メソッド

CreateAccount サーブレットの init メソッドを 195ページの図50 に示します。init メソッドは、サーブレットが開始されたあと、最初の要求がサーブレットによって処理されるときに、1 回だけ実行されます。通常、init メソッドは、サーブレットの一回限りの初期化を行うために使用します。たとえば、HTML 応答を作成する際に使用されるデフォルトの米国英語のストリングを、各国語で置換することができます。

init メソッドは、サーブレットが使用する Enterprise Bean のホーム・インターフェースへの参照の値を初期化する場所としても最適です。 CreateAccount の init メソッドでは、*accountHome* 変数を初期化して、Account bean の EJB ホーム・オブジェクトを参照させています。

他の種類の EJB クライアントの場合と同様に、JNDI 検索に必要なプロパティーは EJB インプリメンテーションに固有です。したがって、これらのプロパティーは、プロパティー・ファイルまたはリソース・バンドル・クラスで外部

的に設定します。これらのプロパティーに関する詳細については、172ページの『bean の EJB オブジェクトへの参照の作成および取得』を参照してください。

CreateAccount サーブレットでは、JNDI 検索を行うために必要なプロパティーを保管するために HashTable オブジェクトが使用され、TransferApplication では、Properties オブジェクトが使用されることに注意してください。これらのプロパティーの保管では、どちらのクラスも有効です。

```
// Variables for finding the EJB home object
private String nameService = null;
private String accountName = null;
private String providerURL = null;
private ResourceBundle bundle = ResourceBundle.getBundle(
          "com.ibm.ejs.doc.client.TransferResourceBundle");
...
public void init(ServletConfig config) throws ServletException {
     super.init(config);
     ...
    try {
         // Get NLS strings from an external resource bundle
          ...
          createTitle = bundle.getString("createTitle");
          number = bundle.getString("number");
          type = bundle.getString("type");
          ...
          //Get values for the naming factory and home name.
          nameService = bundle.getString("nameService");
          accountName = bundle.getString("accountName");
          providerURL = bundle.getString("providerURL");
     }
     catch (Exception e) {
         ...
     }
     // Get home object for access to Account enterprise bean.
     Hashtable env = new Hashtable();
     env.put(Context.INITIAL_CONTEXT_FACTORY, nameService);
     try {
          // Create the initial context.
          Context ctx = new InitialContext(env);
          // Get the home object.
          Object homeObject = ctx.lookup(accountName);
          // Get the AccountHome object.
          accountHome = (AccountHome) javax.rmi.PortableRemoteObject.narrow(
                    (org.omg.CORBA.Object)homeObject, AccountHome.class);
     }
     // Determine cause of failure.
     catch (NamingException e) {
          ...
      }
      catch (Exception e) {
          ...
     }
}
```

*図 50. コード例: CreateAccount サーブレットの init メソッド*

## サーブレットの doGet メソッド

doGet メソッドは、サーブレット要求ごとに呼び出されます。CreateAccount サーブレットでは、このメソッドは、以下の作業を実行してユーザー入力を管理します。これらの作業は、このメソッドに典型的なものです。

- HTML フォームからユーザー入力を読み取り、入力が有効かどうか (ユーザーが適切な初期残高値を入力したかどうかなど) を検査する。
- 各要求に対して要求された初期化を行う。

197ページの図51 に、ユーザー入力を処理する doGet メソッドの部分を示します。*req* 変数は、HTML フォームからユーザー入力を読み取るために使用します。*req* 変数は、doGet メソッドへの引き数の 1 つとして渡された javax.servlet.http.HttpServletRequest オブジェクトです。

```
public void doGet (HttpServletRequest req, HttpServletResponse res)
     throws ServletException, IOException {
     // --- Read and validate user input, initialize. ---
     // Error flags.
     boolean accountFlag  = true;
     boolean balanceFlag  = true;
     boolean inputFlag = false;
     boolean createFlag  = true;
     boolean duplicateFlag = false;
     // Datatypes used to create new account bean.
     AccountKey key;
     int typeAcct  = 0;
     String typeString = "0";
     float initialBalance = 0;
     // Read input parameters from HTML form.
     String[] accountArray = req.getParameterValues("account");
     String[] typeArray = req.getParameterValues("type");
     String[] balanceArray = req.getParameterValues("balance");
     // Convert input parameters to needed datatypes for new account.
     // (account)
     long accountLong = 0;
     ...
     key = new AccountKey(accountLong);
     // (type)
     if (typeArray[0].equals("1")) {
          typeAcct   = 1;           // Savings account.
          typeString = "savings";
     }
     else if (typeArray[0].equals("2")) {
          typeAcct   = 2;           // Checking account
          typeString = "checking";
     }
     // (balance)
     try {
          initialBalance = (Float.valueOf(balanceArray[0])).floatValue();
     } catch (Exception e) {
          balanceFlag = false;
     }
     ...
      // --- If input parameters are good, try to create account bean. ---
     ...
     // --- Prepare message to accompany response. ---
     ...
     // --- Prepare and send HTML response. ---
     ...
}
```

*図 51. コード例: CreateAccount サーブレットの doGet メソッド*

## Enterprise Bean の作成

ユーザー入力が有効な場合は、 図52 に示すように、doGet メソッドは、ユーザー入力に基づいて新規口座を作成しようとします。init メソッドでのホーム・オブジェクト参照の初期化とここで紹介する処理が、サーブレットで Enterprise Bean を使用する場合に固有のコードです。

```
public void doGet(HttpServletRequest req, HttpServletResponse res)
     throws ServletException, IOException {
     // --- Read and validate user input, initialize ---.
     ...
     // --- If input parameters are good, try to create account bean. ---
     if (accountFlag && balanceFlag) {
          inputFlag = true;
          try {
               // Create the bean.
               Account account = accountHome.create(key, typeAcct, initialBalance);
          }
          // Determine cause of failure.
          catch (RemoteException e) {
               ...
          }
          catch (DuplicateKeyException e) {
                ...
         }
         catch (Exception e) {
               ...
          }
     }
     // --- Prepare message to accompany response. ---
     ...
     // --- Prepare and send HTML response. ---
     ...
 }
```

*図 52. コード例: doGet メソッドでの Enterprise Bean の作成*

## ユーザー応答の内容の決定

次に、doGet メソッドは、ユーザーに送信する応答メッセージを作成します。3 つの応答が可能です。

- ユーザー入力が無効である。
- ユーザー入力は有効であったが、なんらかの理由によって口座が作成されなかった。
- 口座が正常に作成された。前記の 2 つのエラーが発生しない場合は、この応答が作成されます。

199ページの図53 に、ユーザーに送信する応答を決定するためにサーブレットが使用するコードを示します。エラーが発生しなかった場合は、成功したことが応答に示されます。

```
public void doGet(HttpServletRequest req, HttpServletResponse res)
     throws ServletException, IOException {
      // --- Read and validate user input, initialize. ---
      ...
      // --- If input parameters are good, try to create account bean. ---
      ...
      // --- Prepare message to accompany response. ---
      ...
    String messageLine  = "";
    if (inputFlag) {
          // If you are here, the client input is good.
          if (createFlag) {
          // New account enterprise bean was created.
          messageLine =  createdaccount + " " + accountArray[0] + ", " +
                    createdtype + " " + typeString + ", " +
                    createdbalance + " " + balanceArray[0];
          }
          else if (duplicateFlag) {
               // Account with same key already exists.
               messageLine = failureexists + " " + accountArray[0];
          }
          else {
          // Other reason for failure.
               messageLine = failureinternal + " " + accountArray[0];
          }
     }
     else {
          // If you are here, something was wrong with the client input.
          String separator = "";
          if (!accountFlag) {
               messageLine = failureaccount + " " + accountArray[0];
               separator = ", ";
     }
     if (!balanceFlag) {
          messageLine = messageLine + separator +
                    failurebalance + " " + balanceArray[0];
     }
     // --- Prepare and send HTML response. ---
     ...
 }
```

*図 53. コード例: doGet メソッドでのユーザー応答の決定*

## ユーザー応答の送信

応答の種類が決まると、doGet メソッドは、適切なメッセージを組み込んで完全な HTML 応答を作成し、ユーザーのブラウザーに戻します。完全な HTML 応答のうち、関係のある部分を 200ページの図54 に示します。

*res* 変数は、ユーザーに戻す応答を渡すために使用します。この変数は、doGet メソッドに引き数として渡された HttpServletResponse オブジェクトです。ここに示す応答コードには、表示 (HTML) とサーブレット内のコンテンツの両方が混在しています。 JavaServer Pages (JSP) を使用することによって、表示とコンテンツを分離することができます。JSP によって、表示とコンテンツを別々に開発および保守することができます。

```
public void doGet(HttpServletRequest req, HttpServletResponse res)
     throws ServletException, IOException {
      // --- Read and validate user input, initialize. ---
      ...
      // --- If input parameters are good, try to create account bean. ---
      ...
      // --- Prepare message to accompany response. ---
      ...
     // --- Prepare and send HTML response. ---
     // HTML returned looks like initial HTML that invoked this servlet.
     // Message line says whether servlet was successful or not.
     res.setContentType("text/html");
     res.setHeader("Pragma", "no-cache");
     res.setHeader("Cache-control", "no-cache");
     PrintWriter out = res.getWriter();
     out.println("<html>");
     ...
     out.println("<title> " + createTitle + "</title>");
     ...
     out.println(" </html>");
}
```

*図 54. コード例: doGet メソッドでのユーザーへの応答*

## スレッド化に関する問題

Account bean のホーム・インターフェースへの参照を取得するために必要なインスタンス変数、および複数の国語をサポートするために必要な変数 (すべてのユーザー要求について不変です) を除いて、CreateAccount サーブレットで使用されている他の変数は、すべて doGet メソッドにローカルです。各要求スレッドはそれ自体の一連のローカル変数を持つので、サーブレットは、同時に複数のユーザー要求を処理することができます。

結果として、CreateAccount サーブレットはスレッド・セーフです。同様の方法でサーブレットを設計することによって、独自に作成したサーブレットもスレッド・セーフにすることができます。

# 第9章 Enterprise Bean に関する高度なプログラミングの概念

本章では、Enterprise Bean の開発および使用に関連する高度なプログラミングの概念について説明します。ここでは、bean 管理のパーシスタンス (BMP) を持つエンティティー bean の開発、BMP bean がデータベースとの対話に必要とするコードの作成、] およびトランザクションに直接関係するセッション bean の開発について説明しています。

## BMP を持つエンティティー bean の開発

コンテナー管理のパーシスタンス (CMP) を持つエンティティー bean では、コンテナーが Enterprise Bean とデータ・ソースの間の対話を処理します。bean 管理のパーシスタンス (BMP) を持つエンティティー bean では、Enterprise Bean は、Enterprise Bean とデータ・ソースの間の対話に必要なすべてのコードを含んでいなければなりません。このような理由から、CMP を持つエンティティー bean の開発は、BMP を持つエンティティー bean の開発よりも単純です。しかし、エンティティー bean が以下に該当する場合には、BMP を使用しなければなりません。

- bean の永続データが複数のデータ・ソースに格納される。
- 使用している EJB サーバーがサポートしていないデータ・ソースに bean の永続データが格納される。

本セクションでは、BMP を持つエンティティー bean の開発について説明します。 CMP を持つエンティティー bean の開発に必要な作業の詳細については、118ページの『CMP を持つエンティティー bean の開発』を参照してください。

エンティティー bean は、以下の基本パーツを含まなければなりません。

- Enterprise Bean クラス。詳細については、202ページの『Enterprise Bean クラスの作成 (BMP を持つエンティティー)』を参照してください。
- Enterprise Bean のホーム・インターフェース。詳細については、214ページの『ホーム・インターフェースの作成 (BMP を持つエンティティー)』を参照してください。
- Enterprise Bean のリモート・インターフェース。詳細については、216ページの『リモート・インターフェースの作成 (BMP を持つエンティティー)』を参照してください。

© Copyright IBM Corp. 1999, 2001

BMP を持つエンティティー bean では、ユーザー独自の 1 次キー・クラスを作成したり、1 次キーに既存のクラスを使用したりすることができます。詳細については、218ページの『1 次キー・クラスの作成または選択 (BMP を持つエンティティー)』を参照してください。

## Enterprise Bean クラスの作成 (BMP を持つエンティティー)

BMP を持つエンティティー bean では、bean クラスは、Enterprise Bean のビジネス・メソッドを定義およびインプリメントし、Enterprise Bean のインスタンスを作成するために使用するメソッドを定義およびインプリメントし、bean のライフ・サイクルの各段階にわたって bean を移動するためにコンテナーによって呼び出されるメソッドをインプリメントします。

規則では、Enterprise Bean クラスには *Name*Bean という名前が付けられます。ここで、*Name* は Enterprise Bean に割り当てた名前です。 AccountBM Enterprise Bean の例に対する Enterprise Bean クラスには、AccountBMBean という名前が付いています。

BMP を持つエンティティー bean クラスは、いずれも以下の要件を満たさなければなりません。

- パブリックでなければならず、抽象クラスであってはならず、javax.ejb.EntityBean インターフェースをインプリメントしなければなりません。詳細については、212ページの『EntityBean インターフェースのインプリメント』を参照してください。
- Enterprise Bean に関連する永続データに対応するインスタンス変数を定義しなければなりません。詳細については、203ページの『インスタンス変数の定義』を参照してください。
- Enterprise Bean に関連するデータのアクセスおよび操作に使用するビジネス・メソッドをインプリメントしなければなりません。詳細については、206ページの『ビジネス・メソッドのインプリメント』を参照してください。
- 永続データの格納に使用するデータ・ソース (複数の場合もある) に対して接続の取得、対話、および接続の解放を行うコードを含まなければなりません。詳細については、219ページの『BMP エンティティー bean によるデータベースの使用』を参照してください。
- Enterprise Bean をインスタンス化可能にするための方法ごとに、ejbCreate メソッドを定義およびインプリメントしなければなりません。各 ejbCreate メソッドについて、対応する ejbPostCreate メソッドの定義およびインプリメントを行うこともできます (行わなくても構いません)。詳細については、206ページの『ejbCreate および ejbPostCreate メソッドのインプリメント』を参照してください。

- 1 次キーを受け取り、有効かつ固有であるかどうかを判別する ejbFindByPrimaryKey メソッドをインプリメントしなければなりません。必要に応じて、追加の finder メソッドを定義およびインプリメントすることもできます。詳細については、208ページの『ejbFindByPrimaryKey およびその他の ejbFind メソッドのインプリメント』を参照してください。

**注:** Enterprise Bean クラスは Enterprise Bean のリモート・インターフェースをインプリメントすることができますが、このインプリメントはお勧めしません。 Enterprise Bean クラスがリモート・インターフェースをインプリメントすると、誤って *this* 変数をメソッド引き数として渡してしまう可能性があります。

図55 に、AccountBM Enterprise Bean の例のための import ステートメントおよびクラス宣言を示します。

```
...
import java.rmi.RemoteException;
import java.util.*;
import javax.ejb.*;
import java.lang.*;
import java.sql.*;
import com.ibm.ejs.doc.account.InsufficientFundsException;
public class AccountBMBean implements EntityBean {
   ...
}
```

*図 55. コード例: AccountBMBean クラス*

## インスタンス変数の定義

エンティティー bean クラスは、永続インスタンス変数と非永続インスタンス変数の両方を含むことができます。しかし、それらが final (つまり定数) でない限り、静的変数は Enterprise Bean ではサポートされません。永続変数は、データベースに格納されます。CMP エンティティー bean クラスにおける永続変数とは異なり、BMP エンティティー bean クラスにおける永続変数はプライベートです。

非永続変数は、データベースには格納されない 一時的なものです。非永続変数は、注意して使用しなければならず、EJB クライアントの状態をメソッド呼び出し間で維持するために使用してはなりません。この制約事項が必要な理由は、EJB クライアントが非永続変数を変更することができ、エンティティー

bean が非活動化されると非永続変数も失われるので、非永続変数がトランザクション外のメソッド呼び出し間で同じままであると想定することができないためです。

AccountBMBean クラスは、AccountBM Enterprise Bean に関連する永続データを表す以下の 3 つのインスタンス変数を含みます。

- *accountId*。口座に関連付けられた口座 ID を識別します。
- *type*。口座の種類が普通預金 (1) であるか当座預金 (2) であるかを識別します。
- *balance*。口座の現在の残高を識別します。

AccountBMBean クラスは、以下のような非永続インスタンス変数を含みます。

- *entityContext*。AccountBM Enterprise Bean の各インスタンスのエンティティー・コンテキストを識別します。エンティティー・コンテキストは、bean インスタンスに現在関連付けられている EJB オブジェクトへの参照を取得したり、その EJB オブジェクトに関連付けられた 1 次キー・オブジェクトを取得したりするために使用することができます。
- *jdbcUrl*。データ・ソースへの接続に使用されるデータベース URL (universal resource locator) をカプセル化します。この変数は、*dbAPI* :*databaseType* :*databaseName* の形式でなければなりません。たとえば、Java データベース・コネクティビティー (JDBC) API を持つ IBM DB2 データベースにある sample という名前のデータベースを指定するには、引き数を `jdbc:db2:sample` とします。
- *driverName*。データベースへの接続に必要なデータベース・ドライバー・クラスをカプセル化します。
- *DBLogin*。データベースへの接続に必要なデータベース・ユーザー ID を識別します。
- *DBPassword*。データベースへの接続に必要な指定されたユーザー ID (*DBLogin*) のパスワードを識別します。
- *tableName*。bean の永続データを格納するデータベース・テーブルの名前を識別します。
- *jdbcConn*。java.sql.Connection オブジェクト内のデータ・ソースへの Java データベース・コネクティビティー (JDBC) による接続をカプセル化します。

```
...
public class AccountBMBean implements EntityBean {
     private EntityContext entityContext = null;
     ...
     private static final String DBRULProp = "DBURL";
     private static final String DriverNameProp = "DriverName";
     private static final String DBLoginProp = "DBLogin";
     private static final String DBPasswordProp = "DBPassword";
     private static final String TableNameProp = "TableName";
     private String jdbcUrl, driverName, DBLogin, DBPassword, tableName;
     private long accountId = 0;
     private int type = 1;
     private float balance = 0.0f;

     private Connection jdbcConn = null;
      ...
}
```

*図 56. コード例: AccountBMBean クラスのインスタンス変数*

データベースとデータベース・ドライバーの間の AccountBM bean の移送を容易にするために、Enterprise Bean に含まれる、対応する環境変数を取得することによって、データベース固有の変数 (*jdbcUrl*、*driverName*、*DBLogin*、*DBPassword*、および *tableName*) を設定します。これらの変数の値は、getEnvProps メソッドによって検索されます。このメソッドは、AccountBMBean クラスにインプリメントされていて、 setEntityContext メソッドを呼び出したときに起動されます。詳細については、220ページの『EJB サーバー (CB) 環境での接続の管理』または 223ページの『EJB サーバー (AE) 環境でのデータベース接続の管理』を参照してください。

Enterprise Bean の環境変数の設定方法について、詳しくは、67ページの『Enterprise Bean 用の環境変数の設定』を参照してください。

図56 に、JDBC 仕様のバージョン 1.0 と互換性のあるデータベース・アクセスを示しますが、JDBC 仕様のバージョン 2.0 と互換性があるデータベース・アクセスを実行することもできます。管理者は、javax.sql.DataSource 参照 (以前は jdbcURL 変数および driverName 変数に格納されていた情報をカプセル化したもの) を JNDI ネーム・スペースにバインドします。 BMP を持つエンティティー bean は、次のことを実行して java.sql.Connection を取得します。

```
DataSource ds = (dataSource)initialContext.lookup("java:comp/env/jdbc/MyDataSource");
Connection con = ds.getConnection();
```

ここで、*MyDataSource* は管理者がデータ・ソースに割り当てた名前です。

### ビジネス・メソッドのインプリメント

エンティティー bean クラスのビジネス・メソッドは、クラスにカプセル化されたデータを操作する方法を定義します。 Enterprise Bean クラスにインプリメントされたビジネス・メソッドは、EJB クライアントから直接呼び出すことができません。代わりに、EJB クライアントは、その Enterprise Bean のインスタンスに関連付けられた EJB オブジェクトを使用することによって、Enterprise Bean のリモート・インターフェースで定義された対応するメソッドを呼び出します。これを受けて、コンテナーは、その Enterprise Bean のインスタンスにある対応するメソッドを呼び出します。

したがって、Enterprise Bean クラスにインプリメントされたすべてのビジネス・メソッドについて、対応するメソッドを Enterprise Bean のリモート・インターフェースで定義しなければなりません。 Enterprise Bean のリモート・インターフェースは、Enterprise Bean の配置時に EJB オブジェクト・クラスにあるコンテナーによってインプリメントされます。

123ページの図20 に示した CMP bean クラスの AccountBean で定義されているビジネス・メソッドと、AccountBMBean bean クラスで定義されているビジネス・メソッドに違いはありません。

### ejbCreate および ejbPostCreate メソッドのインプリメント

Enterprise Bean の新しいインスタンスを作成する各方法について、ejbCreate メソッドを定義およびインプリメントしなければなりません。 ejbCreate メソッドごとに、対応する ejbPostCreate メソッドを定義することもできます。各 ejbCreate メソッドは、EJB ホーム・インターフェースの create メソッドに対応しなければなりません。

bean クラスのビジネス・メソッドと同様に、ejbCreate および ejbPostCreate メソッドは、クライアントから直接呼び出すことができません。代わりに、クライアントは、EJB ホーム・オブジェクトを使用することによって、Enterprise Bean のホーム・インターフェースの create メソッドを呼び出します。これを受けて、コンテナーは、ejbCreate メソッドと ejbPostCreate メソッドを順に呼び出します。

CMP を持つエンティティー bean のメソッドとは異なり、BMP を持つエンティティー bean の ejbCreate メソッドは、bean の永続データをデータ・ソースに挿入するために必要なすべてのコードを含まなければなりません。この要件は、ejbCreate メソッドが、 (データ・ソースへの接続がまだ bean インスタンスで使用可能になっていない場合は) データ・ソースへの接続を取得し、bean の変数の値をデータ・ソースの適切なフィールドに挿入しなければならないことを意味します。

BMP を持つエンティティー bean の ejbCreate メソッドは、いずれも以下の要件を満たさなければなりません。

- パブリックであり、bean の 1 次キー・クラスを戻さなければなりません。
- 引き数および戻り型は、Java リモート・メソッド呼び出し (RMI) に対して有効でなければなりません。
- 永続変数の値をデータ・ソースに挿入するために必要なコードを含まなければなりません。詳細については、219ページの『BMP エンティティー bean によるデータベースの使用』を参照してください。

各 ejbPostCreate メソッドは、パブリックで、戻り値なし (void) で、対応する ejbCreate メソッドと同じ引き数を持たなければなりません。

必要であれば、ejbCreate メソッドと ejbPostCreate メソッドの両方で、java.rmi.RemoteException 例外、javax.ejb.CreateException 例外、javax.ejb.DuplicateKeyException 例外、および任意のユーザー定義の例外を throw することができます。

208ページの図57 に、AccountBMBean bean クラスの例に必要な 2 つの ejbCreate メソッドを示します。ejbPostCreate メソッドは不要です。

AccountBean クラスの場合と同様に、最初の ejbCreate メソッドは 2 番目の ejbCreate メソッドを呼び出します。後者は、データ・ソースとのすべての対話を処理します。 2 番目のメソッドは、bean のインスタンス変数を初期化してから、checkConnection メソッドを呼び出すことによってデータ・ソースへの有効な接続があることを確認します。その後、このメソッドは、データ・ソースに対する SQL INSERT 呼び出しを作成、準備、および実行します。INSERT 呼び出しが正しく実行されると、1 行のみがデータ・ソースに挿入され、メソッドが bean の 1 次キー・クラスのオブジェクトを戻します。

```
public AccountBMKey ejbCreate(AccountBMKey key) throws CreateException,
     RemoteException {
     return ejbCreate(key, 1, 0.0f);
}
...
public AccountBMKey ejbCreate(AccountBMKey key, int type, float balance)
     throws CreateException, RemoteException
{
     accountId = key.accountId;
     this.type = type;
     this.balance = balance;
     checkConnection();
     // INSERT into database
     try {
          String sqlString = "INSERT INTO " + tableName +
               " (balance, type, accountid) VALUES (?,?,?)";
          PreparedStatement sqlStatement = jdbcConn.prepareStatement(sqlString);
          sqlStatement.setFloat(1, balance);
          sqlStatement.setInt(2, type);
          sqlStatement.setLong(3, accountId);
          // Execute query
          int updateResults = sqlStatement.executeUpdate();
          ...
     }
     catch (Exception e) { // Error occurred during insert
          ...
     }
     return key;
}
```

*図 57. コード例: AccountBMBean クラスの ejbCreate メソッド*

### ejbFindByPrimaryKey およびその他の ejbFind メソッドのインプリメント

少なくとも、BMP を持つエンティティー bean は、1 次キーを受け取り、そのキーが Enterprise Bean のインスタンスに対して有効かつ固有であるかどうかを判別するための ejbFindByPrimaryKey メソッドを定義およびインプリメントしなければなりません。1 次キーが有効かつ固有である場合は、その 1 次キーを戻します。エンティティー bean は、Enterprise Bean インスタンスを検索するためのその他の finder メソッドを定義およびインプリメントすることもできます。すべての finder メソッドは、javax.ejb.FinderException 例外を throw して、アプリケーション・レベルのエラーを示します。単一の bean を探すように指定された finder メソッドは、 javax.ejb.ObjectNotFoundException 例外、つまり FinderException クラスのサブクラスを throw することもできます。複数の bean を戻すように指定された finder メソッドは、 ObjectNotFoundException を使用して、適切な bean が見つからなかったことを示すことはできません。代わりに、このようなメソッドは空の戻り値を戻します。

java.rmi.RemoteException 例外の throw は行わないでください。詳細については、123ページの『エンティティー bean の標準アプリケーション例外』を参照してください。

bean クラスのビジネス・メソッドと同様に、ejbFind メソッドは、クライアントから直接呼び出すことができません。代わりに、クライアントは、EJB ホーム・オブジェクトを使用することによって、Enterprise Bean のホーム・インターフェースの finder メソッドを呼び出します。これを受けて、コンテナーは、対応する ejbFind メソッドを呼び出します。コンテナーは、プール状態にあるエンティティー bean の汎用インスタンスを使用して ejbFind メソッドを呼び出します。

コンテナーがプール状態にあるエンティティー bean のインスタンスを使用して ejbFind メソッドを呼び出すため、メソッドは以下を行わなければなりません。

1. データ・ソース (複数の場合もある) への接続を取得する。
2. finder メソッドの仕様に一致するレコードについて、データ・ソースを照会する。
3. データ・ソース (複数の場合もある) への接続を除去する。

これらのデータ・ソースに関する作業の詳細については、219ページの『BMP エンティティー bean によるデータベースの使用』を参照してください。

210ページの図58 に、AccountBMBean クラスの例の ejbFindByPrimaryKey メソッドを示します。 ejbFindByPrimaryKey メソッドは、210ページの図58 に示す makeConnection メソッドを呼び出すことによって、そのデータ・ソースへの接続を取得します。その後、指定された 1 次キーを使用することによって、そのデータ・ソースに対する SQL SELECT ステートメントを作成して呼び出します。

レコードが 1 つだけ検索された場合は、メソッドは、引き数で渡された 1 次キーを戻します。レコードが検索されなかった場合や複数のレコードが検索された場合は、メソッドは、FinderException を throw します。1 次キーを戻すか FinderException を throw するかを決定する前に、メソッドは、219ページの『BMP エンティティー bean によるデータベースの使用』で説明している dropConnection メソッドを呼び出すことによって、データ・ソースへの接続を除去します。

```
public AccountBMKey ejbFindByPrimaryKey (AccountBMKey key)
   throws FinderException {
       boolean wasFound = false;
       boolean foundMultiples = false;
       makeConnection();
       try {
            // SELECT from database
            String sqlString = "SELECT balance, type, accountid FROM " + tableName
                 + " WHERE accountid = ?";
            PreparedStatement sqlStatement = jdbcConn.prepareStatement(sqlString);
            long keyValue = key.accountId;
            sqlStatement.setLong(1, keyValue);

            // Execute query
            ResultSet sqlResults = sqlStatement.executeQuery();

            // Advance cursor (there should be only one item)
            // wasFound will be true if there is one
            wasFound = sqlResults.next();

            // foundMultiples will be true if more than one is found.
            foundMultiples = sqlResults.next();
       }
       catch (Exception e)  { // DB error
            ...
       }
       dropConnection();
       if (wasFound && !foundMultiples)
       {
            return key;
       }
        else
       {
            // Report finding no key or multiple keys
            ...
            throw(new FinderException(foundStatus));
      }
}
```

*図 58. コード例: AccountBMBean クラスの ejbFindByPrimaryKey メソッド*

211ページの図59 に、AccountBMBean クラスの例の ejbFindLargeAccounts メソッドを示します。ejbFindLargeAccounts メソッドも、makeConnection メソッドを呼び出すことによってそのデータ・ソースへの接続を取得し、dropConnection メソッドを使用することによって接続を除去します。この SQL SELECT ステートメントも、ejbFindByPrimaryKey メソッドによって使用されるステートメントと非常に類似しています。 (これらのデータ・ソースに関する作業およびメソッドの詳細については、219ページの『BMP エンティティー bean によるデータベースの使用』を参照してください。)

ejbFindByPrimaryKey メソッドが 1 次キーを 1 つだけ戻す必要があるのに対し、ejbFindLargeAccounts メソッドは、1 次キーを戻さないことも、Enumeration オブジェクトで任意の数の 1 次キーを戻すこともできます。1 次キーの列挙を戻すには、ejbFindLargeAccounts メソッドで以下を行います。

1. while ループを使用して、executeQuery によって戻された結果セット (*sqlResults*) を検査する。
2. 戻された口座 ID を Long オブジェクト、AccountBMKey オブジェクトの順にラップすることによって、*resultTable* という名前のハッシュ・テーブルに結果セットの各 1 次キーを挿入する。 (Long オブジェクト *memberId* は、ハッシュ・テーブルの索引として使用されます。)
3. ハッシュ・テーブルの elements メソッドを呼び出して 1 次キーの列挙を取得し、それを戻す。

```
public Enumeration ejbFindLargeAccounts(float amount) throws FinderException {
     makeConnection();
     Enumeration result;
     try {
          // SELECT from database
          String sqlString = "SELECT accountid FROM " + tableName
               + " WHERE balance >= ?";
          PreparedStatement sqlStatement = jdbcConn.prepareStatement(sqlString);
          sqlStatement.setFloat(1, amount);
          // Execute query
          ResultSet sqlResults = sqlStatement.executeQuery();
          // Set up Hashtable to contain list of primary keys
          Hashtable resultTable = new Hashtable();
          // Loop through result set until there are no more entries
          // Insert each primary key into the resultTable
          while(sqlResults.next() == true)  {
               long acctId = sqlResults.getLong(1);
               Long memberId = new Long(acctId);
               AccountBMKey key = new AccountBMKey(acctId);
               resultTable.put(memberId, key);
          }
          // Return the resultTable as an Enumeration
          result = resultTable.elements();
          return result;
     } catch (Exception e) {
          ...
     } finally {
          dropConnection();
     }
}
```

*図 59. コード例: AccountBMBean クラスの ejbFindLargeAccounts メソッド*

**EntityBean インターフェースのインプリメント**

各エンティティー bean クラスは、javax.ejb.EntityBean インターフェースから継承したメソッドをインプリメントしなければなりません。コンテナーは、これらのメソッドを呼び出して、bean のライフ・サイクルの各段階にわたって bean を移動します。 CMP を持つエンティティー bean とは異なり、BMP を持つエンティティー bean では、これらのメソッドは、bean が永続データの格納に使用するデータ・ソース (複数の場合もある) との対話に必要なすべてのコードを含まなければなりません。

- ejbActivate ー このメソッドは、コンテナーがインスタンス・プールからエンティティー bean のインスタンスを選択し、そのインスタンスを特定の EJB オブジェクトに割り当てるときにコンテナーによって呼び出されます。このメソッドは、データ・ソースへの接続を取得し、bean の javax.ejb.EntityContext クラスを使用して対応する EJB オブジェクトの 1 次キーを取得することによって Enterprise Bean のインスタンスを活動化するために必要なコードを含まなければなりません。

  AccountBMBean クラスの例では、ejbActivate メソッドは、bean インスタンスの口座 ID を取得し、*accountId* 変数の値を設定し、checkConnection メソッドを呼び出してデータ・ソースへの有効な接続があることを確認します。

- ejbLoad ー このメソッドは、エンティティー bean の永続変数をデータ・ソース内の対応するデータと同期させるために、コンテナーによって呼び出されます。 (つまり、データ・ソース内のフィールドの値が、対応する Enterprise Bean インスタンスの永続変数にロードされます。) このメソッドは、データ・ソースから値をロードし、それらの値を bean のインスタンス変数に割り当てるために必要なコードを含まなければなりません。

  AccountBMBean クラスの例では、ejbLoad メソッドは、bean インスタンスの口座 ID を取得し、*accountId* 変数の値を設定し、checkConnection メソッドを呼び出してデータ・ソースへの有効な接続があることを確認し、SQL SELECT ステートメントを構成して実行し、データ・ソースから取得した値に一致するように *type* 変数および *balance* 変数の値を設定します。

- ejbPassivate ー このメソッドは、EJB オブジェクトからのエンティティー bean インスタンスの関連付けを解除し、その Enterprise Bean インスタンスをインスタンス・プールに置くために、コンテナーによって呼び出されます。このメソッドは、Enterprise Bean インスタンスを非活動化 (つまり活動停止) するために必要なコードを含まなければなりません。通常、この非活動化は、単にデータ・ソースへの接続を除去するだけです。

AccountBMBean クラスの例では、ejbPassivate メソッドは、dropConnection メソッドを呼び出してデータ・ソースへの接続を除去します。

- ejbRemove ー このメソッドは、 (javax.ejb.EJBHome インターフェースから) Enterprise Bean のホーム・インターフェースによって継承されたか、 (javax.ejb.EJBObject インターフェースから) リモート・インターフェースによって継承された remove メソッドをクライアントが呼び出したときに、コンテナーによって呼び出されます。このメソッドは、Enterprise Bean の永続データをデータ・ソースから除去するために必要なコードを含まなければなりません。Enterprise Bean インスタンスの除去が許可されなかった場合は、このメソッドは javax.ejb.RemoveException 例外を throw することができます。通常、除去の処理は、bean インスタンスのデータをデータ・ソースから削除した後、データ・ソースへの bean インスタンスの接続を除去することによって行います。

  AccountBMBean クラスの例では、ejbRemove メソッドは、checkConnection メソッドを呼び出してデータ・ソースへの有効な接続があることを確認し、SQL 削除ステートメントを構成して実行し、dropConnection メソッドを呼び出してデータ・ソースへの接続を除去します。

- setEntityContext ー このメソッドは、javax.ejb.EntityContext インターフェースへの参照を Enterprise Bean インスタンスに渡すために、コンテナーによって呼び出されます。このメソッドは、参照をコンテキストに格納するために必要なコードを含まなければなりません。

  AccountBMBean クラスの例では、setEntityContext メソッドは、*entityContext* 変数の値を、コンテナーによって渡された値に設定します。

- ejbStore ー このメソッドは、コンテナーがデータ・ソース内のデータを Enterprise Bean インスタンスの永続変数の値と同期させる必要がある場合に、コンテナーによって呼び出されます。 (つまり、Enterprise Bean インスタンスの変数の値がデータ・ソースにコピーされ、直前の値が上書きされます。) このメソッドは、データ・ソース内のデータを Enterprise Bean インスタンスの対応する値で上書きするために必要なコードを含まなければなりません。

  AccountBMBean クラスの例では、ejbStore メソッドは、checkConnection メソッドを呼び出してデータ・ソースへの有効な接続があることを確認し、SQL UPDATE ステートメントを構成して実行します。

- unsetEntityContext ー このメソッドは、Enterprise Bean インスタンスを除去してその Enterprise Bean インスタンスに関連するリソースをすべて解放する前に、コンテナーによって呼び出されます。これは、Enterprise Bean インスタンスの除去前に呼び出される最後のメソッドです。

AccountBMBean クラスの例では、unsetEntityContext メソッドは、*entityContext* 変数の値をヌルに設定します。

## ホーム・インターフェースの作成 (BMP を持つエンティティー)

エンティティー bean のホーム・インターフェースは、bean の新しいインスタンスを作成し、既存のインスタンスを検索して除去し、インスタンスに関するメタデータを取得するために EJB クライアントが使用するメソッドを定義します。ホーム・インターフェースは、Enterprise Bean の開発者によって定義され、Enterprise Bean の配置中にコンテナーによって作成される EJB ホーム・クラスにインプリメントされます。コンテナーによって、JNDI (Java Naming and Directory Interface) を介してクライアントからホーム・インターフェースにアクセスすることができます。

規則では、ホーム・インターフェースには *Name* Home という名前が付けられます。ここで、*Name* は Enterprise Bean に割り当てた名前です。たとえば、AccountBM Enterprise Bean のホーム・インターフェースには、AccountBMHome という名前が付けられます。

BMP を持つエンティティー bean のホーム・インターフェースは、いずれも以下の要件を満たさなければなりません。

- javax.ejb.EJBHome インターフェースを拡張しなければなりません。ホーム・インターフェースは、javax.ejb.EJBHome インターフェースからいくつかのメソッドを継承します。これらのメソッドに関する詳細については、154ページの『javax.ejb.EJBHome インターフェース』を参照してください。
- インターフェースの各メソッドは、Enterprise Bean クラスの ejbCreate メソッド (場合によっては ejbPostCreate メソッド) に対応する create メソッドか、Enterprise Bean クラスの ejbFind メソッドに対応する finder メソッドのいずれかでなければなりません。詳細については、215ページの『create メソッドの定義』および 216ページの『finder メソッドの定義』を参照してください。
- ホーム・インターフェースで定義される各メソッドのパラメーターおよび戻り値は、Java RMI に対して有効でなければなりません。詳細については、155ページの『java.io.Serializable インターフェースおよび java.rmi.Remote インターフェース』を参照してください。さらに、各メソッドの throws 文節は、java.rmi.RemoteException 例外クラスを含まなければなりません。

215ページの図60 に、AccountBM bean の例のホーム・インターフェース (AccountBMHome) の定義のうち、関連する部分を示します。このインターフェースは、2 つの抽象 create メソッドを定義しています。最初のものは、関連する AccountBMKey オブジェクトを使用することによって AccountBM オブジェ

クトを作成し、2 番目のものは、関連する AccountBMKey オブジェクトを使用し、口座の種類と初期残高を指定することによって AccountBM オブジェクトを作成します。このインターフェースは、必要な findByPrimaryKey メソッドおよび findLargeAccounts メソッドを定義します。

```
...
import java.rmi.*;
import javax.ejb.*;
import java.util.*;
public interface  AccountBMHome extends EJBHome  {
     ...
     AccountBM create(AccountBMKey key) throws CreateException,
          RemoteException;
     ...
     AccountBM create(AccountBMKey key, int type, float amount)
          throws CreateException, RemoteException;
     ...
     AccountBM findByPrimaryKey(AccountBMKey key)
          throws FinderException, RemoteException;
     ...
     Enumeration findLargeAccounts(float amount)
          throws FinderException, RemoteException;
}
```

*図 60. コード例: AccountBMHome ホーム・インターフェース*

## create メソッドの定義

create メソッドは、クライアントが Enterprise Bean インスタンスを作成し、そのインスタンスに関連するデータをデータ・ソースに挿入するために使用されます。各 create メソッドには create という名前が付いていなければならず、Enterprise Bean クラスの対応する ejbCreate メソッドと同じ数および型の引き数を持たなければなりません。(ejbCreate メソッドは、対応する ejbPostCreate メソッドを自身で持つことができます。) create メソッドおよび対応する ejbCreate メソッドの戻り型は、常に異なります。

各 create メソッドは、以下の要件を満たさなければなりません。

- create という名前が付いていなければなりません。
- Enterprise Bean のリモート・インターフェースの型を戻さなければなりません。たとえば、AccountBMHome インターフェースの create メソッドの戻り型は AccountBM です (131ページの図23 を参照)。
- java.rmi.RemoteException 例外、javax.ejb.CreateException 例外、および対応する ejbCreate メソッドと ejbPostCreate メソッドの throws 文節で定義されているすべての例外が throws 文節に含まれていなければなりません。

### finder メソッドの定義

finder メソッドは、1 つ以上のエンティティー EJB オブジェクトを検索するために使用します。各 finder メソッドには、find*Name* という名前を付けなければなりません。ここで、*Name* は finder メソッドの目的を表します。

少なくとも、各ホーム・インターフェースは、findByPrimaryKey メソッドを定義して、クライアントが 1 次キーのみを使用して EJB オブジェクトを検索できるようにしなければなりません。 findByPrimaryKey メソッドには 1 つの引き数 (bean の 1 次キー・クラスのオブジェクト) があり、bean のリモート・インターフェースの型を戻します。

その他の finder メソッドは、いずれも以下の要件を満たさなければなりません。

- Enterprise Bean のリモート・インターフェースの型、 java.util.Enumeration インターフェースの型、または java.util.Collection インターフェースの型 (finder メソッドが複数の EJB オブジェクトまたは 1 つの EJB コレクションを戻すことができる場合) を戻さなければならない。
- java.rmi.RemoteException および javax.ejb.FinderException 例外クラスを含む throws 文節がなければならない。

すべてのエンティティー bean は、デフォルトの finder メソッドを含まなければなりませんが、必要な場合は、追加の finder メソッドを作成することができます。たとえば、215ページの図60 に示すように、AccountBM bean のホーム・インターフェースは、指定された金額より残高が多い口座をカプセル化するオブジェクトを検索するための findLargeAccounts メソッドを定義しています。この finder メソッドは複数の EJB オブジェクトへの参照を戻すことができるので、戻り型は java.util.Enumeration です。

CMP を持つエンティティー bean のインプリメンテーションとは異なり、BMP を持つエンティティー bean では、bean 開発者は、findByPrimaryKey メソッドに対応する ejbFindByPrimaryKey メソッドを完全にインプリメントしなければなりません。さらに、bean 開発者は、ホーム・インターフェースで定義された finder メソッドに対応する追加の ejbFind メソッドを作成しなければなりません。AccountBMBean クラスでの ejbFind メソッドのインプリメンテーションについては、208ページの『ejbFindByPrimaryKey およびその他の ejbFind メソッドのインプリメント』で説明しています。

## リモート・インターフェースの作成 (BMP を持つエンティティー)

エンティティー bean のリモート・インターフェースによって、bean クラスで使用可能なビジネス・メソッドにアクセスすることができます。また、bean イ

ンスタンスに関連付けられた EJB オブジェクトを除去するためのメソッドや、bean インスタンスのホーム・インターフェース、オブジェクト・ハンドル、および 1 次キーを取得するためのメソッドも提供されます。リモート・インターフェースは、EJB の開発者によって定義され、Enterprise Bean の配置中にコンテナーによって作成される EJB オブジェクト・クラスにインプリメントされます。

規則では、リモート・インターフェースには *Name* という名前が付けられます。ここで、*Name* は Enterprise Bean に割り当てた名前です。たとえば、AccountBM Enterprise Bean のリモート・インターフェースには、AccountBM という名前が付けられます。

各リモート・インターフェースは、以下の要件を満たさなければなりません。

- javax.ejb.EJBObject インターフェースを拡張しなければなりません。リモート・インターフェースは、javax.ejb.EJBObject インターフェースからいくつかのメソッドを継承します。これらのメソッドに関する詳細については、154ページの『javax.ejb.EJBObject から継承されたメソッド』を参照してください。
- Enterprise Bean クラスにインプリメントされたすべてのビジネス・メソッドについて、対応するビジネス・メソッドを定義しなければなりません。
- インターフェースで定義される各メソッドのパラメーターおよび戻り値は、Java RMI に対して有効でなければなりません。詳細については、155ページの『java.io.Serializable インターフェースおよび java.rmi.Remote インターフェース』を参照してください。
- 各メソッドの throws 文節は、java.rmi.RemoteException 例外クラスを含まなければなりません。

218ページの図61 に、AccountBM Enterprise Bean の例のリモート・インターフェース (AccountBM) の定義のうち、関連する部分を示します。このインターフェースは、AccountBMBean クラスにインプリメントされたビジネス・メソッドに正確に一致する、口座残高を表示および操作するための 4 つのメソッドを定義しています。

すべてのビジネス・メソッドは java.rmi.RemoteException 例外クラスを throw します。さらに、subtract メソッドは、bean クラスの対応するメソッドが com.ibm.ejs.doc.account.InsufficientFundsException を throw するため、ユーザー定義のこの例外を throw しなければなりません。これに加えて、このメソッドを呼び出すすべてのクライアントは、この例外を処理するか、あるいは throw によって例外を渡さなければなりません。

```
...
import java.rmi.*;
import javax.ejb.*;
import com.ibm.ejs.doc.account.InsufficientFundsException;
public interface AccountBM extends EJBObject {
     ...
     float add(float amount) throws RemoteException;
     ...
     float getBalance() throws RemoteException;
     ...
     void setBalance(float amount) throws RemoteException;
     ...
     float subtract(float amount) throws InsufficientFundsException,
          RemoteException;
}
```

*図 61. コード例: AccountBM リモート・インターフェース*

## 1 次キー・クラスの作成または選択 (BMP を持つエンティティー)

すべてのエンティティー EJB オブジェクトは、オブジェクトのホーム・インターフェース名およびその 1 次キーを組み合わせて定義される、コンテナーに固有の ID を持っています。この 1 次キーは、作成時にオブジェクトに割り当てられるものです。 2 つの EJB オブジェクトの ID が同じ場合は、同一のオブジェクトであると見なされます。

1 次キー・クラスは、EJB オブジェクトの 1 次キーをカプセル化するために使用されます。エンティティー bean (BMP または CMP を持つもの) では、別個の 1 次キー・クラスを作成したり、既存のクラスが逐次化可能であればそのクラスを 1 次キー・クラスとして使用したりすることができます。詳細については、155ページの『java.io.Serializable インターフェースおよび java.rmi.Remote インターフェース』を参照してください。

AccountBM bean の例では、136ページの図26 に示す Account bean に含まれる AccountKey クラスと同じ 1 次キー・クラスを使用します。ただし、キー・クラスの名前は AccountBMKey です。

**注:** EJB サーバー (AE) 環境の場合は、 BMP を持つエンティティー bean の 1 次キー・クラスに、 hashCode および equals メソッドをインプリメントする必要はありません。さらに、1 次キーを構成する変数はパブリックでなければなりません。

AccountBM bean の 1 次キー・クラスには、java.lang.Long クラスも使用可能です。

## BMP エンティティー bean によるデータベースの使用

BMP を持つエンティティー bean では、各 ejbFind メソッドおよびすべてのライフ・サイクル・メソッド (ejbActivate、ejbCreate、ejbLoad、ejbPassivate、および ejbStore) は、bean が永続データの保守に使用するデータ・ソース (複数の場合もある) と対話しなければなりません。サポートされているデータベースと対話するには、BMP エンティティー bean は、データベース接続を管理し、データベース内のデータを操作するためのコードを含まなければなりません。

データベース接続の管理に必要なコードは、EJB サーバーのインプリメンテーションによって異なります。

- EJB サーバー (CB) は、JDBC 1.0 を使用してデータベース接続を直接管理します。 EJB サーバー (CB) に関する詳細については、220ページの『EJB サーバー (CB) 環境での接続の管理』を参照してください。
- EJB サーバー (AE) は、特殊化された一連の bean を使用して、データベースに関する情報と JDBC に対する IBM 固有のインターフェースをカプセル化して、エンティティー bean が接続マネージャーと対話できるようにします。EJB サーバー (AE) に関する詳細については、223ページの『EJB サーバー (AE) 環境でのデータベース接続の管理』を参照してください。

一般に、データベースへの接続を取得および解放するには、以下の 3 つの方法があります。

- bean は、setEntityContext メソッド内でデータベース接続を取得し、unsetEntityContext メソッド内で解放することができます。この方法は、Enterprise Bean の開発者によるインプリメントが最も容易です。しかし、接続マネージャーがない場合は、bean インスタンスが使用されていない場合 (つまり bean インスタンスが非活動化された場合) でもデータベースに接続したままになるため、この方法は現実的ではありません。接続マネージャーがある場合でも、この方法ではスケーラビリティーが低くなります。
- bean は、ejbActivate および ejbCreate メソッドでデータベース接続を取得し、各 ejbFind メソッドでデータベース接続を取得および解放し、ejbPassivate および ejbRemove メソッドでデータベース接続を解放することができます。この方法は、インプリメントがやや難しくなりますが、活動化されている bean インスタンスしかデータベースに接続しないことが保証されます。EJB サーバー (CB) を使用する場合は、BMP エンティティー bean で接続マネージャーを使用することはできないので、この方法が最適です。
- bean は、データベース接続が必要な ejbActivate、ejbCreate、ejbFind、ejbLoad、および ejbStore の各メソッドで接続を取得および解放することができます。この方法は、最初の方法よりイン

プリメントが難しくなりますが、2 番目の方法より容易です。 EJB サーバー (AE) を使用している場合は、接続マネージャーが含まれるので、この方法が接続の使用の点で最も効率的であり、最もスケーラビリティーが高くなります。

AccountBM bean の例では、上記の 2 番目の方法を使用しています。AccountBMBean クラスには、DB2 データベースへの接続を作成するための 2 つのメソッド (checkConnection および makeConnection) と、接続を除去するための 1 つのメソッド (dropConnection) があります。これらのメソッドは、使用する EJB サーバー環境に基づいて異なる方法でコーディングしなければなりません。

- EJB サーバー (CB) で接続マネージャーとともに AccountBM bean を機能させるために必要なコードを 『EJB サーバー (CB) 環境での接続の管理』に示します。
- EJB サーバー (AE) で接続マネージャーとともに AccountBM bean を機能させるために必要なコードを 223ページの『EJB サーバー (AE) 環境でのデータベース接続の管理』に示します。

データベース内のデータを操作するために必要なコードは、いずれの EJB サーバー環境でも同じです。詳細については、226ページの『データベース内のデータの操作』を参照してください。

## EJB サーバー (CB) 環境での接続の管理

EJB サーバー (CB) 環境では、JDBC 1.0 接続 (java.sql.DriverManager インターフェースを使用) および JDBC 2.0 接続 (javax.sql.DataSource インターフェースを使用) の両方がサポートされますが、 JDBC 2.0 の完全サポートには DB2 バージョン 7.1、フィックスパック 2 が必要です。

JDBC 2.0 では、データベース接続は 223ページの『EJB サーバー (AE) 環境でのデータベース接続の管理』に説明されているように確立されます。アドバンスド版固有の com.ibm.db2.jdbc.app.stdext.javax.sql.DataSource インターフェースを、標準 JDBC 2.0 インターフェースの javax.sql.DataSource インターフェースに置き換えなければなりません (DB2 7.1、フィックスパック 2 を使用している場合は、 COM.ibm.db2.jdbc.DB2DataSource クラスによってインプリメントされるので、管理者はこのクラスを JNDI ネーム・スペースにバインドしなければなりません)。

JDBC 1.0 では、java.sql.DriverManager インターフェースを使用して、データベース・ドライバーのロードおよび登録を行ったり、データベースとの接続を解放したりします。次に、このプロセスについて説明します。

### データ・ソースのロードおよび登録

AccountBM bean の例では、IBM DB2 リレーショナル・データベースを使用して永続データを格納しています。 DB2 と対話するために、bean の例では、使用可能な JDBC ドライバーのいずれかをロードしなければなりません。図62 に、ドライバー・クラスのロードに必要なコードを示します。*driverName* 変数の値は、getEnvProps メソッドによって取得されます。このメソッドは、配置された Enterprise Bean にある、対応する環境変数にアクセスします。

Class.forName メソッドは、ドライバー・クラスをロードして登録します。AccountBM bean は、その setEntityContext メソッドにドライバーをロードして、 bean インスタンスを作成し、bean のコンテキストを設定した後、その bean のすべてのインスタンスからドライバーに即時にアクセスできるようにします。

**注:** EJB サーバー (CB) 環境では、JDBC を使用してデータベースにアクセスする BMP を持つエンティティー bean は、環境が XA 対応の JDBC をサポートしていないため、分散トランザクションで実行することができません。

```
public void setEntityContext(EntityContext ctx)
     throws EJBException {
     entityContext = ctx;
     try {
          getEnvProps();
          // Load the applet driver for DB2
          Class.forName(driverName);
     } catch (Exception e) {
          ...
     }
}
```

*図 62. コード例: setEntityContext メソッドでの JDBC ドライバーのロードおよび登録*

### データベースへの接続の作成およびクローズ

データベース・ドライバーをロードおよび登録した後に、BMP エンティティー bean は、データベースへの接続を取得しなければなりません。その接続が不要になった時には、BMP エンティティー bean は接続をクローズしなければなりません。

AccountBMBean クラスでは、データベース接続が必要な他の bean クラスのメソッドから checkConnection メソッドが呼び出されますが、そのメソッドには

接続が既に存在すると見なすことができます。このメソッドは、*jdbcConn* 変数がヌルに設定されているかどうかを検査することによって、接続がまだ使用可能であることを確認します。変数がヌルの場合は、makeConnection メソッドが呼び出されて接続が取得されます。

makeConnection メソッドは、新しいデータベース接続が必要な場合に呼び出されます。これは、静的メソッド java.sql.DriverManager.getConnection を呼び出し、*jdbcUrl* 変数で定義される DB2 URL 値を渡します (203ページの『インスタンス変数の定義』を参照)。getConnection が多重定義されています。ここに示したメソッドは、データベースの URL を使用するのみですが、他のバージョンでは、URL およびデータベース・ユーザー ID、あるいは URL、データベース・ユーザー ID、およびユーザー・パスワードが必要です。

```
# import java.sql.*;
...
private void checkConnection() throws EJBException {
     if (jdbcConn == null) {
          makeConnection();
     }
     return;
}
...
private void makeConnection() throws EJBException {
     ...
     try {
          // Open database connection
          jdbcConn = DriverManager.getConnection(jdbcUrl);
     } catch(Exception e) { // Could not get database connection
          ...
     }
}
```

*図 63. コード例: AccountBMBean クラスの checkConnection メソッドおよび makeConnection メソッド*

BMP を持つエンティティー bean は、特定の bean インスタンスが不要になった場合に、データベース接続の除去も行わなければなりません。
AccountBMBean クラスは、この作業を行うための dropConnection メソッドを含みます。データベース接続を除去するために、dropConnection メソッドは以下を行います。

1. 接続オブジェクト (*jdbcConn*) の commit メソッドを呼び出して、データベースで保留になっているロックをすべて除去する。
2. 接続オブジェクトの close メソッドを呼び出して、接続をクローズする。
3. 接続オブジェクトの参照をヌルに設定する。

```
private void dropConnection() {
     try {
          // Close and delete jdbcConn
          jdbcConn.commit();
     } catch (Exception e) {
          // Could not commit transactions to database
          ...
     } finally {
          jdbcConn.close();
          jdbcConn = null;
     }
}
```

*図 64. コード例: AccountBMBean クラスの dropConnection メソッド*

## EJB サーバー (AE) 環境でのデータベース接続の管理

EJB サーバー (AE) 環境では、管理者は、データベースおよびデータベース・ドライバーに関する情報をカプセル化する、特殊化された一連のエンティティー bean を作成します。これらの特殊化されたエンティティー bean は、WebSphere 管理コンソールを使用して作成します。

データベースにアクセスする必要があるエンティティー bean は、JNDI を使用して、正しいデータベース bean インスタンスに関連付けられた EJB オブジェクトへの参照を作成しなければなりません。これによって、エンティティー bean は、IBM 固有のインターフェース (com.ibm.db2.jdbc.app.stdext.javax.sql.DataSource) を使用して、データベースへの接続を取得および解放することができます。

DataSource インターフェースによって、エンティティー bean は、EJB サーバー (AE) の接続マネージャーと透過的に対話することができます。接続マネージャーは、データベース接続のプールを作成し、必要に応じて個々のエンティティー bean への割り振りおよび割り振り解除を行います。

**注:** 本セクションに記載されているコード例は、AccountBMBean には存在しません。AccountBMBean は、220ページの『EJB サーバー (CB) 環境での接続の管理』で説明されている DriverManager インターフェースを使用してデータベース接続を管理します。本セクションでは、DataSource インターフェースを使用するように AccountBM bean を再作成する場合に必要なコードを示しています。

## データ・ソース bean インスタンスへの EJB オブジェクト参照の取得

BMP エンティティー bean がデータベースへの接続を取得できるようにするには、エンティティー bean は、BMP エンティティー bean の永続データの格納に使用されるデータベースに関連付けられたデータ・ソース・エンティティー bean に対する JNDI 検索を実行しなければなりません。 図65 に、InitialContext オブジェクトを作成して、データベース bean インスタンスへの EJB オブジェクト参照を取得するために必要なコードを示します。データベース bean の JNDI 名は、管理者によって定義されます。この名前を定義する場合は、JNDI の命名規則に従うようにしてください。必要なデータベース固有の変数の値は、getEnvProps メソッドによって取得されます。このメソッドは、配置された Enterprise Bean の対応する環境変数にアクセスします。

接続マネージャーが実際のデータベース接続を作成および除去し、必要に応じて簡単にこれらの接続の割り振りおよび割り振り解除を行うため、BMP エンティティー bean がデータベース・ドライバーをロードして登録する必要はありません。したがって、203ページの『インスタンス変数の定義』で説明している *driverName* および *jdbcUrl* 変数を定義する必要はありません。

```
...
# import com.ibm.db2.jdbc.app.stdext.javax.sql.DataSource;
# import javax.naming.*;
...
InitialContext initContext = null;
DataSource ds = null;
...
    public void setEntityContext(EntityContext ctx)
     throws EJBException {
     entityContext = ctx;
           try {
               getEnvProps();
               ds = initContext.lookup("jdbc/sample");
          } catch (NamingException e) {
               ...
          }
     }
...
```

*図 65. コード例: setEntityContext メソッドにおけるデータ・ソース bean インスタンスへの EJB オブジェクト参照の取得 (DataSource を使用するように再作成したもの)*

## データベースへの接続の割り振りおよび割り振り解除

適切なデータベース bean インスタンスに対する EJB オブジェクト参照を作成した後に、そのオブジェクト参照を使用して、対応するデータベースへの接続を取得および解放します。 DriverManager インターフェースを使用する場合と

は異なり、DataSource インターフェースを使用する場合は、BMP エンティティー bean は、実際にはデータ接続の作成およびクローズを行いません。代わりに、接続マネージャーが、エンティティー bean によって要求された接続の割り振りおよび割り振り解除を行います。しかしこの場合でも、BMP エンティティー bean は、接続マネージャーに割り振り要求および割り振り解除要求を送信するためのコードを含まなければなりません。

AccountBMBean クラスでは、データベース接続が必要な他の bean クラスのメソッドから checkConnection メソッドが呼び出されますが、そのメソッドには接続が既に存在すると見なすことができます。このメソッドは、*jdbcConn* 変数がヌルに設定されているかどうかを検査することによって、接続がまだ使用可能であることを確認します。変数がヌルの場合は、makeConnection メソッドが呼び出されて接続が取得されます (つまり、接続割り振り要求が接続マネージャーに送信されます)。

makeConnection メソッドは、データベース接続が必要な場合に呼び出されます。これは、データ・ソース・オブジェクトに対する getConnection メソッドを呼び出します。getConnection メソッドが多重定義されています。このメソッドは、ユーザー ID およびパスワードを受け取ることも、引き数をとらないこともできます。後者の場合は、ユーザー ID およびパスワードは暗黙的にヌルに設定されます (図66 では後者を使用しています)。

```
private void checkConnection() throws EJBeException {
     if (jdbcConn == null) {
          makeConnection();
     }
     return;
}
...
private void makeConnection() throws EJBeException {
     ...
     try {
          // Open database connection
          jdbcConn = ds.getConnection();
     } catch(Exception e) { // Could not get database connection
          ...
     }
}
```

*図 66. コード例: AccountBMBean クラスの checkConnection メソッドおよび makeConnection メソッド (DataSource を使用するように再作成したもの)*

BMP を持つエンティティー bean は、特定の bean インスタンスが不要になった場合に、データベース接続の解放も行わなければなりません (つまり、接続マネージャーに割り振り解除要求を送信しなければなりません)。

AccountBMBean クラスは、この作業を行うための dropConnection メソッドを含みます。データベース接続を解放するために、dropConnection メソッドは以下を行います (図67 を参照)。

1. 接続オブジェクトの close メソッドを呼び出して、接続が不要になったことを接続マネージャーに通知する。
2. 接続オブジェクトの参照をヌルに設定する。

close メソッドを try/catch/finally ブロックの内部に置くと、なんらかの理由で close メソッドが失敗した場合でも、接続オブジェクト参照が必ずヌルに設定されることが保証されます。接続マネージャーがアイドル接続を終結処理しなければならないため、catch ブロックでは何も行われません。これは Enterprise Bean コードが処理するジョブではありません。

```
private void dropConnection() {
     try {
          // Close the connection
          jdbcConn.close();
     catch (SQLException ex) {
          // Do nothing
     } finally {
          jdbcConn = null;
     }
}
```

*図 67. コード例: AccountBMBean クラスの dropConnection メソッド (DataSource を使用するように再作成したもの)*

## データベース内のデータの操作

BMP エンティティー bean のインスタンスがデータベースへの接続を取得すると、データの読み書きが可能になります。 AccountBMBean クラスは、java.sql.PreparedStatement インターフェースを使用して Java 構造化照会言語 (JSQL) 呼び出しを構成して実行することによって、DB2 データベースと対話します。

227ページの図68 に示すように、SQL 呼び出しは String (*sqlString*) として作成されます。この String 変数は、java.sql.Connection.prepareStatement メソッドに渡されます。SQL 呼び出しにおける各変数の値は、PreparedStatement クラスのさまざまな set メソッドを使用して設定します。 (*sqlString* 変数では、変数が疑問符に置換されています。) PreparedStatement.executeUpdate メソッドを呼び出すと、SQL 呼び出しが実行されます。

```
private void ejbCreate(AccountBMKey key, int type, float initialBalance)
     throws CreateException, EJBException {
     // Initialize persistent variables and check for good DB connection
     ...
     // INSERT into database
     try {
          String sqlString = "INSERT INTO " + tableName +
               " (balance, type, accountid) VALUES (?,?,?)";
          PreparedStatement sqlStatement = jdbcConn.prepareStatement(sqlString);
          sqlStatement.setFloat(1, balance);
          sqlStatement.setInt(2, type);
          sqlStatement.setLong(3, accountId);
          // Execute query
          int updateResults = sqlStatement.executeUpdate();
          ...
     }
     catch (Exception e) { // Error occurred during insert
          ...
     }
     ...
}
```

*図 68. コード例: ejbCreate メソッドでの SQL 更新呼び出しの構成および実行*

executeUpdate メソッドは、データベースにデータを挿入したり更新したりするために呼び出します。executeQuery メソッドは、データベースからデータを取得するために呼び出します。データがデータベースから取得されると、executeQuery メソッドは java.sql.ResultSet オブジェクトを戻します。このオブジェクトは、そのクラスのメソッドを使用して検査および操作しなければなりません。

**注:** スケーラビリティーとパフォーマンスを向上させるために、各データベース更新ごとに PreparedStatement を呼び出す必要はありません。呼び出しを行わなくても、最初の PreparedStatement 呼び出しの結果をキャッシュに入れておくことができます。

228ページの図69 に、ResultSet 内のデータを AccountBMBean クラスの ejbLoad メソッドで操作する方法の例を示します。

```
public void ejbLoad () throws EJBeException {
     // Get data from database
     try {
          // SELECT from database
          ...
          // Execute query
          ResultSet sqlResults = sqlStatement.executeQuery();
          // Advance cursor (there should be only one item)
          sqlResults.next();
          // Pull out results
          balance = sqlResults.getFloat(1);
          type = sqlResults.getInt(2);
     } catch (Exception e) {
          // Something happened while loading data.
          ...
     }
}
```

*図 69. コード例: ejbLoad メソッドでの ResultSet オブジェクトの操作*

## bean 管理トランザクションの使用

多くの場合、Enterprise Bean は、コンテナーに依存して bean 内でトランザクションを管理します。このような場合にユーザーが行う必要があるのは、159ページの『第6章 Enterprise Bean でのトランザクションおよびセキュリティーの使用可能化』で説明しているように、適切なトランザクション・プロパティーをデプロイメント・ディスクリプターで設定することです。

しかし、特定の状況では、Enterprise Bean がトランザクションに直接参加する必要がある場合もあります。 Enterprise Bean のデプロイメント・ディスクリプターにあるトランザクション 属性を TX_BEAN_MANAGED に設定することによって、bean がトランザクションの活動状態の参加者であることをコンテナーに通知します。

**注:** TX_BEAN_MANAGED という値は、エンティティー bean の トランザクション ・デプロイメント・ディスクリプター属性の値としては無効です。つまり、エンティティー bean はトランザクションを管理することができません。

Enterprise Bean がそれ自体のトランザクションを管理するために必要とするコードを作成する場合は、以下の基本的な規則に従ってください。

- 状態なしセッション bean のインスタンスは、EJB クライアントによって呼び出される複数のメソッドにわたって同じトランザクション・コンテキスト

を再利用することができません。したがって、トランザクション・コンテキストは、トランザクション・コンテキストを必要とする各メソッドにローカルな変数にするようにしてください。

- 状態付きセッション bean のインスタンスは、EJB クライアントによって呼び出される複数のメソッドにわたって同じトランザクション・コンテキストを再利用することができます。したがって、トランザクション・コンテキストは、ユーザーが任意にインスタンス変数またはローカル・メソッド変数にすることができます。 (トランザクションが複数のメソッドにわたる場合は、javax.ejb.SessionSynchronization インターフェースを使用して、会話状態をトランザクションと同期させることができます。)

**注:** EJB サーバー (CB) 環境では、TX_BEAN_MANAGED 属性をインプリメントする状態付きセッション bean は、単一のメソッドの範囲内でトランザクションを開始して完了しなければなりません。

230ページの図70 に、トランザクション・コンテキストをカプセル化するオブジェクトを取得するために必要な標準的コードを示します。 3 つの基本的なステップがあります。

1. Enterprise Bean クラスは、setSessionContext メソッドで javax.ejb.SessionContext オブジェクト参照の値を設定しなければなりません。
2. SessionContext オブジェクト参照に対して getUserTransaction を呼び出すことによって、javax.transaction.UserTransaction オブジェクトを作成します。
3. UserTransaction オブジェクトを使用し、必要に応じて begin および commit などのトランザクション・メソッドを呼び出すことによって、トランザクションに参加します。 Enterprise Bean がトランザクションを開始する場合は、commit メソッドか rollback メソッドのいずれかを呼び出すことによって、そのトランザクションの完了も行わなければなりません。

   **注:** いずれの EJB サーバーでも、javax.ejb.EJBContext インターフェースの getUserTransaction メソッド (これは、SessionContext インターフェースによって継承されたものです) は、型 javax.jts.UserTransaction ではなく型 javax.transaction.UserTransaction のオブジェクトを戻します。これは、1.0 バージョンの EJB 仕様からは逸脱していますが、1.1 バージョンの EJB 仕様では、getUserTransaction メソッドが型 javax.transaction.UserTransaction のオブジェクトを戻すことを要求していて、型 javax.jts.UserTransaction のオプションを戻す要件が除去されています。

```
...
import javax.transaction.*;
...
public class MyStatelessSessionBean implements SessionBean {
     private SessionContext mySessionCtx = null;
     ...
     public void setSessionContext(.SessionContext ctx) throws EJBException {
          mySessionCtx = ctx;
     }
     ...
     public float doSomething(long arg1) throws FinderException, EJBException {
          UserTransaction userTran = mySessionCtx.getUserTransaction();
          ...
          // User userTran object to call transaction methods
          userTran.begin();
         // Do transactional work
          ...
          userTran.commit();
          ...
     }
     ...
}
```

*図 70. コード例: トランザクション・コンテキストをカプセル化するオブジェクトの取得*

UserTransaction インターフェースでは、以下のメソッドを使用することができます。

- begin ー トランザクションを開始します。このメソッドは引き数を受け取らず、戻り値もありません (void)。
- commit ー トランザクションのコミットを試行します。トランザクションがロールバックされる要因がなければ、このメソッドが正常に終了することによってトランザクションがコミットされます。このメソッドは引き数を受け取らず、戻り値もありません (void)。
- getStatus ー 参照するトランザクションの状況を戻します。このメソッドは引き数を受け取らず、int を戻します。トランザクションが参照に関連付けられていない場合は、STATUS_NO_TRANSACTION が戻されます。以下に、このメソッドの有効な戻り値を示します。
  - STATUS_ACTIVE ー トランザクション処理がまだ進行中であることを示します。
  - STATUS_COMMITTED ー トランザクションがコミットされ、トランザクションへの変更が永続的なものになったことを示します。
  - STATUS_COMMITTING ー トランザクションがコミット中である (つまり、トランザクションのコミットが開始されたが処理は完了していない) ことを示します。

- STATUS_MARKED_ROLLBACK − トランザクションにロールバックのマークが付けられていることを示します。
- STATUS_NO_TRANSACTION − 現行のトランザクション・コンテキストにトランザクションが存在しないことを示します。
- STATUS_PREPARED − トランザクションが準備されているが、完了していないことを示します。
- STATUS_PREPARING − トランザクションが準備中である (つまり、トランザクションの準備が開始されたが処理は完了していない) ことを示します。
- STATUS_ROLLEDBACK − トランザクションがロールバックされたことを示します。
- STATUS_ROLLING_BACK − トランザクションがロールバック中である (つまり、トランザクションのロールバックが開始されたが処理は完了していない) ことを示します。
- STATUS_UNKNOWN − トランザクションの状況が不明であることを示します。

- rollback − 参照するトランザクションをロールバックします。このメソッドは引き数を受け取らず、戻り値もありません (void)。
- setRollbackOnly − 出力可能なトランザクションのみが、ロールバックされることを指定します。このメソッドは引き数を受け取らず、戻り値もありません (void)。
- setTransactionTimeout − トランザクションに関連付けられたタイムアウトを (秒単位で) 設定します。トランザクションの参加者が特にこの値を設定していない場合は、デフォルトのタイムアウトが使用されます。このメソッドは、 (int 型で) 秒数を受け取り、戻り値はありません (void)。

# 第10章 WebSphere プログラミング・モデル拡張機能

このセクションでは、WebSphere Application Server のプログラミング・モデル拡張機能の一部として提供される機能について説明します。

- 例外チェーニング・パッケージ。分散アプリケーションが使用し、一連の例外を取り込むことができます。詳細については、『分散例外パッケージ』を参照してください。
- コマンド・パッケージ。分散アプリケーションが使用し、これらのアプリケーションが行なうリモート呼び出しの数を削減することができます。詳細については、245ページの『コマンド・パッケージ』を参照してください。
- ローカライズ可能テキスト・パッケージ。複数のロケールにまたがる分散アプリケーションで使用し、ユーザー指定の言語で出力を配布することができます。詳細については、277ページの『ローカライズ可能テキスト・パッケージ』を参照してください。

例外チェーニング・パッケージおよびコマンド・パッケージは、 WebSphere Application Server アドバンスド版およびエンタープライズ版の一部として使用することができます。ローカライズ可能テキスト・パッケージは、WebSphere Application Server アドバンスド版の一部として使用することができます。
3 つのパッケージはすべて汎用ユーティリティーで、再利用可能な方法で共通機能を提供することが目的です。これらの機能は Enterprise Bean のコンテキストで表されていますが、どのような WebSphere Application Server Java アプリケーションでも使用できます。 Enterprise Bean での使用に限定されるものではありません。

## 分散例外パッケージ

分散アプリケーションには、例外処理のためのストラテジーが必要です。アプリケーションが複雑になり、より多くの参加者によって使用されるために、例外処理があいまいになります。あらゆる例外に含まれる情報を取り込むには、メソッドがキャッチするあらゆる例外を再 throw する必要があります。あらゆるメソッドがこの方法を採用すると、例外は管理が不可能な数になり、コードそのものの安全性が低下します。さらに、新規のメソッドが新しい例外を導入すると、その新規メソッドを呼び出すすべての既存のメソッドを、その新しい例外を処理するように変更する必要があります。複雑なアプリケーションで可能なあらゆる例外を明示的に迅速に管理しようとすることは困難になります。

© Copyright IBM Corp. 1999, 2001

例外の数を管理できるようにしておくために、メソッドが単一の文節のすべての例外をキャッチし、応答として 1 つの例外を throw するストラテジーを採用するプログラマーがいます。これによって、各メソッドが認識しなければならない例外の数は削減されますが、発生している例外に関する情報が失われることにもなります。この情報の損失が望ましいのは、たとえば、エンド・ユーザーからのインプリメンテーション明細を隠したい場合です。ただし、このストラテジーは、アプリケーションのデバッグをより困難にする可能性があります。

分散例外パッケージは、例外のチェーンを作成できるようにする機能を提供します。*例外チェーン* は、前回の例外のスタックをカプセル化します。例外チェーンを使用すると、以前の例外を破棄せずに、1 つの例外をもう 1 つの例外に応答して throw することができます。したがって、例外が伝える情報を失うことなく例外の数を管理することができます。チェーニングをサポートする例外を、*分散例外* といいます。

## 概説

分散例外のチェーニングのサポートは、com.ibm.websphere.exception Java パッケージによって提供されます。以下のクラスおよびインターフェースが、このパッケージを構成します。

- DistributedException － このクラスは、DistributedExceptionInfo オブジェクトでのメソッドへのアクセスを提供します。分散アプリケーション内の例外のルート (root) クラスとして機能します。詳細については、235ページの『DistributedException クラス』を参照してください。
- DistributedExceptionEnabled － このインターフェースを使用すると、DistributedException クラスから継承できない例外を、例外チェーンで使用できるようにします。したがって、定義済みの例外を基にした例外を取り込むことができます。詳細については、236ページの『DistributedExceptionEnabled インターフェース』を参照してください。
- DistributedExceptionInfo － このクラスは、分散例外に必要な作業をカプセル化します。 DistributedException クラスを拡張する例外クラスは、自動的にこのクラスへアクセスします。 DistributedExceptionEnabled インターフェースをインプリメントするクラスは、DistributedExceptionInfo 属性を明示的に宣言しなければなりません。詳細については、237ページの『DistributedExceptionInfo クラス』を参照してください。
- ExceptionInstantiationException － このクラスは、例外チェーンを作成できない場合に throw される例外を定義します。この例外は内部でインスタンスを生成されますが、ユーザーがこの例外をキャッチし再 throw することができます。

このセクションでは、例外チェーニング・パッケージ内のインターフェースおよびクラスに関する一般的な説明を提供します。

### DistributedException クラス

DistributedException クラスは、アプリケーションが定義する例外階層のルート (root) 例外を提供します。このクラスでは、キャッチした例外を保管し、throw される新しい例外にその例外を組み込んで、例外のチェーンを作成します。この方法によって、古い例外に関する情報は、新しい例外と共に転送されます。このクラスは 6 つのコンストラクターを宣言します。 図71 は、これらのコンストラクターのシグニチャーを示しています。例外が DistributedException クラスのサブクラスである場合には、ユーザーの例外クラスで対応するコンストラクターを提供しなければなりません。

```
...
public class DistributedException extends Exception
implements DistributedExceptionEnabled
{
    // Constructors
    public DistributedException() {...}
    public DistributedException(String message) {...}
    public DistributedException(Throwable exception) {...}
    public DistributedException(String message,Throwable exception) {...}
    public DistributedException(String resourceBundleName,
                                String resourceKey,
                                Object[] formatArguments,
                                String defaultText)
    {...}
    public DistributedException(String resourceBundleName,
                                String resourceKey,
                                Object[] formatArguments,
                                String defaultText,
                                Throwable exception)
    {...}
    // Other methods
    ...
}
```

*図 71. コードの例: DistributedException クラスのコンストラクター*

このクラスは、チェーンから例外を抽出し、チェーンを照会するためのメソッドも提供します。これらのメソッドには、以下が組み込まれます。

- getMessage ー このメソッドは、現行例外に関連したメッセージ・ストリングを戻します。
- getPreviousException ー このメソッドは、チェーン内の前述の例外を Throwable オブジェクトとして戻します。前述の例外がない場合には、ヌルを戻します。

- getOriginalException － このメソッドは、チェーン内の元の例外を Throwable オブジェクトとして戻します。前の例外がない場合には、ヌルを戻します。
- getException － このメソッドは、チェーンからの名前付き例外の最新のインスタンスを、Throwable オブジェクトとして戻します。インスタンスがない場合には、ヌルを戻します。
- getExceptionInfo － このメソッドは、例外の DistributedExceptionInfo オブジェクトを戻します。
- printStackTrace － これらのメソッドは、現行例外のスタック・トレースを印刷します。このスタック・トレースには、チェーン内の以前のすべての例外のスタック・トレースが組み込まれます。

***ローカライズ・サポート:*** ローカライズされたメッセージのサポートは、分散例外の 2 つのコンストラクターによって提供されます。これらのコンストラクターは、リソース・バンドル、リソース・キー、デフォルト・メッセージ、およびメッセージ内の変数の置き換えストリングのセットを表す引き数を取ります。リソース・バンドルは、特定のロケールに関連した情報を表すリソースまたはリソース名の集まりです。リソース・バンドルは、ResourceBundle クラスまたはプロパティー・ファイルのサブクラスのいずれかとして提供されます。リソース・キーは、検索するバンドル内のリソースを示します。デフォルト・メッセージは、リソース・バンドルの名前またはキーがヌルであるか、あるいは無効である場合に戻されます。

### DistributedExceptionEnabled インターフェース

DistributedException クラスを拡張できない場合に、DistributedExceptionEnabled インターフェースを使用して、分散例外を作成します。Java は複数の継承を許可しないため、複数の例外クラスを拡張することはできません。既存の例外クラス、たとえば javax.ejb.CreateException を拡張している場合には、DistributedException クラスを拡張することもできません。新しい例外クラスがその他の例外をチェーンすることができるようにするには、代わりに DistributedExceptionEnabled インターフェースをインプリメントしなければなりません。

DistributedExceptionEnabled インターフェースは、ユーザーの例外クラスにインプリメントしなければならない以下の 8 つのメソッドを宣言します。

- getMessage － このメソッドは、現行例外に関連したメッセージ・ストリングを戻します。
- getPreviousException － このメソッドは、チェーン内の前述の例外を Throwable オブジェクトとして戻します。前述の例外がない場合には、ヌルを戻します。

- getOriginalException ー このメソッドは、チェーン内の元の例外を Throwable オブジェクトとして戻します。前の例外がない場合には、ヌルを戻します。
- getException ー このメソッドは、チェーンからの名前付き例外の最新のインスタンスを、Throwable オブジェクトとして戻します。インスタンスがない場合には、ヌルを戻します。
- getExceptionInfo ー このメソッドは、例外の DistributedExceptionInfo オブジェクトを戻します。
- printStackTrace ー これらのメソッドは、現行例外のスタック・トレースを印刷します。このスタック・トレースには、チェーン内の以前のすべての例外のスタック・トレースが組み込まれます。
- printSuperStackTrace ー このメソッドは、DistributedExceptionInfo オブジェクトが使用して、現行のスタック・トレースを検索し保管します。

DistributedExceptionEnabled インターフェースをインプリメントする場合には、DistributedExceptionInfo 属性を宣言しなければなりません。この属性は、これらのメソッドのほとんどのインプリメンテーションを提供します。したがって、これらのメソッドの例外クラスでのインプリメントは、DistributedExceptionInfo オブジェクトで対応するメソッドの呼び出しで構成されます。詳細については、241ページの『DistributedExceptionEnabled インターフェースからのメソッドのインプリメント』を参照してください。

### DistributedExceptionInfo クラス

DistributedExceptionInfo クラスは、分散例外に必要な機能性を提供します。このクラスは、DistributedExceptionEnabled インターフェース (DistributedException クラスが組み込まれる) をインプリメントするどんな例外によっても使用されなければなりません。 DistributedExceptionInfo オブジェクトは例外そのものも含み、例外チェーンを作成するためのコンストラクターと、そのチェーン内の情報を検索するためのメソッドを提供します。また、チェーンされた例外を管理するための基礎となるメソッドも提供します。

## DistributedException クラスの拡張

DistributedException クラスは、アプリケーションが定義する例外階層のルート (root) 例外を提供します。このクラスは、チェーンから例外を抽出し、チェーンを照会するためのメソッドも提供します。 DistributedException クラス内のコンストラクターに対応するコンストラクターを提供しなければなりません (235ページの図71 を参照)。コンストラクターは、238ページの図72 に示すように、スーパー・メソッドを使用して引き数を DistributedException クラス内のコンストラクターに渡すことしかできません。

```
...
import com.ibm.websphere.exception.*;
public class MyDistributedException extends DistributedException
{
    // Constructors
    public MyDistributedException() {
        super();
    }
    public MyDistributedException(String message) {
        super(message);
    }
    public MyDistributedException(Throwable exception) {
        super(exception);
    }
    public MyDistributedException(String message, Throwable exception) {
        super(message, exception);
    }
    public MyDistributedException(String resourceBundleName,
                      String resourceKey, Object[] formatArguments,
                      String defaultText)
    {
        super(resourceBundleName, resourceKey, formatArguments, defaultText);
    }
    public MyDistributedException(String resourceBundleName,
                    String resourceKey, Object[] formatArguments,
                    String defaultText, Throwable exception)
    {
        super(resourceBundleName, resourceKey, formatArguments, defaultText,
            exception);
    }
}
```

*図 72. コードの例: DistributedException クラスを拡張する例外内のコンストラクター*

## DistributedExceptionEnabled インターフェースのインプリメント

DistributedException クラスを拡張できない場合に、DistributedExceptionEnabled インターフェースを使用して、分散例外を作成します。新しい例外クラスをチェーンすることができるようにするには、代わりに DistributedExceptionEnabled インターフェースをインプリメントしなければなりません。239ページの図73は、既存の javax.ejb.CreateException クラスを拡張し、DistributedExceptionEnabled インターフェースをインプリメントする例外クラスの構造を示しています。このクラスは、必要な DistributedExceptionInfo オブジェクトも宣言します。

```
...
import javax.ejb.*;
import com.ibm.websphere.exception.*;
public class AccountCreateException extends CreateException
implements DistributedExceptionEnabled
{
    DistributedExceptionInfo exceptionInfo = null;
    // Constructors
    ...
    // Methods from the DistributedExceptionEnabled interface
    ...
}
```

*図 73. コードの例: DistributedExceptionEnabled インターフェースをインプリメントする例外クラスの構造*

## 例外クラスのコンストラクターのインプリメント

例外チェーニング・パッケージは、例外クラスのインスタンスを作成する 6 つの異なる方法をサポートします (235ページの図71 を参照)。DistributedExceptionEnabled インターフェースをインプリメントして例外クラスを作成する場合には、これらのコンストラクターをインプリメントしなければなりません。各コンストラクターでは、DistributedExceptionInfo オブジェクトを使用して、例外のチェーニングに関する情報を取り込まなければなりません。 240ページの図74 は、6 つのコンストラクターの標準インプリメンテーションを示しています。

```
...
public class AccountCreateException extends CreateException
implements DistributedExceptionEnabled
{
    DistributedExceptionInfo exceptionInfo = null;
    // Constructors
    AccountCreateException() {
        super ();
        exceptionInfo = new DistributedExceptionInfo(this);
    }
    AccountCreateException(String msg) {
        super (msg);
        exceptionInfo = new DistributedExceptionInfo(this);
    }
    AccountCreateException(Throwable e) {
        super ();
        exceptionInfo = new DistributedExceptionInfo(this, e);
    }
    AccountCreateException(String msg, Throwable e) {
        super (msg);
        exceptionInfo = new DistributedExceptionInfo(this, e);
    }
   AccountCreateException(String resourceBundleName, String resourceKey,
                             Object[] formatArguments, String defaultText)
    {
        super ();
        exceptionInfo = new DistributedExceptionInfo(resourceBundleName,
                            resourceKey, formatArguments, defaultText, this);
    }
   AccountCreateException(String resourceBundleName, String resourceKey,
                             Object[] formatArguments, String defaultText,
                             Throwable exception)
   {
        super ();
        exceptionInfo = new DistributedExceptionInfo(resourceBundleName,
                       resourceKey, formatArguments, defaultText, this, exception);
    }
    // Methods from the DistributedExceptionEnabled interface
    ...
}
```

*図 74. コードの例: DistributedExceptionEnabled インターフェースをインプリメントする例外クラスのコンストラクター*

### DistributedExceptionEnabled インターフェースからのメソッドのインプリメント

DistributedExceptionInfo オブジェクトは、DistributedExceptionEnabled インターフェース内のほとんどのメソッドのインプリメンテーションを提供します。したがって、DistributedExceptionInfo オブジェクト上で対応するメソッドを呼び出して、例外クラスで必要なメソッドをインプリメントすることができます。242ページの図75 は、この手法を示しています。 DistributedExceptionInfo オブジェクト上では対応するメソッドの呼び出しを行わない 2 つのメソッドがあります。このオブジェクトを戻す getExceptionInfo メソッドと、super.printStackTrace メソッドを呼び出す printSuperStackTrace メソッドです。

```
...
public class AccountCreateException extends CreateException
implements DistributedExceptionEnabled
{
    DistributedExceptionInfo exceptionInfo = null;
    // Constructors
    ...
    // Methods from the DistributedExceptionEnabled interface
    String getMessage() {
        if (exceptionInfo != null)
            return exceptionInfo.getMessage();
        else return null;
    }
    Throwable getPreviousException() {
        if (exceptionInfo != null)
            return exceptionInfo.getPreviousException();
        else return null;
    }
    Throwable getOriginalException() {
        if (exceptionInfo != null)
            return exceptionInfo.getOriginalException();
        else return null;
    }
    Throwable getException(String exceptionClassName) {
        if (exceptionInfo != null)
            return exceptionInfo.getException(exceptionClassName);
        else return null;
    }
    DistributedExceptionInfo getExceptionInfo() {
        if (exceptionInfo != null)
            return exceptionInfo;
        else return null;
    }
    void printStackTrace() {
        if (exceptionInfo != null)
            return exceptionInfo.printStackTrace();
        else return null;
    }
    void printStackTrace(PrintWriter pw) {
        if (exceptionInfo != null)
            return exceptionInfo.printStackTrace(pw);
        else return null;
    }
    void printSuperStackTrace(PrintWriter pw)
        if (exceptionInfo != null)
            return super.printStackTrace(pw);
        else return null;
    }
}
```

*図 75. コードの例: DistributedExceptionEnabled インターフェースでのメソッドのインプリメンテーション*

## 分散例外の使用法

分散例外を定義すると、例外を一緒にチェーンすることができます。DistributedExceptionInfo クラスは、情報を例外チェーンに追加するためのメソッドと、チェーンから情報を抽出するためのメソッドを提供します。このセクションでは、分散例外の使用法を示します。

### 分散例外のキャッチ

DistributedException クラスを拡張する、あるいは DistributedExceptionEnabled インターフェースを別々にインプリメントする例外をキャッチすることができます。また、キャッチした例外をテストして、DistributedExceptionEnabled インターフェースがインプリメントされているかどうかを調べることもできます。このインターフェースがインプリメントされている場合には、キャッチした例外を他の分散例外として処理することができます。図76 は、例外チェーニングをテストするメソッドのインスタンスの使用法を示しています。

```
....
try {
    someMethod();
}
catch (Exception e) {
    ...
    if (e instanceof DistributedExceptionEnabled) {
            ...
    }
...
```

*図 76. コードの例: DistributedExceptionEnabled インターフェースをインプリメントする例外のテスト*

### チェーンへの例外の追加

チェーンに例外を追加するには、分散例外クラスのコンストラクターの 1 つを呼び出さなければなりません。これによって、以前の例外情報を取り込み、その例外を新しい例外とともに圧縮します。244ページの図77 は、MyDistributedException(Throwable) コンストラクターの使用法を示しています。

```
void someMethod() throws MyDistributedException {
    try {
        someOtherMethod();
    }
    catch (DistributedExceptionEnabled e) {
        throw new MyDistributedException(e);
    }
    ...
}...
```

*図 77. コードの例: チェーンへの例外の追加*

## チェーンからの情報の検索

チェーン例外を使用すると、チェーン内の以前の例外に関する情報を検索することができます。たとえば、getPreviousException、getOriginalException、および getException(String) メソッドを使用すると、特定の例外をチェーンから検索することができます。 getMessage メソッドを呼び出して、現行の例外に関連したメッセージを検索することができます。また、printStackTrace メソッドの 1 つを呼び出して、チェーン全体に関する情報を入手することもできます。245ページの図78 は、getPreviousException および getOriginalException メソッドの呼び出しを示しています。

```
...
try {
    someMethod();
}
catch (DistributedExceptionEnabled e) {
    try {
       Throwable prev = e.getPreviousException();
    }
    catch (ExceptionInstantiationException eie) {
        DistributedExceptionInfo prevExInfo = e.getPreviousExceptionInfo();
        if (prevExInfo != null) {
            String prevExName = prevExInfo.getClassName();
            String prevExMsg = prevExInfo.getClassMessage();
            ...
        }
    }
    try {
        Throwable orig = e.getOriginalException();
    }
    catch (ExceptionInstantiationException eie) {
        DistributedExceptionInfo origExInfo = null;
        DistributedExceptionInfo prevExInfo = e.getPreviousExceptionInfo();
        while (prevExInfo != null) {
            origExInfo = prevExInfo;
            prevExInfo = prevExInfo.getPreviousExceptionInfo();
        }
        if (origExInfo != null) {
            String origExName = origExInfo.getClassName();
            String origExMsg = origExInfo.getClassMessage();
        ...
        }
    }
}
...
```

*図 78. コードの例: チェーンからの例外の抽出*

## コマンド・パッケージ

分散アプリケーションは、リモート・リソースをローカルであるかのように使用する能力で定義されますが、このリモート作業は、分散アプリケーションのパフォーマンスに影響を及ぼします。分散アプリケーションは、リモート呼び出しの使用を控えめにすることによって、パフォーマンスを改善することができます。たとえば、サーバーがクライアントのためのいくつかの作業を行なう場合、クライアントが要求をバンドルすれば、アプリケーションはより迅速に実行され、個々のリモート呼び出しの数を削減できます。コマンド・パッケージは、要求のセットを収集し、それを 1 つの単位として実行依頼するメカニズムを提供します。

クライアントが行なうリモート呼び出しの数を削減する方法を提供する他に、コマンド・パッケージは要求を作成する一般的な方法も提供します。クライアントはコマンドをインスタンス化し、その入力データを設定し、そのコマンドに実行を指示します。コマンドの基礎構造はターゲット・サーバーを決定し、そのコマンドのコピーをターゲット・サーバーに渡します。サーバーはそのコマンドを実行し、なんらかの出力データを設定し、そのデータのコピーをクライアントに戻します。パッケージは、ローカルまたはリモートで、しかもサーバーのインプリメンテーションとは無関係に、コマンドを実行するための共通した方法を提供します。サーバーがそのリソースへの Java アクセスをサポートし、クライアントの Java 仮想マシン (JVM) と独自の JVM との間でコマンドをコピーする方法を提供する場合には、任意のサーバー (Enterprise Bean、Java データベース・コネクティビティー (JDBC) サーバー、サーブレットなど) がコマンドのターゲットとなることができます。

## 概説

コマンド機能は、com.ibm.websphere.command Java パッケージにインプリメントされます。コマンド・パッケージ内のクラスおよびインターフェースは、以下の 4 つの汎用カテゴリーに分類されます。

- コマンドを作成するためのインターフェース。詳細については、『コマンドを作成するための機能』を参照してください。
- コマンドをインプリメントするためのクラスおよびインターフェース。詳細については、247ページの『コマンドをインプリメントするための機能』を参照してください。
- コマンドを実行する場所を決定するクラスおよびインターフェース。詳細については、248ページの『ターゲットを設定し決定するための機能』を参照してください。
- パッケージ固有の例外を判別するクラス。詳細については、250ページの『コマンド・パッケージの例外』を参照してください。

このセクションでは、コマンド・パッケージ内のインターフェースおよびクラスに関する一般的な説明を提供します。

### コマンドを作成するための機能

コマンド・インターフェースは、コマンドの最も基本的な性質のみを指定します。このインターフェースは、TargetableCommand インターフェースおよび CompensableCommand インターフェースの両方で拡張され、どちらも追加機構を提供します。アプリケーション用のコマンドを作成するには、以下のことを行なわなければなりません。

- コマンド・パッケージで 1 つ以上のインターフェースを拡張するインターフェースを定義する。
- ユーザーのインターフェースにインプリメンテーション・クラスを提供する。

実際問題として、ほとんどのコマンドは TargetableCommand インターフェースをインプリメントします。このインターフェースを使用すると、コマンドをリモートで実行することができます。 図79 は、ターゲットを定めることができるコマンド用のコマンド・インターフェースの構造を示しています。

```
...
import com.ibm.websphere.command.*;
public interface MySimpleCommand extends TargetableCommand {
      // Declare application methods here
}
```

*図 79. コードの例: ターゲットを定めることができるコマンド用のインターフェースの構造*

CompensableCommand インターフェースを使用すると、1 つのコマンドを、最初のコマンドの作業を元に戻すことができる別のコマンドと関連付けることができます。補正可能なコマンドは、一般的には、TargetableCommand インターフェースもインプリメントします。図80 は、ターゲットを定め、補正が可能なコマンド用のコマンド・インターフェースの構造を示しています。

```
...
import com.ibm.websphere.command.*;
public interface MyCommand extends TargetableCommand, CompensableCommand {
      // Declare application methods here
}
```

*図 80. コードの例: ターゲットを定め、補正が可能なコマンド用のインターフェースの構造*

### コマンドをインプリメントするための機能

コマンドは、クラス TargetableCommandImpl を拡張することによってインプリメントされます。このクラスは、TargetableCommand インターフェースをインプリメントします。 TargetableCommandImpl クラスは、TargetableCommand インターフェース内の一部のメソッド (たとえば、戻り値を設定する) にいくつかのインプリメンテーションを提供し、アプリケーション自身がインプリメントしなければならない追加メソッド (たとえば、コマンドの実行方法) を宣言する抽象クラスです。

TargetableCommandImpl クラスを拡張し、ユーザーのコマンド・インターフェースをインプリメントするクラスを作成して、コマンド・インターフェースをインプリメントします。このクラスには、ユーザーのインターフェースのメソッド、拡張インターフェース (TargetableCommand インターフェースおよび CompensableCommand インターフェース) から継承されたメソッド、および TargetableCommandImpl クラスに必要な (抽象) メソッド用のコードが含まれています。 TargetableCommandImpl クラスで提供された他のメソッドのデフォルト・インプリメンテーションを上書きすることもできます。 図81 は、247ページの図80 のインターフェースのインプリメンテーション・クラスを示しています。

```
...
import java.lang.reflect.*;
import com.ibm.websphere.command.*;
public class MyCommandImpl extends TargetableCommandImpl
implements MyCommand {
     // Set instance variables here
     ...
     // Implement methods in the MyCommand interface
     ...
     // Implement methods in the CompensableCommand interface
     ...
     // Implement abstract methods in the TargetableCommandImpl class
     ...
}
```

*図 81. コードの例: コマンド・インターフェースのインプリメンテーション・クラスの構造*

### ターゲットを設定し決定するための機能

TargetableCommand のターゲットであるオブジェクトは、CommandTarget インターフェースをインプリメントしなければなりません。このオブジェクトは、エンティティー bean のような実際のサーバー側のオブジェクトでも構いません。あるいは、サーバー用のクライアント側のアダプターでも構いません。 CommandTarget インターフェースのインプリメントを行う人は、望ましいターゲット・サーバー環境でコマンドを適切に実行する責任があります。一般的には、このためには以下のステップを行なう必要があります。

1. サーバー固有のプロトコルを使用して、ターゲット・サーバーにコマンドをコピーする。
2. サーバーでコマンドを実行する。
3. サーバー固有のプロトコルを使用して、実行されたコマンドをターゲット・サーバーからクライアントにコピーする。

CommandTarget インターフェースをインプリメントするための共通した方法には、以下が含まれます。

- ローカル・ターゲット。クライアントの JVM で実行されます。
- サーバー用のクライアント側のアダプター。クライアント側アダプターとしてターゲットをインプリメントする例については、272ページの『コマンド・ターゲットの書き込み (クライアント側のアダプター)』を参照してください。
- Enterprise Bean (セッション bean またはエンティティー bean)。 図82 は、CommandTarget インターフェースをインプリメントするエンティティー bean のリモート・インターフェースと Enterprise Bean クラスの構造を示しています。

```
...
import java.rmi.RemoteException;
import java.util.Properties;
import javax.ejb.*;
import com.ibm.websphere.command.*;
// Remote interface for the MyBean enterprise bean (also a command target)
public interface MyBean extends EJBObject, CommandTarget {
     // Declare methods for the remote interface
     ...
}
// Entity bean class for the MyBean enterprise bean (also a command target)
public class MyBeanClass implements EntityBean, CommandTarget {
     // Set instance variables here
     ...
     // Implement methods in the remote interface
     ...
     // Implement methods in the EntityBean interface
     ...
     // Implement the method in the CommandTarget interface
     ...
}
```

*図 82. コードの例: コマンド・ターゲット・エンティティー bean の構造*

ターゲットを定めることができるコマンドは、別の JVM でリモートで実行できるため、コマンド・パッケージは、コマンドを実行する場所を決定するメカニズムを提供します。ターゲット・ポリシー は、コマンドをターゲットと関連付け、TargetPolicy インターフェースを介して指定されます。このインターフェースをインプリメントして、カスタマイズ済みのターゲット・ポリシーを設計することができます。あるいは、提供された TargetPolicyDefault クラスを使用することができます。詳細については、266ページの『ターゲットおよびターゲット・ポリシー』を参照してください。

### コマンド・パッケージの例外

コマンド・パッケージは、例外クラスのセットを定義します。

CommandException クラスは DistributedException クラスを拡張し、追加のコマンドに関連した例外、UnauthorizedAccessException、UnsetInputPropertiesException、および UnavailableCompensableCommandException として機能します。アプリケーションは CommandException クラスを拡張して、さらに追加の例外を定義することができます。

CommandException クラスは DistributedException クラスを拡張しますが、ユーザーのアプリケーションで DistributedException クラスの機能を使用する必要がなければ、分散例外パッケージ com.ibm.websphere.exception をインポートする必要はありません。分散例外に関する詳細については、233ページの『分散例外パッケージ』を参照してください。

## コマンド・インターフェースの書き込み

コマンド・インターフェースを書き込むには、コマンド・パッケージに組み込まれた 3 つのインターフェースの 1 つ以上を拡張します。すべてのコマンドの基本インターフェースは、Command インターフェースです。このインターフェースは、汎用コマンドのクライアント側のインターフェースだけを提供し、以下の 3 つの基本メソッドを宣言します。

- isReadyToCallExecute − このメソッドは、実行用のサーバーにコマンドを渡す前に、クライアント側で呼び出されます。
- execute − このメソッドは、ターゲットにコマンドを渡し、何らかのデータを戻します。
- reset − このメソッドは、どんな出力プロパティーも execute メソッドが呼び出される前の値に復帰させ、オブジェクトを再利用できるようにします。

ユーザーのインターフェースのインプリメンテーション・クラスは、isReadyToCallExecute のインプリメンテーションを含み、メソッドをリセットしなければなりません。 execute メソッドはどこか他の場所でユーザーのためにインプリメントされます。詳しくは、253ページの『コマンド・インターフェースのインプリメント』を参照してください。ほとんどのコマンドは、Command インターフェースを直接拡張しませんが、提供された拡張機能である TargetableCommand インターフェースおよび CompensableCommand インターフェースのいずれかを使用します。

### TargetableCommand インターフェース

TargetableCommand インターフェースは Command インターフェースを拡張し、コマンドをリモートで実行するために提供します。ほとんどのコマンド

は、ターゲットを定めることができるコマンドです。 TargetableCommand インターフェースはいくつかの追加メソッドを宣言します。

- setCommandTarget － このメソッドを使用すると、ターゲット・オブジェクトをコマンドに指定することができます。
- setCommandTargetName － このメソッドを使用すると、名前によるターゲットをコマンドに指定することができます。
- getCommandTarget － このメソッドは、コマンドのターゲット・オブジェクトを戻します。
- getCommandTargetName － このメソッドは、コマンドのターゲット・オブジェクト名を戻します。
- hasOutputProperties － このメソッドは、出力をもつコマンドのコピーをクライアントに戻す必要があるかどうかを示します。 (インプリメンテーション・クラスも、このコマンドの出力を設定するためのメソッド setHasOutputProperties を戻します。デフォルトでは、hasOutputProperties は真を戻します。)
- setOutputProperties － このメソッドは、クライアントに戻すために、コマンドからの出力値を保管します。
- performExecute － このコマンドは、アプリケーション固有の作業をカプセル化します。このコマンドは、Command インターフェースで宣言された execute メソッドによって、ユーザーのために呼び出されます。

ユーザーがインプリメントしなければならない performExecute メソッドの例外では、これらのコマンドのすべてが TargetableCommandImpl クラスでインプリメントされます。このクラスは、Command インターフェースで宣言された execute メソッドもインプリメントします。

### CompensableCommand インターフェース

CompensableCommand インターフェースも、Command インターフェースを拡張します。補正が可能なコマンドとは、それに関連した別のコマンド (補正プログラム) をもつコマンドです。たとえば、ホテルの予約に続いて航空会社の予約を行なおうとするコマンドは、ホテルの予約ができない場合に、ユーザーが航空会社の予約を取り消すことができる補正コマンドを提供することができます。

CompensableCommand インターフェースは、以下のメソッドの 1 つを宣言します。

- getCompensatingCommand － このメソッドは、元のコマンド効果を取り消すために使用できるコマンドを戻します。

補正可能なコマンドを作成するには、CompensableCommand インターフェースを拡張するインターフェースを書き込みます。一般的には、このようなインターフェースはさらに TargetableCommand インターフェースも拡張します。ユーザーのインターフェース用のインプリメンテーション・クラスでは、getCompensatingCommand メソッドをインプリメントしなければなりません。補正可能なコマンドもインプリメントしなければなりません。

**アプリケーションの例**

この資料の残りの部分を通して使用する例は、コンテナー管理の永続性 (CMP) をもつ、CheckingAccountBean と呼ばれるエンティティー bean を使用します。この bean を使用すると、クライアントは、預金、引き出し、残高の設定、残高の入手、および口座での名前の検索を行なうことができます。このエンティティー bean は、クライアントからのコマンドも受け入れます。コードの例は、コマンド関連のプログラミングを示しています。サーブレット・ベースの例については、272ページの『コマンド・ターゲットの書き込み (クライアント側のアダプター)』を参照してください。

図83 は、ModifyCheckingAccountCmd コマンドのインターフェースを示しています。このコマンドは、ターゲットを定めることも補正することも可能です。したがって、インターフェースは TargetableCommand インターフェースと CompensableCommand インターフェースの両方を拡張します。

```
...
import com.ibm.websphere.exception.*;
import com.ibm.websphere.command.*;
public interface ModifyCheckingAccountCmd
extends TargetableCommand, CompensableCommand {
      float getAmount();
      float getBalance();
      float getOldBalance();       // Used for compensating
      float setBalance(float amount);
      float setBalance(int amount);
      CheckingAccount getCheckingAccount();
      void setCheckingAccount(CheckingAccount newCheckingAccount);
      TargetPolicy getCmdTargetPolicy();
      ...
}
```

*図 83. コードの例: ModifyCheckingAccountCmd インターフェース*

## コマンド・インターフェースのインプリメント

コマンド・パッケージは、クラス TargetableCommandImpl を提供します。このクラスは、performExecute メソッドを除く、TargetableCommand インターフェースのすべてのメソッドをインプリメントします。また、Command インターフェースから execute メソッドもインプリメントします。アプリケーションのコマンド・インターフェースをインプリメントするには、TargetableCommandImpl クラスを拡張し、ユーザーのコマンド・インターフェースをインプリメントするクラスを書き込まなければなりません。 図84 は、ModifyCheckingAccountCmdImpl クラスの構造を示しています。

```
...
public class ModifyCheckingAccountCmdImpl extends TargetableCommandImpl
implements ModifyCheckingAccountCmd
{
    // Variables
    ...
    // Methods
    ...
}
```

*図 84. コードの例: ModifyCheckingAccountCmdImpl クラスの構造*

クラスは任意の変数を宣言し、以下のメソッドをインプリメントしなければなりません。

- ユーザーのコマンド・インターフェースで定義した任意のメソッド
- Command コマンドからの isReadyToCallExecute メソッドおよび reset メソッド
- TargetableCommand インターフェースからの performExecute メソッド
- ユーザーのコマンドが補正可能である場合には、CompensableCommand コマンドからの getCompensatingCommand メソッド。補正可能なコマンドもインプリメントしなければなりません。

TargetableCommandImpl クラスで提供された最終以外のインプリメンテーションを上書きすることもできます。デフォルトのインプリメンテーションは最終フィールド、一時フィールド、または静的フィールドを保管しないため、再インプリメンテーションに最も適切な候補は、setOutputProperties メソッドです。

### インスタンス変数およびクラス変数の定義

ModifyCheckingAccountCmdImpl クラスは、CheckingAccount エンティティー bean のリモート・インターフェースを含めて、このクラスのメソッドが使用する変数、当座預金で操作を取り込むために使用する変数 (残高および総額)、お

よび補正コマンドを宣言します。 図85 は、ModifyCheckingAccountCmd コマンドが使用する変数を示しています。

```
...
public class ModifyCheckingAccountCmdImpl extends TargetableCommandImpl
implements ModifyCheckingAccountCmd
{
    // Variables
    public float balance;
    public float amount;
    public float oldBalance;
    public CheckingAccount checkingAccount;
    public ModifyCheckingAccountCompensatorCmd
                        modifyCheckingAccountCompensatorCmd;
    ...
}
```

*図 85. コードの例: ModifyCheckingAccountCmdImpl クラスの変数*

## コマンド固有のメソッドのインプリメント

ModifyCheckingAccountCmd インターフェースは、コマンド・パッケージの他のインターフェースを拡張する他に、いくつかのコマンド固有のメソッドを定義します。このようなコマンド固有のメソッドは、ModifyCheckingAccountCmdImpl クラスでインプリメントされます。

ユーザーは、コマンドをインスタンス化する方法を提供しなければなりません。コマンド・パッケージではそのメカニズムを指定しません。したがって、ユーザーはユーザーのアプリケーションに最適な技法を選択することができます。高速で最も効果的な技法は、コンストラクターを使用することです。最も柔軟な技法は、ファクトリーを使用することです。また、コマンドは内部的に JavaBeans コンポーネントとしてインプリメントされるため、標準の Beans.instantiate メソッドを使用することもできます。ModifyCheckingAccountCmd コマンドはコンストラクターを使用します。

255ページの図86 は、コマンド用の 2 つのコンストラクターを示しています。これらのコンストラクターの相違は、最初のコンストラクターはコマンドのターゲットを決定するためにデフォルトのターゲット・ポリシーを使用し、2 番目のコンストラクターを使用するとカスタム・ポリシーを指定できることです。 (ターゲットおよびターゲット・ポリシーに関する詳細については、266ページの『ターゲットおよびターゲット・ポリシー』を参照してください。)

どちらのコンストラクターも引き数として CommandTarget オブジェクトを取り、そのオブジェクトを CheckingAccount タイプに振り当てます。CheckingAccount インターフェースは、CommandTarget インターフェースと EJBObject の両方を拡張します (264ページの図95 を参照)。結果として生じる checkingAccount オブジェクトは、bean のリモート・インターフェースを使用して、望ましいサーバーにコマンドを発送します。 (CommandTarget オブジェクトに関する詳細については、263ページの『コマンド・ターゲットの書き込み (サーバー)』を参照してください。)

```
...
public class ModifyCheckingAccountCmdImpl extends TargetableCommandImpl
implements ModifyCheckingAccountCmd
{
    // Variables
    ...
    // Constructors
    // First constructor: relies on the default target policy
    public ModifyCheckingAccountCmdImpl(CommandTarget target,
                    float newAmount)
    {
        amount = newAmount;
        checkingAccount = (CheckingAccount)target;
        setCommandTarget(target);
    }
    // Second constructor: allows you to specify a custom target policy
    public ModifyCheckingAccountCmdImpl(CommandTarget target,
                    float newAmount,
                    TargetPolicy targetPolicy)
    {
        setTargetPolicy(targetPolicy);
        amount = newAmount;
        checkingAccount = (CheckingAccount)target;
        setCommandTarget(target);
    }
    ...
}
```

*図 86. コードの例: ModifyCheckingAccountCmdImpl クラスのコンストラクター*

256ページの図87 は、コマンド固有のメソッドのインプリメンテーションを示しています。

- setBalance ー このメソッドは口座の残高を設定します。
- getAmount ー このメソッドは、預金または引き出しの金額を戻します。
- getOldBalance, getBalance ー これらのメソッドは、操作前後の残高を取り込みます。
- getCmdTargetPolicy ー このメソッドは、現在のターゲット・ポリシーを検索します。

- setCheckingAccount、getCheckingAccount － これらのメソッドは、現在の当座預金口座を設定し取り出します。

```
...
public class ModifyCheckingAccountCmdImpl extends TargetableCommandImpl
implements ModifyCheckingAccountCmd
{
    // Variables
    ...
    // Constructors
    ...
    // Methods in ModifyCheckingAccountCmd interface
    public float getAmount() {
        return amount;
    }
    public float getBalance() {
        return balance;
    }
    public float getOldBalance() {
        return oldBalance;
    }
    public float setBalance(float amount) {
        balance = balance + amount;
        return balance;
    }
    public float setBalance(int amount) {
        balance += amount ;
        return balance;
    }
    public TargetPolicy getCmdTargetPolicy() {
        return getTargetPolicy();
    }
    public void setCheckingAccount(CheckingAccount newCheckingAccount) {
        if (checkingAccount == null) {
            checkingAccount = newCheckingAccount;
        }
        else
            System.out.println("Incorrect Checking Account (" +
                newCheckingAccount + ") specified");
    }
    public CheckingAccount getCheckingAccount() {
        return checkingAccount;
    }
    ...
}
```

*図 87. コードの例: ModifyCheckingAccountCmdImpl クラスのコマンド固有のメソッド*

ModifyCheckingAccountCmd コマンドは、当座預金で機能します。コマンドは JavaBeans コンポーネントとしてインプリメントされるため、ユーザーは、標準 JavaBeans 技法を使用してコマンドの入出力プロパティーを管理します。た

とえば、セット・メソッド (setCheckingAccount など) を使って入力プロパティーを初期化し、入手メソッド (getCheckingAccount など) を使って出力プロパティーを検索します。入手メソッドは、コマンドの実行メソッドが呼び出されるまで、作動しません。

### コマンド・インターフェースからのメソッドのインプリメント

Command インターフェースは 2 つのメソッド isReadyToCallExecute および reset を宣言します。これらのメソッドは、アプリケーション・プログラマーがインプリメントしなければなりません。 図88 は、ModifyCheckingAccountCmd コマンドのインプリメンテーションを示しています。 isReadyToCallExecute メソッドのインプリメンテーションは、checkingAccount 変数を確実に設定します。リセット・メソッドはすべての変数を開始値に戻します。

```
...
public class ModifyCheckingAccountCmdImpl extends TargetableCommandImpl
implements ModifyCheckingAccountCmd
{
    ...
    // Methods from the Command interface
    public boolean isReadyToCallExecute() {
        if (checkingAccount != null)
            return true;
        else
            return false;
    }
    public void reset() {
        amount = 0;
        balance = 0;
        oldBalance = 0;
        checkingAccount = null;
        targetPolicy = new TargetPolicyDefault();
    }
   ...
}
```

*図 88. コードの例: ModifyCheckingAccountCmdImpl クラスの Command インターフェースからのメソッド*

### TargetableCommand インターフェースからのメソッドのインプリメント

TargetableCommand インターフェースは、1 つのメソッド performExecute を宣言します。このメソッドは、アプリケーション・プログラマーがインプリメントしなければなりません。 258ページの図89 は、ModifyCheckingAccountCmd

コマンドのインプリメンテーションを示しています。 performExecute メソッドのインプリメンテーションは、以下のことを行ないます。

- 現在の残高を保管する (したがって、compensator コマンドによってやり直すことができます)
- 新規残高を計算する
- 現在の残高を新規の残高に設定する
- hasOutputProperties メソッドが確実に真を戻し、その値をクライアントに戻すようにする

さらに、ModifyCheckingAccountCmdImpl クラスは、setOutputProperties メソッドのデフォルトのインプリメンテーションを上書きします。

```
...
public class ModifyCheckingAccountCmdImpl extends TargetableCommandImpl
implements ModifyCheckingAccountCmd
{
    ...
    // Method from the TargetableCommand interface
    public void performExecute() throws Exception {
        CheckingAccount checkingAccount = getCheckingAccount();
        oldBalance =  checkingAccount.getBalance();
        balance = oldBalance+amount;
        checkingAccount.setBalance(balance);
        setHasOutputProperties(true);
    }
    public void setOutputProperties(TargetableCommand fromCommand) {
            try {
            if (fromCommand != null) {
                ModifyCheckingAccountCmd modifyCheckingAccountCmd =
                    (ModifyCheckingAccountCmd) fromCommand;
                this.oldBalance = modifyCheckingAccountCmd.getOldBalance();
                this.balance = modifyCheckingAccountCmd.getBalance();
                this.checkingAccount =
                    modifyCheckingAccountCmd.getCheckingAccount();
                this.amount = modifyCheckingAccountCmd.getAmount();
            }
        }
        catch (Exception ex) {
             System.out.println("Error in setOutputProperties.");
        }
    }
    ...
}
```

*図 89. コードの例: ModifyCheckingAccountCmdImpl クラスの TargetableCommand インターフェースからのメソッド*

### CompensableCommand インターフェースのインプリメント

CompensableCommand インターフェースは、1 つのメソッド getCompensatingCommand を宣言します。このメソッドは、アプリケーション・プログラマーがインプリメントしなければなりません。図90 は、ModifyCheckingAccountCmd コマンドのインプリメンテーションを示しています。インプリメンテーションは、現行コマンドに関連した ModifyCheckingAccountCompensatorCmd コマンドのインスタンスだけを戻します。

```
...
public class ModifyCheckingAccountCmdImpl extends TargetableCommandImpl
implements ModifyCheckingAccountCmd
{
    ...
    // Method from CompensableCommand interface
    public Command getCompensatingCommand() throws CommandException {
        modifyCheckingAccountCompensatorCmd =
            new ModifyCheckingAccountCompensatorCmd(this);
        return (Command)modifyCheckingAccountCompensatorCmd;
    }
}
```

*図 90. コードの例: ModifyCheckingAccountCmdImpl クラスの CompensableCommand インターフェースからのメソッド*

***補正コマンドの書き込み:*** 補正可能なコマンドを使用するアプリケーションには、基本コマンド (CompensableCommand として宣言された) と補正コマンドの 2 つの異なるコマンドが必要です。例のアプリケーションでは、基本コマンドは ModifyCheckingAccountCmd インターフェースで宣言され、ModifyCheckingAccountCmdImpl クラスでインプリメントされます。このコマンドは補正可能なコマンドでもあるため、その作業をやり直すために設計された、関連する 2 次コマンドがあります。補正可能なコマンドを作成する場合には、補正コマンドも書き込む必要があります。

補正コマンドを書き込むには、インターフェースを書き込むことと、インプリメンテーション・クラスを書き込むという、元のコマンドを書き込むのとまったく同じステップが必要になります。場合によっては、より簡単になる可能性があります。たとえば、ModifyCheckingAccountCmd を補正するためのコマンドには、元のコマンドに定義されたメソッド以外は必要ありません。したがって、インターフェースは必要ありません。ModifyCheckingAccountCompensatorCmd という補正コマンドに必要なのは、TargetableCommandImpl クラスを拡張するクラスでインプリメントされることだけです。このクラスは、以下のことを行なわなければなりません。

- コマンドをインスタンス化する方法を提供する。例では、コンストラクターを使用します。
- 3 つの必須メソッドをインプリメントする。
  - isReadyToCallExecute および reset － どちらも Command インターフェースのものです。
  - performExecute － TargetableCommand インターフェースのものです。

図91 は、インプリメンテーション・クラスの構造、その変数 (もとのコマンドと相当する当座預金を表す)、およびコンストラクターを示しています。コンストラクターは、基本コマンドと口座の参照を簡単にインスタンス化します。

```
...
public class ModifyCheckingAccountCompensatorCmd extends TargetableCommandImpl
{
    public ModifyCheckingAccountCmdImpl modifyCheckingAccountCmdImpl;
    public CheckingAccount checkingAccount;

    public ModifyCheckingAccountCompensatorCmd(
                 ModifyCheckingAccountCmdImpl originalCmd)
    {
        // Get an instance of the original command
        modifyCheckingAccountCmdImpl = originalCmd;
        // Get the relevant account
        checkingAccount = originalCmd.getCheckingAccount();
     }
     // Methods from the Command and Targetable Command interfaces
    ....
}
```

*図 91. コードの例: ModifyCheckingAccountCompensatorCmd クラスの変数およびコンストラクター*

261ページの図92 は、継承メソッドのインプリメンテーションを示しています。 isReadyToCallExecute メソッドのインプリメンテーションは、checkingAccount 変数がインスタンス化されたことを確認します。

performExecute メソッドは、実際の当座預金の残高が、元のコマンドが戻すものと一致するかどうかを検証します。一致している場合には、ModifyCheckingAccountCmd コマンドを使用して、現行の残高を前回保管された残高と置き換えます。最後に、補正コマンドをやり直す必要がある場合には、最新の残高を保管します。 reset メソッドが行なう作業はありません。

```
...
public class ModifyCheckingAccountCompensatorCmd extends TargetableCommandImpl
{
    // Variables and constructor
    ....
   // Methods from the Command and TargetableCommand interfaces
   public boolean isReadyToCallExecute() {
        if (checkingAccount != null)
            return true;
        else
            return false;
    }
    public void performExecute() throws CommandException
    {
        try {
            ModifyCheckingAccountCmdImpl originalCmd =
modifyCheckingAccountCmdImpl;
            // Retrieve the checking account modified by the original command
            CheckingAccount checkingAccount = originalCmd.getCheckingAccount();

            if (modifyCheckingAccountCmdImpl.balance ==
                checkingAccount.getBalance()) {
                    // Reset the values on the original command
                    checkingAccount.setBalance(originalCmd.oldBalance);
                    float temp = modifyCheckingAccountCmdImpl.balance;
                    originalCmd.balance = originalCmd.oldBalance;
                    originalCmd.oldBalance = temp;
            }
            else {
                // Balances are inconsistent, so we cannot compensate
                throw new CommandException(
"Object modified since this command ran.");
            }
        }
        catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
    public void reset() {}
}
```

*図 92. コードの例: ModifyCheckingAccountCompensatorCmd クラスのメソッド*

## コマンドの使用法

コマンドを使用するために、クライアントはコマンドのインスタンスを作成し、コマンドの実行メソッドを呼び出します。コマンドによっては、他のメソッドを呼び出す必要がある場合があります。細部は、アプリケーションによって異なります。

例のアプリケーションでは、サーバーは CheckingAccountBean でエンティティー Enterprise Bean です。この Enterprise Bean を使用するために、クライアントは bean のホーム・インターフェースの参照を入手します。次に、クライアントは、ホーム・インターフェースの参照と bean の探知メソッドの 1 つを使用して、bean のリモート・インターフェースの参照を獲得します。適切な bean がない場合には、クライアントは、ホーム・インターフェースで作成メソッドを使用して適切な bean を作成することができます。この作業のすべてが、本書の別のところで扱う標準 Enterprise Bean プログラミングです。

図93 は、ModifyCheckingAccountCmd コマンドの使用法を示しています。この作業は、適切な CheckingAccount Bean が見つかるかまたは作成されると実行されます。コードはコマンドをインスタンス化し、コマンドに定義されたコンストラクターの 1 つを使用して入力値を設定します。ヌルの引き数は、コマンドはデフォルトのターゲット・ポリシーを使用してサーバーを検索する必要があることを示します。 1000 は、コマンドが当座預金の残高に追加しようする総額です。 (コマンド・パッケージがデフォルトを使用してコマンドのターゲットを決定する方法に関する詳細については、266ページの『デフォルトのターゲット・ポリシー』を参照してください。) コマンドがインスタンス化されると、コードは setCheckingAccount メソッドを呼び出して、変更する口座を識別します。最後に、コマンドの実行メソッドが呼び出されます。

```
{
    ...
    CheckingAccount checkingAccount
    ...
    try {
        ModifyCheckingAccountCmd cmd =
            new ModifyCheckingAccountCmdImpl(null, 1000);
        cmd.setCheckingAccount(checkingAccount);
        cmd.execute();
    }
    catch (Exception e) {
       System.out.println(e.getMessage());
    }
    ...
}
```

*図 93. コードの例: ModifyCheckingAccountCmd コマンドの使用法*

## 補正コマンドの使用法

補正コマンドを使用するには、基本コマンドに関連した補正プログラムを検索し、その実行メソッドを呼び出さなければなりません。 263ページの図94 は、オリジナルのコマンドを実行するために使用するコードと、補正コマンドを実

行して作業をやり直すオプションをユーザーに与えるために使用するコードを示しています。

```
{
    ...
    CheckingAccount checkingAccount
    ....
    try {
        ModifyCheckingAccountCmd cmd =
            new ModifyCheckingAccountCmdImpl(null, 1000);
        cmd.setCheckingAccount(checkingAccount);
        cmd.execute();
        ...
        System.out.println("Would you like to undo this work?  Enter Y or N");
            try {
            // Retrieve and validate user's response
            ...
        }
        ...
        if (answer.equalsIgnoreCase(Y)) {
            Command compensatingCommand = cmd.getCompensatingCommand();
            compensatingCommand.execute();
        }
    }
    catch (Exception e) {
       System.out.println(e.getMessage());
    }
    ...
}
```

*図 94. コードの例: ModifyCheckingAccountCompensator コマンドの使用法*

## コマンド・ターゲットの書き込み (サーバー)

コマンドを受け入れるために、サーバーは、CommandTarget インターフェースとその唯一のメソッド executeCommand をインプリメントしなければなりません。

アプリケーションの例では、Enterprise Bean の CommandTarget インターフェースをインプリメントしています。 (サーブレット・ベースの例については、272ページの『コマンド・ターゲットの書き込み (クライアント側のアダプター)』を参照してください。) ターゲット Enterprise Bean は、セッション bean またはエンティティー bean でも構いません。別のエンティティー bean など、特定のサーバーにコマンドを転送するターゲット Enterprise Bean を書き込むことができます。この場合には、特定のターゲットで送信されたコマンド

はすべて、ターゲット Enterprise Bean を通り抜けます。コマンドの作業をローカルで行なうターゲット Enterprise Bean を書き込むこともできます。

以下を行ない、Enterprise Bean をコマンドのターゲットにします。

- bean のリモート・インターフェースを定義するときに、CommandTarget インターフェースを拡張する。このリモート・インターフェースは EJBObject インターフェースも拡張しなければなりません。
- bean クラスをインプリメントするときに、CommandTarget インターフェースをインプリメントする。このクラスは、SessionBean インターフェースまたは EntityBean インターフェースのどちらかもインプリメントしなければなりません。

例のアプリケーションのターゲットは、CheckingAccountBean と呼ばれる Enterprise Bean です。この bean のリモート・インターフェース CheckingAccount は、EJBObject インターフェースの他に CommandTarget インターフェースも拡張します。リモート・インターフェースで宣言されたメソッドは、コマンドが使用するメソッドとは無関係です。 executeCommand は bean のホームでもリモート・インターフェースのどちらでも宣言されません。図95 は、CheckingAccount インターフェースを示しています。

```
...
import com.ibm.websphere.command.*;
import javax.ejb.EJBObject;
import java.rmi.RemoteException;
public interface CheckingAccount extends CommandTarget, EJBObject {
    float deposit (float amount) throws RemoteException;
    float deposit (int amount) throws RemoteException;
    String getAccountName() throws RemoteException;

    float getBalance() throws RemoteException;
    float setBalance(float amount) throws RemoteException;

    float withdrawal (float amount) throws RemoteException, Exception;
    float withdrawal (int amount) throws RemoteException, Exception;
}
```

*図 95. コードの例: CheckingAccount エンティティー bean のリモート・インターフェース、およびコマンド・ターゲット*

Enterprise Bean クラスの CheckingAccountBean は、CommandTarget インターフェースだけでなく、EntityBean インターフェースもインプリメントします。このクラスには、リモート・インターフェースのメソッドのビジネス・ロジック、必要なライフ・サイクル・メソッド (ejbActivate、ejbStore など)、および CommandTarget インターフェースによって宣言された executeCommand が含ま

れています。executeCommand メソッドは、Enterprise Bean クラスの唯一のコマンド固有コードです。このメソッドはコマンドの performExecute メソッドを実行を試み、エラーが発生する場合には CommandException を throw します。performExecute メソッドが正常に実行されると、executeCommand メソッドは hasOutputProperties メソッドを使用して、戻さなければならない出力プロパティーがあるかどうかを決定します。コマンドに出力プロパティーがある場合には、このメソッドはコマンド・オブジェクトをクライアントに戻します。 図96 は、CheckingAccountBean クラスで相当する部分を示しています。

```
...
public class CheckingAccountBean implements EntityBean, CommandTarget {
    // Bean variables
    ...
    // Business methods from remote interface
    ...
    // Life-cycle methods for CMP entity beans
    ...
    // Method from the CommandTarget interface
    public TargetableCommand executeCommand(TargetableCommand command)
    throws RemoteException, CommandException
    {
            try {
                command.performExecute();
        }
        catch (Exception ex) {
            if (ex instanceof RemoteException) {
                RemoveException remoteException = (RemoteException)ex;
                if (remoteException.detail != null) {
                    throw new CommandException(remoteException.detail);
                }
                throw new CommandException(ex);
            }
        }
        if (command.hasOutputProperties()) {
            return command;
        }
        return null;
    }
}
```

*図 96. コードの例: CheckingAccount エンティティー bean の bean クラス、およびコマンド・ターゲット*

## ターゲットおよびターゲット・ポリシー

ターゲットを定めることができるコマンドは、TargetableCommand インターフェースを拡張します。このインターフェースを使用すると、クライアントはコマンドを特定のサーバーに送信することができます。TargetableCommand インターフェース (および TargetableCommandImpl クラス) は、setCommandTarget メソッドと setCommandTargetName メソッドという、クライアントがターゲットを指定するための 2 つの方法を提供します。 (これらのメソッドは、250ページの『TargetableCommand インターフェース』で紹介されています。) setCommandTarget メソッドを使用すると、クライアントは、ターゲット・オブジェクトをコマンドで直接設定することができます。 setCommandTargetName メソッドを使用すると、クライアントは、サーバーを名前で参照することができます。この方法は、クライアントがサーバー・オブジェクトを直接知らないときに役に立ちます。ターゲットを定めることができるコマンドには、対応する getCommandTarget メソッドと getCommandTargetName メソッドもあります。

コマンド・パッケージは、コマンドのターゲットを識別できる必要があります。ターゲットを指定する方法が 1 つしかないため、およびアプリケーションが異なると要件も異なる可能性があるため、コマンド・パッケージは選択アルゴリズム指定しません。代わりに、1 つのメソッド getCommandTarget が指定された TargetPolicy インターフェースとデフォルトのインプリメンテーションを提供します。これによって、アプリケーションは、適切な場合にコマンドのターゲットを決定するためのカスタム・アルゴリズムを考案することができます。

### デフォルトのターゲット・ポリシー

コマンド・パッケージは、TargetPolicyDefault クラスにある TargetPolicy インターフェースのデフォルトのインプリメンテーションを提供します。このデフォルトのインプリメンテーションを使用する場合には、以下の 4 つのオプションのオーダー・シーケンスを調べて、コマンドがターゲットを決定します。

1. CommandTarget 値
2. CommandTargetName 値
3. 特定のコマンドのターゲットの登録済みマッピング
4. 定義済みのデフォルト・ターゲット

ターゲットが検出されない場合には、ヌルを戻します。

TargetPolicyDefault クラスは、ターゲットをもつコマンドの割り当てを管理するためのメソッド (registerCommand、unregisterCommand、および listMappings) と、ターゲットのデフォルト名を設定するためのメソッド

(setDefaultTargetName) を提供します。デフォルトのターゲット名は com.ibm.websphere.command.LocalTarget で、LocalTarget はコマンドの performExecute メソッドをローカルで実行するクラスです。図97 は、TargetPolicyDefault クラスの関係のある変数およびメソッドを示しています。

```
...
public class TargetPolicyDefault implements TargetPolicy, Serializable
{
    ...
    protected String defaultTargetName = "com.ibm.websphere.command.LocalTarget";
    public CommandTarget getCommandTarget(TargetableCommand command) {
        ... }
    public Dictionary listMappings() {
        ... }
    public void registerCommand(String commandName, String targetName) {
        ... }
    public void unregisterCommand(String commandName) {
        ... }
    public void seDefaultTargetName(String defaultTargetName) {
        ... }
}
```

*図 97. コードの例: TargetPolicyDefault クラス*

***コマンド・ターゲットの設定:*** ModifyCheckingAccountImpl クラスは、2 つのコマンド・コンストラクターを提供します (255ページの図86 を参照)。このコマンド・コンストラクターの 1 つは、コマンドを引き数として取り、明示的にデフォルト・ターゲットを使用してターゲットを位置指定します。262ページの図93 で使用するコンストラクターは、ヌルのターゲットを渡します。その結果、デフォルトのターゲット・ポリシーはその選択項目を走査し、デフォルトのターゲット名 LocalTarget を検出します。

268ページの図98 の例では、同じコンストラクターを使用して、ターゲットを明示的に設定します。この例は、以下のように、262ページの図93 とは異なります。

- コマンド・ターゲットはヌルではなく当座預金に設定されます。デフォルトのターゲット・ポリシーは選択項目の走査を開始し、調べた最初の場所でターゲットを検出します。
- コマンド・ターゲットは、setCheckingAccount メソッドを呼び出して、コマンドが作動する口座を示す必要はありません。コンストラクターがターゲット変数をターゲットと口座の両方として使用します。

```
{
    ...
    CheckingAccount checkingAccount
    ....
    try {
        ModifyCheckingAccountCmd cmd =
            new ModifyCheckingAccountCmdImpl(checkingAccount, 1000);
        cmd.execute();
    }
    catch (Exception e) {
       System.out.println(e.getMessage());
    }
    ...
}
```

*図 98. コードの例: CommandTarget を使ったターゲットの識別*

***コマンド・ターゲット名の設定:*** クライアントがコマンドのターゲットを名前で設定する必要がある場合には、コマンドの setCommandTargetName メソッドを使用することができます。269ページの図99 は、この手法を示しています。この例は、以下のように、262ページの図93 と比較しています。

- どちらも、コンストラクターのコマンド・ターゲットを明示的にヌルに設定します。
- どちらも、setCheckingAccount メソッドを使用して、コマンドが作動するべき口座を示します。
- この例は、setCommandTargetName メソッドを使用して、明示的にターゲット名を設定します。デフォルトのターゲット・ポリシーがその選択項目を走査すると、最初の選択項目ではヌルを、2 番目では名前を検出します。

```
{
    ...
    CheckingAccount checkingAccount
    ....
    try {
        ModifyCheckingAccountCmd cmd =
            new ModifyCheckingAccountCmdImpl(null, 1000);
        cmd.setCheckingAccount(checkingAccount);
        cmd.setCommandTargetName("com.ibm.sfc.cmd.test.CheckingAccountBean");
        cmd.execute();
    }
    catch (Exception e) {
       System.out.println(e.getMessage());
    }
    ...
}
```

*図 99. コードの例: CommandTargetName を使ったターゲットの識別*

***ターゲット名へのコマンドのマッピング:*** デフォルトのターゲット・ポリシーは、ターゲットを使ってコマンドを登録することも許可します。コマンドをターゲットにマッピングすることは、構成ツールを最適に終了する管理用タスクです。 WebSphere Application Server 管理コンソールは、コマンドとターゲット間のマッピングの構成をまだサポートしていません。ターゲットを使ってコマンドを登録するサポートが必要なアプリケーションは、マッピングを管理するためのツールを提供しなければなりません。これらのツールは、ビジュアル・インターフェースまたはコマンド行ツールでも構いません。

270ページの図100 は、ターゲットを使ったコマンドの登録を示しています。コマンド・クラスおよびターゲットの名前は、コードで明示的になりますが、実際には、これらの値は、ユーザー・インターフェースのフィールドまたはコマンド行ツールへの引き数から出されます。プログラムが、262ページの図93 で示すような、ターゲットにヌルを指定したコマンドを作成する場合、デフォルトのターゲット・ポリシーがその選択項目を走査すると、最初の選択項目ではヌルを、2 番目では名前を、3 番目ではマッピングを検出します。

```
{
    ...
   targetPolicy.registerCommand(
        "com.ibm.sfc.cmd.test.ModifyCheckingAccountImpl",
        "com.ibm.sfc.cmd.test.CheckingAccountBean");
    ...
}
```

*図 100. コードの例: 外部アプリケーションでのターゲットへのコマンドのマッピング*

## ターゲット・ポリシーのカスタマイズ

TargetPolicy インターフェースをインプリメントすることと、ユーザーのアプリケーションに適切な getCommandTarget メソッドを提供することによって、カスタム・ターゲット・ポリシーを定義することができます。 TargetableCommandImpl クラスは、カスタム・ターゲット・ポリシーを管理するための setTargetPolicy メソッドと getTargetPolicy メソッドを提供します。

ここまでで、すべてのコマンドのターゲットは、当座預金エンティティー bean になりました。誰かが、コマンド・ターゲットとしても機能するセッション Enterprise Bean (MySessionBean) を導入しすると想定します。 271ページの図 101 は、各コマンドのターゲットを MySessionBean に設定する簡単なカスタム・ポリシーを示しています。

```
...
import java.io.*;
import java.util.*;
import java.beans.*;
import com.ibm.websphere.command.*;
public class CustomTargetPolicy implements TargetPolicy, Serializable {
    public CustomTargetPolicy {
        super ();
    }
    public CommandTarget getCommandTarget(TargetableCommand command) {
        CommandTarget = null;
            try {
            target = (CommandTarget)Beans.instantiate(null,
                        "com.ibm.sfc.cmd.test.MySessionBean");
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

*図 101. コードの例: カスタム・ターゲット・ポリシーの作成*

コマンドは JavaBeans コンポーネントとしてインプリメントされるため、カスタム・ターゲット・ポリシーを使用するには、java.beans パッケージのインポートと多少の基本の JavaBeans コードを書き込む必要があります。また、ユーザーのカスタム・ターゲット・ポリシー・クラスも、java.io.Serializable インターフェースをインプリメントしなければなりません。

***カスタム・ターゲット・ポリシーの使用法:*** ModifyCheckingAccountImpl クラスは、2 つのコマンド・コンストラクターを提供します (255ページの図86 を参照)。このコンストラクターの一方は、デフォルトのターゲット・ポリシーを使用し、他方は引き数としてターゲット・ポリシー・オブジェクトを取ります。このターゲット・ポリシー・オブジェクトを使用すると、カスタム・ターゲット・ポリシーを使用することができます。 272ページの図102 の例は 2 番目のコンストラクターを使用し、ヌル・ターゲットおよびカスタム・ターゲット・ポリシーを渡します。その結果、ターゲットを決定するために、カスタム・ポリシーが使用されます。コマンドが実行されると、コードは reset メソッドを使用して、ターゲット・ポリシーをデフォルトに戻します。

```
{
    ...
    CheckingAccount checkingAccount
    ....
    try {
        CustomTargetPolicy customPolicy = new CustomTargetPolicy();
        ModifyCheckingAccountCmd cmd =
            new ModifyCheckingAccountCmdImpl(null, 1000, customPolicy);
        cmd.setCheckingAccount(checkingAccount);
        cmd.execute();
        cmd.reset();
    }
    catch (Exception e) {
       System.out.println(e.getMessage());
    }
}
```

*図 102. コードの例: カスタム・ターゲット・ポリシーの使用法*

## コマンド・ターゲットの書き込み (クライアント側のアダプター)

任意の Java アプリケーションでコマンドを使用することができますが、クライアントからサーバーにコマンドを送信する手段が変わります。 252ページの『アプリケーションの例』で説明するアプリケーションは、 Enterprise Bean を使用しました。このセクションの例は、HTTP プロトコルを介してサーブレットにコマンドを送信できる方法を示しています。

この例では、クライアントは CommandTarget インターフェースをローカルでインプリメントします。273ページの図103 は、クライアント側のクラスの構造を示しています。このクラスは、executeCommand メソッドをインプリメントして CommandTarget インターフェースをインプリメントします。

```
...
import java.io.*;
import java.rmi.*;
import com.ibm.websphere.command.*;
public class ServletCommandTarget implements CommandTarget, Serializable
{
    protected String hostName = "localhost";
  public static void main(String args[]) throws Exception
    {
        ....
    }
    public TargetableCommand executeCommand(TargetableCommand command)
     throws CommandException
    {
        ....
    }
      public static final byte[] serialize(Serializable serializable)
         throws IOException {
        ...  }
    public String getHostName() {
        ... }
    public void setHostName(String hostName) {
        ... }
    private static void showHelp() {
         ... }
}
```

*図 103. コードの例: ターゲット用のクライアント側アダプターの構造*

クライアント側のアダプターのメイン・メソッドは、 図104 に示すように、 CommandTarget オブジェクトを構成し初期化します。

```
public static void main(String args[]) throws Exception
{
    String hostName = InetAddress.getLocalHost().getHostName();
    String fileName = "MyServletCommandTarget.ser";
    // Parse the command line
    ...
    // Create and initialize the client-side CommandTarget adapter
    ServletCommandTarget servletCommandTarget = new ServletCommandTarget();
    servletCommandTarget.setHostName(hostName);
    ...
    // Flush and close output streams
    ...
}
```

*図 104. コードの例: クライアント側のアダプターのインスタンス化*

### クライアント側のアダプターのインプリメント

CommandTarget インターフェースは、1 つのメソッド executeCommand を宣言します。このメソッドは、クライアントがインプリメントします。

executeCommand メソッドは、TargetableCommand オブジェクトを入力として取ります。このメソッドも TargetableCommand を戻します。 275ページの図 105 は、クライアント側のアダプターで使用するメソッドのインプリメンテーションを示しています。このインプリメンテーションは、以下のことを行ないます。

- インプリメンテーションが受け取るコマンドを逐次化する
- サーブレットへの HTTP 接続を作成する
- サーバーに送信され戻されたコマンドを処理するための、入出力ストリームを作成する
- 出力ストリームにコマンドを配置する
- サーバーにコマンドを送信する
- 入力ストリームから戻されたコマンドを検索する
- 戻されたコマンドを executeCommand メソッドの呼び出し元に戻す

```
public TargetableCommand executeCommand(TargetableCommand command)
     throws CommandException
{
    try {
        // Serialize the command
        byte[] array = serialize(command);
        // Create a connection to the servlet
        URL url = new URL
            ("http://" + hostName +
            "/servlet/com.ibm.websphere.command.servlet.CommandServlet");
        HttpURLConnection httpURLConnection =
            (HttpURLConnection) url.openConnection();
        // Set the properties of the connection
        ...
        // Put the serialized command on the output stream
        OutputStream outputStream = httpURLConnection.getOutputStream();
        outputStream.write(array);
        // Create a return stream
        InputStream inputStream = httpURLConnection.getInputStream();
        // Send the command to the servlet
        httpURLConnection.connect();
            ObjectInputStream objectInputStream =
                new ObjectInputStream(inputStream);
        // Retrieve the command returned from the servlet
        Object object = objectInputStream.readObject();
        if (object instanceof CommandException) {
            throw ((CommandException) object);
        }
        // Pass the returned command back to the calling method
        return (TargetableCommand) object;
    }
    // Handle exceptions
    ....
}
```

*図 105. コードの例: executeCommand メソッドのクライアント側のインプリメンテーション*

## サーブレットでのコマンドの実行

コマンドを実行するサーブレットは、276ページの図106 に示してあります。サービス・メソッドは入力ストリームからのコマンドを検索し、コマンドで performExecute メソッドを実行します。その結果生じるオブジェクトは、クライアントに戻さなければならない出力プロパティーを伴い、出力ストリームに配置され、クライアントに送り返されます。

```
...
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import com.ibm.websphere.command.*;
public class CommandServlet extends HttpServlet {
    ...
    public void service(HttpServletRequest request,
                        HttpServletResponse response)
        throws ServletException, IOException
     {
        try {
            ...
            // Create input and output streams
            InputStream inputStream = request.getInputStream();
            OutputStream outputStream = response.getOutputStream();
            // Retrieve the command from the input stream
            ObjectInputStream objectInputStream =
                new ObjectInputStream(inputStream);
            TargetableCommand command = (TargetableCommand)
                objectInputStream.readObject();
            // Create the command for the return stream
            Object returnObject = command;

            // Try to run the command's performExecute method
            try {
                command.performExecute();
            }
            // Handle exceptions from the performExecute method
            ...

            // Return the command with any output properties
            ObjectOutputStream objectOutputStream =
                new ObjectOutputStream(outputStream);
            objectOutputStream.writeObject(returnObject);
            // Flush and close output streams
            ...
        }
        catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

*図 106. コードの例: サーブレットでのコマンドの実行*

この例では、ターゲットはコマンドで performExecute メソッドを起動しますが、いつでも必要なわけではありません。一部のアプリケーションでは、コマンドの作業をローカルでインプリメントするのが望ましいです。たとえば、入力データを送信するためだけにコマンドを使用することが可能です。その結果、ターゲットはコマンドからのデータを検索し、入力に基づいてローカル・

データベース手順を実行します。ユーザーのアプリケーションでコマンドを使用するのに適した方法を決定しなければなりません。

## ローカライズ可能テキスト・パッケージ

### 概説

分散アプリケーションのユーザーは、さまざまな地域に分散している可能性があります。言語が異なったり、日付と時刻の表示方法が地域に固有のものであったり、通貨が異なる可能性があります。このようなユーザーが使用することを目的としたアプリケーションは、すべてのユーザーが同じインターフェース (たとえば、英語ベースのインターフェース) を使用するように設定しなければならない場合と、ユーザーの言語上の規則に合わせて構成できるように作成できる場合があります。後者の場合には、英語を母国語とするユーザーは英語のインターフェースを使用し、フランス語を母国語とするユーザーはフランス語のインターフェースを介してアプリケーションと対話することができます。

ユーザーの言語上の規則に従った形式でユーザーに情報を表示することができるアプリケーションのことを*ローカライズ可能* であると言います。アプリケーションは、さまざまな地域のユーザーと適切な言語で対話するように構成することができます。ローカライズしたアプリケーションでは、エラー・メッセージ、出力、およびインターフェースのエレメント (メニュー・オプションなど) は、その地域で指定された言語で表示されることになります。さらに、言語としての用法にはそれほど厳しく規制されない内容、すなわち、日付および時刻の形式や通貨などのその他のエレメントも、指定地域のユーザーに適したスタイルで表示されます。他の地域では、その地域に適した言語や形式で出力が表示されることになります。

歴史的に見て、ローカライズ可能なアプリケーションの作成は、大企業が複雑なシステムを作成する場合のみに限定されていました。ローカライズ可能なコードを作成する戦略、つまり一般的に言えば*国際化対応テクニック* は、これまでは費用がかかり、インプリメントも難しかったため、大規模な開発にしか適用されなかったのです。しかし、分散コンピューティングと、WWW の使用が普及すると、アプリケーション開発者は、幅広い種類のアプリケーションをローカライズ可能にすることを迫られました。このためには、国際化対応、つまりローカライズ可能なプログラムを作成するテクニックを、アプリケーション開発者にとって大幅に使いやすいものする必要があります。 WebSphere のローカライズ可能テキスト・パッケージは、 WebSphere アプリケーション開発者が分散 WebSphere アプリケーションを簡単にローカライズできるようにすることを目的とした、 Java クラスおよびインターフェースのセットです。分

散 WebSphere アプリケーション用の言語カタログを中央に格納できるため、カタログは効率的に維持および管理することができます。

### ローカライズ可能プログラムの作成

ローカライズ不可能なアプリケーションでは、ユーザーに表示するアプリケーションの部分は、アプリケーションにそのままコード化され、変更することはできません。たとえば、エラー・メッセージを出力するルーチンは、ストリングを、おそらく英語で、ファイルやコンソールに出力するだけです。ローカライズ可能プログラムは、設計に抽象のための層を 1 つ追加します。単にエラー条件から出力ストリングに向かうのではなく、ローカライズ可能プログラムは、エラー・メッセージを言語に中立的な情報で表します。一番簡単なのが、それぞれのエラー条件がキーに対応するケースです。ユーザー向けのエラー・ストリングを出力するには、アプリケーションは、構成されたメッセージ・カタログでキーを検索します。メッセージ・カタログは、キーおよび対応するストリングからなるリストです。メッセージ・カタログごとに、提供するストリングの言語は異なります。アプリケーションは、適切なカタログでキーを検索し、必要な言語で記述されている対応するエラー・メッセージを取得して、このストリングをユーザーに出力します。

ローカライズのテクニックは、エラー・メッセージの変換以外にも使用することができます。たとえば、キーを使用してグラフィカル・ユーザー・インターフェー内の各エレメント (ボタン、ラベル、メニュー項目など) を表示し、ボタン名、ラベル、メニュー項目の変換結果が収められたメッセージ・カタログを提供することで、グラフィカル・インターフェースを複数の言語に自動的に変換することができます。また、サポートをその他の言語に拡張する場合は、その他の言語のメッセージ・カタログを提供するだけで済み、アプリケーションそのものを変更する必要はありません。

アプリケーションのローカライズは、時間帯およびロケールの 2 つの変数で制御されます。時間帯変数は、グリニッジ標準時などの標準時刻からのオフセットとしてローカル時間を計算する方法を示します。ロケールは、地理的、政治的、または文化的な地域を示す情報の集合です。ロケールは、言語や通貨だけでなく、日付などの情報を表す場合の規則に関する情報を提供し、ローカライズ可能プログラムでは、アプリケーションがメッセージを探す場所であるメッセージ・カタログも示します。 1 つの時間帯で数多くのロケールをサポートすることができます。また、単一のロケールは複数の時間帯にまたがることができます。時間帯とロケールの両方を使用して、特定の地域のユーザーに合った日付、時刻、通貨、および言語を判別することができます。

***ローカライズ可能テキストの識別:*** ローカライズ可能アプリケーションを作成するには、アプリケーション開発者は、アプリケーションのどのアスペクトを

変換可能にする必要があるかを判別しなければなりません。これらは一般に、ユーザーが読み取って理解しなければならないアプリケーションの部分です。アプリケーション開発者は、アプリケーションのインターフェースのような、すべてのユーザーが直接対話するアプリケーションの部分と、ログ・ファイル内のメッセージのような、もっと特別な目的を果たす部分を考慮しなければなりません。ローカライズに適した候補は次のとおりです。

- グラフィカル・ユーザー・インターフェー内のエレメント
  - ウィンドウのタイトル・バー
  - メニュー名、およびメニュー上の項目 (たとえば、「select File (ファイルの選択) → Open (オープン)」など)
  - ボタンのラベル (たとえば、″click the OK button″ など)
  - ユーザーにフィールドへの入力を指示する説明 (たとえば、″enter the account number″ など)
  - ユーザーに表示しなければならないその他のエレメント
- コマンド行インターフェースのプロンプト
- プログラムからの出力
  - ユーザー入力への応答
  - エラー・メッセージ
  - 例外が throw されたときに戻すテキスト
  - その他の状況メッセージ (警告、監査メッセージなど)

ローカライズするアプリケーションの各エレメントを識別したら、アプリケーション開発者は各エレメントに固有のキーを割り当て、サポートする各言語のメッセージ・カタログを提供しなければなりません。各メッセージ・カタログは、キーおよび対応する言語固有のストリングから成り立ちます。したがって、キーは、プログラムとメッセージ・カタログの間のリンクです。プログラムは、内部でキーによってローカライズ可能エレメントを参照し、メッセージ・カタログを使用して、ユーザーに表示する出力を生成します。変換後のストリングは、キーおよびリソース・バンドル (メッセージ・カタログのセット) を表す、 LocalizableTextFormatter オブジェクトの format メソッドを呼び出すことで生成します。プログラムのロケールの設定により、キーを検索する場所であるメッセージ・カタログが判別されます。

***メッセージ・カタログの作成:*** ローカライズする各エレメントを識別したら、サポートする各言語ごとにメッセージ・カタログを作成しなければなりません。 Java リソース・バンドルとしてインプリメントするこれらのカタログを作成する方法は 2 つあります。 ResourceBundle クラスのサブクラスとして作成する方法と、Java プロパティー・ファイルとして作成する方法です。リソー

ス・バンドルは、Java ではさまざまな使い道があるので、メッセージ・カタログの場合は、通常はプロパティー・ファイルとして作成します。プロパティー・ファイルを使用する場合は、アプリケーション・コードを変更することなく、言語を追加したり除去したりでき、プログラミングの経験がない人でもカタログを作成することができます。

プロパティー・ファイルにインプリメントするメッセージ・カタログは、各キーに対する行から成り立ちます。キーで、ローカライズ可能エレメントを識別します。ファイル内の各行の構造は次のとおりです。

*key = String corresponding to the key*

たとえば、銀行システムのグラフィカル・ユーザー・インターフェースに、普通預金や当座預金など、口座のタイプを選択するためのプルダウン・メニューがあるとします。プルダウン・メニューのラベルおよびメニュー上の口座タイプがローカライズの適切な対象です。キーを必要とするエレメントは 3 つあります。口座メニューのラベルと、そのメニュー上の 2 つの項目です。キーが accountString、savingsString および checkingString の場合、英語プロパティー・ファイルは、各キーを英語のストリングに関連付けます。

```
accountString = Accounts
savingsString = Savings
checkingString = Checking
...
```

*図 107. 英語メッセージ・カタログ内の 3 つのエレメント*

ドイツ語プロパティー・ファイルでは、各キーに、対応するドイツ語の値が割り当てられます。

```
accountString = Konten
savingsString = Sparkonto
checkingString = Girokonto
...
```

*図 108. ドイツ語メッセージ・カタログ内の 3 つのエレメント*

プロパティー・ファイルは、他の必要な言語に合わせて追加することもできます。

***プロパティー・ファイルのネーム解決:*** 特定のプロパティー・ファイルに分解できるように、 Java では、リソース・バンドル内のプロパティー・ファイルの命名規則 *resourceBundleName_localeID*.properties を指定しています。

それぞれのファイルには、決まった拡張子 .properties が付きます。リソース・バンドルを構成するファイルの集合には、集合の名前を付与します。単純な銀行アプリケーションの場合は、BankingResources などの分かりやすいリソース・バンドル名で十分です。各ファイルには、リソース・バンドルにロケール ID を付けた名前を付与します。ロケール ID の特定の値は、ロケールによって異なります。これらは、Java.util.ResourceBundle クラスが内部で、リソース・バンドル内のファイルを、ロケールと時間帯の設定の組み合わせに一致させる場合に使用します。アルゴリズムの詳細は、JDK のリリースによって異なります。使用するシステムに固有の情報については、Java の資料を参照してください。

銀行アプリケーションでは、BankingResources リソース・バンドルには一般的に、英語メッセージ・カタログ用の BankingResources_en.properties や、ドイツ語メッセージ・カタログ用の BankingResources_de.properties などのファイルが含まれます。また、指定されたカタログが見つからなかった場合に使用する、デフォルト・カタログ BankingResources.properties も用意されます。デフォルト・カタログは、大抵の場合は英語のカタログです。

ローカライズ可能テキストで使用するメッセージ・カタログが入っているリソース・バンドルをインストールする必要があるのは、ストリングのフォーマットを実際に実行するシステムだけです。リソース・バンドルは一般的に、アプリケーションの JAR ファイルに挿入します。詳細については、282ページの『WebSphere サポート』を参照してください。

### WebSphere および Java でのローカライズのサポート

Java パッケージ com.ibm.websphere.i18n.localizabletext には、ローカライズ可能テキストを構築するクラスおよびインターフェースが入っています。このパッケージにより、Java 言語の国際化対応機能およびローカライズ機能を幅広く使用することができます。ここでは詳しく説明しませんが、WebSphere のローカライズ可能テキスト・パッケージを使用するプログラマーは、ベースとなっている Java サポートを理解しておかなければなりません。

***Java サポート:*** WebSphere のローカライズ可能テキスト・パッケージは、主に次の Java コンポーネントを利用します。

- java.util.Locale
- java.util.TimeZone
- java.util.ResourceBundle
- java.text.MessageFormat

これだけではありません。 WebSphere およびこれらの Java クラスは、関連する Java クラスも使用することができますが、その関連するクラス (たとえば java.util.Calendar) は、一般的に特殊な目的を持つクラスです。このセクションでは、主なクラスだけを簡単に説明します。

***Locale:*** Java の Locale オブジェクトは、言語と地理的な地域をカプセル化したものです。たとえば、java.util.Locale.US オブジェクトには、米国のロケール情報が収められています。ロケールを指定するアプリケーションは、 Java 言語に組み込まれているロケール依存フォーマッターを利用することができます。これらのフォーマッターは、java.text パッケージでは、数字、通貨値、日付、および時刻の表示を処理します。

***TimeZone:*** Java の TimeZone オブジェクトは、時刻の表記をカプセル化したもので、時刻を出力したり、夏時間への移行に合わせるなどの作業のメソッドを提供します。アプリケーションは、時間帯を使用して、ローカルの日付と時刻を判別します。

***ResourceBundle:*** リソース・バンドルは、特定のロケールで使用するリソース (つまりストリング、フォント、画像など、アプリケーションが使用する情報) の名前付きの集合です。 ResourceBundle クラスを使用することで、アプリケーションは、ロケールに適した名前付きのリソース・バンドルを取得することができます。 278ページの『ローカライズ可能プログラムの作成』で説明したように、リソース・バンドルを使用して、メッセージ・カタログを保持します。リソース・バンドルは、ResourceBundle クラスのサブクラスとしてインプリメントする方法と、 Java プロパティー・ファイルとしてインプリメントする方法の 2 つの方法でインプリメントすることができます。

***MessageFormat:*** MessageFormat クラスを使用して、パラメーターに基づいてストリングを構成することができます。簡単な例として、数値キーで特定のエラー条件を表すローカライズ・アプリケーションがあるとします。アプリケーションは、エラー条件を報告するときに、メッセージ・フォーマッターを使用して、数値キーを意味のあるストリングに変換します。メッセージ・フォーマッターは、適切なリソース・バンドルでコード (パラメーター) を検索し、メッセージ・カタログから対応するストリングを取得することで、出力ストリングを構成します。出力メッセージをアセンブルする場合には、その他のパラメーター (たとえば、プログラム・モジュールを表す別のキー) を使用することもできます。

***WebSphere サポート:*** WebSphere のローカライズ可能テキスト・パッケージは、 Java サポートをラップし、分散環境で効率良く簡単に使用できるように拡張したものです。アプリケーション・プログラマーが使用する主なクラスは

LocalizableTextFormatter クラスです。このクラスのオブジェクトは、一般にサーバー・プログラムで作成されますが、クライアントが作成することもできます。 LocalizableTextFormatter オブジェクトは、特定のリソース・バンドル名およびキーに対して作成されます。 LocalizableTextFormatter オブジェクトを受け取ったクライアント・プログラムは、そのオブジェクトの format メソッドを呼び出します。このメソッドは、クライアント・アプリケーションのロケールを使用して、適切なリソース・バンドルを取得し、キーに基づいてロケール固有のメッセージをアセンブルします。

たとえば、フランス語と英語の両方のロケールをサポートする WebSphere クライアント / サーバー・アプリケーションがあるとします。サーバーは英語のロケールを使用し、クライアントはフランス語のロケールを使用します。サーバーは、英語用とフランス語用の 2 つのリソース・バンドルを作成します。クライアントが、メッセージを生成する要求を作成すると、サーバーは、リソース・バンドルの名前とメッセージのキーを含む LocalizableTextFormatter オブジェクトを作成し、そのオブジェクトをクライアントに戻します。

クライアントは、LocalizableTextFormatter オブジェクトを受け取ると、そのオブジェクトの format メソッドを呼び出し、この format メソッドがフランス語リソース・バンドルからキーに対応するメッセージを戻します。 format メソッドは、クライアントのロケールを取得し、ロケールとリソース・バンドルの名前を使用して、ロケールに対応するリソース・バンドルを判別します (クライアントで英語のロケールが設定されている場合に format メソッドを呼び出すと、英語のメッセージが取得されます)。メッセージのフォーマット化は、クライアントが意識することなく実行されます。

この単純なクライアント / サーバー例では、リソース・バンドルはサーバーと一緒に中央に常駐しています。クライアント・マシンにリソース・バンドルをインストールする必要はありません。 WebSphere のローカライズ可能テキスト・パッケージが提供するものの 1 つに、中央のカタログをサポートするインフラストラクチャーがあります。 WebSphere は、Enterprise Bean、つまり、ローカライズ可能テキスト・パッケージと一緒に提供される状態なしセッション bean を使用して、メッセージ・カタログにアクセスします。クライアントが LocalizableTextFormatter オブジェクトの format メソッドを呼び出すと、内部では次のイベントが発生します。

1. クライアント・アプリケーションは、明示的に指定するか、デフォルトを使用することで、LocalizableTextFormatter オブジェクトに時間帯の値とロケールの値を設定します。
2. 呼び出し LocalizableTextFormatterEJBFinder が作成されて、フォーマット Enterprise Bean の参照を取得します。

3. クライアントの時間帯およびロケールを含む、LocalizableTextFormatter オブジェクトの情報が、フォーマット bean に送られます。
4. フォーマット bean は、リソース・バンドルの名前、メッセージ・キー、時間帯、およびロケールを使用して、言語固有のメッセージをアセンブルします。
5. Enterprise Bean は、フォーマットしたメッセージをクライアントに戻します。
6. フォーマットしたメッセージは、format メソッドにより、LocalizableTextFormatter オブジェクトに挿入されて戻されます。

フォーマット Enterprise Bean を呼び出すには、リモートで LocalizableTextFormatter.format メソッドを 1 回呼び出し、フォーマット用の Enterprise Bean と接続するだけで十分です。 LocalizableTextFormatter オブジェクトは、フォーマットしたメッセージを必要に応じてキャッシュに入れるので、後で使用するときにはこのフォーマッターを呼び出す必要はありません。また、アプリケーションは、フォールバック・ストリングを設定することもできます。つまり、アプリケーションは、言語固有のストリングを取得するためにメッセージ・カタログにアクセスできない場合でも、読み取り可能なストリングを戻すことができます。また、リソース・バンドルをローカルに格納することもできます。ローカライズ可能テキスト・パッケージは、バンドルがローカルに格納されているか (LocalizableConfiguration.LOCAL)、リモートに格納されているか (LocalizableConfiguration.REMOTE) を示す静的変数を提供しますが、この変数の設定は、Java 仮想マシン (JVM) 内で実行されているすべてのアプリケーションに適用されます。

***LocalizableTextFormatter クラス:*** パッケージ com.ibm.websphere.i18n.localizabletext 内にある LocalizableTextFormatter クラスは、ローカライズ可能テキスト・パッケージを使用するための基本プログラミング・インターフェースです。このクラスのオブジェクトには、キーおよびリソース・バンドルから言語固有のストリングを作成するために必要な情報が入っています。

***メッセージ・カタログの場所と ApplicationName 値:*** WebSphere のローカライズ可能テキスト・パッケージで作成したアプリケーションは、メッセージ・カタログをローカルでもリモートでも格納することができます。分散環境では、リモートの中央に格納したカタログの使用が適しています。すべてのアプリケーションで同じカタログを使用できるため、カタログの管理と保持は簡単です。各コンポーネントごとに、メッセージ・カタログのコピーを保管したり保持したりする必要はありません。ローカルのフォーマットは、テストの場合に役立ち、ローカルの方が適している状況もあります。ローカル・フォーマッ

トとリモート・フォーマットの両方をサポートするためには、LocalizableTextFormatter オブジェクトはフォーマット・アプリケーションの名前を指定しなければなりません。たとえば、アプリケーションが、リモートの中央に格納したカタログを使用してメッセージをフォーマットする場合、メッセージは実際には単純な Enterprise Bean でフォーマットされます (詳細については、282ページの『WebSphere サポート』を参照してください)。ローカライズ可能テキスト・パッケージには、Enterprise Bean の検索およびその呼び出しの発行を自動化するコードが含まれてますが、アプリケーションは、フォーマット Enterprise Bean の名前を認識していなければなりません。LocalizableTextFormatter クラスのいくつかのメソッドは、 *application name* として表されている値を使用します。これが、フォーマット・アプリケーションの名前です。必ずしも値が設定されているアプリケーションの名前ではありません。

***メッセージのキャッシング:*** LocalizableTextFomatter オブジェクトは、再び必要になったときに再フォーマットする手間を省くため、フォーマットしたメッセージを必要に応じてキャッシュに入れることができます。デフォルトでは、キャッシングは使用されませんが、 LocalizableTextFormatter.setCacheSetting メソッドを使用してキャッシングを使用可能にすることができます。キャッシングを使用可能にして LocalizableTextFormatter.format メソッドを呼び出すと、メソッドは、メッセージがすでにフォーマットされているかどうかを判別します。すでにフォーマットされている場合は、キャッシュ内のメッセージが戻されます。キャッシュ内に見つからない場合は、メッセージがフォーマットされて呼び出し側に戻され、メッセージのコピーが将来に備えてキャッシュに格納されます。

メッセージをキャッシュに入れた後でキャッシングを使用不能にした場合、これらのメッセージは、キャッシュを LocalizableTextFormatter.clearCache メソッド呼び出しでクリアするまで、キャッシュ内に残ります。キャッシュは、いつでもクリアすることができます。 LocalizableTextFormatter オブジェクト内のキャッシュは、オブジェクトの次のメソッドのいずれかを呼び出したときに、自動的にクリアされます。

- setResourceBundleName(String resourceBundleName)
- setPatternKey(String patternKey)
- setArguments(Object[] args)
- setApplicationName(String appName)

***フォールバック情報:*** メッセージをフォーマットできない場合もあります。ローカライズ可能テキスト・パッケージは、フォールバック・ストラテジーをインプリメントしているので、メッセージを必要な言語用に正しくフォーマット

できなかった場合でも何らかの情報を取得することができます。LocalizableTextFomatter オブジェクトは、メッセージ・ストリング、時間帯、およびロケールのフォールバック値を必要に応じて格納することができます。これらの値は、LocalizableTextFormatter オブジェクトが例外を throw しない限り、無視することができます。

***アプリケーション固有の変数:*** ローカライズ可能テキスト・パッケージでは、時間帯とロケールに基づくローカライズは初めからサポートされていますが、アプリケーション開発者は、他の値に基づいてメッセージを構成することもできます。ローカライズ可能テキスト・パッケージには、日付と時刻を出力する例として使えるクラス LocalizableTextDateTimeArgument が用意されています。日付と時刻の情報は、ロケールと時間帯の値を使用してローカライズされますが、このクラスは、出力の表示方法の決定に他の変数も使用します。日付と時刻の情報は、非常に詳細なスタイルから簡潔なスタイルまで、さまざまなスタイルで要求することができます。この例では、メッセージ・ストリングの構成は、ロケール、時間帯、スタイルの 3 つの変数で制御しています。アプリケーションは、ロケールや時間帯のほかにさまざまな変数を使用してメッセージを構成することができます。詳細については、290ページの『オプション引き数の使用』を参照してください。

## ローカライズ可能アプリケーションの作成

ローカライズ可能テキストを使用する WebSphere アプリケーションを開発するには、次のことを行わなければなりません。

- ローカライズするアプリケーションの部分を判別する。
  - ローカライズするアプリケーションのエレメントを識別し、それぞれにキーを割り当てる。
  - ストリングに各キーを関連付けることで、各言語のメッセージ・カタログを作成する。

  これらの作業については、すでに説明しました。詳細については、278ページの『ローカライズ可能テキストの識別』および 279ページの『メッセージ・カタログの作成』を参照してください。
- キー、リソース・バンドル、およびその他の引き数から言語固有のストリングをアセンブルする。
  - LocalizableTextFormatter オブジェクトを作成する。
  - オブジェクト内に、キー、リソース・バンドルの名前、リモート・フォーマット・アプリケーションの名前、およびオプション引き数に対する値を設定する。

– LocalizableTextObject の format メソッドを呼び出して、アセンブルしたストリングを戻す。

このセクションでは、これらの作業について説明します。

### LocalizableTextFormatter オブジェクトの作成

サーバー・プログラムは一般に、LocalizableTextFormatter オブジェクトを作成し、このオブジェクトを何らかの操作の結果としてクライアントに戻します。クライアントは、適切なときにそのオブジェクトをフォーマットします。あまり一般的ではありませんが、クライアントがローカルで LocalizableTextFormatter オブジェクトを作成することもできます。LocalizableTextFormatter オブジェクトを作成するには、アプリケーションは LocalizableTextFormatter クラス内のコンストラクターの 1 つを使用します。

- LocalizableTextFormatter()
- LocalizableTextFormatter(String resourceBundleName, String patternKey, String appName)
- LocalizableTextFormatter(String resourceBundleName, String patternKey, String appName, Object[] args)

LocalizableTextFormatter オブジェクトは、オブジェクトをフォーマットできるように、リソース・バンドルの名前、キー、フォーマット・アプリケーションの名前、およびオプション値に対して設定された値を持っていなければなりません。 LocalizableTextFormatter オブジェクトの作成と値の設定は、必要な引き数を取るコンストラクターを使用して一度に行うことも、または別々に行うこともできます。値は、LocalizableTextFormatter オブジェクトのメソッドを使用して設定します。値を手作業で設定する場合は、コンストラクターを使用するのではなく、次のメソッドを使用します。

- setResourceBundleName(String resourceBundleName)
- setPatternKey(String patternKey)
- setApplicationName(String appName)
- setArguments(Object[] args)

**注:** オプション引き数の配列内の値を LocalizableTextFormatter オブジェクト内に設定した場合、それらの値は参照されるのではなく、オブジェクトにコピーされます。値を LocalizableTextFormatter オブジェクトにコピーした後で値を保持する配列変数が変更された場合、 LocalizableTextFormatter オブジェクト内の値は、リセットしない限り、変更を反映しません。

LocalizableTextFormatter オブジェクトには、オブジェクトの作成時には設定できない値を設定するためのメソッドもあります。次に例を示します。

- LocalizableTextFormatter オブジェクトのキャッシュ設定を切り替えるには、setCacheSetting(boolean setting) メソッドを使用する (詳細については、285ページの『メッセージのキャッシング』を参照)
- キャッシュをクリアするには、clearLocalizableTextFormatter メソッドを使用する
- フォールバック値を設定するには、次のメソッドを使用する
  - setFallBackString
  - setFallBackLocale
  - setFallBackTimeZone

  (詳細については、285ページの『フォールバック情報』を参照)

これらの set メソッドにはそれぞれ、値を取得するための対応する get メソッドがあります。 clearLocalizableTextFormatter メソッドは、すべての値の設定を解除し、 LocalizableTextFormatter オブジェクトをブランクの状態に戻します。オブジェクトをクリアした後は、新しい値を設定して、format メソッドを再び呼び出すことで、オブジェクトを再使用します。

図109 では、デフォルトのコンストラクターを使用して LocalizableTextFormatter オブジェクトを作成し、新しいオブジェクトのメソッドを使用して、キー、リソース・バンドルの名前、フォーマット・アプリケーションの名前、およびフォールバック・ストリングに対する値をオブジェクトに設定します。

```
import com.ibm.websphere.i18n.localizabletext.LocalizableException;
import com.ibm.websphere.i18n.localizabletext.LocalizableTextFormatter;
import java.util.Locale;
public void drawAccountNumberGUI(String accountType) {
  ...
  LocalizableTextFormatter ltf = new LocalizableTextFormatter();
  ltf.setPatternKey("accountNumber");
  ltf.setResourceBundleName("BankingSample.BankingResources");
  ltf.setApplicationName("BankingSample");
  ltf.setFallBackString("Enter account number: ");
  ...
}
```

*図 109. コード例: LocalizableTextFormatter オブジェクトの作成およびオブジェクトでの値の設定*

## ローカライズ値の設定

ローカライズ・メッセージを要求するアプリケーションは、フォーマットするメッセージに対応するロケールと時間帯を指定したり、 JVM で設定されているデフォルト値を使用したりすることができます。たとえば、グラフィカル・

ユーザー・インターフェーでは、ユーザーはメニューを表示するときの言語を選択できるようにすることができます。アプリケーションが最初に起動したときにはメニューを生成できるように、環境の中で、またはプログラムを用いてデフォルト値を設定しておかなければなりませんが、ユーザーは必要に合わせてメニューの言語を変更することができます。 図110 に、メニュー項目の選択に基づいてアプリケーションで使用するロケールを変更する方法を示します。

```
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
...
import java.util.Locale;
public void actionPerformed(ActionEvent event) {
  String action = event.getActionCommand();
  ...
   if (action.equals("en_us")) {
      applicationLocale = new Locale("en", "US");
     ...
  }
  else if (action.equals("de_de")) {
     applicationLocale = new Locale("de", "DE");
     ...
  }
  else if (action.equals("fr_fr")) {
     applicationLocale = new Locale("fr", "FR");
     ...
  }
  ...
}
```

*図 110. コード例: プログラムによるロケールの設定*

アプリケーションは format メソッドを呼び出すときに、引き数を指定せずに、メッセージをロケールと時間帯に対する JVM のデフォルト値でフォーマットしたり、ロケールと時間帯の組み合わせを指定して、JVM のデフォルトをオーバーライドしたりすることができます (format メソッドの引き数の詳細については、『ローカライズ・テキストの生成』を参照してください)。

### ローカライズ・テキストの生成

LocalizableTextFormatter オブジェクトを作成して、適切な値を設定し終えたら、ロケールと時間帯に適したストリングを生成するようにオブジェクトをフォーマットすることができます。 LocalizableTextFormatter クラスの format メソッドが、ロケールと時間帯に基づいて、メッセージ・キーとリソース・バンドルのセットからストリングを生成するために必要な作業を実行します。
LocalizableTextFormatter クラスは、4 つの format メソッドを提供します。各 format メソッドは、フォーマット設定メッセージ・ストリングを戻します。メ

ソッドは、java.util.Locale オブジェクトと java.util.TimeZone オブジェクトの組み合わせを取得し、 LocalizableException オブジェクトを throw します。

- String format();
- String format(locale);
- String format(timeZone);
- String format(locale, timeZone);

引き数を取らない format メソッドは、JVM のデフォルトとして設定されているロケールと時間帯の値を使用します。その他の format メソッドでは、これらの値のいずれか一方または両方をオーバーライドすることができます。

図111 に、288ページの図109 で作成した LocalizableTextFormatter オブジェクトに対するローカライズ・ストリングの作成を示します。フォーマット化は、289ページの図110 で設定したロケールに基づいて行われます。フォーマットできなかった場合、アプリケーションは、ローカライズ・ストリングの代わりに、フォールバック・ストリングを取得して、使用します。

```
import com.ibm.websphere.i18n.localizabletext.LocalizableException;
import com.ibm.websphere.i18n.localizabletext.LocalizableTextFormatter;
import java.util.Locale;
public void drawAccountNumberGUI(String accountType) {
  ...
  LocalizableTextFormatter ltf = new LocalizableTextFormatter();
  ltf.setPatternKey("accountNumber");
  ltf.setResourceBundleName("BankingSample.BankingResources");
  ltf.setApplicationName("BankingSample");
  ltf.setFallBackString("Enter account number: ");
  try {
     msg = new Label (ltf.format(this.applicationLocale) , Label.CENTER);
  }
  catch (LocalizableException le) {
     msg = new Label(ltf.getFallBackString(), Label.CENTER);
  }
  ...
}
```

*図 111. コード例: LocalizableTextFormatter オブジェクトのフォーマット*

## オプション引き数の使用

ローカライズ可能テキスト・パッケージでは、ユーザーは LocalizableTextFormatter オブジェクトでのオプション引き数の配列を指定することができます。これらのオプション引き数を使用すると、WebSphere アプリ

ケーションで行うローカライズの種類を大幅に拡張することができます。このセクションでは、オプション引き数の 2 つの使い方について説明します。

- 変数サブストリングを持つ複合ストリングをアセンブルし、フォーマットする場合
- ストリングのフォーマットをカスタマイズして、ロケールと時間帯以外の変数を考慮する場合

**複合ストリングのアセンブル**

これまで紹介してきたキーはすべて、定まったストリングを表していました。つまり、ローカライズ時には、適切な言語のストリングがキーに合わせて置換されます。ローカライズ可能テキスト・パッケージは、変数をプレースホルダーとして組み込むことができる、ストリング置換もサポートしています。たとえば、指定された口座での操作が正常終了したことを報告する必要があるアプリケーションは、″The operation on account *number* was successful″ というようなストリングを提供しなければなりません。変数 *number* は、実際の口座番号に置き換えます。変数を持つストリングの作成がサポートされていないと、各ストリングごとに独自のキーがなければならず、またストリングはフレーズ単位で作成しなければならないことになります。

変数がさまざまな値を取ることができたり、ストリングに複数の変数コンポーネントが入っていたりする場合には、これらのアプローチはどちらも簡単には実現できなくなります。そこで、ローカライズ可能テキスト・パッケージは、オプション引き数を使ったストリング内の変数の置換をサポートしています。メッセージ・カタログ内のストリングは、中括弧で囲んだ整数 (たとえば、{0}、{1} など) を使用して、変数コンポーネントを表します。 図112 に、単一の変数置換を持つストリングの英語メッセージ・カタログの例を示します (他の言語のメッセージ・カタログ内の同じキーでも、このストリングの変換は実行されますが、変数の場所は言語ごとに異なります)。

```
successfulTransaction = The operation on account {0} was successful.
```

*図 112. 変数サブストリングを持つメッセージ・カタログ項目*

ストリングに置換される値は、オプション引き数の配列の値です。LocalizableTextFormatter オブジェクトのコンストラクターの 1 つは、オブジェクトの配列を引き数として取るので、このようなオブジェクトの配列を LocalizableTextFormatter オブジェクト内で設定することができます。配列を使用して、ストリングの変数部に対する値を保持します。オブジェクトで format メソッドが呼び出されると、配列は format メソッドに渡され、メソッドは配列のエレメントを取得して、そのエレメントをストリング内では一致するイン

デックスを持つプレースホルダーに置換します。配列内のインデックス 0 の値は、ストリングでは {0} 変数に置換され、インデックス 1 の値は {1} に置換されます。

図113 に、単一エレメント引き数配列の作成、および LocalizableTextFormatter の作成と使用を示します。引き数配列内のエレメントは、ユーザーが入力する口座番号です。 LocalizableTextFormatter は、オプション引き数の配列を取るコンストラクターを使用して作成します。この配列は、LocalizableTextFormatter オブジェクトの setArguments メソッドを使用することで、直接設定することもできます。コードの後半では、アプリケーションは format メソッドを呼び出します。 format メソッドは、引き数の配列の値を、適切なメッセージ・カタログから戻されたストリングに自動的に置換します。

```
public void updateAccount(String transactionType) {
  ...
  Object[] arg = { new String(this.accountNumber)};
  ...
   LocalizableTextFormatter successLTF =
     new LocalizableTextFormatter("BankingResources",
                                  "successfulTransaction",
                                  "BankingSample",
                                 arg);
  ...
  successLTF.format(this.applicationLocale);
  ...
}
```

*図 113. コード例: 変数サブストリングを持つメッセージのフォーマット*

***LocalizableTextFormatter オブジェクトのネスト:*** 変数をメッセージ・カタログ内のストリングに置換できることから、ローカライズ可能テキスト・パッケージの柔軟性は向上しますが、少なくとも国際的な環境では、置換引き数そのものをローカライズできない限り、この柔軟性は制限されます。たとえば、特定の口座での操作が正常終了したことを報告する必要があるアプリケーションの場合、 ″The operation on account *number* was successful″ のようなストリング (ここで、唯一の変数は口座の「番号」です) は、変換して、複数の言語のメッセージ・カタログで使用することができます。中に入っている変数もまたストリングであるようなストリング、たとえば、″The *type* operation on account *number* was successful″ (ここで、新しい *type* 変数は、 ″deposit″ や ″withdrawal″ などの値を取ります) は、簡単には変換できません。 *type* 変数に有効な値も、ローカライズする必要があります。

293ページの図114 に、2 つの引き数 (そのうちの 1 つがローカライズする変数) を含む英語カタログ内のメッセージ・ストリングと、 2 つの有効な値に対

するキーを示します (ストリング内の 2 つ目の引き数、account number は、ストリングに置換しなければならない単なる数字なので、ローカライズする必要はありません)。

```
sucessfulTransaction = The {0} operation on account {1} was successful.
depositOpString = deposit
withdrawOpString = withdrawal
```

*図 114. 2 つの変数サブストリングを持つメッセージ・カタログ項目*

サブストリングのローカライズをサポートするため、ローカライズ可能テキスト・パッケージでは、LocalizableTextFormatter オブジェクトのネストを行うことができます。これは、単に LocalizableTextFormatter オブジェクトを別の LocalizableTextFormatter オブジェクトの引き数の配列に挿入するだけです。format メソッドは、変数の置換を実行する際に、変数に対して配列のエレメントを置換するようにすべての LocalizableTextFormatter オブジェクトをフォーマットします。これにより、サブストリングは、埋め込み先のストリングとは別個にフォーマットすることができます。

294ページの図115 は、292ページの図113 の例を修正したもので、ローカライズ可能サブストリングを持つメッセージをフォーマットします。まず、ローカライズ可能サブストリング (預金操作を指している) の LocalizableTextFormatter オブジェクトを作成します。このオブジェクトを、口座番号情報と共に、引き数の配列に挿入します。次に、この引き数の配列を、完全なストリングに対する LocalizableTextFormatter オブジェクトを構成する場合に使用します。format メソッドを呼び出すと、埋め込まれている LocalizableTextFormatter オブジェクトがフォーマットされて、 1 つ目の変数が置き換えられ、2 つ目の変数では口座番号への置換が実行されます。

```
public void updateAccount(String transactionType) {
   ...
   // Successful Deposit.
   LocalizableTextFormatter opLTF =
      new LocalizableTextFormatter("BankingResources,
                                   "depositOpString", "BankingSample");
   Object[] args = {opLTF, new String(this.accountNumber)};
   LocalizableTextFormatter successLTF =
      new LocalizableTextFormatter("BankingResources",
                                   "successfulTransaction",
                                   "BankingSample",
                                   args);
   ...
   successLTF.format(this.applicationLocale);
   ...
}
```

*図 115. コード例: ローカライズ可能変数サブストリングを持つメッセージのフォーマット*

### format メソッドの振る舞いのカスタマイズ

オプション引き数の配列には、フォーマット・ストリングに置換される口座番号のような簡単な値と、より大きなフォーマット・ストリングに置換されるローカライズ可能サブストリングを表す、他の LocalizableTextFormatter オブジェクトを含めることができます。これらのテクニックについては、291ページの『複合ストリングのアセンブル』を参照してください。また、オプション引き数配列には、ユーザー定義クラスのオブジェクトを含めることもできます。

オプション引き数として使用するユーザー定義クラスは、アプリケーション固有の format メソッドを提供するので、プログラマーはこのメソッドを使用して、ロケールや時間帯だけではなく、あらゆる値に基づいてローカライズを実行することができます。これらのユーザー定義クラスをインストールする必要があるのは、そのクラスを構成して LocalizableTextFormatter オブジェクトに挿入し、実際のフォーマットを実行するシステムだけです。クライアント・アプリケーションにこれらのクラスをインストールする必要はありません。

ローカライズ可能テキスト・パッケージは、 LocalizableTextDateTimeArgument クラスにこのようなユーザー定義クラスのサンプルを用意しています。このクラスを使用することで、日付および時刻の情報を、 java.text.DateFormat クラスで定義されているスタイル値に従って、また LocalizableTextDateTimeArgument クラスで定義されている定数に従って、選択してフォーマットすることができます。

DateFormat スタイルは、日付の情報をどのように出力するかを指定します。たとえば、DateFormat.FULL スタイルが選択されている場合は、 2000 年 2 月 22 日は、英語で *Tuesday, February 22, 2000* と表示されます。 DateFormat.SHORT スタイルが使用されている場合は、同じ日付は *2/22/00* と表示されます。有効な値は次のとおりです。

- DateFormat.FULL
- DateFormat.LONG
- DateFormat.MEDIUM
- DateFormat.SHORT
- DateFormat.DEFAULT

LocalizableTextDateTimeArgument クラスは、日付または時刻の一方を要求する場合、または日付および時刻の両方を日付 - 時刻の順序または時刻 - 日付の順序で要求する場合に使用できる定数を定義します。定義されている値は次のとおりです。

- LocalizableTextDateTimeArgument.TIME
- LocalizableTextDateTimeArgument.DATE
- LocalizableTextDateTimeArgument.TIMEANDDATE
- LocalizableTextDateTimeArgument.DATEANDTIME

LocalizableTextDateTimeArgument クラスのようなユーザー定義クラスのオブジェクトは、 LocalizableTextFormatter オブジェクトのオプション引き数の配列の中に設定することができ、 LocalizableTextFormatter オブジェクトは、ユーザー定義オブジェクトをフォーマットしようとするときに、そのユーザー定義オブジェクトの format メソッドを呼び出します。この format メソッドは、アプリケーション開発者が作成するので、アプリケーション固有の値を使って、適切などのようなことでも実行できます。 LocalizableTextDateTimeArgument クラスの場合、 format メソッドは、日付、時刻、またはその両方が必要かどうかを判別し、 DateFormat 値に従ってそれらをフォーマットし、 LocalizableTextDateTimeArgument スタイルで指定された順序でアセンブルします。日付および時刻の情報も、ロケール値と時間帯値の影響を受けますが、 DateFormat クラスとユーザー定義の値によってフォーマット内で調整が実行されます。

LocalizableTextDateTimeArgument クラスのようなユーザー定義クラスからアセンブルしたストリングは、ネストされた LocalizableTextFormatter オブジェクトの戻り値の場合と同じように、より大きいストリングに置換することができます。このようなユーザー定義クラスを作成する場合は、それらのクラスを汎用

の LocalizableTextFormatter クラスの特殊バージョンと考えると良いでしょう。 LocalizableTextFormatter クラスの作成方法が、ユーザー定義クラスを作成する場合のモデルとなります。

***LocalizableTextFormatter クラスの構造:*** LocalizableTextFormatter クラスは、ローカライズ可能テキストの汎用クラスです。 java.lang.Object クラスを拡張したもので、 java.io.Serializable インターフェースと 4 つのローカライズ可能テキスト・インターフェースをインプリメントしています。

- LocalizableTextLTZ
- LocalizableTextL
- LocalizableTextTZ
- LocalizableText

LocalizableTextFormatter クラスがインプリメントするローカライズ可能テキスト・インターフェースはそれぞれ、 Localizable インターフェース (単に Serializable インターフェースを拡張したもの) をインプリメントし、単一の format メソッドを定義します。

- LocalizableTextLTZ インターフェースは format(locale, timezone) を定義する
- LocalizableTextL は format(locale) を定義する
- LocalizableTextTZ は format(timezone) を定義する
- LocalizableText は format() を定義する

LocalizableTextFormatter クラスは、これらの 4 つのインターフェースをすべてインプリメントするので、対応する各 format メソッドもインプリメントしなければなりません。

***ユーザー定義クラスの作成:*** ユーザー定義クラスは、Serializable インターフェースの他に、少なくとも 1 つのローカライズ可能テキスト・インターフェースと、それに対応する format メソッドをインプリメントしなければなりません。クラスで複数のローカライズ可能テキスト・インターフェースと format メソッドをインプリメントする場合、インターフェースの評価の順序は次のようになります。

1. LocalizableTextLTZ
2. LocalizableTextL
3. LocalizableTextTZ
4. LocalizableText

たとえば、LocalizableTextDateTimeArgument クラスは、 図116 に示すように、LocalizableTextLTZ インターフェースしかインプリメントしません。

```
package com.ibm.websphere.i18n.localizabletext;
import java.util.Locale;
import java.util.Date;
import java.text.DateFormat;
import java.util.TimeZone;
import java.io.Serializable;
public class LocalizableTextDateTimeArgument implements LocalizableTextLTZ,
                                                         Serializable
{
   ...
}
```

*図 116. コード例: LocalizableTextDateTimeArgument クラスの構造*

ユーザー定義クラスにはコンストラクターが入っており、クラスがインプリメントするローカライズ可能テキスト・インターフェースで定義されている format メソッドがインプリメントされていなければなりません。また、必要に応じてその他のメソッドを入れることもできます。LocalizableTextDateTimeArgument クラスには、コンストラクター、単一の format メソッド、equality メソッド、ハッシュ・コード・ジェネレーター、および string-conversion メソッドが入っています。

```
...
public class LocalizableTextDateTimeArgument implements LocalizableTextLTZ,
                                                        Serializable
{
   public final static int DATE = 1;
   public final static int TIME = 2;
   public final static int DATEANDTIME = 3;
   public final static int TIMEANDDATE = 4;
   private Date date = null;
   private dateTimeStyle = LocalizableTextDateTimeArgument.DATE;
   private int dateFormatStyle = DateFormat.FULL;
   ...
   public LocalizableTextDateTimeArgument(Date date, int dateTimeStyle,
                                          int dateFormatStyle)
   { ... }
   public boolean equals(Object param)
   { ... }
public format (Locale locale, TimeZone timeZone)
   throws IllegalArgumentException
   { ... }
   public int hashCode()
   { ... }
   public String toString()
   { ... }
}
```

*図 117. コード例: LocalizableTextDateTimeArgument クラス内のメソッド*

ユーザー定義クラス内の各 format メソッドは、アプリケーションに適したどのようなことでも実行できます。 LocalizableTextDateTimeArgument クラスでは、 format メソッド (インプリメンテーションについては、299ページの図118を参照) は、オブジェクト内で設定されている日付 - 時刻スタイルの設定値 (たとえば、DATEANDTIME) を調べます。その後、date-foramt 値に従って、指定された情報を指定された順序でアセンブルします。

```
public format (Locale locale, TimeZone timeZone)
   throws IllegalArgumentException
{
   String returnString = null;

   switch(dateTimeStyle) {
      case LocalizableTextDateTimeArgument.DATE :
      {
         returnString = DateFormat.getDateInstance(dateFormatStyle,
                                                   locale).format(date);
         break;
      }
      case LocalizableTextDateTimeArgument.TIME :
      {
         df = DateFormat.getTimeInstance(dateFormatStyle, locale);
         df.setTimeZone(timeZone);
         returnString = df.format(date);
         break;
      }
      case LocalizableTextDateTimeArgument.DATEANDTIME :
      {
         dateString = DateFormat.getDateInstance(dateFormatStyle,
                                                   locale).format(date);
         df = DateFormat.getTimeInstance(dateFormatStyle, locale);
         df.setTimeZone(timeZone);
         timeString = df.format(date);
         returnString = dateString + " " + timeString;
         break;
      }
      case LocalizableTextDateTimeArgument.TIMEANDDATE :
      {
         dateString = DateFormat.getDateInstance(dateFormatStyle,
                                                   locale).format(date);
         df = DateFormat.getTimeInstance(dateFormatStyle, locale);
         df.setTimeZone(timeZone);
         returnString = timeString + " " + dateString;
         break;
      }
      default :
      {
         throw new IllegalArgumentException();
      }
   }
   return returnString;
}
```

*図 118. コード例: LocalizableTextDateTimeArgument クラス内の format メソッド*

アプリケーションは、LocalizableTextDateTimeArgument オブジェクト (または、その他のユーザー定義クラスのオブジェクト) を作成し、それを LocalizableTextFormatter オブジェクトのオプション引き数の配列に挿入することができます。 LocalizableTextFormatter オブジェクトは、ユーザー定義オブジ

ェクトを見つけると、そのオブジェクトの format メソッドを呼び出して、フォーマットを試みます。そして、LocalizableTextFormatter はオプション引き数の配列内の各エレメントを処理するので、変数の場合は戻されたストリングへの置換が実行されます。

## フォーマット Enterprise Bean の配置

ローカライズ可能テキスト・パッケージには、状態なしセッション Enterprise Bean、つまり分散環境でメッセージをフォーマットするための LocalizableTextResourceAccessorBean が用意されています。LocalizableTextFormatter オブジェクトの format メソッドは、セッション bean を透過的に検索し、この bean に問い合わせます。ただし、セッション bean を使用するには、WebSphere Application Server に配置しておかなければなりません。アプリケーションが LocalizableTextFormatter オブジェクトの format メソッドを呼び出すと、 format メソッドは、フォーマット・アプリケーションの名前を使用して、フォーマット bean が配置されているサーバーを探します。このローカライズ可能テキスト bean は、 LocalizableTextFormatter オブジェクト内の情報からストリングをアセンブルして、そのアセンブルしたストリングを戻します。

ローカライズ可能テキスト・パッケージには、コマンド行ツール、つまりローカライズ可能テキスト・セッション bean を配置するための LocalizableTextEJBDeploy ツールが用意されており、セッション bean の実行に必要なすべてのコードが用意されています。管理者はこのツールを使用して、フォーマット bean を配置し、命名することができます。この bean に付与する名前は、LocalizableTextFormatter オブジェクトでフォーマット・アプリケーションの名前として指定した名前と一致しなければなりません。また、このツールでは、もはや必要なくなった配置済みの bean を除去することもできます。

### ツールのセットアップ

LocalizableTextEJBDeploy ツールを使用してローカライズ可能アプリケーションのフォーマット・セッション bean を配置する場合は、次の条件を満たしていなければなりません。

- WebSphere インストール・ディレクトリーの下に temp というディレクトリーがなければならない。このディレクトリーは一般に、WebSphere Application Server のインストール時に作成されます。このディレクトリーがない場合は、作成してください。
- CLASSPATH 変数上にファイル ujc.jar がなければならない。このファイルには、配置ツールのコンパイル済み Java コードが入っています。

### フォーマット・セッション bean の配置

ツールの前提条件を満たしていれば、ツールを使用してフォーマット・セッション bean を配置することができます。ツールは、4 つの必須引き数と、2 つのオプション引き数を取ります。

```
LocalizableTextEJBDeploy -a <appName> -h <hostName>
                         -i <installationDir> -x <action>
                         [ -s <serverName> ] [ -c <containerName> ]
```

必須引き数は次のとおりです。指定する順序は問いません。

- appName: フォーマット・セッション bean の名前。この名前は、LocalizableTextFormatter オブジェクトで、実際にフォーマットを実行する場所を指定する場合に使用します。 LocalizableTextFormatter オブジェクトが、解決できない名前を指定した場合、format メソッドは例外を throw します。
- hostName: フォーマット・セッション bean を配置するマシンの名前。ここで指定する値は、すべてのプラットフォームで大文字小文字の区別があります。
- installationDir: WebSphere Application Server がインストールされているマシン上の場所。
- action: ツールを使用して実行するタスク。ツールでは、フォーマット・セッション bean の配置情報を作成したり、bean がもはや必要なくなった場合の配置情報を除去したりすることができます。この引き数の有効な値は 2 つあります。
  - create: ツールは、フォーマット・セッション bean について次の JAR ファイルと XML ファイルを作成し、配置が完了したらそれらのファイルを削除します。
    - <installRoot>/temp/LocalizableText-Jetace-<appName>.xml
    - <installRoot>/temp/LocalizableText-XMLConfig-<appName>.xml
    - <installRoot>/deployableEJBs/LocalizableText-<appName>.jar
    - <installRoot>/deployedEJBs/DeployedLocalizableText-<appName>.jar
  - delete: ツールは、フォーマット・セッション bean について次の XML ファイルを作成します。
    - <installRoot>/temp/LocalizableText-XMLConfig-<appName>.xml

オプション引き数は次のとおりです。この引き数も、指定する順序は問いません。

- serverName: WebSphere Application Server の名前。この引き数を指定しなかった場合は、″Default Server″ という値が使用されます。

- containerName: WebSphere Application Server 内のコンテナーの名前。この引き数を指定しなかった場合は、″Default Container″ という値が使用されます。

フォーマット bean は、各システムに必要なリソース・バンドルのコピーがある限り、複数のシステムに配置することができます。 図119 に、CheckingApplication という名前のフォーマット bean を ResourcesHost1 という UNIX マシンと ResourcesHost2 という PC の 2 台のマシンに配置する場合のコマンドを示します。

```
% java LocalizableTextEJBDeploy -a CheckingApplication -x create
-h ResourcesHost1 -i /usr/WebSphere/AppServer
C:¥java LocalizableTextEJBDeploy -a CheckingApplication -x create
-h ResourcesHost2-i C:¥WebSphere¥AppServer
```

*図 119. フォーマット Enterprise Bean の配置*

フォーマット bean は、必要なくなったら、LocalizableTextEJBDeploy ツールを使って削除できます。 図120 に、図119 で配置したフォーマット bean を一方のマシンから削除する場合のコマンドを示します。

```
C:¥java LocalizableTextEJBDeploy -a CheckingApplication -x delete
-h ResourcesHost2-i C:¥WebSphere¥AppServer
```

*図 120. 配置したフォーマット Enterprise Bean の削除*

# 付録A. EJB 仕様バージョン 1.1 の変更内容

WebSphere Application Server は、EJB 仕様のバージョン 1.1 をサポートしています。この付録では、EJB 仕様のバージョン 1.1 における新しいフィーチャーまたは変更されたフィーチャーを紹介し、 EJB 仕様のバージョン 1.0 に合わせて作成された Enterprise Beans の移行問題について説明します。

## 新しいフィーチャーおよび更新されたフィーチャー

バージョン 1.1 における Enterprise Bean の新しいフィーチャーまたは変更されたフィーチャーは次のとおりです。

- Enterprise Bean の環境依存性は、JNDI ネーミング・コンテキスト内の項目を使用して指定するようになりました。Enterprise Bean のインスタンスは、引き数を指定せずにコンストラクターを呼び出すことによって、javax.naming.InitialContext オブジェクトを作成します。名前 java:comp/env で InitialContext オブジェクトを使用することで、環境ネーミング・コンテキストを検索します。
- 1 次キーの処理方法は、EJB 仕様のバージョン 1.1 で変更されました。エンティティー bean のプロバイダーは、コンテナー管理のパーシスタンス (CMP) を持つエンティティー bean の 1 次キー・クラスを指定する必要がなくなり、 bean をコンテナーに配置するときに配置機能を用いて 1 次キー・フィールドを選択できるようになりました。
- アプリケーション・アセンブリーに対するデプロイメント・ディスクリプターのサポートが拡張されました。

## バージョン 1.0 からバージョン 1.1 への移行

クライアント側から見ると、EJB 仕様のバージョン 1.1 に合わせて作成された Enterprise Bean は、バージョン 1.0 に合わせて作成された Enterprise Bean とほとんど同じです。ただし、次の EJB 1.1 の変更点がクライアントに影響します。

- EJB 仕様のバージョン 1.1 に合わせて作成された Enterprise Bean は、JNDI ネーム・スペースの別の部分に登録されます。たとえば、クライアントが JNDI でバージョン 1.0 の Enterprise Bean の初期コンテキストを検索する場合は、次のように **initialContext.lookup** メソッドを使用します。

  ```
  initialContext.lookup("com/ibm/Hello")
  ```

© Copyright IBM Corp. 1999, 2001

バージョン 1.1 における同じ JNDI 検索は、次のようになります。

```
initialContext.lookup("java:comp/env/ejb/Hello")
```

- EJB 仕様のバージョン 1.1 に合わせて作成された Enterprise Bean では、UserTransaction オブジェクトの取得方法が変更されました。バージョン 1.0 では、次のように取得していました。

  ```
  initialContext.lookup("jta/UserTransaction")
  ```

  バージョン 1.1 では、次のように取得します。

  ```
  initialContext.lookup("java:comp/UserTransaction")
  ```

- EJB 仕様のバージョン 1.1 に合わせて作成されたエンティティー bean は、プリミティブな 1 次キーをサポートする (1 次キー・クラスにカプセル化する必要がない) ので、クライアントは、これらのプリミティブなキーを直接検索する必要があります。たとえば、クライアントは、次のように、型 java.lang.Integer のプリミティブなキーを検索することができます。

  ```
  accountHome.findByPrimaryKey(new Integer(5))
  ```

  1 次キー・クラスもサポートされていますが、プリミティブなデータ型は使用しないでください。

アプリケーション開発者側から見ると、 EJB 仕様のバージョン 1.0 に合わせて作成された Enterprise Bean をバージョン 1.1 に対応させるためには、次のような変更を加える必要があります。

- すべてのデプロイメント・ディスクリプターを、EJB 仕様のバージョン 1.1 で指定されている XML フォーマットに変換しなければなりません。
- 一般的に、EJB 仕様のバージョン 1.0 に合わせて作成された Enterprise Bean は、バージョン 1.1 と互換性があります。ただし、次の場合には、Enterprise Bean のコードを変更または再コンパイルする必要があります。
  - CMP を持つすべてのエンティティー bean で、 ejbCreate メソッドの戻り値を変更しなければなりません。 ejbCreate メソッドが戻す型は、1 次キーと同じでなければなりません。実際の戻り値はヌルでなければなりません。これらの bean は再コンパイルも必要です。詳細については、124ページの『ejbCreate メソッドおよび ejbPostCreate メソッドのインプリメント』を参照してください。
  - javax.jts.UserTransaction インターフェースを使用している場合。このインターフェースの名前は javax.transaction.UserTransaction になりました。このインターフェースを使用している Enterprise Bean は、新しいインターフェース名を使用するように変更しなければなりません。また、このインターフェースが throw する例外にも小さい変更があります。

- javax.ejb.EJBContext インターフェースの getCallerIdentity メソッドまたは isCallerInRole メソッドを使用している場合。 javax.security.Identity は Java 2 プラットフォームでは使用できなくなったので、これらのメソッドも使用しないでください。
- エンティティー bean が、EJB 仕様のバージョン 1.1 では認められていない UserTransaction インターフェースを使用している場合。
- エンティティー bean の finder メソッドが、そのメソッドの throws クラスで FinderException を定義していない場合。バージョン 1.1 では、エンティティー bean の finder メソッドはこの例外を定義していなければなりません。
- エンティティー bean が UserTransaction インターフェースを使用し、SessionSynchronization インターフェースをインプリメントしている場合。エンティティー bean は、バージョン 1.1 では、UserTransaction インターフェースを使用することも、 SessionSynchronization インターフェースをインプリメントすることもできません。
- 状態付きセッション bean が SessionSynchronization インターフェースをインプリメントしている場合。これは、バージョン 1.1 では認められていません。
- Enterprise Bean が、EJB 仕様のバージョン 1.1 で定義されている新しいセマンティック制限に違反している場合。
- バージョン 1.1 では、javax.ejb.RemoteException 例外を bean のインプリメンテーションから throw してはなりません。この例外は、javax.ejb.EJBException、または javax.ejb.CreateException などのより特定性の強い例外に置き換えてください。 javax.ejb.EJBException は、javax.ejb.RuntimeException から継承するので、 throws 文節で明示的に宣言する必要はありません。

  RMI で要求されるように、リモート・インターフェースおよびホーム・インターフェースで javax.ejb.RemoteException 例外を宣言します。この例外を bean のインプリメンテーションから直接 throw してはなりません。ただし、システム例外のために、または bean のインプリメンテーションから throw される例外をマップすることによって、コンテナーから throw することは可能です。

# 付録B. WebSphere Application Server に提供されているコード例

この付録では、WebSphere Application Server の アドバンスド版とエンタープライズ版の両方で提供されているコード例について説明しています。

## 本書に記載する例の説明

本書で説明しているコードの例は、この製品に提供されている例のセットからのものです。この例のセットは、主に以下のコンポーネントから構成されています。

- Account エンティティー bean。当座預金または普通預金の銀行口座を作成し、各口座の残高を保守します。口座 ID は、bean クラスの各インスタンスを一意に識別するために使用し、1 次キーとして機能します。この bean 内の永続データはコンテナーによって管理されており、以下の変数から構成されています。
  - *accountId* ー 口座を一意に識別する口座 ID。この変数の型は long です。
  - *type* ー 口座が普通預金口座 (1) か当座預金口座 (2) かを識別する整数。この変数の型は int です。
  - *balance* ー 口座の現時点での残高。この変数の型は float です。

  この bean の主なコンポーネントについては、118ページの『CMP を持つエンティティー bean の開発』で説明しています。
- AccountBM エンティティー bean。Account エンティティー bean とほぼ同じですが、AccountBM bean は bean 管理のパーシスタンスをインプリメントします。本書の例のセットに含まれるその他の Enterprise Bean、アプリケーション、またはサーブレットは、この bean を使用しません。この bean の主なコンポーネントについては、201ページの『BMP を持つエンティティー bean の開発』で説明しています。
- Transfer セッション bean。2 つの Account bean のインスタンス間で指定額を送金するための送金セッションを形成します。この bean には、2 つの口座間で送金する transferFunds メソッドと、指定口座について残高を照会する getBalance メソッドの 2 つがあります。この bean は状態なし bean です。この bean の主なコンポーネントについては、138ページの『セッション bean の開発』で説明しています。

© Copyright IBM Corp. 1999, 2001

- CreateAccount サーブレット。これを使用して、新しい銀行口座 (および対応する Account bean インスタンス) を、口座 ID、口座のタイプ、および初期残高を指定して簡単に作成することができます。このサーブレットは、ユーザーが口座を作成して例のセットにある他のコンポーネントを簡単に実行できるように設計されていますが、サーブレットとエンティティー bean との対話についても例証します。このサーブレットについては、189ページの『第8章 Enterprise Bean を使用するサーブレットの開発』で説明しています。
- TransferApplication Java アプリケーション。AWT (Abstract Windowing Toolkit) で構築されたグラフィカル・ユーザー・インターフェースを提供します。このアプリケーションは Transfer セッション bean のインスタンスを作成します。このインスタンスを操作して、選択した 2 つの口座間で送金するか、指定の口座について残高を取得します。 TransferApplication コードは、EJB クライアントで Enterprise Bean を使用するために必要となるものを多数インプリメントします。このアプリケーションの部分は、 Enterprise Bean との対話に関連するものです。これについては、169ページの『第7章 EJB クライアントの開発』で説明しています。
- TransferFunds サーブレット。サーブレット版の TransferApplication Java アプリケーションです。このサーブレットによって、基本的に同じ作業を実行する Java アプリケーションと Java サーブレットにおける Enterprise Bean の使用法を比較することができます。本書では、このサーブレットについて詳しく説明していません。

**注:** 本書のコード例は、できるだけ単純に作成されています。これらの例の目的は、コードを提供することによって、Enterprise Bean と EJB クライアント開発の基本概念を理解することです。銀行 (または同等の企業) が使用する銀行取引アプリケーションの作成方法の例を提供するものではありません。たとえば、Account bean には float 型の変数 *balance* が含まれています。実際の銀行取引アプリケーションでは、金額の記録に float 型を使用してはなりません。しかし、java.math.BigDecimal などのクラスや通貨処理クラスを例で使用すると、例が不必要に複雑になってしまいます。この点に注意して、これらの例を使用してください。

## EJB サーバー (AE) 環境における他の例の説明

表5 は、EJB サーバー (AE) に提供されている、Enterprise Bean 固有の例を要約したものです。

*表 5. EJB サーバー (AE) で使用できる例*

| 名前 | bean のタイプ | EJB クライアントのタイプ | 追加情報 |
|---|---|---|---|
| Hello | 状態なしセッション | Java サーブレット | 非常に単純なセッション bean の例。 |
| Increment | CMP エンティティー | Java サーブレット | 非常に単純なエンティティー bean の例。 |

## EJB サーバー (CB) 環境における他の例の説明

表6 は、EJB サーバー (CB) に提供されている、Enterprise Bean 固有の例を要約したものです。これらの例に関する詳細については、各例に付随する README ファイルを参照してください。

*表 6. EJB サーバー (CB) で使用できる例*

| 名前 | bean のタイプ | EJB クライアントのタイプ | 追加情報 |
|---|---|---|---|
| Hello | 状態なしセッション | Java アプリケーション | 非常に単純なセッション bean の例。 |
| Calculator | 状態付きセッション | アプレット、ActiveX コントロール | セッション bean における状態情報の保守を例示する。 |
| Account | 状態付きセッション、CMP エンティティー、BMP エンティティー | サーブレット、Active X コントロール | サーブレット・クライアント付きのアドバンスド版のサンプル。 1 つの Enterprise Bean が別の bean を参照する。 |
| Card Game | 状態付きセッション、CMP エンティティー | アプレット、ActiveX コントロール | さまざまなタイプの照会を使用するカスタムの finder メソッドを使ってエンティティー bean を選択するセッション bean を例示する。 1 つの Enterprise Bean が別の bean を参照する。 |

*表 6. EJB サーバー (CB) で使用できる例 (続き)*

| 名前 | bean のタイプ | EJB クライアントのタイプ | 追加情報 |
|---|---|---|---|
| Travel | 状態付きセッション、BMP エンティティー、CMP エンティティー | アプレット、ActiveX コントロール | クライアント側のトランザクションを実行する。 Enterprise Bean はデータ・ソースとして PAA を使用する。 1 つの Enterprise Bean が別の bean を参照する。 |
| VisualAge for Java デモ | CMP エンティティー | | クライアント開始トランザクション、継承、関連付け、およびポリモアフィック照会を例示する。 1 つの Enterprise Bean が別の bean を参照する。 |
| Big 3 | 状態なしセッション、CMP エンティティー | マルチスレッド | EJB 仕様のバージョン 1.1 に合わせて作成された Enterprise Bean を例示する。 1 つの Enterprise Bean が別の bean を参照する。 |
| Postcard | 状態なしセッション | | Java Messaging Service (JMS) のポイント・ツー・ポイント・メッセージ交換を使用する Enterprise Bean を例示する。 |
| CORBA インターオペラビリティー (ポリシー・ラッパー) | BMP エンティティー | | C++ ビジネス・オブジェクト (BO) と通信する Enterprise Bean および Enterprise Bean と通信する Java BO (C++ クライアント付き) を例示する。 |
| JDBC AA | BMP エンティティー | | CB セッション・サービスの使い方を例示する。 Enterprise Bean はデータ・ソースとして PAA を使用する。 |

# 付録C. Enterprise Bean での XML の使用 (CB のみ)

**注:** この付録は、EJB サーバー (CB) 環境にのみ適用されます。さらにこの付録は、EJB 仕様のバージョン 1.0 に合わせて作成された Enterprise Bean の XML デプロイメント・ディスクリプターの作成にのみ適用されます (バージョン 1.1 の Enterprise Bean では、標準 XML デプロイメント・ディスクリプターを使用します)。

この付録では、XML (Extensible Markup Language) を使用して、Enterprise Bean のデプロイメント・ディスクリプターを作成する手順について説明します。

**注:** 次の方法に従う代わりに、VisualAge for Java を使用して XML デプロイメント・ディスクリプターを作成する方法もあります。詳細については、VisualAge for Java 製品の資料を参照してください。

この付録には、XML の作成や使用に関する一般情報は記載されていません。XML に関する詳細については、市販の資料を参照してください。

XML ファイルは標準 ASCII ファイルであり、手作業で作成することも、**jetace** ツールのグラフィカル・ユーザー・インターフェース (GUI) で作成することもできます。コマンド行から **jetace** ツールを使用して、XML ファイルから EJB JAR ファイルを作成することができます。詳細については、54ページの『Enterprise Bean 用の EJB JAR ファイルの作成』を参照してください。

XML ベースのデプロイメント・ディスクリプターは、以下の主要なコンポーネントを含んでいなければなりません。

- 標準ヘッダーおよび EJB JAR タグ。詳細については、312ページの『標準ヘッダーおよび EJB JAR タグの作成』を参照してください。
- 入力ファイル・タグおよび出力ファイル・タグ。詳細については、312ページの『入力ファイル・タグおよび出力ファイル・タグの作成』を参照してください。
- デプロイメント・ディスクリプターを生成する bean の種類に応じて、セッション bean タグまたはエンティティー bean タグ。 XML ファイルには、すべての種類の複数の Enterprise Bean を持つ EJB JAR ファイルを生成するための命令を入れることができます。詳細については、313ページの『エンティティー bean タグの作成』および 314ページの『セッション bean タグの作成』を参照してください。

© Copyright IBM Corp. 1999, 2001 

- すべての Enterprise Bean によって使用されるタグ。詳細については、315ページの『すべての Enterprise Bean によって使用されるタグの作成』を参照してください。

## 標準ヘッダーおよび EJB JAR タグの作成

各 XML ベースのデプロイメント・ディスクリプターには、標準ヘッダー・タグがなければなりません。このタグは、XML のバージョンと XML ファイルのスタンドアロン状況を定義します。 Enterprise Bean の場合は、これらの特性を 図121 に示す値に設定しなければなりません。ヘッダー・タグはファイルの最初のタグでなければなりませんが、このタグ以外の XML ファイルの残りのコンテンツは、開始および終了 EJB JAR タグで囲まなければなりません。

```
<?xml version='1.0' standalone='yes' ?>
<ejb-JAR>
<!-- Content of the XML file -->
...
</ejb-JAR>
```

*図 121. コード例: 標準ヘッダーおよび EJB JAR タグ*

## 入力ファイル・タグおよび出力ファイル・タグの作成

入力ファイル・タグは、JAR ファイル、ZIP ファイル、または 1 つ以上の Enterprise Bean の必要なコンポーネントを含むディレクトリーを識別します。出力ファイル・タグは、作成する EJB JAR ファイルを識別します。デフォルトでは JAR ファイルが作成されますが、出力ファイル名に拡張子 .zip を追加することによって、強制的に ZIP ファイルを作成することができます。 Account bean の例に対する入力ファイルと出力ファイルを図122 に示します。

```
<?xml version='1.0' standalone='yes' ?>
<ejb-JAR>
<input-file>AccountIn.jar</input-file>
<output-file>Account.jar</output-file>
...
</ejb-JAR>
```

*図 122. コード例: 入力ファイル・タグおよび出力ファイル・タグ*

## エンティティー bean タグの作成

エンティティー bean に対するデプロイメント・ディスクリプターを作成する場合は、エンティティー bean タグを使用しなければなりません。エンティティー bean 開始タグは、dname 属性を含まなければなりません。この属性は、エンティティー bean に関連付けられたデプロイメント・ディスクリプターの完全修飾名に設定しなければなりません。

エンティティー bean 開始および終了タグの間には、以下のエンティティー bean 固有の属性タグを作成しなければなりません。

- <primary-key> － このエンティティー bean に対する 1 次キー・クラスの完全修飾名を識別します。
- <re-entrant> － このエンティティー bean が再入可能かどうかを指定します。このタグは、value 属性を含まなければなりません。この属性は、`true` (再入可能) または `false` (再入不可) のいずれかに設定しなければなりません。
- <container-managed> － コンテナー管理される CMP エンティティー bean の永続変数を識別します。永続変数ごとに別個のタグを使用しなければなりません。

エンティティー bean 固有のタグに加えて、315ページの『すべての Enterprise Bean によって使用されるタグの作成』で説明する、すべての Enterprise Bean が必要とするタグを作成しなければなりません。

314ページの図123 に、Account bean の例に対するエンティティー bean 固有のタグを示します。

```
<?xml version='1.0' standalone='yes' ?>
<ejb-JAR>
<input-file>AccountIn.jar</input-file>
<output-file>Account.jar</output-file>
...
<entity-bean dname="com/ibm/ejs/doc/account/Account.ser">
<primary-key>com.ibm.ejs.doc.account.AccountKey</primary-key>
<re-entrant value=false/>
<container-managed>accountId</container-managed>
<container-managed>type</container-managed>
<container-managed>balance</container-managed>
<!--Other tags used by all enterprise beans--!>
...
</entity-bean>
...
</ejb-JAR>
```

*図 123. コード例: エンティティー bean 固有のタグ*

## セッション bean タグの作成

セッション bean に対するデプロイメント・ディスクリプターを作成する場合は、セッション bean タグを使用しなければなりません。セッション bean 開始タグは、dname 属性を含まなければなりません。この属性は、セッション bean に関連付けられたデプロイメント・ディスクリプターの完全修飾名に設定しなければなりません。セッション bean 開始および終了タグの間には、以下のセッション bean 属性タグも作成しなければなりません。

- <session-timeout> ー セッション bean に関連付けられたアイドル・タイムアウトを秒単位で定義します。
- <state-management> ー セッション bean の種類 (STATELESS_SESSION または STATEFUL_SESSION) を識別します。

セッション bean 固有のタグに加えて、315ページの『すべての Enterprise Bean によって使用されるタグの作成』で説明する、すべての Enterprise Bean が必要とするタグを作成しなければなりません。

315ページの図124 に、Transfer bean の例に対するセッション bean タグを示します。

```
<?xml version='1.0' standalone='yes' ?>
<ejb-JAR>
<input-file>TransferIn.jar</input-file>
<output-file>Transfer.jar</output-file>
...
<session-bean dname="com/ibm/ejs/doc/transfer/Transfer.ser">
<session-timeout>0<¥session-timeout>
<state-management>STATELESS_SESSION<¥state-management>
<!--Other tags used by all enterprise beans--!>
...
</session-bean>
...
</ejb-JAR>
```

*図 124. コード例: セッション bean 固有のタグ*

## すべての Enterprise Bean によって使用されるタグの作成

以下のタグは、すべての種類の Enterprise Bean によって使用されます。これらのタグは、適切なセッション bean またはエンティティー bean の開始タグと終了タグの組の間に、それらの種類の bean に固有のタグとともに置かなければなりません。

- <remote-interface> ー Enterprise Bean のリモート・インターフェースの完全修飾名を識別します。
- <エンタープライズ - bean> ー Enterprise Bean の bean クラスの完全修飾名を識別します。
- <JNDI-name> ー Enterprise Bean の JNDI ホーム名を識別します。
- <transaction-attr> ー Enterprise Bean 全体に対するトランザクション属性を定義します。この属性は、個々の bean メソッドに対して設定することもできます。有効な値は、TX_MANDATORY、TX_NOT_SUPPORTED、 TX_REQUIRES_NEW、TX_REQUIRED、TX_SUPPORTS、および TX_BEAN_MANAGED です。これらの値の意味および制約事項に関する詳細については、160ページの『トランザクション属性の設定』を参照してください。
- <isolation-level> ー Enterprise Bean 全体のトランザクション分離レベル属性を定義します。この属性は、個々の bean メソッドに対して設定することもできます。値は、開始タグ内の value 属性を使用して設定しなければなりません。有効な値は、 SERIALIZABLE、REPEATABLE_READ、READ_COMMITTED、および READ_UNCOMMITTED です。これらの値の意味および制約事項に関する詳細については、164ページの『トランザクション分離レベル属性の設定』を参照してください。

- <run-as-mode> ー Enterprise Bean 全体の実行モード属性を定義します。この属性は、個々の bean メソッドに対して設定することもできます。値は、開始タグ内の value 属性を使用して設定しなければなりません。有効な値は、CLIENT_IDENTITY、SYSTEM_IDENTITY、および SPECIFIED_IDENTITY です。これらの値の意味に関する詳細については、167ページの『デプロイメント・ディスクリプターのセキュリティー属性の設定』を参照してください。
- <run-as-id> ー Enterprise Bean 全体の実行識別属性を定義します。この属性は、個々の bean メソッドに対して設定することもできます。この属性は、WebSphere Application Server に含まれる EJB サーバー環境では使用されません。
- <method-control> ー bean 全体に対する属性値と異なるトランザクション属性またはセキュリティー属性を持つ個々の bean メソッドを識別します。
- <dependency> ー この Enterprise Bean が依存するクラスの完全修飾名を識別します。
- <env-setting> ー Enterprise Bean が必要とする環境変数 (およびその値) を識別します。環境変数名は name 属性で指定します。環境変数値は、開始タグと終了タグの間に置きます。

図125 に、Transfer bean の例に対する Enterprise Bean タグを示します。Account bean も同様のセットを必要とします。

```
<?xml version='1.0' standalone='yes' ?>
<ejb-JAR>
<input-file>TransferIn.jar</input-file>
<output-file>Transfer.jar</output-file>
...
<session-bean dname="com/ibm/ejs/doc/transfer/Transfer.ser">
<!--Session bean-specific tags --!>
...
<remote-interface>com.ibm.ejs.doc.transfer.Transfer</remote-interface>
<enterprise-bean>com.ibm.ejs.doc.transfer.TransferBean</enterprise-bean>
<JNDI-name>Transfer </JNDI-name>
<transaction-attr value="TX_REQUIRED"/>
<isolation-level value="SERIALIZABLE"/>
<run-as-mode value="CLIENT_IDENTITY"/>
<dependency>com/ibm/ejs/doc/account/InsufficientFundsException.class</dependency>
...
<env-setting name="ACCOUNT_NAME">Account<env-setting>
...
</session-bean>
...
</ejb-JAR>
```

*図 125. コード例: すべての Enterprise Bean に使用されるタグ*

Enterprise Bean 全体に対するトランザクション属性またはセキュリティー属性を、その bean の特定のメソッドについて上書きする場合は、<method-control> タグを使用しなければなりません。開始タグと終了タグの間では、<method-name> タグでメソッドを識別し、<parameter> タグでメソッドのパラメーターの型を識別しなければなりません。さらに、<transaction-attr>、<isolation-level>、<run-as-mode>、および <run-as-id> の各タグを使用して、Enterprise Bean 全体とメソッドで異なる属性値を識別することができます。

たとえば、 図126 に示す XML では、Transfer bean のトランザクション属性 (TX_REQUIRED) を getBalance メソッドについてのみ TX_SUPPORTED に上書きすることを要求しています。トランザクション属性のみが上書きされるため、メソッドは、<isolation-level> および <run-as-mode> タグの値を Transfer bean から自動的に継承します。

```
<?xml version='1.0' standalone='yes' ?>
<ejb-JAR>
<input-file>TransferIn.jar</input-file>
<output-file>Transfer.jar</output-file>
...
<session-bean dname="com/ibm/ejs/doc/transfer/Transfer.ser">
<!--Session bean-specific tags --!>
...
<transaction-attr value="TX_REQUIRED"/>
<isolation-level value="SERIALIZABLE"/>
<run-as-mode value="CLIENT_IDENTITY"/>
...
<method-control>
<method-name>getBalance</method-name>
<parameter>long</parameter>
<transaction-attr value="TX_SUPPORTED"/>
</method-control>
</session-bean>
...
</ejb-JAR>
```

*図 126. コード例: メソッド固有のタグ*

# 付録D. EJB 仕様に対する機能拡張

この付録では、WebSphere Application Server で使用可能な、EJB 仕様に対する機能拡張について簡単に説明します。これらの拡張は、WebSphere Application Server に固有なものであり、これらのフィーチャーの使用は、VisualAge for Java エンタープライズ版だけでサポートされます。これらのフィーチャーのインプリメントについては、VisualAge for Java の資料を参照してください。

## アクセス bean

アクセス *bean* は、Sun Microsystems JavaBeans™ 仕様に準拠する Java コンポーネントであり、EJB クライアントの開発を容易にするためのものです。アクセス bean は、アクセス bean ユーザー (つまり、EJB クライアント開発者) からホーム・インターフェースおよびリモート・インターフェースを隠すことにより、 Enterprise Bean を JavaBeans プログラミング・モデルに適合させます。アクセス bean は、アドバンスド版および Component Broker EJB 環境の両方でサポートされています。

アクセス bean には、次の 3 つのタイプがあります (単純なものから順に示します)。

- Java bean ラッパー ー 3 つのタイプのアクセス bean のうちで、Java bean ラッパーは最も簡単に作成できます。これは、セッションまたはエンティティーの Enterprise Bean を標準 Java bean と同じように使用できるようにするために設計され、 Enterprise Bean のホームおよびリモート・インターフェースをユーザーから隠します。ユーザーが作成した各 Java bean ラッパーは、com.ibm.ivj.ejb.access.AccessBean クラスを拡張します。
- コピー・ヘルパー ー コピー・ヘルパー・アクセス bean は、Java bean ラッパーのすべての特性を備え、さらにリモート・エンティティー bean から得られた属性のローカル・コピーを含む、単一のコピー・ヘルパー・オブジェクトも含んでいます。ユーザー・プログラムは、アクセス bean に入っているローカル・コピー・ヘルパー・オブジェクトからエンティティー bean 属性を検索することができます。これにより、リモート・エンティティー bean から属性にアクセスする必要がなくなります。
- 行セット ー 行セット・アクセス bean は、Java bean ラッパーとコピー・ヘルパーの両方のアクセス bean の、すべての特性を備えています。ただし、単一のコピー・ヘルパー・オブジェクトではなく、複数のコピー・ヘル

© Copyright IBM Corp. 1999, 2001

パー・オブジェクトを含んでいます。それぞれのコピー・ヘルパー・オブジェクトは、単一の Enterprise Bean インスタンスに対応しています。

VisualAge for Java は、アクセス bean の作成または編集を支援するための SmartGuide を備えています。

## Enterprise Bean 間の関連

EJB サーバー環境における関連は、2 つの CMP エンティティー bean 間に存在する関係を意味します。関連には、1 対 1 と 1 対多の、2 つのタイプがあります。 1 対 1 関連では、1 つの CMP エンティティー bean が、別の CMP エンティティー bean の単一インスタンスに関連付けられます。たとえば、1 つの Employee (従業員) bean は、ある Department (部門) bean の単一インスタンスにだけ関連付けられます。これは、一般に従業員が 1 つの部門にだけ所属しているためです。

1 対多関連では、1 つの CMP エンティティー bean が、別の CMP エンティティー bean の複数インスタンスに関連付けられます。たとえば、1 つの Department は、1 つの Employee bean の複数インスタンスに関連付けることができます。これは、ほとんどの部門が複数の従業員によって構成されるためです。

VisualAge for Java における CMP エンティティー bean 間の関連を作成または編集するためには、関連エディターが使用されます。

## Enterprise Bean における継承

Java における*継承* は、既存のクラスから新規クラスを作成したり、既存のインターフェースから新規インターフェースを作成したりすることを意味します。EJB サーバー環境では、標準クラス継承と EJB 継承の 2 つの形式の継承が許されます。標準クラス継承では、ホーム・インターフェース、リモート・インターフェース、または Enterprise Bean クラスが、それ自体は Enterprise Bean クラスまたはインターフェースでない基本クラスから、プロパティーとメソッドを継承します。

これに対し、Enterprise Bean 継承では、Enterprise Bean クラスが、同じグループに含まれる別の Enterprise Bean からプロパティー (CMP フィールドや関連の両端など)、メソッド、およびメソッド・レベルの制御記述子属性を継承します。

VisualAge for Java は、Enterprise Bean での継承のインプリメントを支援するための SmartGuide を備えています。

# 特記事項

本書はアメリカ合衆国で提供されている製品およびサービス用に作成されたものであり、本書に記載の製品、サービス、またはフィーチャーが日本においては提供されていない場合があります。日本で利用可能な製品、サービス、およびフィーチャーについては、日本 IBM の営業担当員にお尋ねください。本書で IBM 製品、プログラム、またはサービスに言及していても、その IBM 製品、プログラム、またはサービスのみが使用可能であることを意味するものではありません。これらに代えて、IBM の 知的所有権を侵害することのない、機能的に同等な製品、プログラム、またはサービスを使用することができます。ただし、IBM 製以外の製品と組み合わせた場合、その操作の評価と検証については、お客様の責任で行っていただきます。

IBM は、本書に記載されている内容に関して特許権 (特許出願中のものを含む。) を保有している場合があります。本書の提供は、お客様にこれらの特許権について実施権を許諾することを意味するものではありません。実施権の許諾については、下記の宛先に、書面にてご照会ください。

〒106-0032 東京都港区六本木 3 丁目 2-31
AP 事業所
IBM World Trade Asia Corporation
Intellectual Property Law & Licensing

**以下の保証は、国または地域の法律に沿わない場合は、適用されません。**

IBM およびその直接または間接の子会社は、本書を特定物として現存するままの状態で提供し、商品性の保証、特定目的適合性の保証および法律上の瑕疵担保責任を含むすべての明示もしくは黙示の保証責任を負わないものとします。国または地域によっては、法律の強行規定により、保証責任の制限が禁じられる場合、強行規定の制限を受けるものとします。

本書は定期的に見直され、必要な変更 (たとえば、技術的に不適確な表現や誤植など) は、本書の次版に組み込まれます。IBM は予告なしに、随時、この文書に記載されている製品またはプログラムに対して、改良または変更を行うことがあります。

本書において IBM 以外の Web サイトに言及している場合がありますが、便宜のため記載しただけであり、決してそれらの Web サイトを推奨するもので

はありません。それらの Web サイトにある資料は、この IBM 製品の資料の一部ではありません。それらの Web サイトは、お客様の責任でご使用ください。

IBM は、お客様が提供するいかなる情報も、お客様に対してなんら義務も負うことのない、自ら適切と信ずる方法で、使用もしくは配布することができるものとします。

本プログラムのライセンス保持者で、(i) 独自に作成したプログラムとその他のプログラム (本プログラムを含む) との間での情報交換、および (ii) 交換された情報の相互利用を可能にすることを目的として、本プログラムに関する情報を必要とする方は、下記に連絡してください。

**Component Broker** については、
IBM Corporation
Department LZKS
11400 Burnet Road
Austin, TX 78758
U.S.A.

**TXSeries** については、
IBM Corporation
ATTN: Software Licensing
11 Stanwix Street
Pittsburgh, PA 15222
U.S.A.

本プログラムに関する上記の情報は、適切な使用条件の下で使用することができますが、有償の場合もあります。

本書で説明されているライセンス・プログラムまたはその他のライセンス資料は、IBM 所定のプログラム契約の契約条項、IBM プログラムのご使用条件、またはそれと同等の条項に基づいて、IBM より提供されます。

この文書に含まれるいかなるパフォーマンス・データも、管理環境下で決定されたものです。そのため、他の操作環境で得られた結果は、異なる可能性があります。一部の測定が、開発レベルのシステムで行われた可能性がありますが、その測定値が、一般に利用可能なシステムのものと同じである保証はありません。さらに、一部の測定値が、推定値である可能性があります。実際の結果は、異なる可能性があります。お客様は、お客様の特定の環境に適したデータを確かめる必要があります。

IBM 以外の製品に関する情報は、その製品の供給者、出版物、もしくはその他の公に利用可能なソースから入手したものです。IBM は、それらの製品のテストは行っておりません。また、IBM 以外の製品に関するパフォーマンスの正確性、互換性、またはその他の要求は確証できません。IBM 以外の製品の性能に関する質問は、それらの製品の供給者にお願いします。

IBM の将来の方向または意向に関する記述については、予告なしに変更または撤回される場合があり、単に目標を示しているものです。

本書には、日常の業務処理で用いられるデータや報告書の例が含まれています。より具体性を与えるために、それらの例には、個人、企業、ブランド、あるいは製品などの名前が含まれている場合があります。これらの名称はすべて架空のものであり、名称や住所が類似する企業が実在しているとしても、それは偶然にすぎません。

この情報をソフトコピーでご覧になっている場合は、写真やカラーの図表は現れない場合があります。

## 商標およびサービス・マーク

以下の用語は、IBM Corporation の米国およびその他の国における商標または登録商標です。

Advanced Peer-to-Peer Networking
AFS
AIX
APPN
AS/400
CICS
CICS OS/2
CICS/400
CICS/6000
CICS/ESA
CICS/MVS
CICS/VSE
CICSPlex
DB2
DCE Encina Lightweight Client
DFS
Encina
IBM
IBM System Application Architecture
IMS
IMS/ESA
Language Environment
MQSeries
MVS/ESA
NetView
Open Class
OS/2
OS/390
OS/400
Parallel Sysplex
PowerPC
RACF
RAMAO
RMF
RISC System/6000
RS/6000
S/390
SecureWay
TeamConnection
Transarc
TXSeries
VSE/ESA
VTAM
VisualAge
WebSphere

Domino、Lotus、および LotusScript は Lotus Development Corporation の米国およびその他の国における商標です。

Tivoli は、Tivoli Systems, Inc. の米国およびその他の国における登録商標です。

Microsoft、Windows、Windows NT、および Windows ロゴは Microsoft Corporation の米国および他の国における商標です。

Java およびすべての Java 関連の商標およびロゴは、Sun Microsystems, Inc. の米国およびその他の国における商標および登録商標です。

UNIX は、The Open Group がライセンスしている米国および他の国における登録商標です。

本書の一部は、以下の著作権表示を付しているオブジェクト管理グループの資料に基づいています。

Copyright 1995, 1996 AT&T/NCR
Copyright 1995, 1996 BNR Europe Ltd.
Copyright 1991, 1992, 1995, 1996 by Digital Equipment Corporation
Copyright 1996 Gradient Technologies, Inc.
Copyright 1995, 1996 Groupe Bull
Copyright 1995, 1996 Expersoft Corporation
Copyright 1996 FUJITSU LIMITED
Copyright 1996 Genesis Development Corporation
Copyright 1989, 1990, 1991, 1992, 1995, 1996 by Hewlett-Packard Company
Copyright 1991, 1992, 1995, 1996 by HyperDesk Corporation
Copyright 1995, 1996 IBM Corporation
Copyright 1995, 1996 ICL, plc
Copyright 1995, 1996 Ing. C. Olivetti &C.Sp
Copyright 1997 International Computers Limited
Copyright 1995, 1996 IONA Technologies, Ltd.
Copyright 1995, 1996 Itasca Systems, Inc.
Copyright 1991, 1992, 1995, 1996 by NCR Corporation
Copyright 1997 Netscape Communications Corporation
Copyright 1997 Northern Telecom Limited
Copyright 1995, 1996 Novell USG
Copyright 1995, 1996 02 Technolgies
Copyright 1991, 1992, 1995, 1996 by Object Design, Inc.
Copyright 1991, 1992, 1995, 1996 Object Management Group, Inc.
Copyright 1995, 1996 Objectivity, Inc.
Copyright 1995, 1996 Oracle Corporation
Copyright 1995, 1996 Persistence Software
Copyright 1995, 1996 Servio, Corp.
Copyright 1996 Siemens Nixdorf Informationssysteme AG
Copyright 1991, 1992, 1995, 1996 by Sun Microsystems, Inc.
Copyright 1995, 1996 SunSoft, Inc.
Copyright 1996 Sybase, Inc.
Copyright 1996 Taligent, Inc.
Copyright 1995, 1996 Tandem Computers, Inc.
Copyright 1995, 1996 Teknekron Software Systems, Inc.
Copyright 1995, 1996 Tivoli Systems, Inc.
Copyright 1995, 1996 Transarc Corporation
Copyright 1995, 1996 Versant Object Technology Corporation

Copyright 1997 Visigenic Software, Inc.
Copyright 1996 Visual Edge Software, Ltd.

上記の各著作権者は、本書に記載される仕様を使用したこと、またはコンピューター・ソフトウェアがその仕様に従ったことを理由に、本書に記載された資料に係る当該著作権者の著作権を侵害するとは考えていません。

本書の情報は正確を期していますが、オブジェクト管理グループおよび上記の企業は、本書の内容に関していかなる種類の保証もせず、しかも、商品性および特定目的適合性の黙示の保証もしないものとします。オブジェクト管理グループおよび上記の企業は、本書に含まれる誤り、もしくは本書に記述された内容を提供、実行、使用したことによって起こる付随的あるいは間接的損害に対する責任を負わないものとします。

本ソフトウェアには、RSA 暗号化コードが含まれています。

他の会社名、製品名およびサービス名等はそれぞれ各社の商標またはサービス・マークです。

# 索引

日本語, 数字, 英字, 特殊文字の順に配列されています。なお, 濁音と半濁音は清音と同等に扱われています。

## ［ア行］


アトミシティ 9
アプリケーション・アセンブリー・ツール 36, 39
一時プロセス 11
インスタンス変数
  サーブレット 192
  セッション bean 139
  BMP を持つエンティティー bean 203
  CMP を持つエンティティー bean 120
インストール
  EJB サーバー (CB) への Enterprise Bean の 96
エンティティー bean 17
  インスタンス変数 (BMP) 203
  インスタンス変数 (CMP) 120
  開発 (BMP) 201
  開発 (CMP) 118
  コンポーネント 18
  コンポーネント (BMP) 201
  コンポーネント (CMP) 118
  作成状態 31
  作動可能状態 32
  除去状態 33
  デプロイメント・ディスクリプター属性 23, 59
  ビジネス・メソッド (BMP) 206
  ビジネス・メソッド (CMP) 122
  プール状態 32
  ホーム・インターフェース (BMP) 214
エンティティー bean 17 *(続き)*
  ホーム・インターフェース (CMP) 129
  ライフ・サイクル 31
  リモート・インターフェース (BMP) 216
  リモート・インターフェース (CMP) 133
  1 次キー・クラス (BMP) 218
  1 次キー・クラス (CMP) 134
  bean クラス (BMP) 202
  bean クラス (CMP) 118


## ［カ行］


解決フェーズ 11
開発
  エンティティー bean (BMP) 201
  エンティティー bean (CMP) 118
  セッション bean 138
  CICS アプリケーションからの Enterprise Bean の 108
  EJB アプリケーション 26
  EJB クライアント 169, 189
  Enterprise Bean 35, 36, 43, 117
  Enterprise Bean でのサーブレットの 189
  IMS アプリケーションからの Enterprise Bean の 108
  MQSeries 用の Enterprise Bean の 109
回復可能プロセス 11
環境 148
環境変数
  デプロイメント・ディスクリプター属性 67, 145, 146
環境名コンテキスト 148
管理
  セキュリティー・サービス 6
  ワークロード管理サービス 7
管理 *(続き)*
  EJB クライアントでのトランザクション 182
  EJB サーバー (AE) でのデータベース接続の 223
  EJB サーバー (CB) でのデータベース接続の 220
  Enterprise Bean 内のトランザクション 228
  WebSphere Application Server 15
許可 5
コーディネーター 11
国際化対応
  テクニック 278
コマンド 245
  コマンド・インターフェース 246, 250, 257
  実行メソッド 250
  ターゲット 248, 263, 266, 272, 274, 275
  ターゲット (サーブレット) 272, 274, 275
  ターゲット (Enterprise Bean) 263
  ターゲット・ポリシー 248, 249, 266, 267, 268, 269, 270, 271
  ユーザー定義の例外クラス 250
  リセット・メソッド 250, 257
  例外クラス 250
  CommandException クラス 250, 264
  CommandTarget インターフェース 248, 263, 264, 272, 274
  CompensableCommand インターフェース 246, 251, 259
  DistributedException クラス 250
  executeCommand メソッド 263, 264, 272, 274
  getCommandTarget メソッド 250, 266, 270
  getCommandTargetName メソッド 250, 266


© Copyright IBM Corp. 1999, 2001

コマンド 245 *(続き)*
getCompensatingCommand メソッド 251, 259
getTargetPolicy メソッド 270
hasOutputProperties メソッド 250, 264
isReadyToCallExecute メソッド 250, 257
listMappings メソッド 266
LocalTarget クラス 266
performExecute メソッド 250, 257, 264, 275
registerCommand メソッド 266, 269
setCommandTarget メソッド 250, 266
setCommandTargetName メソッド 250, 266
setDefaultTargetName メソッド 266, 268
setHasOutputProperties メソッド 250
setOutputProperties メソッド 250, 253, 257
setTargetPolicy メソッド 270
TargetableCommand インターフェース 246, 247, 248, 250, 253, 257, 266, 274
TargetableCommandImpl クラス 247, 253, 254, 270
TargetPolicy インターフェース 249, 266, 270
TargetPolicyDefault クラス 249, 266
UnauthorizedAccessException クラス 250
unregisterCommand メソッド 266, 269
UnsetInputPropertiesException クラス 250
コマンド・インターフェース 246, 250, 257
コミット
トランザクション 10, 184, 228
コンテナー管理のパーシスタンス 7, 19, 118
コンポーネント
エンティティー bean 18
エンティティー bean (BMP) 201
エンティティー bean (CMP) 118
セッション bean 20, 138
EJB サーバー 1

## ［サ行］

サービス
セキュリティー 4
トランザクション 9
ネーム解決 8
パーシスタンス 7
ワークロード管理 7
サーブレット
インスタンス変数 192
初期化 193
スレッド・セーフにする 200
標準のメソッド 189
ユーザー入力の処理 196, 198, 199
Enterprise Bean の作成 193, 198
Enterprise Bean を使用する 189, 191
HTML への組み込み 190, 200
JSP との比較 200
Web サーバーの要件 15, 189
再入可能性
Enterprise Bean での 155
作成
サーブレット内の Enterprise Bean 193, 198
デプロイメント・ディスクリプター 39, 54
EJB JAR ファイル 54
EJB クライアントでの EJB オブジェクトの 172
EJB クライアントでの EJB ホーム・オブジェクトの 176
EJB モジュール 39, 157
XML でのデプロイメント・ディスクリプターの 311
作成状態
エンティティー bean 31
セッション bean 29
作動可能状態
エンティティー bean 32
セッション bean 29
システム管理エンド・ユーザー・インターフェース 15
実行メソッド 250
準備フェーズ 11
状態付きセッション bean 21, 139, 144, 150, 152, 228
状態なしセッション bean 21, 139, 144, 150, 152, 178, 228
初期化
サーブレット 193
除去
EJB クライアントでの EJB オブジェクトの 181
除去状態
エンティティー bean 33
セッション bean 31
スレッド 13
サーブレットの 200
Enterprise Bean での 155
整合性 9
静的変数 (制約事項) 120, 139, 203
制約事項
EJB サーバー (CB) 112
セキュリティー 13
デプロイメント・ディスクリプター属性 22, 65, 159, 167
セキュリティー・サービス 4
管理 6
セッション bean 17
インスタンス変数 139
開発 138
コンポーネント 20, 138
作成状態 29
作動可能状態 29
状態付き 21, 139, 144, 150, 152, 228
状態なし 21, 139, 144, 150, 152, 178, 228
除去状態 31
デプロイメント・ディスクリプター属性 23, 61
プール状態 30
ホーム・インターフェース 151

セッション bean 17 *(続き)*
ライフ・サイクル 29
リモート・インターフェース 152
セッション・サービス 185
接続 (データベース)
エンティティー bean (BMP) 219
解放 222
クローズ 221
作成 221
割り振り 224
割り振り解除 225
EJB サーバー (AE) での管理 223
EJB サーバー (CB) での管理 220
接続マネージャー 223


# [タ行]


ターゲット・ポリシー 248, 266, 267, 268, 269, 270, 271
カスタム 270, 271
デフォルト 248, 266, 267, 268, 269
耐障害性 9
逐次列挙型 54
ツール
アプリケーション・アセンブリー・ツール 36
EJB サーバー (AE) 35, 36
EJB サーバー (CB) 43
VisualAge for Java 35
データベース 12
クラス・ドライバーの登録 221
クラス・ドライバーのロード 221
接続の解放 222
接続のクローズ 221
接続の作成 221
接続の取得 219
接続の割り振り 224
接続の割り振り解除 225
データの操作 226
EJB オブジェクト参照 224
EJB サーバー (AE) 41
データベース 12 *(続き)*
EJB サーバー (CB) 82, 83, 86, 89, 93
データ・ソース 12
デプロイメント・ディスクリプター 22
エンティティー bean 属性 23, 59
環境変数の属性 67, 145, 146
コンポーネント名属性 58
作成 39, 54
セキュリティー属性 22, 65, 159, 167
セッション bean 属性 23, 61
トランザクション属性 22, 63, 159, 160, 164
ファイル依存関係属性 68
JNDI 名属性 58
XML での作成 311
登録
データベースのクラス・ドライバーの 221
トランザクション 9, 13
解決フェーズ 11
コーディネーター 11
コミット 10, 184, 228
準備フェーズ 11
デプロイメント・ディスクリプター属性 22, 63, 159, 160, 164
分散 10
ロールバック 10, 184, 228
2 フェーズ・コミット 11
bean 管理 228
EJB クライアントでの管理 182
Enterprise Bean での管理 228
トランザクション・サービス 9


# [ナ行]


認証 4
ネーム解決サービス 8


# [ハ行]


パーシスタンス 19
パーシスタンス管理サービス 7
配置
配置の検証 (CB) 72, 73
配置 *(続き)*
配置の削除 (CB) 72, 73
EJB サーバー (CB) への Enterprise Bean の 74, 81
Enterprise Bean 25, 35, 36, 43, 156
Enterprise Bean、EAR ファイルから (CB) 72
Enterprise Bean、JAR ファイルから 70
バインド
EJB サーバー (CB) での JNDI への Enterprise Bean の 97, 101
EJB サーバー (CB) でのファクトリー・ファインダーと Enterprise Bean の 101
パッケージ (Java)
EJB クライアントに必要な 171
Enterprise Bean 157
パッケージ化
Enterprise Bean 22, 54
ビジネス・メソッド
エンティティー bean (BMP) 206
エンティティー bean (CMP) 122
セッション bean 141
プール状態
エンティティー bean 32
セッション bean 30
ファイル依存関係
デプロイメント・ディスクリプター属性 68
ファインダー・ヘルパー・インターフェース 39
ファインダー・ヘルパー・クラス 48
プリンシパル・コンテキスト 167
プログラミング・モデル拡張機能 233
コマンド・パッケージ 245
分散例外パッケージ 233
ローカライズ可能テキスト・パッケージ 277
分散トランザクション 10
分散例外 233
ユーザー定義の 237, 238, 239, 241, 243

分散例外 233 *(続き)*
ローカライズ 236
DistributedException クラス 234, 235
DistributedExceptionEnabled インターフェース 234, 236
DistributedExceptionInfo クラス 234, 237
ExceptionInstantiationException クラス 234
getException メソッド 235, 236
getExceptionInfo メソッド 235, 236
getMessage メソッド 235, 236
getOriginalException メソッド 235, 236
getPreviousException メソッド 235, 236
printStackTrace メソッド 235, 236
printSuperStackTrace メソッド 236
分離 9, 164
変数
サーブレットの 192
静的 (制約事項) 120, 139, 203
リソース・バンドルからの値の取得 121, 122, 173, 175, 192
bean クラス (BMP を持つエンティティー) 203
bean クラス (CMP を持つエンティティー) 120
Enterprise Bean からの値の取得 205, 221, 224
ホーム・インターフェース
エンティティー bean (BMP) 18, 214
エンティティー bean (CMP) 18, 129
セッション bean 20, 151
JNDI での検索 174

## [マ行]

メソッド・レベル属性
デプロイメント・ディスクリプター 63, 65
メッセージ・カタログ 279
ネーム解決 280
場所 280

## [ヤ行]

ユーザー定義の例外
EJB JAR ファイルでの 157
ユーザー定義の例外クラス 122, 134, 172, 217, 250
分散例外 237, 238, 239, 241, 243
ユーザー・コンテキスト 167

## [ラ行]

ライフ・サイクル
エンティティー bean 31
作成状態 (エンティティー) 31
作成状態 (セッション) 29
作動可能状態 (エンティティー) 32
作動可能状態 (セッション) 29
除去状態 (エンティティー) 33
除去状態 (セッション) 31
セッション bean 29
プール状態 (エンティティー) 32
プール状態 (セッション) 30
Enterprise Bean 29
ライフ・サイクル・サービス 101
アプリケーション固有の関連 104
デフォルトの関連付け 102
リセット・メソッド 250, 257
リソース・バンドル
ネーム解決 280
場所 280
変数値の取得 121, 122, 173, 175, 192
EJB JAR ファイルでの 157
リフレッシュ
セッション bean の EJB オブジェクト 179
リモート・インターフェース 134
エンティティー bean (BMP) 18, 216
エンティティー bean (CMP) 18, 133
セッション bean 20, 152
例
本書のコード 307
EJB アプリケーション 26
EJB サーバー (AE) に提供されている 309
EJB サーバー (CB) に提供されている 309
例外
チェーニング 233
分散 233
例外クラス
ユーザー定義の 122, 134, 172, 217, 237, 238, 250
CommandException 250, 264
CreateException 123, 125, 131, 152, 207, 215
DistributedException 250
DuplicateKeyException 123, 125, 207
EJBException 123, 125, 126, 139, 142, 144
ExceptionInstantiationException 234
FinderException 123, 131, 142, 208, 209, 216
NoSuchObjectException 31, 180
ObjectNotFoundException 123, 208
RemoteException 125, 126, 130, 131, 133, 142, 151, 152, 153, 207, 212, 214, 215, 216, 217
RemoveException 31, 123, 126, 212
RuntimeException 139
TransactionRequiredException 161
UnauthorizedAccessException 250
UnsetInputPropertiesException 250
ローカライズ 277

ローカライズ 277 *(続き)*
アプリケーション分析 278
合理的な 277
テクニック 278
ローカライズ可能テキスト 277
アプリケーション固有の引き数 286
アプリケーション分析 278
オプション引き数 290, 292, 294
カスタマイズしたフォーマット 290, 294
合理的な 277
作業 286
テクニック 278
フォーマット bean の配置 300
フォーマットの詳細 283
フォーマットのネスト 292
フォーマット・アプリケーション 284
フォールバック情報 285
複合ストリングのアセンブル 290, 291
変数サブストリング 291, 294
変数サブストリング (ローカライズ) 292
メッセージのキャッシング 285
メッセージ・カタログ 279
メッセージ・カタログの位置決め 280, 284
メッセージ・カタログのネーム解決 280
ユーザー定義のフォーマット 294
リソース・バンドル 279
format メソッド 282, 284
Java サポート 281
java.text.MessageFormat クラス 281, 282
java.util.Locale クラス 281, 282
java.util.ResourceBundle クラス 281, 282
java.util.TimeZone クラス 281, 282
LocalizableConfiguration クラス 284
ローカライズ可能テキスト 277 *(続き)*
LocalizableTextDateTimeArgument クラス 294
LocalizableTextEJBDeploy ツール 300
LocalizableTextFormatter クラス 282, 284
LocalizableTextResourceAccessor Bean 300
WebSphere サポート 281, 282
ロード
データベースのクラス・ドライバーの 221
ロールバック
トランザクション 10, 184, 228

## [ワ行]

ワークロード管理サービス 7
管理 7

## [数字]

1 次キー 18
および remove メソッド 154
配置時に指定 135
未知の 136
1 次キー・クラス 18
エンティティー bean (BMP) 218
エンティティー bean (CMP) 134
2 フェーズ・コミット 11

## A

ACID 特性 9
ActiveX EJB クライアント 184, 185
afterBegin メソッド 29
afterCompletion メソッド 29
appbind ツール 44, 101, 104, 105

## B

bean 管理のパーシスタンス 7, 19, 201
bean クラス
エンティティー bean (BMP) 18, 202
エンティティー bean (CMP) 18, 118
セッション bean 20
変数 (BMP を持つエンティティー) 203
変数 (CMP を持つエンティティー) 120
beforeCompletion メソッド 29

## C

CBDeployEar ツール 44, 72
CBDeployJar ツール 44, 70
cbejb ツール 44, 74, 81, 83
CB_EJB_JAVA_CP 環境変数 47
CDS (DCE) 8
CICS 12, 89, 108
CLASSPATH 環境変数
EJB サーバー (AE) 38
EJB サーバー (CB) 47
Class.forName メソッド 221
clearLocalizableTextFormatter メソッド 287
CommandException クラス 250, 264
CommandTarget インターフェース 248, 263, 264, 272, 274
CompensableCommand インターフェース 246, 251, 259
Component Broker
セッション・サービス 185
ライフ・サイクル・サービス 101, 102, 104
com.ibm.websphere.i18n.localizabletext パッケージ 281, 282
CORBA EJB クライアント 184
create メソッド
エンティティー bean 32
エンティティー bean (BMP) 206, 214, 215
エンティティー bean (CMP) 124, 130, 131
セッション bean 29, 144, 151, 152

CreateException クラス　123, 125, 131, 152, 207, 215
Current インターフェース (CORBA)　167

## D

DataSource インターフェース　223, 225
DB2 データベース　12, 86, 93
DCE CDS　8
destroy メソッド (サーブレット)　189
DistributedException クラス　234, 235, 250
DistributedExceptionEnabled インターフェース　234, 236
DistributedExceptionInfo クラス　234, 237
DNS　8
doGet メソッド (サーブレット)　189, 196, 198, 199
doPost メソッド (サーブレット)　189
DriverManager インターフェース　220, 222
DuplicateKeyException　123
DuplicateKeyException クラス　125, 207

## E

EAR ファイル
- Enterprise Bean の配置　44
- Enterprise Bean の配置 (CB)　72

EJB JAR ファイル　22
- 作成　54

EJB アプリケーション
- 開発　26
- 例　26

EJB オブジェクト　20, 25
- データベースへの参照　224
- 無効な　179
- EJB クライアントでの作成　172
- EJB クライアントでの除去　181

EJB オブジェクト・クラス　18, 20, 25, 152
EJB クライアント　13
- 開発　169, 189
- スレッド　13
- セキュリティー　13
- トランザクション　13
- トランザクションの管理　182
- ネーム解決および通信　13
- 必要な Java パッケージ　171
- EJB オブジェクトの作成　172
- EJB オブジェクトの除去　181
- EJB オブジェクトのホーム・オブジェクトの作成　176
- EJB サーバー (CB) でのみサポートされている　184

EJB サーバー　2
- コンテナー　3
- コンポーネント　1
- サービス　3
- ツール　3

EJB サーバー (AE)
- コード例　309
- 前提条件ソフトウェア　37
- ツール　35, 36
- データベース　41
- データベース接続の管理　223
- ファインダー・ヘルパー・インターフェース　39
- CLASSPATH 環境変数　38

EJB サーバー (CB)
- コード例　309
- 制約事項　112
- 前提条件ソフトウェア　47
- ツール　43
- 追加の EJB クライアント　184
- データベース　82, 83, 86, 89, 93
- データベース接続の管理　220
- ファインダー・ヘルパー・クラス　48
- ファクトリー・ファインダーと Enterprise Bean のバインディング　101
- CLASSPATH 環境変数　47
- Enterprise Bean のインストール　96
- Enterprise Bean の配置　74, 81

EJB サーバー (CB) *(続き)*
- Enterprise Bean の配置、EAR ファイルから　72
- Enterprise Bean の配置、JAR ファイルから　70
- JNDI での Enterprise Bean のバインド　97, 101

EJB ホーム・オブジェクト　20, 25, 214
- EJB クライアントでの作成　176

EJB ホーム・クラス　18, 20, 25, 130
EJB モジュール　22
- 作成　39, 157
- デプロイメント・ディスクリプター　22

ejbActivate メソッド
- エンティティー bean　32
- エンティティー bean (BMP)　212
- エンティティー bean (CMP)　126
- セッション bean　30, 149

ejbbind ツール　44, 97
ejbCreate メソッド
- エンティティー bean　32
- エンティティー bean (BMP)　202, 206, 214, 215
- エンティティー bean (CMP)　119, 124, 130, 131
- セッション bean　29, 138, 139, 144, 151, 152

EJBException クラス　123, 125, 126, 139, 142, 144
ejbfhgen ユーティリティー　51, 53
ejbFindByPrimaryKey メソッド
- エンティティー bean (BMP)　208
- エンティティー bean (CMP)　131
- 1 次キー　131

EJBHome インターフェース　130, 151, 154, 214
ejbLoad メソッド　32
- エンティティー bean (BMP)　212
- エンティティー bean (CMP)　126

EJBObject インターフェース　133, 153, 154, 217
ejbPassivate メソッド
- エンティティー bean　32

ejbPassivate メソッド *(続き)*
- エンティティー bean (BMP) 212
- エンティティー bean (CMP) 126
- セッション bean 30, 149

ejbPostCreate メソッド 32
- エンティティー bean (BMP) 202, 206, 214, 215
- エンティティー bean (CMP) 119, 124, 130, 131

ejbRemove メソッド
- エンティティー bean (BMP) 212
- エンティティー bean (CMP) 126
- セッション bean 31, 149

ejbStore メソッド 32
- エンティティー bean (BMP) 212
- エンティティー bean (CMP) 126

Enterprise Bean 17
- 開発 35, 36, 43, 117
- サーブレットでの作成 193, 198
- サーブレットでの使用 189, 191
- 再入可能性 155
- スレッド 155
- トランザクションの管理 228
- 配置 25, 35, 36, 43, 156
- 配置、EAR ファイルから (CB) 72
- 配置、JAR ファイルから 70
- 配置解除 (CB) 72, 73
- パッケージ (Java) 157
- パッケージ化 22, 54
- 変数値の取得 205, 221, 224
- ライフ・サイクル 29
- CICS アプリケーションからの開発 108
- EJB サーバー (CB) での JNDI へのバインド 97, 101
- EJB サーバー (CB) でのファクトリー・ファインダーとのバインディング 101
- EJB サーバー (CB) へのインストール 96
- EJB サーバー (CB) への配置 74, 81
- EJB モジュール 22
- IMS アプリケーションからの開発 108

Enterprise Bean 17 *(続き)*
- MQSeries 用の開発 109

EntityBean インターフェース 119, 126, 202, 212

Enumeration インターフェース 131, 216

equals メソッド 135

ExceptionInstantiationException クラス 234

executeCommand メソッド 263, 264, 272, 274

## F

findByPrimaryKey メソッド 131, 142, 216
- エンティティー bean (BMP) 214
- エンティティー bean (CMP) 130

finder メソッド
- エンティティー bean (BMP) 208, 216
- エンティティー bean (CMP) 131

FinderException 123

FinderException クラス 131, 142, 208, 209, 216

FinderHelperGenerator クラス 51

## G

getCommandTarget メソッド 250, 266, 270

getCommandTargetName メソッド 250, 266

getCompensatingCommand メソッド 251, 259

getEJBHome メソッド 154

getEJBMetaData メソッド 154

getException メソッド 235, 236

getExceptionInfo メソッド 235, 236

getHandle メソッド 154

getInitialContext メソッド 145

getMessage メソッド 235, 236

getOriginalException メソッド 235, 236

getPreviousException メソッド 235, 236

getPrimaryKey メソッド 154

getTargetPolicy メソッド 270

## H

hashCode メソッド 135

hasOutputProperties メソッド 250, 264

HTML
- サーブレットの組み込み 190, 200

HTTP 13

HttpServlet クラス 191

## I

IIOP 13

IMS 12, 89, 108

init メソッド (サーブレット) 189, 193

InitialContext インターフェース 145, 174

INITIAL_CONTEXT_FACTORY プロパティー 145, 173

isIdentical メソッド 154

isReadyToCallExecute メソッド 250, 257

## J

J2EE 72

J2EE ツール 44

jar コマンド 36, 44, 54

JAR ファイル
- 配置解除 (CB) 72, 73
- 配置の検証 (CB) 72, 73
- Enterprise Bean の配置 (CB) 70

javac コマンド 36, 38, 44, 48

javax.ejb パッケージ 31, 119, 123, 125, 126, 130, 131, 133, 138, 139, 142, 144, 149, 151, 152, 153, 154, 171, 202, 207, 208, 209, 212, 214, 215, 216, 217

javax.naming パッケージ 145, 171, 173, 174

javax.rmi.PortableRemoteObject.narrow メソッド 147, 177

javax.servlet パッケージ 191

javax.servlet.http パッケージ 191
javax.transaction パッケージ 12, 182, 229
java.io パッケージ 155
java.jts パッケージ 161
java.lang パッケージ 139
java.rmi パッケージ 31, 125, 126, 130, 131, 133, 142, 151, 152, 153, 155, 171, 180, 207, 212, 214, 215, 216, 217
java.sql パッケージ 220, 222, 226
java.text.MessageFormat クラス 281, 282
java.util パッケージ 131, 171, 216
java.util.Locale クラス 281, 282
java.util.ResourceBundle クラス 281, 282
java.util.TimeZone クラス 281, 282
JDBC 7, 219, 226
jetace ツール 44, 54, 311
JNDI 8, 147, 173, 182
　デプロイメント・ディスクリプター属性 58
　ホーム・インターフェースの検索 174
　INITIAL_CONTEXT_FACTORY プロパティー 173
　PROVIDER_URL プロパティー 173
JSP 15, 200
JSQL 226
JTA 7, 182

## L

LDAP 8
listMappings メソッド 266
LocalizableConfiguration クラス 284
LocalizableException クラス 289
LocalizableText インターフェース 296
　ユーザー定義のインプリメンテーション 296
　format メソッド 296
LocalizableTextDateTimeArgument クラス 294
LocalizableTextEJBDeploy ツール 300
　構文 301
　使用 301
　前提条件 300
　例 302
LocalizableTextFormatter クラス 282, 284
　値の設定 287, 288
　アプリケーション固有の引き数 286
　コンストラクター 287
　時間帯 289
　フォールバック情報 285, 290
　メッセージのキャッシング 285
　ロケール 289
　clearLocalizableTextFormatter メソッド 287
　format メソッド 282, 284, 289, 296
　setApplicationName メソッド 285, 287
　setArguments メソッド 285, 287
　setCacheSetting メソッド 287
　setFallBackLocale メソッド 287
　setFallBackString メソッド 287
　setFallBackTimeZone メソッド 287
　setPatternKey メソッド 285, 287
　setResourceBundleName メソッド 285, 287
LocalizableTextLT インターフェース 296
　ユーザー定義のインプリメンテーション 296
　format メソッド 296
LocalizableTextLTZ インターフェース 296
　ユーザー定義のインプリメンテーション 296
　format メソッド 296
　LocalizableTextDateTimeArgument クラス 296
LocalizableTextZ インターフェース 296
LocalizableTextZ インターフェース 296 *(続き)*
　ユーザー定義のインプリメンテーション 296
　format メソッド 296
LocalTarget クラス 266
lookup メソッド 147

## M

MQSeries 12
　Enterprise Bean の開発 109

## N

NoSuchObjectException クラス 31, 180

## O

ObjectNotFoundException 123
ObjectNotFoundException クラス 208
Oracle データベース 12, 86, 93

## P

PAOToEJB ツール 108
performExecute メソッド 250, 257, 264, 275
PreparedStatement インターフェース 226
printStackTrace メソッド 235, 236
printSuperStackTrace メソッド 236
PROVIDER_URL プロパティー 145, 173

## R

registerCommand メソッド 266, 269
RemoteException クラス 125, 126, 130, 131, 133, 142, 151, 152, 153, 155, 207, 212, 214, 215, 216, 217
remove メソッド 154
　エンティティー bean 32
　セッション bean 31, 149

remove メソッド　154 *(続き)*
　EJB クライアントでの呼び出し
　　181
RemoveException　123
RemoveException クラス　31, 126,
　212
ResultSet インターフェース　226
RMI　13
　有効なパラメーター　155
RuntimeException クラス　139


## S

Serializable インターフェース　155
SessionBean インターフェース　138,
　149
SessionSynchronization インターフェース　139
setApplicationName メソッド　285,
　287
setArguments メソッド　285, 287
setCacheSetting メソッド　287
setCommandTarget メソッド　250,
　266
setCommandTargetName メソッド
　250, 266
setDefaultTargetName メソッド　266,
　268
setEntityContext メソッド　31
　エンティティー bean (BMP)　212
　エンティティー bean
　　(CMP)　126, 128
setFallBackLocale メソッド　287
setFallBackString メソッド　287
setFallBackTimeZone メソッド　287
setHasOutputProperties メソッド　250
setOutputProperties メソッド　250,
　253, 257
setPatternKey メソッド　285, 287
setResourceBundleName メソッド
　285, 287
setSessionContext メソッド　29, 149,
　150
setTargetPolicy メソッド　270
SQL Server　12


## T

TargetableCommand インターフェース　246, 247, 248, 250, 253, 257,
　266, 274
TargetableCommandImpl クラス　247,
　253, 254, 270
TargetPolicy インターフェース　249,
　266, 270
TargetPolicyDefault クラス　249, 266
TransactionRequiredException クラス
　　161


## U

UnauthorizedAccessException クラス
　250
unregisterCommand メソッド　266,
　269
unsetEntityContext メソッド　33
　エンティティー bean (BMP)　212
　エンティティー bean
　　(CMP)　126, 128
UnsetInputPropertiesException クラス
　　250
UserTransaction インターフェース
　12, 182, 229


## V

VisualAge for Java　35


## W

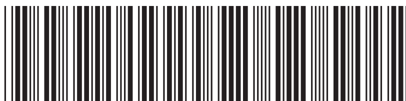
Web サーバー
　サーブレットおよび JSP　15, 189
WebSphere Application Server
　管理　15
　コード例　307
WebSphere 管理コンソール　15, 36,
　41
WebSphere プログラミング・モデル
　拡張機能　233
　コマンド・パッケージ　245
　分散例外パッケージ　233
　ローカライズ可能テキスト・パッケージ　277


## X

XML　57, 311

Printed in Japan

SD88-7343-02



日本アイ·ビー·エム株式会社
〒106-8711　東京都港区六本木3-2-12