WebSphere™ Application Server

Writing Enterprise Beans in WebSphere

Version 4.0

SC09-4431-04

Note

Before using this information and the product it supports, be sure to read the general information under "Notices" on page 201.

Fifth Edition (June 2001)

This edition replaces SC09-4431-03.

Order publications through your IBM representative or through the IBM branch office serving your locality.

Contents

Figures . . . . . **v**

Tables . . . . . **ix**

About this book . . . . . **xi**
Who should read this book xi
Document organization xi
Related information xii
Conventions used in this book xii
How to send your comments xiv

Chapter 1. An architectural overview of the EJB programming environment **1**
Components of the EJB environment 1
The EJB server 2
The security service 3
The workload management service 5
The persistence service 5
The naming service 6
The transaction service 6
The data source 9
The EJB clients 9
The Web server 11
The administration interface 11

Chapter 2. An introduction to enterprise beans **13**
Bean basics 13
Entity beans 13
Session beans 15
Creating an EJB module 17
The EJB module 17
The deployment descriptor 17
Deploying an EJB module 19
Developing EJB applications 20
An example: enterprise beans for a bank 21
Using the banking beans to develop EJB banking applications 22
Life cycles of enterprise bean instances . . . 23
Session bean life cycle 23
Entity bean life cycle 25

Chapter 3. Tools for developing and deploying enterprise beans **27**
Using VisualAge for Java 27
Developing and deploying enterprise beans 28
Installing and configuring the software for the EJB server 28
Setting the CLASSPATH environment variable in the EJB server environment . . 29
Creating the components of an enterprise bean 29
Creating finder logic in the EJB server . . 30
Creating an EJB module 30
Creating a database for use by entity beans 31

Chapter 4. Developing enterprise beans . . **33**
Developing entity beans with CMP 33
Writing the enterprise bean class (entity with CMP) 34
Writing the home interface (entity with CMP) 43
Writing the remote interface (entity with CMP) 46
Writing the primary key class (entity with CMP) 47
Interacting with databases 50
Developing session beans 50
Writing the enterprise bean class (session) 51
Writing the home interface (session) . . . 62
Writing the remote interface (session) . . 63
Implementing interfaces common to multiple types of enterprise beans 64
Methods inherited from javax.ejb.EJBObject 65
The javax.ejb.EJBHome interface 65
The java.io.Serializable and java.rmi.Remote interfaces 66
Using threads and reentrancy in enterprise beans 66
Creating an EJB module for enterprise beans 66
Making bean components part of a Java package 67
Creating an EJB module and deployment descriptor 67

Chapter 5. Enabling transactions and security in enterprise beans **69**
Setting transactional attributes in the deployment descriptor 69

Setting the transaction attribute 70
Setting the transaction isolation level attribute 72
Setting the security attribute in the deployment descriptor 74

Chapter 6. Developing EJB clients 77
Importing required Java packages 78
Creating and getting a reference to a bean's EJB object 79
Locating and creating an EJB home object 80
Creating an EJB object 83
Handling an invalid EJB object for a session bean 84
Removing a bean's EJB object 86
Managing transactions in an EJB client . . . 86

Chapter 7. Developing servlets that use enterprise beans 91
An overview of standard servlet methods . . 91
Writing an HTML page that embeds a servlet 91
Developing the servlet 93
The servlet's instance variables 94
The servlet's init method 95
The servlet's doGet method 97
Creating an enterprise bean 98
Determining the content of the user response 99
Sending the user response 100
Threading issues 101

Chapter 8. More-advanced programming concepts for enterprise beans 103
Developing entity beans with BMP 103
Writing the enterprise bean class (entity with BMP) 104
Writing the home interface (entity with BMP) 114
Writing the remote interface (entity with BMP) 117
Writing or selecting the primary key class (entity with BMP) 118
Using a database with a BMP entity bean 119
Managing database connections in the EJB server environment 120
Manipulating data in a database 123
Using bean-managed transactions 124

Chapter 9. WebSphere Programming Model Extensions 129
The distributed-exception package 129
Overview 130
Extending the DistributedException class 133
Implementing the DistributedExceptionEnabled interface . . 134
Using distributed exceptions 139
The command package 140
Overview 141
Writing command interfaces 144
Implementing command interfaces . . . 147
Using a command 155
Using the WebSphere EJBCommandTarget bean as a command target 157
Writing a command target (server) . . . 159
Targets and target policies 161
Writing a command target (client-side adapter) 166
The localizable-text package 170
Overview 170
Writing a localizable application 178
Using optional arguments 182
Deploying the formatter enterprise bean 191

Appendix A. Changes for version 1.1 of the EJB specification 193
New and updated features 193
Migrating from version 1.0 to version 1.1 193

Appendix B. Example code provided with WebSphere Application Server 197
Information about the examples described in the documentation 197
Information about other examples 198

Appendix C. Extensions to the EJB Specification 199
Access beans 199
Associations between enterprise beans . . . 200
Inheritance in enterprise beans 200

Notices 201
Trademarks and service marks 203

Index 207

Figures

1. The components of the EJB environment 1
2. Example of a distributed transaction 7
3. The components of an entity bean 14
4. The components of a session bean 16
5. The major components of a deployed entity bean 20
6. Conceptual view of EJB applications 21
7. Code example: AccountBeanFinderHelper interface for the EJB server 30
8. Code example: The AccountBean class 35
9. Code example: The variables of the AccountBean class 36
10. Code example: The business methods of the AccountBean class 38
11. Code example: The ejbCreate and ejbPostCreate methods of the AccountBean class 41
12. Code example: Implementing the EntityBean interface in the AccountBean class 43
13. Code example: The AccountHome home interface 44
14. Code example: The findLargeAccounts method 45
15. Code example: The Account remote interface 47
16. Code example: The ItemKey primary key class 49
17. Code example: The TransferBean class 53
18. Code example: The business methods of the TransferBean class 55
19. Code example: Creating the InitialContext object in the ejbCreate method of the TransferBean class . . . 58
20. Code example: The getProviderURL method 59
21. Code example: Creating the AccountHome object in the ejbCreate method of the TransferBean class . . . 60
22. Code example: Looking up an enterprise bean's environment naming context . . 61
23. Code example: Implementing the SessionBean interface in the TransferBean class 62
24. Code example: The TransferHome home interface 63
25. Code example: The Transfer remote interface 64
26. Code example: The import statements for the Java application TransferApplication. 79
27. Code example: Creating the InitialContext object 82
28. Code example: Creating the EJBHome object 83
29. Code example: Creating the EJB object 84
30. Code example: Refreshing the EJB object reference for a session bean 85
31. Code example: Removing a session EJB object 86
32. Code example: Managing transactions in an EJB client 89
33. Code example: Content of the create.html file used to access the CreateAccount servlet 92
34. The initial form and output of the CreateAccount servlet 93
35. Code example: The CreateAccount class 94
36. Code example: The instance variables of the CreateAccount class 95
37. Code example: The init method of the CreateAccount servlet 96
38. Code example: The doGet method of the CreateAccount servlet 98
39. Code example: Creating an enterprise bean in the doGet method 99
40. Code example: Determining a user response in the doGet method. 100
41. Code example: Responding to the user in the doGet method 101
42. Code example: The AccountBMBean class 105
43. Code example: The instance variables of the AccountBMBean class 106
44. Code example: The ejbCreate methods of the AccountBMBean class 109
45. Code example: The ejbFindByPrimaryKey method of the AccountBMBean class 111

46. Code example: The ejbFindLargeAccounts method of the AccountBMBean class 112
47. Code example: The AccountBMHome home interface 115
48. Code example: The AccountBM remote interface 118
49. Code example: Getting an EJB object reference to a data source bean instance in the setEntityContext method (rewritten to use DataSource) 121
50. Code example: The checkConnection and makeConnection methods of the AccountBMBean class (rewritten to use DataSource) 122
51. Code example: The dropConnection method of the AccountBMBean class (rewritten to use DataSource) 122
52. Code example: Constructing and executing an SQL update call in an ejbCreate method 123
53. Code example: Manipulating a ResultSet object in the ejbLoad method . 124
54. Code example: Getting an object that encapsulates a transaction context . . 126
55. Code example: Constructors for the DistributedException class 131
56. Code example: Constructors in an exception class that extends the DistributedException class 134
57. Code example: The structure of an exception class that implements the DistributedExceptionEnabled interface . 135
58. Code example: Constructors for an exception class that implements the DistributedExceptionEnabled interface . 136
59. Code example: Implementations of the methods in the DistributedExceptionEnabled interface . 138
60. Code example: Testing for an exception that implements the DistributedExceptionEnabled interface . 139
61. Code example: Adding an exception to a chain 139
62. Code example: Extracting exceptions from a chain. 140
63. Code example: The structure of an interface for a targetable command . . 142
64. Code example: The structure of an interface for a targetable, compensable command 142
65. Code example: The structure of an implementation class for a command interface 143
66. Code example: The structure of a command-target entity bean 144
67. Code example: The ModifyCheckingAccountCmd interface . 146
68. Code example: The structure of the ModifyCheckingAccountCmdImpl class 147
69. Code example: The variables in the ModifyCheckingAccountCmdImpl class 148
70. Code example: Constructors in the ModifyCheckingAccountCmdImpl class 149
71. Code example: Command-specific methods in the ModifyCheckingAccountCmdImpl class 150
72. Code example: Methods from the Command interface in the ModifyCheckingAccountCmdImpl class 151
73. Code example: Methods from the TargetableCommand interface in the ModifyCheckingAccountCmdImpl class 152
74. Code example: Method from the CompensableCommand interface in the ModifyCheckingAccountCmdImpl class 153
75. Code example: Variables and constructor in the ModifyCheckingAccountCompensatorCmd class 154
76. Code example: Methods in ModifyCheckingAccountCompensatorCmd class 155
77. Code example: Using the ModifyCheckingAccountCmd command 156
78. Code example: Using the ModifyCheckingAccountCompensator command 157
79. Code example: Using an EJBCommandTarget bean 158
80. Code example: The remote interface for the CheckingAccount entity bean, also a command target 160
81. Code example: The bean class for the CheckingAccount entity bean, also a command target 161

82. Code example: The TargetPolicyDefault class 162
83. Code example: Identifying a target with CommandTarget 163
84. Code example: Identifying a target with CommandTargetName 164
85. Code example: Mapping a command to a target in an external application . . 164
86. Code example: Creating a custom target policy 165
87. Code example: Using a custom target policy 166
88. Code example: The structure of a client-side adapter for a target 167
89. Code example: Instantiating the client-side adapter 167
90. Code example: A client-side implementation of the executeCommand method 168
91. Code example: Running the command in the servlet 169
92. Three elements in an English message catalog 173
93. Three elements in a German message catalog 173
94. Code example: Creating a LocalizableTextFormatter object and setting values on it 180
95. Code example: Setting the locale programmatically 181
96. Code example: Formatting a LocalizableTextFormatter object . . . 182
97. A message-catalog entry with a variable substring 183
98. Code example: Formatting a message with a variable substring 184
99. A message-catalog entry with two variable substrings 184
100. Code example: Formatting a message with a localizable variable substring . . 185
101. Code example: The structure of the LocalizableTextDateTimeArgument class 188
102. Code example: The methods in the LocalizableTextDateTimeArgument class 188
103. Code example: The format method in the LocalizableTextDateTimeArgument class 190

Tables

1. Conventions used in this book xii
2. Effect of the enterprise bean's transaction attribute on the transaction context 72
3. Examples available with the EJB server 198

About this book

This document focuses on the development of enterprise beans written to the Sun Microsystems Enterprise JavaBeans™ specification in the WebSphere™ Application Server programming environment. It also discusses development of EJB clients that can access enterprise beans.

Who should read this book

This document is written for developers and system architects who want an introduction to programming enterprise beans and EJB clients in WebSphere Application Server. It is assumed that programmers are familiar with the concepts of object-oriented programming, distributed programming, and Web-based programming. Knowledge of the Sun Microsystems Java™ programming language is also assumed.

Document organization

This document is organized as follows:

- "Chapter 1. An architectural overview of the EJB programming environment" on page 1 provides a high-level introduction to the EJB server environment in WebSphere Application Server.
- "Chapter 2. An introduction to enterprise beans" on page 13 explains the main concepts associated with enterprise beans.
- "Chapter 3. Tools for developing and deploying enterprise beans" on page 27 explains how to set up and use the tools used in developing and deploying enterprise beans.
- "Chapter 4. Developing enterprise beans" on page 33 explains how to develop entity beans with container-managed persistence (CMP) and session beans. It also provides information on how to package enterprise beans for later deployment.
- "Chapter 5. Enabling transactions and security in enterprise beans" on page 69 explains how to enable transactions in enterprise beans by using the appropriate deployment descriptor attributes.
- "Chapter 6. Developing EJB clients" on page 77 explains the basic code required by an EJB client to use an enterprise bean. This chapter covers generic issues relevant to enterprise beans, Java applications, and Java servlets that use enterprise beans.
- "Chapter 7. Developing servlets that use enterprise beans" on page 91 discusses the basic code required in a servlet that accesses an enterprise bean.

- "Chapter 8. More-advanced programming concepts for enterprise beans" on page 103 explains how to develop a simple entity bean with bean-managed persistence and discusses the basic code required of an enterprise bean that manages its own transactions.
- "Appendix A. Changes for version 1.1 of the EJB specification" on page 193 describes features that are new or have changed in version 1.1 of the EJB specification and discusses migration issues for enterprise beans written to version 1.0 of the EJB specification.
- "Appendix B. Example code provided with WebSphere Application Server" on page 197 describes the major example used throughout this book and the additional examples that are delivered with the various editions of WebSphere Application Server.
- "Appendix C. Extensions to the EJB Specification" on page 199 describes the extensions to the EJB Specification that are specific to WebSphere Application Server. Use of these extensions is supported in VisualAge for Java only.

Related information

For further information on the topics discussed in this manual, see the following documents:

- *Getting Started with WebSphere Application Server*
- *Building Business Solutions with WebSphere*

Conventions used in this book

This document uses the following typographical and keying conventions.

Table 1. Conventions used in this book

Convention	Meaning
Bold	Indicates command names. When referring to graphical user interfaces (GUIs), bold also indicates menus, menu items, labels, and buttons.
`Monospace`	Indicates text you must enter at a command prompt and values you must use literally, such as commands, functions, and resource definition attributes and their values. Monospace also indicates screen text and code examples.
Italics	Indicates variable values you must provide (for example, you supply the name of a file for *fileName*). Italics also indicates emphasis and the titles of books.
Ctrl-*x*	Where *x* is the name of a key, indicates a control-character sequence. For example, Ctrl-c means hold down the Ctrl key while you press the c key.
Return	Refers to the key labeled with the word Return, the word Enter, or the left arrow.

Table 1. Conventions used in this book (continued)

Convention	Meaning
%	Represents the UNIX command-shell prompt for a command that does not require **root** privileges.
#	Represents the UNIX command-shell prompt for a command that requires **root** privileges.
C:\>	Represents the Windows NT® command prompt.
>	When used to describe a menu, shows a series of menu selections. For example, "Click **File > New**" means "From the **File** menu, click the **New** command."
	When used to describe a tree view, shows a series of folder or object expansions. For example, "**Expand Management Zones > Sample Cell and Work Group Zone > Configuration**" means: 1. Expand the Management Zones folder 2. Expand the management zone named Sample Cell and Work Group Zone 3. Expand the Configurations folder **Note:** An object in a view can be expanded when there is a plus sign (+) beside that object. After an object is expanded, the plus sign is replaced by a minus sign (-).
+	Expands a tree structure to show more objects. To expand, click the plus sign (+) beside any object.
-	Collapses a branch of a tree structure to remove from view the objects contained in that branch. To collapse the branch of a tree structure, click the minus sign (-) beside the object at the head of the branch.
Entering commands	When instructed to "enter" or "issue" a command, type the command and then press Return. For example, the instruction "Enter the **ls** command" means type **ls** at a command prompt and then press Return.
[ ]	Enclose optional items in syntax descriptions.
{ }	Enclose lists from which you must choose an item in syntax descriptions.
|	Separates items in a list of choices enclosed in braces ({ }) in syntax descriptions.
...	Ellipses in syntax descriptions indicate that you can repeat the preceding item one or more times. Ellipses in examples indicate that information was omitted from the example for the sake of brevity.
IN	In function descriptions, indicates parameters whose values are used to pass data to the function. These parameters are not used to return modified data to the calling routine. (Do *not* include the IN declaration in your code.)
OUT	In function descriptions, indicates parameters whose values are used to return modified data to the calling routine. These parameters are not used to pass data to the function. (Do *not* include the OUT declaration in your code.)

Table 1. Conventions used in this book (continued)

Convention	Meaning
INOUT	In function descriptions, indicates parameters whose values are passed to the function, modified by the function, and returned to the calling routine. These parameters serve as both IN and OUT parameters. (Do *not* include the INOUT declaration in your code.)
$CICS	Indicates the full pathname where the CICS product is installed; for example, C:\opt\cics on Windows NT or /opt/cics on Solaris. If the environment variable named CICS is set to the product pathname, you can use the examples exactly as shown; otherwise, you must replace all instances of $CICS with the CICS product pathname.
CICS on Open Systems	Refers collectively to the CICS products for all supported UNIX platforms.
TXSeries CICS	Refers collectively to the CICS for AIX, CICS for Solaris, and CICS for Windows NT products.
CICS	Refers generically to the CICS on Open Systems and CICS for Windows NT products. References to a specific version of a CICS on Open Systems product are used to highlight differences between CICS on Open Systems products. Other CICS products in the CICS Family are distinguished by their operating system (for example, CICS for OS/2 or IBM mainframe-based CICS for the ESA, MVS, and VSE platforms).

How to send your comments

Your feedback is important in helping to provide the most accurate and highest quality information. If you have any comments about this book, send your comments by e-mail to wasdoc@us.ibm.com. Be sure to include the name of the book, the document number of the book, the edition and version of WebSphere Application Server, and, if applicable, the specific location of the information you are commenting on (for example, a page number or table number).

Chapter 1. An architectural overview of the EJB programming environment

The World Wide Web (the Web) has transformed the way in which businesses work with their customers. At first, it was good enough just to have a Web home page. Then, businesses began to deploy active Web sites that allowed customers to order products and services. Today, businesses not only need to use the Web in all of these ways, they need to integrate their Web-based systems with their other business systems. The IBM® WebSphere Application Server, and specifically the support for enterprise beans, provides the model and the tools to accomplish this integration.

Components of the EJB environment

IBM's implementation of the Sun Microsystems Enterprise JavaBeans (EJB) Specification enables users of the WebSphere Application Server to integrate their Web-based systems with their other business systems. A major part of this implementation is the WebSphere EJB server and its associated components, which are illustrated in Figure 1.

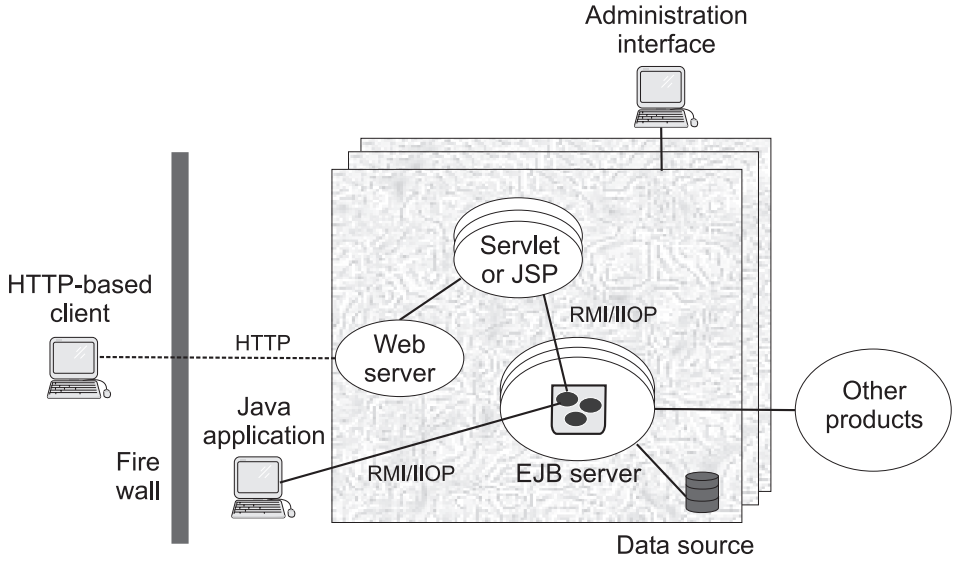

Figure 1. The components of the EJB environment

The WebSphere EJB server environment contains the following components, which are discussed in more detail in the specified sections:

- *EJB server*—A WebSphere EJB server contains and runs one or more *enterprise beans*, which encapsulate the business logic and data used and shared by EJB clients. The enterprise beans installed in an EJB server do not communicate directly with the server; instead, an *EJB container* provides an interface between the enterprise beans and the EJB server, providing many low-level services such as threading, support for transactions, and management of data storage and retrieval. For more information, see "The EJB server".
- *Data source*—There are two types of enterprise beans: session beans, which encapsulate short-lived, client-specific tasks and objects, and entity beans, which encapsulate permanent or *persistent* data. The EJB server stores and retrieves this persistent data in a data source, which can be a database, another application, or even a file. For more information, see "The data source" on page 9.
- *EJB clients*—There are two general types of EJB clients:
  - *HTTP-based clients* that interact with the EJB server by using either Java servlets or JavaServer Pages™ (JSP) by way of the Hypertext Transfer Protocol (HTTP).
  - *Java applications* that interact directly with the EJB server by using Java remote method invocation over the Internet Inter-ORB Protocol (RMI/IIOP).

  For more information, see "The EJB clients" on page 9.
- The *administration interface*—The administrative interface allows you to manage the EJB server environment. For more information, see "The administration interface" on page 11.

The EJB server

The EJB server is the application server tier of WebSphere Application Server's three-tier architecture. The EJB server has three components: the EJB server runtime, the EJB containers, and the enterprise beans. EJB containers insulate the enterprise beans from the underlying EJB server and provide a standard application programming interface (API) between the beans and the container. The EJB Specification defines this API.

Together, the EJB server and container components provide or give access to the following services for the enterprise beans that are deployed into it:

- A tool that deploys enterprise beans. When a bean is deployed, the deployment tool creates several classes that implement the interfaces that make up the predeployed bean. In addition, the deployment tool generates Java ORB, stub, and skeleton classes that enable remote method invocation. For entity beans, the tool also generates persistor and finder classes to handle interaction between the bean and the data source that stores the

bean's persistent data. Before an enterprise bean can be deployed, the developer must create an *EJB module* and associated *deployment descriptor*. The deployment descriptor provides information about each enterprise bean in the module and instructions for the container on how to handle the beans. For more information on deployment, see "Deploying an EJB module" on page 19.

- A security service that handles authentication and authorization for principals that need to access resources in an EJB server environment. For more information, see "The security service".
- A workload management service that ensures that resources are used efficiently. For more information, see "The workload management service" on page 5.
- A persistence service that handles interaction between an entity bean and its data source to ensure that persistent data is properly managed. For more information, see "The persistence service" on page 5.
- A naming service that exports a bean's name, as defined in the deployment descriptor, into the name space. The EJB server uses the Java Naming and Directory Interface™ (JNDI) to implement a naming service. For more information, see "The naming service" on page 6.
- A transaction service that implements the transactional attributes in a bean's deployment descriptor. For more information, see "The transaction service" on page 6.

The security service

When enterprise computing was handled solely by a few powerful mainframes located at a centralized site, ensuring that only authorized users obtained access to computing services and information was a fairly straightforward task. In distributed computing systems where users, application servers, and resource managers can be spread out across the world, securing computing resources has become a much more complicated task. Nevertheless, the underlying issues are basically the same.

Authentication and authorization

A good security service provides two main functions: authentication and authorization.

Authentication takes place when a *principal* (a user or a computer process) initially attempts to gain access to a computing resource. At that point, the security service challenges the principal to prove that the principal is who it claims to be. Human users typically prove who they are by entering a user ID and password; a process normally presents an encrypted key. If the password or key is valid, the security service gives the user a *token* or *ticket* that identifies the principal and indicates that the principal has been authenticated.

After a principal is authenticated, it can then attempt to use any of the resources within the boundaries of the computing system protected by the security service; however, a principal can use a particular computing resource only if it has been authorized to do so. *Authorization* takes place when an authenticated principal requests the use of a resource and the security service determines if the user has been granted permission to use that resource. Typically, authorization is handled by associating access control lists (ACLs) with resources that define which principal (or groups of principals) are authorized to use the resource. If the principal is authorized, it gains access to the resource.

In a distributed computing environment, principals and resources must be mutually suspicious of each other's identity until both have proven that they are who they say they are. This is necessary because principals can attempt to falsify an identity to get access to a resource, and a resource can be a trojan horse, attempting to get valuable information from the principal. To solve this problem, the security service contains a security server that acts as a *trusted third party*, authenticating principals and resources so that these entities can prove their identities to each other. This security protocol is known as *mutual authentication*.

Using the security server

The security service does *not* use the *access control* and *run-as identity* security attributes defined in the deployment descriptor. However, it does use the *run-as mode* attribute as the basis for mapping a user identity to a user security context. For more information on this attribute, see "The deployment descriptor" on page 17.

The main component of the security service is an EJB server that contains security enterprise beans. When system administrators administer the security service, they manipulate the security beans in the security EJB server.

Once an EJB client is authenticated, it can attempt to invoke methods on the enterprise beans that it manipulates. A method is successfully invoked if the principal associated with the method invocation has the required permissions to invoke the method. These permissions can be set at the application level (an administrator-defined set of Web and object resources) and at the method group level (an administrator-defined set of Java interface/method pairs). An application can contain multiple method groups.

In general, the principal under which a method is invoked is associated with that invocation across multiple Web servers and EJB servers (this association is known as *delegation*). Delegating the method invocations in this way ensures that the user of an EJB client needs to authenticate only once. HTTP cookies are used to propagate a user's authentication information across multiple Web

servers. These cookies have a lifetime equal to the life of the browser session, and a logout method is provided to destroy these cookies when the user is finished.

For information on administering security, see the WebSphere InfoCenter and the online help available with the WebSphere Administrative Console.

The workload management service

The workload management service improves the scalability of the EJB server environment by grouping multiple EJB servers into *server groups*. Clients then access these server groups as if they are a single EJB server, and the workload management service ensures that the workload is evenly distributed across the EJB servers in the server groups. An EJB server can belong to only one server group.

The creation of server groups is an administrative task that is handled from within the WebSphere Administrative Console. For more information on workload management, consult the WebSphere InfoCenter and the online help for the appropriate administrative interface.

The persistence service

There are two types of enterprise beans: session beans and entity beans. Session beans encapsulate temporary data associated with a particular client. Entity beans encapsulate permanent data that is stored in a data source. For more information, see "Chapter 2. An introduction to enterprise beans" on page 13.

The persistence service ensures that the data associated with entity beans is properly synchronized with their corresponding data in the data source. To accomplish this task, the persistence service works with the transaction service to insert, update, extract, and remove data from the data source at the appropriate times.

There are two types of entity beans: those with container-managed persistence (CMP) and those with bean-managed persistence (BMP). In entity beans with CMP, the persistence service handles nearly all of the tasks required to manage persistent data. In entity beans with BMP, the bean itself handles most of the tasks required to manage persistent data.

The persistence service uses the following components to accomplish its task:

- The Java Database Connectivity (JDBC™) API, which gives entity beans a common interface to relational databases.
- Java transaction support, which is discussed in "Using transactions in the EJB server environment" on page 8. The EJB server ensures that persistent data is always handled within the appropriate transactional context.

The naming service

In an object-oriented distributed computing environment, clients must have a mechanism to locate and identify objects so that the clients, objects, and resources appear to be on the same machine. A naming service provides this mechanism. In the EJB server environment, JNDI is used to mask the actual naming service and provide a common interface to the naming service.

JNDI provides naming and directory functionality to Java applications, but the API is independent of any specific implementation of a naming and directory service. This implementation independence ensures that different naming and directory services can be used by accessing them by way of the JNDI API. Therefore, Java applications can use many existing naming and directory services such as the Lightweight Directory Access Protocol (LDAP), the Domain Name Service (DNS), or the DCE Cell Directory Service (CDS).

JNDI was designed for Java applications by using Java's object model. Using JNDI, Java applications can store and retrieve named objects of any Java object type. JNDI also provides methods for executing standard directory operations, such as associating attributes with objects and searching for objects by using their attributes.

In the EJB server environment, the deployment descriptor is used to specify the JNDI name for an enterprise bean. When an EJB server is started, it registers these names with JNDI.

The transaction service

A *transaction* is a set of operations that transforms data from one consistent state to another. This set of operations is an indivisible unit of work, and in some contexts, a transaction is referred to as a *logical unit of work* (LUW). A transaction is a tool for distributed systems programming that simplifies failure scenarios.

Transactions provide the *ACID properties*:

- *Atomicity*: A transaction's changes are atomic: either all operations that are part of the transaction happen or none happen.
- *Consistency*: A transaction moves data between consistent states.
- *Isolation*: Even though transactions can run (or be executed) concurrently, no transaction sees another's work in progress. The transactions appear to run serially.
- *Durability*: After a transaction completes successfully, its changes survive subsequent failures.

As an example, consider a transaction that transfers money from one account to another. Such a transfer involves money being deducted from one account and deposited in the other. Withdrawing the money from one account and

depositing it in the other account are two parts of an *atomic* transaction: if both cannot be completed, neither must happen. If multiple requests are processed against an account at the same time, they must be *isolated* so that only a single transaction can affect the account at one time. If the bank's central computer fails just after the transfer, the correct balance must still be shown when the system becomes available again: the change must be *durable*. Note that *consistency* is a function of the application; if money is to be transferred from one account to another, the application must subtract the same amount of money from one account that it adds to the other account.

Transactions can be completed in one of two ways: they can commit or roll back. A successful transaction is said to *commit*. An unsuccessful transaction is said to *roll back*. Any data modifications made by a rolled back transaction must be completely undone. In the money-transfer example, if money is withdrawn from one account but a failure prevents the money from being deposited in the other account, any changes made to the first account must be completely undone. The next time any source queries the account balance, the correct balance must be shown.

Distributed transactions and the two-phase commit process

A *distributed transaction* is one that runs in multiple processes, often on several machines. Each process participates in the transaction. This is illustrated in Figure 2, where each oval indicates work being done on a different machine, and each arrow indicates a remote method invocation (RMI).

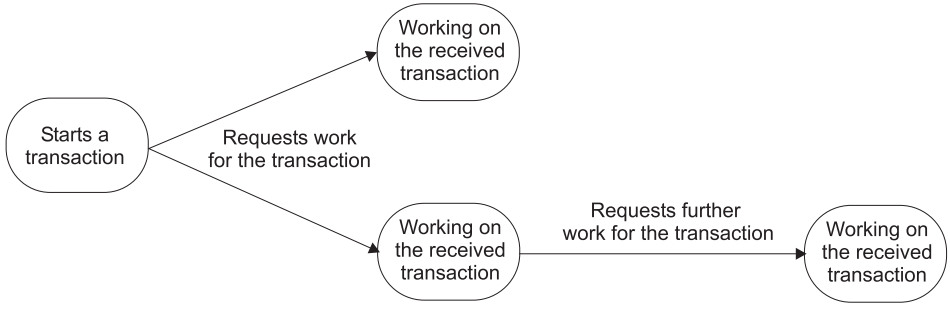

Figure 2. Example of a distributed transaction

Distributed transactions, like local transactions, must adhere to the ACID properties. However, maintaining these properties is greatly complicated for distributed transactions because a failure can occur in any process, and in the event of such a failure, each process must undo any work already done on behalf of the transaction.

A distributed transaction processing system maintains the ACID properties in distributed transactions by using two features:

- *Recoverable processes*: Recoverable processes are those that can restore earlier states if a failure occurs.
- *A commit protocol*: A commit protocol enables multiple processes to coordinate the committing or rolling back (aborting) of a transaction. The most common commit protocol, and the one used by the EJB server, is the two-phase commit protocol.

Transaction state information must be stored by all recoverable processes. However, only processes that manage application data (such as resource managers) must store descriptions of changes to data. Not all processes involved in a distributed transaction need to be recoverable. In general, clients are not recoverable because they do not interact directly with a resource manager. Processes that are not recoverable are referred to as *ephemeral* processes.

The *two-phase commit protocol*, as the name implies, involves two phases: a prepare phase and a resolution phase. In each transaction, one process acts as the coordinator. The *coordinator* oversees the activities of the other participants in the transaction to ensure a consistent outcome.

In the *prepare phase*, the coordinator sends a message to each process in the transaction, asking each process to prepare to commit. When a process prepares, it guarantees that it can commit the transaction and makes a permanent record of its work. After guaranteeing that it can commit, it can no longer unilaterally decide to roll back the transaction. If a process cannot prepare (that is, if it cannot guarantee that it can commit the transaction), it must roll back the transaction.

In the *resolution phase*, the coordinator tallies the responses. If all participants are prepared to commit, the transaction commits; otherwise, the transaction is rolled back. In either case, the coordinator informs all participants of the result. In the case of a commit, the participants acknowledge that they have committed.

Using transactions in the EJB server environment

The enterprise bean transaction model corresponds in most respects to the OMG OTS version 1.1. An enterprise bean instance that is transaction enabled corresponds to an object of the OTS TransactionalObject interface. However, the enterprise bean transaction model does not support transaction nesting.

In the EJB server environment, transactions are handled by three main components of the transaction service:

- A transaction manager interface that enables the EJB server to control transaction boundaries within its enterprise beans based on the transactional attributes specified for the beans.

- An interface (UserTransaction) that allows an enterprise bean or an EJB client to manage transactions. The container makes this interface available to enterprise beans and EJB clients by way of the name service.
- Coordination by way of the X/Open XA interface that enables a transactional resource manager (such as a database) to participate in a transaction controlled by an external transaction manager.

For most purposes, the enterprise bean developers can delegate the tasks involved in managing a transaction to the container. The developer performs this delegation by setting the deployment descriptor attributes for transactions. These attributes and their values are described in "Setting transactional attributes in the deployment descriptor" on page 69.

In other cases, the enterprise bean developer will want or need to manage the transactions at the bean level or involve the EJB client in the management of transactions. For more information on this approach, see "Using bean-managed transactions" on page 124.

The data source

Entity beans contain persistent data that must be permanently stored in a recoverable data source. Although the EJB Specification often refers to databases as the place to store persistent data associated with an entity bean, it leaves open the possibility of using other data sources, including operating system files and other applications.

If you want to let the container handle the interaction between an entity bean and a data source, you must use the data sources supported by that container.

If you write the additional code required to handle the interaction between a BMP entity bean and the data source, you can use any data source that meets your needs and is compatible with the persistence service. For more information, see "Developing entity beans with BMP" on page 103.

The EJB clients

An EJB client can take one of the following forms: it can be a Java application, a Java servlet, a Java applet-servlet combination, or a JSP file. The EJB client code required to access and manipulate enterprise beans is very similar across the different Java EJB clients. EJB client developers must consider the following issues:

- *Naming and communications*—A Java EJB client must use either HTTP or RMI to communicate with enterprise beans. Fortunately, there is very little difference in the coding required to enable communications between the

EJB client and the enterprise bean, because JNDI masks the interaction between the EJB client and the name service.

  - Java applications communicate with enterprise beans by using RMI/IIOP.
  - Java servlets and JSP files communicate with enterprise beans by using HTTP. To use servlets with an EJB server, a Web server must be installed and configured on a machine in the EJB server environment. For more information, see "The Web server" on page 11.

- *Threading*—Java clients can be either single-threaded or multithreaded depending on the tasks that the client needs to perform. Each client thread that uses a service provided by a session bean must create or find a separate instance of that bean and maintain a reference to that bean until the thread completes; multiple client threads can access the same entity bean.
- *Security* – EJB clients that access an EJB server over HTTP (for example, servlets and JSP files) encounter the following two layers of security:
  1. Universal Resource Locator (URL) security enforced by the WebSphere Application Server Security Plug-in attached to the Web server in collaboration with the security service.
  2. Enterprise bean security enforced at the server working with the security service.

  When the user of an HTTP-based EJB client attempts to access an enterprise bean, the Web server (using the WebSphere Server plug-in) authenticates the user. This authentication can take the form of a request for a user ID and password or it can happen transparently in the form of a certificate exchange followed by the establishment of a Secure Sockets Layer (SSL) session.

  The authentication policy is governed by an additional option: secure channel constraint. If the secure channel constraint is required, an SSL session must be established as the final phase of authentication; otherwise, SSL is optional.
- *Transactions*—Both types of Java clients can use the transaction service by way of the JTA interfaces to manage transactions. The code required for transaction management is identical in the two types of clients. For general information on transactions and the Java transaction service, see "The transaction service" on page 6. For information on managing transactions in a Java EJB client, see "Managing transactions in an EJB client" on page 86.

The Web server

To access the functionality in the EJB server, Java servlets and JSP files must have access to a Web server. The Web server enables communication between a Web client and the EJB server. The EJB server, Web server, and Java servlet can each reside on different machines.

For information on the Web servers supported by the EJB servers, see the Advanced Application Server *Getting Started* document.

The administration interface

The EJB server uses the WebSphere Administrative Console. For more information on this interface, consult the WebSphere InfoCenter and the online help available with the WebSphere Administrative Console. You can also administer the EJB server using the **wscp** command-line tool. For more information, see the Advanced Edition Information Center.

Chapter 2. An introduction to enterprise beans

This chapter looks at the characteristics and purpose of enterprise beans. It describes the two basic types of enterprise beans and their life cycles, and it provides an example of how enterprise beans can be combined to create distributed, three-tiered applications.

Bean basics

An enterprise bean is a Java component that can be combined with other enterprise beans and other Java components to create a distributed, three-tiered application. There are two types of enterprise beans:

- An *entity* bean encapsulates permanent data, which is stored in a data source such as a database or a file system, and associated methods to manipulate that data. In most cases, an entity bean must be accessed in some transactional manner. Instances of an entity bean are unique and they can be accessed by multiple users.

  For example, the information about a bank account can be encapsulated in an entity bean. An account entity bean might contain an account ID, an account type (checking or savings), and a balance variable and methods to manipulate these variables.
- A *session* bean encapsulates ephemeral (nonpermanent) data associated with a particular EJB client. Unlike the data in an entity bean, the data in a session bean is not stored in a permanent data source, and no harm is caused if this data is lost. However, a session bean can update data in an underlying database, usually by accessing an entity bean. A session bean can also participate in a transaction.

  When created, instances of a session bean are identical, though some session beans can store semipermanent data that makes them unique at certain points in their life cycle. A session bean is always associated with a single client; attempts to make concurrent calls result in an exception being thrown.

  For example, the task associated with transferring funds between two bank accounts can be encapsulated in a session bean. Such a transfer session bean can find two instances of an account entity bean (by using the account IDs), and then subtract a specified amount from one account and add the same amount to the other account.

Entity beans

This section discusses the basics of entity beans.

Basic components of an entity bean

Every entity bean must have the following components, which are illustrated in Figure 3:

- *Bean class*—This class encapsulates the data for the entity bean and contains the developer-implemented business methods that access the data. It also contains the methods used by the container to manage the life cycle of an entity bean instance. EJB clients (whether they are other enterprise beans or user components such as servlets) *never* access objects of this class directly; instead, they use the container-generated classes associated with the home and remote interfaces to manipulate the entity bean instance.
- *Home interface*—This interface defines the methods used by the client to create, find, and remove instances of the entity bean. This interface is implemented by the container during deployment in a class known generically as the *EJB home class*; instances are referred to as *EJB home objects*.
- *Remote interface*—Once the client has used the home interface to gain access to an entity bean, it uses this interface to invoke indirectly the business methods implemented in the bean class. This interface is implemented by the container during deployment in a class known generically as the *EJB object class*; instances are referred to as *EJB objects*.
- *Primary key* — One or more variables that uniquely identify a specific entity bean instance. A primary key that consists of a single variable of a primitive Java data type can be specified at deployment. A *primary key class* is used to encapsulate primary keys that consist of multiple variables or more complex Java data types. The primary key class also contains methods to create primary key objects and manipulate those objects.

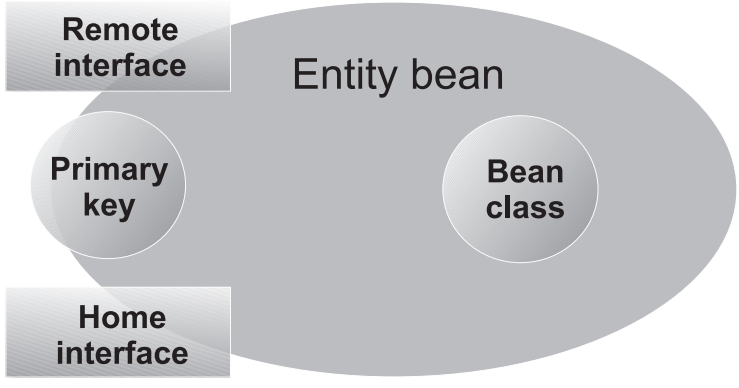

Figure 3. The components of an entity bean

Data persistence

Entity beans encapsulate and manipulate *persistent* (or permanent) business data. For example, at a bank, entity beans can be used to model customer profiles, checking and savings accounts, car loans, mortgages, and customer transaction histories.

To ensure that this important data is not lost, the entity bean stores its data in a data source such as a database. When the data in an enterprise bean instance is changed, the data in the data source is synchronized with the bean data. Of course, this synchronization takes place within the context of the appropriate type of transaction, so that if a router goes down or a server fails, permanent changes are not lost.

When you design an entity bean, you must decide whether you want the enterprise bean to handle this data synchronization or whether you want the container to handle it. An enterprise bean that handles its own data synchronization is said to implement *bean-managed persistence* (BMP), while an enterprise bean whose data synchronization is handled by the container is said to implement *container-managed persistence* (CMP).

Unless you have a good reason for implementing BMP, it is recommended that you design your entity beans to use CMP. The code for an enterprise bean with CMP is easier to write and does not depend on any particular data storage product, making it more portable between EJB servers. However, you must use entity beans with BMP if you want to use a data source that is not supported by the EJB server.

Session beans

This section discusses the basics of session beans.

Basic components of a session bean

Every session bean must have the following components, which are illustrated in Figure 4 on page 16:

- *Bean class*—This class encapsulates the data associated with the session bean and contains the developer-implemented business methods that access this data. It also contains the methods used by the container to manage the life cycle of an session bean instance. EJB clients (whether they are other enterprise beans or user applications) *never* access objects of this class directly; instead, they use the container-generated classes associated with the home and remote interfaces to manipulate the session bean.
- *Home interface*—This interface defines the methods used by the client to create and remove instances of the session bean. This interface is implemented by the container during deployment in a class known generically as the *EJB home class*; instances are referred to as *EJB home object*.
- *Remote interface*—After the client has used the home interface to gain access to an session bean, it uses this interface to invoke indirectly the business

methods implemented in the bean class. This interface is implemented by the container during deployment in a class known generically as the *EJB object class*; instances are referred to as *EJB objects*.

Unlike an entity bean, a session bean does not have a primary key class. A session bean does not require a primary key class because you do not need to search for specific instances of session beans.

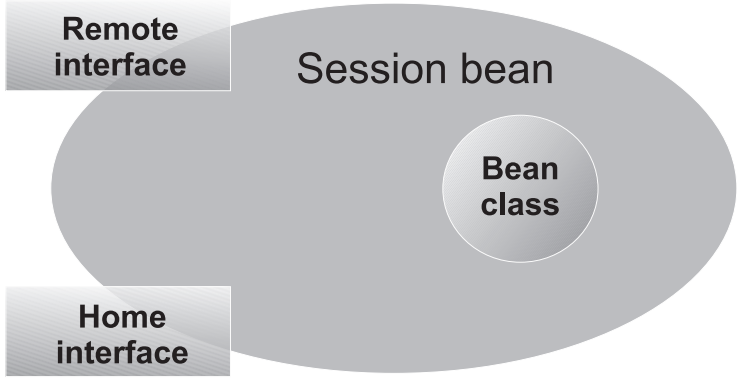

Figure 4. The components of a session bean

Stateless versus stateful session beans

Session beans encapsulate data and methods associated with a user session, task, or ephemeral object. By definition, the data in a session bean instance is ephemeral; if it is lost, no real harm is done. For example, at a bank, session beans can represent a funds transfer, the creation of a customer profile or new account, and a withdrawal or deposit. If information about a fund transfer is already typed (but not yet committed), and a server fails, the balances of the bank accounts remains the same. Only the transfer data is lost, which can always be retyped.

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

- If a session bean needs to maintain specific data across methods, it is referred to as a *stateful* session bean. When a session bean maintains data across methods, it is said to have a *conversational state*. A Web-based shopping cart is a classic use of a stateful session bean. As the shopping cart user adds items to and subtracts items from the shopping cart, the underlying session bean instance must maintain a record of the contents of the cart. After a particular EJB client begins using an instance of a stateful session bean, the client must continue to use that instance as long as the specific state of that instance is required. If the session bean instance is lost before the contents of the shopping cart are committed to an order, the shopper must load a new shopping cart.

- If a session bean does not need to maintain specific data across methods, it is referred to as a *stateless* session bean. The example Transfer session bean developed in "Developing session beans" on page 50 provides an example of a stateless session bean. For stateless session beans, a client can use any instance to invoke any of the session bean's methods because all instances are the same.

Creating an EJB module

The last step in the development of an enterprise bean is the creation of an EJB module. An EJB module consists of the following:

- One or more deployable enterprise beans.
- A deployment descriptor, stored in an Extensible Markup Language (XML) file. This file contains information about the structure and external dependencies of the beans in the module, and application assembly information describing how the beans are to be used in an application.

The EJB module can be created by using the tools within an integrated development environment (IDE) like IBM's VisualAge for Java Enterprise Edition or by using the tools contained in WebSphere. For more information, see "Chapter 3. Tools for developing and deploying enterprise beans" on page 27.

The EJB module

The *EJB module* is used to assemble enterprise beans into a single deployable unit; this file uses the standard Java archive file format. The EJB module can contain individual enterprise beans or multiple enterprise beans. For more information, see "Creating an EJB module and deployment descriptor" on page 67.

The deployment descriptor

The EJB module contains one or more deployable enterprise beans and one *deployment descriptor*. The deployment descriptor contains attribute and environment settings for each bean in the module, and it defines how the container invokes functionality for all beans in the module. The deployment descriptor attributes can be set for the entire enterprise bean or for the individual methods in the bean. The container uses the definition of the bean-level attribute unless a method-level attribute is defined, in which case the latter is used.

The deployment descriptor contains the following information about entity and session beans. These attributes can be set on the bean only; they cannot be set on a specific method of the bean.

- The bean's name, class, home interfaces, remote interfaces, and bean type (entity or session).

- *Primary key class* attribute—Identifies the primary key class for the bean. For more information, see "Writing the primary key class (entity with CMP)" on page 47 or "Writing or selecting the primary key class (entity with BMP)" on page 118.
- *Persistence management*. Specifies whether persistence management is performed by the enterprise bean or by the container.
- *Container-managed fields* attribute—Lists those persistent variables in the bean class that the container must synchronize with fields in a corresponding data source to ensure that this data is persistent and consistent. For more information, see "Defining variables" on page 35.
- *Reentrant* attribute—Specifies whether an enterprise bean can invoke methods on itself or call another bean that invokes a method on the calling bean. Only entity beans can be reentrant. For more information, see "Using threads and reentrancy in enterprise beans" on page 66.
- *State management* attribute—Defines the conversational state of the session bean. This attribute must be set to either STATEFUL or STATELESS. For more information on the meaning of these conversational states, see "Stateless versus stateful session beans" on page 16.
- *Timeout* attribute—Defines the idle timeout value in seconds associated with this session bean. (This attribute is an extension to the standard deployment descriptor.)
- References to external resources, such as resource connection factories, to the homes of other enterprise beans, and to security roles.

The deployment descriptor contains the following application assembly information:

- A display name and icons for identifying the module.
- The location of class files needed for a client program to access the beans in the module.
- *Security roles*— Define a logical grouping of principals. Access to operations (such as EJB methods) is controlled by granting access to a role.
- *Method permissions*—Define a mapping between one or more security roles and one or more methods that a member of the role can invoke. This value is set per method.
- *Transaction* attributes—Define the transactional manner in which the container invokes a method for enterprise beans that require container-managed transaction demarcation. This value is set per method. The values for this attribute are described in "Chapter 5. Enabling transactions and security in enterprise beans" on page 69.
- *Transaction isolation level* attribute—Defines the degree to which transactions are isolated from each other by the container. This value is set per method. The values for this attribute are described in "Chapter 5. Enabling

transactions and security in enterprise beans" on page 69. (This attribute is an extension to the standard deployment descriptor.)

- *RunAsMode* and *RunAsIdentity* attributes—The *RunAsMode* attribute defines the identity used to invoke the method. If a specific identity is required, the *RunAsIdentity* attribute is used to specify that identity. This value is set per bean. The values for the *RunAsMode* attribute are described in "Chapter 5. Enabling transactions and security in enterprise beans" on page 69. (This attribute is an extension to the standard deployment descriptor.)

The following binding attribute is stored in the repository (it is not part of the deployment descriptor):

- *JNDI home name* attribute—Defines the Java Naming and Directory Interface (JNDI) home name that is used to locate instances of an EJB home object. This value is set per bean. The values for this repository attribute are described in "Creating and getting a reference to a bean's EJB object" on page 79.

Deploying an EJB module

When you deploy an EJB module, the deployment tool creates or incorporates the following elements:

- The container-implemented *EJBObject* and *EJBHome* classes (hereafter referred to as the EJB object and EJB home classes) from the enterprise bean's home and remote interfaces (and the persistor and finder classes for entity beans with CMP).
- The stub and skeleton files required for remote method invocation (RMI).

Figure 5 on page 20 shows a simplified version of a deployed entity bean.

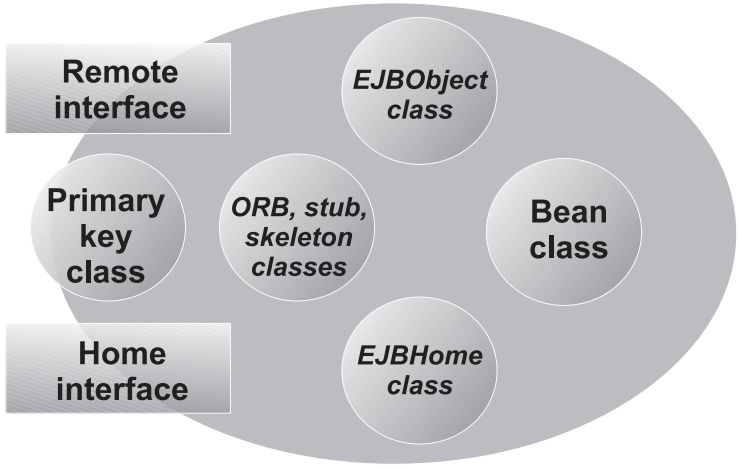

Figure 5. The major components of a deployed entity bean

You can deploy an EJB module with a variety of different tools. For more information, see "Chapter 3. Tools for developing and deploying enterprise beans" on page 27.

Developing EJB applications

To create EJB applications, create the enterprise beans and EJB clients that encapsulate your business data and functionality and then combine them appropriately. Figure 6 on page 21 provides a conceptual illustration of how EJB applications are created by combining one or more session beans, one or more entity beans, or both. Although individual entity beans and session beans can be used directly in an EJB client, session beans are designed to be associated with clients and entity beans are designed to store persistent data, so most EJB applications contain session beans that, in turn, access entity beans.

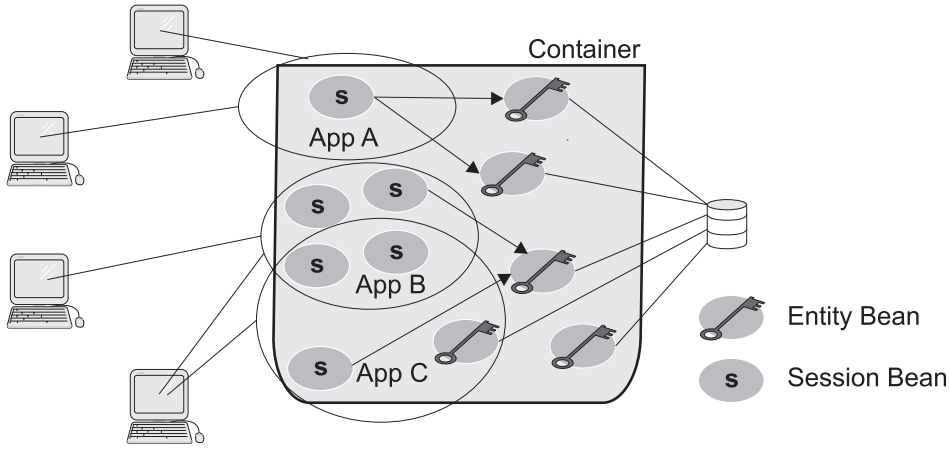

Figure 6. Conceptual view of EJB applications

This section provides an example of the ways in which enterprise beans can be combined to create EJB applications.

An example: enterprise beans for a bank

If you develop EJB applications for the banking industry, you can develop the following entity beans to encapsulate your business data and associated methods:

- Account bean—An entity bean that contains information about customer checking and savings accounts.
- CarLoan bean—An entity bean that contains information about an automobile loan.
- Customer bean—An entity bean that contains information about a customer, including information on accounts held and loans taken out by the customer.
- CustomerHistory bean—An entity bean that contains a record of customer transactions for specified accounts.
- Mortgage bean—An entity bean that contains information about a home or commercial mortgage.

An EJB client can directly access entity beans or session beans; however, the EJB Specification suggests that EJB clients use session beans to in turn access entity beans, especially in more complex applications. Therefore, as an EJB developer for the banking industry, you can create the following session beans to represent client tasks:

- LoanApprover bean—A session bean that allows a loan to be approved by using instances of the CarLoan bean, the Mortgage bean, or both.

- CarLoanCreator bean—A session bean that creates a new instance of a CarLoan bean.
- MortgageCreator bean—A session bean that creates a new instance of a Mortgage bean.
- Deposit bean—A session bean that credits a specified amount to an existing instance of an Account bean.
- StatementGenerator bean—A session bean that generates a statement summarizing the activities associated with a customer's accounts by using the appropriate instances of the Customer and CustomerHistory entity beans.
- Payment bean—A session bean that credits a payment to a customer's loan by using instances of the CarLoan bean, the Mortgage bean, or both.
- NewAccount bean—A session bean that creates a new instance of an Account bean.
- NewCustomer bean—A session bean that creates a new instance of a Customer bean.
- LoanReviewer bean—A session bean that accesses information about a customer's outstanding loans (instances of the CarLoan bean, the Mortgage bean, or both).
- Transfer bean—A session bean that transfers a specified amount between two existing instances of an Account bean.
- Withdraw bean—A session bean that debits a specified amount from an existing instance of an Account bean.

This example is simplified by necessity. Nevertheless, by using this set of enterprise beans, you can create a variety of EJB applications for different types of users by combining the appropriate beans within that application. One or more EJB clients can then be built to access the application.

Using the banking beans to develop EJB banking applications

When using beans built to the Sun Microsystems JavaBeans™ Specification (as opposed to the EJB Specification), you combine predefined components such as buttons and text fields to create GUI applications. When using enterprise beans, you combine predefined components such as the banking beans to create three-tiered applications.

For example, you can use the banking enterprise beans to create the following EJB applications:

- Home Banking application—An Internet application that allows a customer to transfer funds between accounts (with the Transfer bean), to make payments on a loan by using funds in an existing account (with the Payment bean), to apply for a car loan or home mortgage (with the CarLoanCreator bean or the MortgageCreator bean).

- Teller application—An intranet application that allows a teller to create new customer accounts (with the NewCustomer bean and the NewAccount bean), transfer funds between accounts (with the Transfer bean), and record customer deposits and withdrawals (with the Withdraw bean and the Deposit bean).
- Loan Officer application—An intranet application that allows a loan officer to create and approve car loans and home mortgages (with the CarLoanCreator, MortgageCreator, LoanReviewer, and LoanApprover beans).
- Statement Generator application—A batch application that prints monthly customer statements related to account activity (with the StatementGenerator bean).

These examples represent only a subset of the possible EJB applications that can be created with the banking beans.

Life cycles of enterprise bean instances

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

The methods mentioned in this section are discussed in greater detail in "Chapter 4. Developing enterprise beans" on page 33.

Session bean life cycle

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

Creation state

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

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

Ready state

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

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

  The container then invokes the following methods:

  1. The afterBegin method, if that method is implemented by the bean class.
  2. The business method in the bean class that corresponds to the business method defined in the bean's remote interface and called by the EJB client.
  3. The bean instance's beforeCompletion method, if that method is implemented by the bean class and if a commit is requested prior to the container's attempt to commit the transaction.

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

  If a rollback occurs, a stateful session bean can roll back its conversational state to the values contained in the bean instance prior to beginning the transaction. Stateless session beans do not maintain a conversational state, so they do not need to be concerned about rollbacks.
- *Nontransactional method invocations*—When a client invokes a nontransactional business method, the container simply invokes the corresponding method in the bean class.

Pooled state

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

If a client invokes a method on a passivated instance of a stateful session bean, the container activates the instance by restoring the instance's state and

then invoking the bean instance's ejbActivate method. When this method returns, the bean instance is again in the ready state.

Because every stateless session bean instance of a particular type is the same as every other instance of that type, stateless session bean instances are not passivated or activated. These instances exist in a ready state at all times until their removal.

Removal state

A session bean's life cycle ends when an EJB client or the container invokes a remove method defined in the bean's home interface and remote interface. In response to this method invocation, the container calls the bean instance's ejbRemove method.

If you attempt to remove a bean instance while it is associated with a transaction, the javax.ejb.RemoveException is thrown. After a bean instance is removed, any attempt to invoke a method on that instance causes the java.rmi.NoSuchObjectException to be thrown.

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

For more information on the remove methods, see "Removing a bean's EJB object" on page 86.

Entity bean life cycle

This section describes the life cycle of entity bean instances. Differences between entity beans with CMP and BMP are noted.

Creation State

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

Pooled State

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

Ready State

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

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

- When a client invokes the create method in the bean's home interface to create a new and unique entity of the entity bean class (and a new record in the data source). As a result of this method invocation, the container calls the bean instance's ejbCreate and ejbPostCreate methods, and the new EJB object is associated with the bean instance.
- When a client invokes a finder method to manipulate an existing instance of the entity bean class (associated with an existing record in the data source). In this case, the container calls the bean instance's ejbActivate method to associate the bean instance with the existing EJB object.

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

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

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

Removal State

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

For more information on the remove methods, see "Removing a bean's EJB object" on page 86.

Chapter 3. Tools for developing and deploying enterprise beans

There are two basic approaches to developing and deploying enterprise beans:

- You can use one of the available integrated development environments (IDEs) such as IBM VisualAge™ for Java Enterprise Edition. IDE tools automatically generate significant parts of the enterprise bean code and contain integrated tools for packaging and testing enterprise beans. VisualAge for Java is the recommended development tool. For more information on using VisualAge for Java, see "Using VisualAge for Java".
- You can use the tools available in the Java Software Development Kit (SDK) and the Advanced Application Server. For more information, see "Developing and deploying enterprise beans" on page 28.

Using VisualAge for Java

Before you can develop enterprise beans in VisualAge for Java, you must set up the EJB development environment. You need to perform this setup task only once. This setup procedure directs VisualAge for Java to import all of the classes and interfaces required to develop enterprise beans.

After generating an enterprise bean, you complete its development by following these general steps:

1. Implement the enterprise bean class.
2. Create the required abstract methods in the bean's home and remote interfaces by promoting the corresponding methods in the bean class to the appropriate interface.
3. For entity beans, do the following:
   a. Create any additional finder methods in the home interface by using the appropriate menu items.
   b. Create a finder helper interface, if required.
4. Create the EJB module and corresponding deployment descriptor.
5. Generate the deployment code for the bean.

VisualAge for Java contains a complete WebSphere Application Server run time environment and a mechanism to generate a test client to test your enterprise beans. For much more detailed information on developing enterprise beans in VisualAge for Java, refer to the VisualAge for Java documentation.

Developing and deploying enterprise beans

If you have decided to develop enterprise beans *without* an IDE, you need at minimum the following tools:

- An ASCII text editor. (You can use also use a Java development tool that does not support enterprise bean development.)
- The SDK Java compiler (**javac**) and Java Archiving tool (**jar**).
- The WebSphere Application Assembly Tool and the WebSphere Administrative Console.

This section describes steps you can follow to develop enterprise beans by using these tools. The following tasks are involved in the development of enterprise beans:

1. Ensure that you have installed and configured the prerequisite software to develop, deploy, and run enterprise beans in the EJB server environment. For more information, see "Installing and configuring the software for the EJB server".
2. Set the CLASSPATH environment variable required by different components of the EJB server environment. For more information, see "Setting the CLASSPATH environment variable in the EJB server environment" on page 29.
3. Write and compile the components of the enterprise bean. For more information, see "Creating the components of an enterprise bean" on page 29.
4. (*Entity beans with CMP only*) Create a finder helper interface for each entity bean with CMP that contains specialized finder methods (other than the findByPrimaryKey method). For more information, see "Creating finder logic in the EJB server" on page 30.
5. Create an EJB module and corresponding deployment descriptor by using the Application Assembly Tool. For more information, see "Creating an EJB module" on page 30.
6. (*Entity beans only*) Create a database schema to enable storage of the entity bean's persistent data in a database. For more information, see "Creating a database for use by entity beans" on page 31.
7. Generate deployment code for the EJB module by using the Application Assembly Tool. For more information, see the WebSphere InfoCenter and the online help available with the Application Assembly Tool.
8. Install the EJB module into an EJB server and start the server by using the WebSphere Administrative Console.

Installing and configuring the software for the EJB server

You must ensure that you have installed and configured the following prerequisite software products before you can begin developing enterprise beans and EJB clients with the EJB server:

- WebSphere Application Server Advanced Edition
- One or more of the following databases for use by entity beans with container-managed persistence (CMP):
  - DB2
  - Oracle
  - Sybase
  - Informix
  - Microsoft SQL Server
  - InstantDB
- The Java Software Development Kit (SDK)

For information on the appropriate version numbers of these products and instructions for setting up the environment, see the WebSphere InfoCenter.

Setting the CLASSPATH environment variable in the EJB server environment

In addition to the classes.zip file contained in the SDK, the following WebSphere JAR files must be appended to the CLASSPATH environment variable for developing enterprise beans:

- ejs.jar
- ujc.jar
- *otherDeployedBean*.jar (if the enterprise bean uses another enterprise bean). This is the deployed JAR file containing the enterprise bean being used by this enterprise bean.

For developing and running an EJB client, the following WebSphere JAR files must be appended to the CLASSPATH environment variable:

- ejs.jar
- ujc.jar
- servlet.jar (required by EJB clients that are servlets)
- *otherDeployedBean*.jar. This is the deployed JAR file containing the enterprise bean being used by this EJB client.

Creating the components of an enterprise bean

If you use an ASCII text editor or a Java development tool that does not support enterprise bean development, you must create each of the components that compose the enterprise bean you are creating. You must ensure that these components match the requirements described in "Chapter 4. Developing enterprise beans" on page 33.

To manually develop a session bean, you must write the bean class, the bean's home interface, and the bean's remote interface. To manually develop an

entity bean, you must write the bean class, the bean's primary key class, the bean's home interface, the bean's remote interface, and if necessary, the bean's finderHelper interface.

After you have properly coded these components, use the Java compiler to create the corresponding Java class files. For example, because the components of the example Account bean are stored in a specific directory, the bean components can be compiled by issuing the following command:

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

This command assumes that the CLASSPATH environment variable contains all of the packages used by the Account bean.

Creating finder logic in the EJB server

For the EJB server environment, the following finder logic is required for each finder method (other than the findByPrimaryKey method) contained in the home interface of an entity bean with CMP:

- The logic must be defined in a public interface named *Name*BeanFinderHelper, where *Name* is the name of the enterprise bean (for example, AccountBeanFinderHelper).
- The logic must be contained in a String constant named *findMethodName*WhereClause, where *findMethodName* is the name of the finder method. The String constant can contain zero or more question marks (?) that are replaced from left to right with the value of the finder method's arguments when that method is invoked.

Note: Encapsulating the logic in a String constant named *findMethodName*QueryString has been deprecated.

If you define the findLargeAccounts method shown in Figure 14 on page 45, you must also create the AccountBeanFinderHelper interface shown in Figure 7.

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

Figure 7. Code example: AccountBeanFinderHelper interface for the EJB server

Creating an EJB module

The WebSphere Application Server Application Assembly Tool can be used to create an EJB module. An EJB module can contain one or more enterprise beans. The tool automatically creates the required deployment descriptor for the module based on information specified by the user.

Using the Application Assembly Tool

To create an EJB module and corresponding deployment descriptor, use the Create EJB Module wizard in the Application Assembly Tool. This wizard prompts you to specify the following information for each enterprise bean to be included in the module:

- The enterprise bean class, home interface class, and remote interface class.
- The bean type (entity or session), and associated attributes (such as persistence management type and primary key class for entity beans).
- References to another enterprise bean's home interface and to resource connection factories.
- References to security roles for the enterprise bean.
- CMP fields, if applicable.
- Transaction isolation level attributes for enterprise bean methods.

The wizard also prompts you to specify the following application assembly information for the module itself:

- General properties of the EJB module, such as the location of class files needed for a client program to access the enterprise beans in the module and the icons to be associated with the module.
- The deployable enterprise beans that the module will contain.
- Security roles used to access resources in the module.
- Transaction attributes for the enterprise bean methods.

Both bean and module information are used to create the deployment descriptor. See the WebSphere InfoCenter and the online help for details on how to use the Application Assembly Tool.

Creating a database for use by entity beans

For entity beans with *container-managed persistence (CMP)*, you must store the bean's persistent data in one of the supported databases. The Application Assembly Tool automatically generates SQL code for creating database tables for CMP entity beans. The tool names the database schema and table ejb.*beanName*beantbl, where *beanName* is the name of the enterprise bean (for example, ejb.accountbeantbl). If your CMP entity beans require complex database mappings, it is recommended that you use VisualAge for Java to generate code for the database tables. At run time, the WebSphere Administrative Console displays a prompt asking whether you want to execute the generated SQL code that creates the database table.

For entity beans with *bean-managed persistence (BMP)*, you can create the database and database table by using the database tools or use an existing database and database table. Because entity beans with BMP handle the database interaction, any database or database table name is acceptable.

For more information on creating databases and database tables, consult your database documentation and the online help for the WebSphere Administrative Console.

Chapter 4. Developing enterprise beans

This chapter explains the basic tasks required to develop and package the most common types of enterprise beans. Specifically, this chapter focuses on creating stateless session beans and entity beans that use container-managed persistence (CMP); in the discussion of stateless session beans, important information about stateful beans is also provided. For information on developing entity beans that use bean-managed persistence (BMP), see "Developing entity beans with BMP" on page 103.

The information in this chapter is not exhaustive; however, it includes the information you need to develop basic enterprise beans. For information on developing more complicated enterprise beans, consult a commercially available book on enterprise bean development. The example enterprise beans discussed in this chapter and the example Java applications and servlets that use them are described in "Information about the examples described in the documentation" on page 197.

This chapter describes the requirements for building each of the major components of an enterprise bean. If you do *not* intend to use one of the commercially available integrated development environments (IDE), such as IBM's VisualAge for Java, you must build each of these components manually (by using tools in the Java Development Kit and WebSphere). Manually developing enterprise beans is much more difficult and error-prone than developing them in an IDE. Therefore, it is strongly recommended that you choose an IDE with which you are comfortable.

Developing entity beans with CMP

In an entity bean with CMP, the container handles the interactions between the entity bean and the data source. In an entity bean with BMP, the entity bean must contain all of the code required for the interactions between the entity bean and the data source. For this reason, developing an entity bean with CMP is simpler than developing an entity bean with BMP.

This section examines the development of entity beans with CMP. While much of the information in this section also applies to entity beans with BMP, there are some major differences between the two types. For information on the tasks required to develop an entity bean with BMP, see "Developing entity beans with BMP" on page 103.

Every entity bean must contain the following basic parts:

- The enterprise bean class. For more information, see "Writing the enterprise bean class (entity with CMP)".
- The enterprise bean's home interface. For more information, see "Writing the home interface (entity with CMP)" on page 43.
- The enterprise bean's remote interface. For more information, see "Writing the remote interface (entity with CMP)" on page 46.
- The enterprise bean's primary key class. For more information, see "Writing the primary key class (entity with CMP)" on page 47.

Writing the enterprise bean class (entity with CMP)

In a CMP entity bean, the bean class defines and implements the business methods of the enterprise bean, defines and implements the methods used to create instances of the enterprise bean, and implements the methods used by the container to inform the instances of the enterprise bean of significant events in the instance's life cycle. Enterprise bean clients never access the bean class directly; instead, the classes that implement the home and remote interfaces are used to indirectly invoke the methods defined in the bean class.

By convention, the enterprise bean class is named *Name*Bean, where *Name* is the name you assign to the enterprise bean. The enterprise bean class for the example Account enterprise bean is named AccountBean.

Every entity bean class with CMP must meet the following requirements:

- It must be public, it must *not* be abstract, and it must implement the javax.ejb.EntityBean interface. For more information, see "Implementing the EntityBean interface" on page 41.
- It must define instance variables that correspond to persistent data associated with the enterprise bean. For more information, see "Defining variables" on page 35.
- It must implement the business methods used to access and manipulate the data associated with the enterprise bean. For more information, see "Implementing the business methods" on page 36.
- It must define and implement an ejbCreate method for each way in which the enterprise bean can be instantiated. A corresponding ejbPostCreate method must be defined for each ejbCreate method. For more information, see "Implementing the ejbCreate and ejbPostCreate methods" on page 39.

Note:

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

An enterprise bean class cannot implement two different interfaces if the methods in the interfaces have the same name, even if the methods have different signatures, due to the Java-IDL mapping specification. Errors can occur when the enterprise bean is deployed.

Figure 8 shows the main parts of the enterprise bean class for the example Account enterprise bean. (Emphasized code is in bold type.) The sections that follow discuss these parts in greater detail.

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

Figure 8. Code example: The AccountBean class

Defining variables

An entity bean class can contain both persistent and nonpersistent instance variables; however, static variables are not supported in enterprise beans unless they are also final (that is, they are constants). Static variables are not supported because there is no way to guarantee that they remain consistent across enterprise bean instances.

Container-managed fields (which are persistent variables) are stored in a database. Container-managed fields must be public.

Nonpersistent variables are *not* stored in a database and are temporary. Nonpersistent variables must be used with caution and must not be used to maintain the state of an EJB client between method invocations. This restriction is necessary because nonpersistent variables cannot be relied on to remain the same between method invocations outside of a transaction because other EJB clients can change these variables, or they can be lost when the entity bean is passivated.

The AccountBean class contains three container-managed fields (shown in Figure 9 on page 36):

- *accountId*, which identifies the account ID associated with an account
- *type*, which identifies the account type as either savings (1) or checking (2)
- *balance*, which identifies the current balance of the account

```
...
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;
     ...
}
```

Figure 9. Code example: The variables of the AccountBean class

The deployment descriptor is used to identify container-managed fields in entity beans with CMP. In an entity bean with CMP, each container-managed field must be initialized by each ejbCreate method (see "Implementing the ejbCreate and ejbPostCreate methods" on page 39).

A subset of the container-managed fields is used to define the primary key class associated with each instance of an enterprise bean. As is shown in "Writing the primary key class (entity with CMP)" on page 47, the *accountId* variable defines the primary key for the Account enterprise bean.

The AccountBean class contains two nonpersistent variables:

- *entityContext*, which identifies the entity context of each instance of an Account enterprise bean. The entity context can be used to get a reference to the EJB object currently associated with the bean instance and to get the primary key object associated with that EJB object.
- *bundle*, which encapsulates a resource bundle class (com.ibm.ejs.doc.account.AccountResourceBundle) that contains locale-specific objects used by the Account bean.

Implementing the business methods

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

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

Figure 10 on page 38 shows the business methods for the AccountBean class. These methods are used to add a specified amount to an account balance and return the new balance (add), to return the current balance of an account (getBalance), to set the balance of an account (setBalance), and to subtract a specified amount from an account balance and return the new balance (subtract).

The subtract method throws the user-defined exception com.ibm.ejs.doc.account.InsufficientFundsException if a client attempts to subtract more money from an account than is contained in the account balance. The subtract method in the Account bean's remote interface must also throw this exception as shown in Figure 15 on page 47. User-defined exception classes for enterprise beans are created as are any other user-defined exception class. The message content for the InsufficientFundsException exception is obtained from the AccountResourceBundle class file by invoking the getMessage method on the *bundle* object.

Note: If an enterprise bean container catches a system exception from the business method of an enterprise bean, and the method is running within a container-managed transaction, the container rolls back the transaction before passing the exception on to the client. However, if the business method is throwing an application exception, then the transaction is not rolled back (it is committed), unless the application has called setRollbackOnly function. In this case, the transaction is rolled back before the exception is re-thrown.

```
...
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;
     }
     ...
}
```

Figure 10. Code example: The business methods of the AccountBean class

Standard application exceptions for entity beans

Version 1.1 of the EJB specification defines several standard application exceptions for use by enterprise beans. All of these exceptions are subclasses of the javax.ejb.EJBException class. For entity beans with both container- and bean-managed persistence, the EJB specification defines the following application exceptions:

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

Application programmers can use the generic EJBException class or one of the provided subclassed exceptions, or programmers can define their own exceptions by subclassing any of this family of exceptions. All of these

exceptions inherit from the javax.ejb.RuntimeException class and do not have to be explicitly declared in throws clauses.

Each exception is discussed in more detail within the relevant section; for more information on:

- CreateException and DuplicateKeyException (a subclass of the CreateException class), see "Implementing the ejbCreate and ejbPostCreate methods".
- javax.ejb.RemoveException, see "Implementing the EntityBean interface" on page 41.
- FinderException and ObjectNotFoundException (a subclass of the FinderException class), see "Defining finder methods" on page 45.

Note: Version 1.0 of the EJB specification used the java.rmi.RemoteException class to capture application-specific exceptions; the EJBException class and its subclasses are new in the 1.1 version of the specification. Therefore, using the RemoteException class is now deprecated in favor of the more precise exception classes. Older applications that use the RemoteException class can still run, but enterprise beans compliant with version 1.1 of the specification must use the new exception classes.

Implementing the ejbCreate and ejbPostCreate methods

You must define and implement an ejbCreate method for each way in which you want a new instance of an enterprise bean to be created. For each ejbCreate method, you must also define a corresponding ejbPostCreate method. Each ejbCreate and ejbPostCreate method must correspond to a create method in the home interface.

Like the business methods of the bean class, the ejbCreate and ejbPostCreate methods cannot be invoked directly by the client. Instead, the client invokes the create method of the enterprise bean's home interface by using the EJB home object, and the container invokes the ejbCreate method followed by the ejbPostCreate method. If the ejbCreate and ejbPostCreate methods are executed successfully, an EJB object is created and the persistent data associated with that object is inserted into the data source.

For an entity bean with CMP, the container handles the required interaction between the entity bean instance and the data source between calls to the ejbCreate and ejbPostCreate methods. For an entity bean with BMP, the ejbCreate method must contain the code to directly handle this interaction. For more information on entity beans with BMP, see "Developing entity beans with BMP" on page 103.

Each ejbCreate method in an entity bean with CMP must meet the following requirements:

- It must be public and return the same type as the primary key. The actual return value must be null.
- Its arguments must be valid for Java remote method invocation (RMI). For more information, see "The java.io.Serializable and java.rmi.Remote interfaces" on page 66.
- It must initialize the container-managed fields of the enterprise bean instance. The container extracts the values of these variables and writes them to the data source after the ejbCreate method returns.

Each ejbPostCreate method must be public, return void, and have the same arguments as the matching ejbCreate method.

If necessary, both the ejbCreate method and the ejbPostCreate method can throw the javax.ejb.EJBException exception or one of the creation-related subclasses, the CreateException or the DuplicateKeyException exceptions. The DuplicateKeyException class is a subclass of the CreateException class. Throwing the java.rmi.RemoteException exception is deprecated; see "Standard application exceptions for entity beans" on page 38 for more information.

Figure 11 on page 41 shows two sets of ejbCreate and ejbPostCreate methods required for the example AccountBean class. The first set of ejbCreate and ejbPostCreate methods are wrappers that call the second set of methods and set the *type* variable to 1 (corresponding to a savings account) and the *balance* variable to 0 (zero dollars).

```
...
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) { }
     ...
}
```

Figure 11. Code example: The ejbCreate and ejbPostCreate methods of the AccountBean class

Implementing the EntityBean interface

Each entity bean class must implement the methods inherited from the javax.ejb.EntityBean interface. The container invokes these methods to inform the bean instance of significant events in the instance's life cycle. (For more information, see "Entity bean life cycle" on page 25.) All of these methods must be public and return void; they can throw the javax.ejb.EJBException exception or, in the case of the ejbRemove method, the javax.ejb.RemoveException exception. Throwing the java.rmi.RemoteException exception is deprecated; see "Standard application exceptions for entity beans" on page 38 for more information.

- ejbActivate—This method is invoked by the container when the container selects an entity bean instance from the instance pool and assigns that instance to a specific existing EJB object. This method must contain any code that you want to execute when the enterprise bean instance is activated.
- ejbLoad—This method is invoked by the container to synchronize an entity bean's container-managed fields with the corresponding data in the data source. (That is, the values of the fields in the data source are loaded into the container-managed fields in the corresponding enterprise bean instance.)

This method must contain any code that you want to execute when the enterprise bean instance is synchronized with associated data in the data source.

- ejbPassivate—This method is invoked by the container when the container disassociates an entity bean instance from its EJB object and places the enterprise bean instance in the instance pool. This method must contain any code that you want to execute when the enterprise bean instance is ″passivated″ or deactivated.
- ejbRemove—This method is invoked by the container when a client invokes the remove method inherited by the enterprise bean's home interface from the javax.ejb.EJBHome interface. This method must contain any code that you want to execute before an enterprise bean instance is removed from the container (and the associated data is removed from the data source). This method can throw the javax.ejb.RemoveException exception if removal of an enterprise bean instance is not permitted.
- setEntityContext—This method is invoked by the container to pass a reference to the javax.ejb.EntityContext interface to an enterprise bean instance. If an enterprise bean instance needs to use this context at any time during its life cycle, the enterprise bean class must contain an instance variable to store this value. This method must contain any code required to store a reference to a context.
- ejbStore—This method is invoked by the container when the container needs to synchronize the data in the data source with the values of the container-managed fields in an enterprise bean instance. (That is, the values of the variables in the enterprise bean instance are copied to the data source, overwriting the previous values.) This method must contain any code that you want to execute when the data in the data source is overwritten with the corresponding values in the enterprise bean instance.
- unsetEntityContext—This method is invoked by the container, before an enterprise bean instance is removed, to free up any resources associated with the enterprise bean instance. This is the last method called prior to removing an enterprise bean instance.

In entity beans with CMP, the container handles the required data source interaction for these methods. In entity beans with BMP, these methods must directly handle the required data source interaction. For more information on entity beans with BMP, see "Chapter 8. More-advanced programming concepts for enterprise beans" on page 103.

These methods have several possible uses, including the following:

- They can contain audit or debugging code.
- They can contain code for allocating and deallocating additional resources used by the bean instance (for example, an SNA connection to a mainframe).

As shown in Figure 12, except for the setEntityContext and unsetEntityContext methods, all of these methods are empty in the AccountBean class because no additional action is required by the bean for the particular life cycle states associated with the these methods. The setEntityContext and unsetEntityContext methods are used in a conventional way to set the value of the *entityContext* variable.

```
...
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;
     }
}
```

Figure 12. Code example: Implementing the EntityBean interface in the AccountBean class

Writing the home interface (entity with CMP)

An entity bean's home interface defines the methods used by clients to create new instances of the bean, find and remove existing instances, and obtain metadata about an instance. The home interface is defined by the enterprise bean developer and implemented in the EJB home class created by the container during enterprise bean deployment.

The container makes the home interface accessible to enterprise bean clients through the Java Naming and Directory Interface (JNDI). JNDI is independent of any specific naming and directory service and allows Java-based applications to access any naming and directory service in a standard way.

By convention, the home interface is named *Name*Home, where *Name* is the name you assign to the enterprise bean. For example, the Account enterprise bean's home interface is named AccountHome.

Every home interface must meet the following requirements:

- It must extend the javax.ejb.EJBHome interface. The home interface inherits several methods from the javax.ejb.EJBHome interface. See "The javax.ejb.EJBHome interface" on page 65 for information on these methods.
- Each method in the interface must be either a create method that corresponds to a set of ejbCreate and ejbPostCreate methods in the EJB object class, or a finder method. For more information, see "Defining create methods" and "Defining finder methods" on page 45.
- The parameters and return value of each method defined in the home interface must be valid for Java RMI. For more information, see "The java.io.Serializable and java.rmi.Remote interfaces" on page 66. In addition, each method's throws clause must include the java.rmi.RemoteException exception class.

Figure 13 shows the relevant parts of the definition of the home interface (AccountHome) for the example Account bean. This interface defines two abstract create methods: the first creates an Account object by using an associated AccountKey object, the second creates an Account object by using an associated AccountKey object and specifying an account type and an initial balance. The interface defines the required findByPrimaryKey method and a findLargeAccounts method, which returns a collection of accounts containing balances greater than a specified amount.

```
...
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;
}
```

Figure 13. Code example: The AccountHome home interface

Defining create methods

A create method is used by a client to create an enterprise bean instance and insert the data associated with that instance into the data source. Each create method must be named create and it must have the same number and types

of arguments as a corresponding ejbCreate method in the enterprise bean class. (The ejbCreate method must itself have a corresponding ejbPostCreate method.)

Each create method must meet the following requirements:

- It must be named create.
- It must return the type of the enterprise bean's remote interface. For example, the return type for the create methods in the AccountHome interface is Account (as shown in Figure 13 on page 44).
- It must have a throws clause that includes the java.rmi.RemoteException exception, the javax.ejb.CreateException exception, and all of the application exceptions defined in the throws clause of the corresponding ejbCreate and ejbPostCreate methods.

Defining finder methods

A finder method is used to find one or more existing entity EJB objects. Each finder method must be named find*Name*, where *Name* further describes the finder method's purpose.

At minimum, each home interface must define the findByPrimaryKey method that enables a client to locate an EJB object by using the primary key only. The findByPrimaryKey method has one argument, an object of the bean's primary key class, and returns the type of the bean's remote interface.

Every other finder method must meet the following requirements:

- It must return the type of the enterprise bean's remote interface, the java.util.Enumeration interface, or the java.util.Collection interface (when a finder method can return more than one EJB object or an EJB collection).
- It must have a throws clause that includes the java.rmi.RemoteException and javax.ejb.FinderException exception classes.

While every entity bean must contain the default finder method, you can write additional finder methods if needed. For example, the Account bean's home interface defines the findLargeAccounts method to find objects that encapsulate accounts with balances of more than a specified amount, as shown in Figure 14. Because this finder method can be expected to return a reference to more than one EJB object, its return type is Enumeration.

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

Figure 14. Code example: The findLargeAccounts method

Every EJB server can implement the findByPrimaryKey method. During enterprise bean deployment, the container generates the code required to search the database for the appropriate enterprise bean instance.

However, for each additional finder method that you define in the home interface, the enterprise bean deployer must associate finder logic with that finder method. This logic is used by the EJB server during deployment to generate the code required to implement the finder method.

The EJB Specification does not define the format of the finder logic, so the format can vary according to the EJB server you are using. For more information on creating finder logic, see "Creating finder logic in the EJB server" on page 30.

Writing the remote interface (entity with CMP)

An entity bean's remote interface provides access to the business methods available in the bean class. It also provides methods to remove an EJB object associated with a bean instance and to obtain the bean instance's home interface, object handle, and primary key. The remote interface is defined by the enterprise bean developer and implemented in the EJB object class created by the container during enterprise bean deployment.

By convention, the remote interface is named *Name*, where *Name* is the name you assign to the enterprise bean. For example, the Account enterprise bean's remote interface is named Account.

Every remote interface must meet the following requirements:

- It must extend the javax.ejb.EJBObject interface. The enterprise bean's remote interface inherits several methods from the javax.ejb.EJBObject interface. See "Methods inherited from javax.ejb.EJBObject" on page 65 for information on these methods.
- You must define a corresponding business method for every business method implemented in the enterprise bean class.
- The parameters and return value of each method defined in the interface must be valid for Java RMI. For more information, see "The java.io.Serializable and java.rmi.Remote interfaces" on page 66.
- Each method's throws clause must include the java.rmi.RemoteException exception class.

Figure 15 on page 47 shows the relevant parts of the definition of the remote interface (Account) for the example Account enterprise bean. This interface defines four methods for displaying and manipulating the account balance that exactly match the business methods implemented in the AccountBean class.

All of the business methods in the remote interface throw the java.rmi.RemoteException exception class. In addition, the subtract method must throw the user-defined exception com.ibm.ejs.doc.account.InsufficientFundsException because the corresponding method in the bean class throws this exception. Furthermore, any client that calls this method must either handle the exception or pass it on by throwing it.

```
...
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;
}
```

Figure 15. Code example: The Account remote interface

Writing the primary key class (entity with CMP)

Within a container, every entity EJB object has a unique identity that is defined by using a combination of the object's home interface name and its primary key, the latter of which is assigned to the object at creation. If two EJB objects have the same identity, they are considered identical.

Primary keys are specified in two ways:

- Simple primary keys, which map to a single field in the entity bean class and are comprised of primitive Java data types (such as integer or long), are specified in the deployment descriptor.
- Composite primary keys, which map to multiple fields in the entity bean class (or to data structures built from the primitive Java data types), must be encapsulated in a *primary key class*. More complicated enterprise beans are likely to have composite primary keys, with multiple instance variables representing the primary key.

The primary key class is used to manage an EJB object's primary key. By convention, the primary key class is named *Name*Key, where *Name* is the name of the enterprise bean. For example, the Account enterprise bean's primary key class is named AccountKey.

The primary key class must meet the following requirements:

- It must be public and it must be serializable. For more information, see "The java.io.Serializable and java.rmi.Remote interfaces" on page 66.
- Its instance variables must be public, and the variable names must match a subset of the container-managed field names defined in the enterprise bean class.
- It must have a public default constructor, at a minimum.

Note: The primary key class of a CMP entity bean must override the equals method and the hashCode method inherited from the java.lang.Object class.

Figure 16 on page 49 shows a composite primary key class for an example enterprise bean, Item. In effect, this class acts as a wrapper around the string variables *productId* and *vendorId*. The hashCode method for the ItemKey class invokes the corresponding hashCode method in the java.lang.String class after creating a temporary string object by using the value of the *productId* variable. In addition to the default constructor, the ItemKey class also defines a constructor that sets the value of the primary key variables to the specified strings.

```
...
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-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-specific method
      public int hashCode() {
                return (new productId.hashCode());
       }
 }
```

Figure 16. Code example: The ItemKey primary key class

A primary key class can also be used to encapsulate a primary key that is not known ahead of time — for instance, if the entity bean is intended to work with several persistent data stores, each of which requires a different primary key structure. The entity bean's primary key type is derived from the primary key type used by the underlying database that stores the entity objects; it does not necessarily have to be known to the enterprise bean developer.

To specify an unknown primary key, do the following:

- Declare the argument of the findByPrimaryKey class as java.lang.Object.
- Declare the return value of the ejbCreate method as java.lang.Object

- In the deployment descriptor, specify the primary key class as being of the type java.lang.Object.

When the primary key selection is deferred to deployment, client applications cannot use methods that rely on knowledge of the primary key type. In addition, applications cannot always depend on methods that return the type of the primary key (such as the EntityContext.getPrimaryKey method) because the return type is determined at deployment.

Interacting with databases

This section contains general information and tips on enterprise beans and database access.

- Although it is not necessary, it is good practice to specify the user ID and password for a data source either in the enterprise bean to be using the data source, or in the container of the bean.
- The container supports Option A and Option C caching. When Option A caching is in use, the application server hosting the enterprise bean container must be the only updater of the data in the persistent store. As such, Option A caching is incompatible with the following:
  - Workload managed servers (such as a cluster of clones)
  - Databases with data being shared among multiple applications

  The default caching option is C (multiple entity bean instances, possibly in different servers, can update bean state in the database). The default caching option can be changed from Option C to Option A by selecting ″exclusive persistent store″ in the administrative console when creating the entity bean.

Shared database access corresponds to Option C caching. Option A and Option C caching are also known as commit option A and commit option C, respectively.

Developing session beans

In their basic makeup, session beans are similar to entity beans. However, their purposes are very different.

From a component perspective, one of the biggest differences between the two types of enterprise beans is that session beans do not have a primary key class and the session bean's home interface does not define finder methods. Session enterprise beans do not require primary keys and finder methods because session EJB objects are created, associated with a specific client, and then removed as needed, whereas entity EJB objects represent permanent data in a data source and can be uniquely identified with a primary key. Because the

data for session beans is never permanently stored, the session bean class does not have methods for storing data to and loading data from a data source.

Every session bean must contain the following basic parts:

- The enterprise bean class. For more information, see "Writing the enterprise bean class (session)".
- The enterprise bean's home interface. For more information, see "Writing the home interface (session)" on page 62.
- The enterprise bean's remote interface. For more information, see "Writing the remote interface (session)" on page 63.

Writing the enterprise bean class (session)

A session bean class defines and implements the business methods of the enterprise bean, implements the methods used by the container during the creation of enterprise bean instances, and implements the methods used by the container to inform the enterprise bean instance of significant events in the instance's life cycle. By convention, the enterprise bean class is named *Name*Bean, where *Name* is the name you assign to the enterprise bean. The enterprise bean class for the example Transfer enterprise bean is named TransferBean.

Every session bean class must meet the following requirements:

- It must define and implement the business methods that execute the tasks associated with the enterprise bean. For more information, see "Implementing the business methods" on page 53.
- It must define and implement an ejbCreate method for each way in which you want it to be able to instantiate the enterprise bean class. For more information, see "Implementing the ejbCreate methods" on page 56.
- It must be public, it must *not* be abstract, and it must implement the javax.ejb.SessionBean interface. For more information, see "Implementing the SessionBean interface" on page 61.

Note: Version 1.0 of the EJB specification allowed the methods in the session bean class to throw the java.rmi.RemoteException exception to indicate a non-application exception. This practice is deprecated in version 1.1 of the specification. A session bean compliant with version 1.1 of the specification should throw the javax.ejb.EJBException exception (a subclass of the java.lang.RuntimeException class) or another RuntimeException exception instead. Because the javax.ejb.EJBException class is a subclass of the java.lang.RuntimeException, EJBException exceptions do not need to be explicitly listed in the throws clause of methods.

A session bean can be either stateful or stateless. In a stateless session bean, none of the methods depend on the values of variables set by any other method, except for the ejbCreate method, which sets the initial (identical) state of each bean instance. In a stateful enterprise bean, one or more methods depend on the values of variables set by some other method. As in entity beans, static variables are not supported in session beans unless they are also final.

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

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

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

Figure 17 on page 53 shows the main parts of the enterprise bean class for the example Transfer bean. The sections that follow discuss these parts in greater detail.

The Transfer bean is stateless. If the Transfer bean's transferFunds method were dependent on the value of the *balance* variable returned by the getBalance method, the TransferBean would be stateful.

```
...
import java.rmi.RemoteException;
import java.util.Properties;
import java.util.ResourceBundle;
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 {
          ...
     }
}
```

Figure 17. Code example: The TransferBean class

Implementing the business methods

The business methods of a session bean class define the ways in which an EJB client can manipulate the enterprise bean. The business methods implemented in the enterprise bean class cannot be directly invoked by an EJB client. Instead, the EJB client invokes the corresponding methods defined in the

enterprise bean's remote interface, by using an EJB object associated with an instance of the enterprise bean, and the container invokes the corresponding methods in the enterprise bean instance.

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

Figure 18 on page 55 shows the business methods for the TransferBean class. The getBalance method is used to get the balance for an account. It first locates the appropriate Account EJB object and then calls that object's getBalance method.

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

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

To call the findByPrimaryKey method, both business methods need to be able to access the EJB home object that implements the AccountHome interface discussed in "Writing the home interface (entity with CMP)" on page 43. Obtaining the EJB home object is discussed in "Implementing the ejbCreate methods" on page 56.

```
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);
        }
   }
}
```

Figure 18. Code example: The business methods of the TransferBean class

Implementing the ejbCreate methods

You must define and implement an ejbCreate method for each way in which you want an enterprise bean to be instantiated.

Each ejbCreate method must correspond to a create method in the enterprise bean's home interface. (Note that there is no ejbPostCreate method in a session bean as there is in an entity bean.) Unlike the business methods of the enterprise bean class, the ejbCreate methods cannot be invoked directly by the client. Instead, the client invokes the create method in the bean instance's home interface, and the container invokes the ejbCreate method. If an ejbCreate method is executed successfully, an EJB object is created.

An ejbCreate method for a session bean must meet the following requirements:

- The method must be declared as public and cannot be declared as final or static.
- It must return void.
- A stateless session bean must have only one ejbCreate method, which must return void and contain no arguments. A stateful session bean can have multiple ejbCreate methods.

The throws clause can define arbitrary application exceptions. The javax.ejb.EJBException or another runtime exception can be used to indicate non-application exceptions.

An ejbCreate method for an entity bean must meet the following requirements:

- The method must be declared as public and cannot be declared as final or static.
- It must return the entity bean's primary key type.
- It must contain code to set the values of any variables needed by the EJB object.

The throws clause can define arbitrary application exceptions. The javax.ejb.EJBException or another runtime exception can be used to indicate non-application exceptions.

Figure 19 on page 58 shows the ejbCreate method required by the example TransferBean class. The Transfer bean's ejbCreate method obtains a reference to the Account bean's home object. This reference is required by the Transfer bean's business methods. Getting a reference to an enterprise bean's home interface is a two-step process:

1. Construct an InitialContext object by setting the required property values. For the example Transfer bean, these property values are defined in the environment variables of the Transfer bean's deployment descriptor.
2. Use the InitialContext object to create and get a reference to the home object. For the example Transfer bean, the JNDI name of the Account bean is stored in an environment variable in the Transfer bean's deployment descriptor.

Creating the InitialContext object: When a container invokes the Transfer bean's ejbCreate method, the enterprise bean's *initialContext* object is constructed by creating a Properties variable (*env*) that requires the following values:

- The location of the name service (javax.naming.Context.PROVIDER_URL).
- The name of the initial context factory (javax.naming.Context.INITIAL_CONTEXT_FACTORY).

The values of these properties are discussed in more detail in "Creating and getting a reference to a bean's EJB object" on page 79.

```
...
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
          ...
}
```

Figure 19. Code example: Creating the InitialContext object in the ejbCreate method of the TransferBean class

Although the example Transfer bean stores some locale specific variables in a resource bundle class, like the example Account bean, it also relies on the values of environment variables stored in its deployment descriptor. Each of these InitialContext Properties values is obtained from an environment variable contained in the Transfer bean's deployment descriptor. A private get method that corresponds to the property variable is used to get each of the values (getNamingFactory and getProviderURL); these methods must be written by the enterprise bean developer. The following environment variables must be set to the appropriate values in the deployment descriptor of the Transfer bean.

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

Figure 20 on page 59 illustrates the relevant parts of the getProviderURL method that is used to get the PROVIDER_URL property value. The javax.ejb.SessionContext variable (*mySessionCtx*) is used to get the Transfer bean's environment in the deployment descriptor by invoking the getEnvironment method. The object returned by the getEnvironment method

can then be used to get the value of a specific environment variable by invoking the getProperty method.

```
...
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;
     }
     ...
}
```

Figure 20. Code example: The getProviderURL method

Getting the reference to the home object: An enterprise bean is accessed by looking up the class implementing its home interface by name through JNDI. Methods on the home interface provide access to an instance of the class implementing the remote interface.

After constructing the InitialContext object, the ejbCreate method performs a JNDI lookup using the JNDI name of the Account enterprise bean. Like the PROVIDER_URL and INITIAL_CONTEXT_FACTORY properties, this name is also retrieved from an environment variable contained in the Transfer bean's deployment descriptor (by invoking a private method named getHomeName). The lookup method returns an object of type java.lang.Object.

The returned object is narrowed by using the static method javax.rmi.PortableRemoteObject.narrow to obtain a reference to the EJB home object for the specified enterprise bean. The parameters of the narrow method are the object to be narrowed and the class of the object to be created as a result of the narrowing. For a more thorough discussion of the code required to locate an enterprise bean in JNDI and then narrow it to get an EJB home object, see "Creating and getting a reference to a bean's EJB object" on page 79.

```
...
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(
                         ejbHome,  AccountHome.class);
          } catch (NamingException e) { // Error getting the home interface
               ...
          }
          ...
     }
     ...
}
```

Figure 21. Code example: Creating the AccountHome object in the ejbCreate method of the TransferBean class

Looking up an enterprise bean's environment naming context: The enterprise bean's environment is implemented by the container. It enables the bean's business logic to be customized without the need to access or change the bean's source code. The container provides an implementation of the JNDI naming context that stores the enterprise bean environment. Business methods access the environment by using the JNDI interfaces. The deployment descriptor provides the environment entries that the enterprise bean expects at runtime.

Each enterprise bean defines its own environment entries, which are shared between all of its instances (that is, all instances with the same home). Environment entries are not shared between enterprise beans.

An enterprise bean's environment entries are stored directly in the environment naming context (or one of its subcontexts). To retrieve its environment naming context, an enterprise bean instance creates an InitialContext object by using the constructor with no arguments. It then looks up the environment naming via the InitialContext object under the name java:comp/env.

The enterprise bean in Figure 22 on page 61 changes an account number by looking up an environment entry to find the new account number.

```
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");
            ... }
}
```

Figure 22. Code example: Looking up an enterprise bean's environment naming context

Implementing the SessionBean interface

Every session bean class must implement the methods inherited from the javax.ejb.SessionBean interface. The container invokes these methods to inform the enterprise bean instance of significant events in the instance's life cycle. All of these methods must be public, must return void, and can throw the javax.ejb.EJBException. (Throwing the java.rmi.RemoteException exception is deprecated; see 51 for more information.)

- ejbActivate—This method is invoked by the container when the container selects an enterprise bean instance from the instance pool and assigns it a specific existing EJB object. This method must contain any code that you want to execute when the enterprise bean instance is activated.
- ejbPassivate—This method is invoked by the container when the container disassociates an enterprise bean instance from its EJB object and places the enterprise bean instance in the instance pool. This method must contain any code that you want to execute when the enterprise bean instance is passivated (deactivated).
- ejbRemove—This method is invoked by the container when a client invokes the remove method inherited by the enterprise bean's home interface (from the javax.ejb.EJBHome interface). This method must contain any code that you want to execute when an enterprise bean instance is removed from the container.
- setSessionContext—This method is invoked by the container to pass a reference to the javax.ejb.SessionContext interface to a session bean instance. If an enterprise bean instance needs to use this context at any time during its life cycle, the enterprise bean class must contain an instance variable to store this value. This method must contain any code required to store a reference to the context.

A session context can be used to get a handle to a particular instance of a stateful session bean. It can also be used to get a reference to a transaction context object, as described in "Using bean-managed transactions" on page 124.

As shown in Figure 23, except for the setSessionContext method, all of these methods in the TransferBean class are empty because no additional action is required by the bean for the particular life cycle states associated with the these methods. The setSessionContext method is used in a conventional way to set the value of the *mySessionCtx* variable.

```
...
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;
     }
     ...
}
```

Figure 23. Code example: Implementing the SessionBean interface in the TransferBean class

Writing the home interface (session)

A session bean's home interface defines the methods used by clients to create and remove instances of the enterprise bean and obtain metadata about an instance. The home interface is defined by the enterprise bean developer and implemented in the EJB home class created by the container during enterprise bean deployment. The container makes the home interface accessible to clients through JNDI.

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

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

- It must extend the javax.ejb.EJBHome interface. The home interface inherits several methods from the javax.ejb.EJBHome interface. See "The javax.ejb.EJBHome interface" on page 65 for information on these methods.
- Each method in the interface must be a create method that corresponds to a ejbCreate method in the enterprise bean class. For more information, see

"Implementing the ejbCreate methods" on page 56. Unlike entity beans, the home interface of a session bean contains no finder methods.

- The parameters and return value of each method defined in the interface must be valid for Java RMI. For more information, see "The java.io.Serializable and java.rmi.Remote interfaces" on page 66. In addition, each method's throws clause must include the java.rmi.RemoteException exception class.

Figure 24 shows the relevant parts of the definition of the home interface (TransferHome) for the example Transfer bean.

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

Figure 24. Code example: The TransferHome home interface

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

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

Each create method must meet the following requirements:

- It must return the type of the enterprise bean's remote interface. For example, the return type for the create method in the TransferHome interface is Transfer.
- It must have a throws clause that includes the java.rmi.RemoteException exception, the javax.ejb.CreateException exception class, and all of the exceptions defined in the throws clause of the corresponding ejbCreate method.

Writing the remote interface (session)

A session bean's remote interface provides access to the business methods available in the enterprise bean class. It also provides methods to remove an enterprise bean instance and to obtain the enterprise bean's home interface

and handle. The remote interface is defined by the enterprise bean developer and implemented in the EJB object class created by the container during enterprise bean deployment.

By convention, the remote interface is named *Name*, where *Name* is the name you assign to the enterprise bean. For example, the Transfer enterprise bean's remote interface is named Transfer.

Every remote interface must meet the following requirements:

- It must extend the javax.ejb.EJBObject interface. The remote interface inherits several methods from the EJBObject interface. See "Methods inherited from javax.ejb.EJBObject" on page 65 for information on these methods.
- You must define a corresponding business method for every business method implemented in the enterprise bean class.
- The parameters and return value of each method defined in the interface must be valid for Java RMI. For more information, see "The java.io.Serializable and java.rmi.Remote interfaces" on page 66.
- Each method's throws clause must include the java.rmi.RemoteException exception class.

Figure 25 shows the relevant parts of the definition of the remote interface (Transfer) for the example Transfer bean. This interface defines the methods for transferring funds between two Account bean instances and for getting the balance of an Account bean instance.

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

Figure 25. Code example: The Transfer remote interface

Implementing interfaces common to multiple types of enterprise beans

Enterprise beans must implement the interfaces described here in the appropriate enterprise bean component.

Methods inherited from javax.ejb.EJBObject

The remote interface inherits the following methods from the javax.ejb.EJBObject interface, which are implemented by the container during deployment:

- getEJBHome—Returns the enterprise bean's home interface.
- getHandle—Returns the handle for the EJB object.
- getPrimaryKey—Returns the EJB object's primary key. (For session beans, this cannot be used because session beans do not have a primary key.)
- isIdentical—Compares this EJB object with the EJB object argument to determine if they are the same.
- remove—Removes this EJB object.

These methods have the following syntax:

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

These methods are implemented by the container in the EJB object class.

The javax.ejb.EJBHome interface

The home interface inherits two remove methods and the getEJBMetaData method from the javax.ejb.EJBHome interface. Just like the methods defined directly in the home interface, these inherited methods are also implemented in the EJB home class created by the container during deployment.

The remove methods are used to remove an existing EJB object (and its associated data in the database) either by specifying the EJB object's handle or its primary key. (The remove method that takes a *primaryKey* variable can be used only in entity beans.) The getEJBMetaData method is used to obtain metadata about the enterprise bean and is mainly intended for use by development tools.

These methods have the following syntax:

```
public abstract EJBMetaData getEJBMetaData();
public abstract void remove(Handle handle);
public abstract void remove(Object primaryKey);
```

The javax.ejb.EJBHome interface also contains a method to get a handle to the home interface. It has the following syntax:

```
public abstract HomeHandle getHomeHandle();
```

The java.io.Serializable and java.rmi.Remote interfaces

To be valid for use in a remote method invocation (RMI), a method's arguments and return value must be one of the following types:

- A primitive type; for example, an int or a long.
- An object of a class that directly or indirectly implements java.io.Serializable; for example, java.lang.Long.
- An object of a class that directly or indirectly implements java.rmi.Remote.
- An array of valid types or objects.

If you attempt to use a parameter that is not valid, the java.rmi.RemoteException exception is thrown. Note that the following atypical types are *not* valid:

- An object of a class that directly or indirectly implements both Serializable and Remote.
- An object of a class that directly or indirectly implements Remote, but contains a method that does not throw the RemoteException or an exception that inherits from RemoteException.

Using threads and reentrancy in enterprise beans

An enterprise bean must not contain code to start new threads (nor can methods be defined with the keyword synchronized). Session beans can *never* be reentrant; that is, they cannot call another bean that invokes a method on the calling bean. Entity beans can be reentrant, but building reentrant entity beans is not recommended and is not documented here.

The EJB server enforces single-threaded access to all enterprise beans. Illegal callbacks result in a java.rmi.RemoteException exception being thrown to the EJB client.

Creating an EJB module for enterprise beans

There are two tasks involved in preparing an enterprise bean for deployment:

- Making the components of the bean part of the same Java package. For more information, see "Making bean components part of a Java package" on page 67.
- Creating an EJB module and associated deployment descriptor. For more information, see "Creating an EJB module and deployment descriptor" on page 67.

If you develop enterprise beans in an IDE, these tasks are handled from within the tool that you use. If you do not develop enterprise beans in an IDE, you must handle each of these tasks by using tools contained in the Java Software Development Kit (SDK) and WebSphere Application Server. For

more information on the tools used to create an EJB module in the EJB server programming environment, see "Chapter 3. Tools for developing and deploying enterprise beans" on page 27.

Making bean components part of a Java package

You determine the best way to allocate your enterprise beans to Java packages. A Java package can contain one or more enterprise beans. The example Account and Transfer beans are stored in separate packages. All of the Java source files that make up the Account bean contain the following package statement:

```
package com.ibm.ejs.doc.account;
```

All of the Java source files that make up the Transfer bean contain the following package statement:

```
package com.ibm.ejs.doc.transfer;
```

Creating an EJB module and deployment descriptor

An EJB module contains one or more deployable enterprise beans. It also contains a deployment descriptor that provides information about each enterprise bean and instructions for the container on how to handle all enterprise beans in the module. The deployment descriptor is stored in an XML file.

During creation of the EJB module, you specify the files for each enterprise bean to be included in the module. These files include:

- The class files associated with each component of the enterprise bean.
- Any additional classes and files associated with the enterprise bean; for example: user-defined exception classes, properties files, and resource bundle classes.

You also specify other information about the bean, such as references to other enterprise beans, resource connection factories, and security roles. After defining the enterprise beans to be included in the module, you specify application assembly instructions that apply to the module as a whole. Both bean and module information are used to create a deployment descriptor. See "The deployment descriptor" on page 17 for a list of deployment descriptor settings and attributes.

Chapter 5. Enabling transactions and security in enterprise beans

This chapter examines how to enable transactions and security in enterprise beans by setting the appropriate deployment descriptor attributes:

- For transactions, a session bean can either use container-managed transactions or implement bean-managed transactions; entity beans must use container-managed transactions. To enable container-managed transactions, you must set the transaction attribute to any value *except* BeanManged and set the transaction isolation level attribute. To enable bean-managed transactions, you must set the transaction attribute to BeanManaged and set the transaction isolation level attribute. For more information, see "Setting transactional attributes in the deployment descriptor".

  If you want a session bean to manage its own transactions, you must write the code that explicitly demarcates the boundaries of a transaction as described in "Using bean-managed transactions" on page 124.

  If you want an EJB client to manage its own transactions, you must explicitly code that client to do so as described in "Managing transactions in an EJB client" on page 86.
- For security, the *run-as mode* attribute is used by the EJB server environments. For information on the valid values of this attribute, see "Setting the security attribute in the deployment descriptor" on page 74.

These attributes, like the other deployment descriptor attributes, are set by using one of the tools available. For more information, see "Chapter 3. Tools for developing and deploying enterprise beans" on page 27.

Setting transactional attributes in the deployment descriptor

The EJB Specification describes the creation of applications that enforce transactional consistency on the data manipulated by the enterprise beans. However, unlike other specifications that support distributed transactions, the EJB specification does not require enterprise bean and EJB client developers to write any special code to use transactions. Instead, the container manages transactions based on two deployment descriptor attributes associated with the EJB module, and the enterprise bean and EJB application developers are freed to deal with the business logic of their applications.

Enterprise bean developers can specifically design enterprise beans and EJB applications that explicitly manage transactions. For more information, see "Using bean-managed transactions" on page 124.

Under most conditions, transaction management can be handled within the enterprise beans, freeing the EJB client developer of this task. However, EJB clients can participate in transactions if required or desired. For more information, see "Managing transactions in an EJB client" on page 86.

Two attributes determine the way in which an enterprise bean is managed from a transactional perspective:

- The *transaction* attribute defines the transactional manner in which the container invokes a method. This attribute is part of the standard deployment descriptor. "Setting the transaction attribute" defines the valid values of this attribute and explains their meanings.
- The *transaction isolation level* attribute defines the manner in which transactions are isolated from each other by the container. This attribute is an extension to the standard deployment descriptor. "Setting the transaction isolation level attribute" on page 72 defines the valid values of this attribute and explains their meanings.

Setting the transaction attribute

The transaction attribute defines the transactional manner in which the container invokes enterprise bean methods. This attribute is set for individual methods in a bean.

The following are valid values for this attribute in decreasing order of transactional strictness:

BeanManaged

Notifies the container that the bean class directly handles transaction demarcation. This attribute value can be specified only for session beans and it cannot be specified for individual bean methods. For more information on designing session beans to implement this attribute value, see "Using bean-managed transactions" on page 124.

Mandatory

Directs the container to always invoke the bean method within the transaction context associated with the client. If the client attempts to invoke the bean method without a transaction context, the container throws the javax.jts.TransactionRequiredException exception to the client. The transaction context is passed to any EJB object or resource accessed by an enterprise bean method.

EJB clients that access these entity beans must do so within an existing transaction. For other enterprise beans, the enterprise bean or bean method must implement the BeanManaged value or use the

Required or RequiresNew value. For non-enterprise bean EJB clients, the client must invoke a transaction by using the javax.transaction.UserTransaction interface, as described in "Managing transactions in an EJB client" on page 86.

Required
Directs the container to invoke the bean method within a transaction context. If a client invokes a bean method from within a transaction context, the container invokes the bean method within the client transaction context. If a client invokes a bean method outside of a transaction context, the container creates a new transaction context and invokes the bean method from within that context. The transaction context is passed to any enterprise bean objects or resources that are used by this bean method.

RequiresNew
Directs the container to always invoke the bean method within a new transaction context, regardless of whether the client invokes the method within or outside of a transaction context. The transaction context is passed to any enterprise bean objects or resources that are used by this bean method.

Supports
Directs the container to invoke the bean method within a transaction context if the client invokes the bean method within a transaction. If the client invokes the bean method without a transaction context, the container invokes the bean method without a transaction context. The transaction context is passed to any enterprise bean objects or resources that are used by this bean method.

NotSupported
Directs the container to invoke bean methods without a transaction context. If a client invokes a bean method from within a transaction context, the container suspends the association between the transaction and the current thread before invoking the method on the enterprise bean instance. The container then resumes the suspended association when the method invocation returns. The suspended transaction context is *not* passed to any enterprise bean objects or resources that are used by this bean method.

Never Directs the container to invoke bean methods without a transaction context.

- If the client invokes a bean method from within a transaction context, the container throws the java.rmi.RemoteException exception.
- If the client invokes a bean method from outside a transaction context, the container behaves in the same way as if the

NotSupported transaction attribute was set. The client must call the method without a transaction context.

Table 2. Effect of the enterprise bean's transaction attribute on the transaction context

Transaction attribute	Client transaction context	Bean transaction context
Mandatory	No transaction	Not allowed
	Client transaction	Client transaction
RequiresNew	No transaction	New transaction
	Client transaction	New transaction
Required	No transaction	New transaction
	Client transaction	Client transaction
Supports	No transaction	No transaction
	Client transaction	Client transaction
NotSupported	No transaction	No transaction
	Client transaction	No transaction
Never	No transaction	No transaction
	No transaction	No transaction

When setting the deployment descriptor for an entity bean, you can mark getter methods as "Read-Only" methods to improve performance. If a transaction unit of work includes no methods other than "Read-Only" designated methods, then the entity bean state synchronization does not invoke store.

Setting the transaction isolation level attribute

The transaction isolation level determines how strongly one transaction is isolated from another. This attribute is set for individual methods in a bean. However, within a transactional context, the isolation level associated with the first method invocation becomes the required isolation level for all other methods invoked within that transaction. If a method is invoked with a different isolation level from that of the first method, the java.rmi.RemoteException exception is thrown.

The following are valid values for this attribute, in decreasing order of isolation:

Serializable

This level prohibits all of the following types of reads:

- *Dirty reads*, where a transaction reads a database row containing uncommitted changes from a second transaction.

- *Nonrepeatable reads*, where one transaction reads a row, a second transaction changes the same row, and the first transaction rereads the row and gets a different value.
- *Phantom reads*, where one transaction reads all rows that satisfy an SQL WHERE condition, a second transaction inserts a row that also satisfies the WHERE condition, and the first transaction applies the same WHERE condition and gets the row inserted by the second transaction.

RepeatableRead
This level prohibits dirty reads and nonrepeatable reads, but it allows phantom reads.

ReadCommitted
This level prohibits dirty reads, but allows nonrepeatable reads and phantom reads.

ReadUncommitted
This level allows dirty reads, nonrepeatable reads, and phantom reads.

These isolation levels correspond to the isolation levels defined in the Java Database Connectivity (JDBC) java.sql.Connection interface.

The container uses the transaction isolation level attribute as follows:

- Session beans and entity beans with bean-managed persistence (BMP)—For each database connection used by the bean, the container sets the transaction isolation level at the start of each transaction.
- Entity beans with container-managed persistence (CMP)—The container generates database access code that implements the specified isolation level.

None of these values permits two transactions to update the same data concurrently; one transaction must end before another can update the same data. These values determine only how locks are managed for reading data. However, risks to consistency can arise from read operations when a transaction does further work based on the values read. For example, if one transaction is updating a piece of data and a second transaction is permitted to read that data after it has been changed but before the updating transaction ends, the reading transaction can make a decision based on a change that is eventually rolled back. The second transaction risks making a decision on transient data.

Deciding which isolation level to use depends on several factors:

- The acceptable level of risk to data consistency
- The acceptable levels of concurrency and performance
- The isolation levels supported by the underlying database

The first two factors, risk to consistency and level of concurrency, are related. Decreasing the risk to consistency requires you to decrease concurrency because reducing the risk to consistency requires holding locks longer. The longer a lock is held on a piece of data, the longer concurrently running transactions must wait to access that data. The Serializable value protects data by eliminating concurrent access to it. Conversely, the ReadUncommitted value allows the highest degree of concurrency but entails the greatest risk to consistency. You need to balance these two factors appropriately for your application.

By default, most developers deploy enterprise beans with the transaction isolation level set to Serializable. This is the default value in IBM VisualAge for Java Enterprise Edition and other deployment tools. It is also the most restrictive and protected transaction isolation level incurring the most overhead. Some workloads do not require the isolation level and protection afforded by Serializable. A given application might never update the underlying data or be run with other applications that also make concurrent updates. In that case, the application would not have to be concerned with dirty, non-repeatable, or phantom reads. The ReadUncommitted isolation level would probably be sufficient.

Because the transaction isolation level is set in the EJB module's deployment descriptor, the same enterprise bean could be reused in different applications with different transaction isolation levels. The isolation level requirements should be reviewed and adjusted appropriately to increase performance.

The third factor, isolation levels supported in the database, means that although the EJB specification allows you to request one of the four levels of transaction isolation, it is possible that the database being used in the application does not support all of the levels. Also, vendors of database products implement isolation levels differently, so the precise behavior of an application can vary from database to database. You need to consider the database and the isolation levels it supports when deciding on the value for the transaction isolation attribute in deployment descriptors. Consult your database documentation for more information on supported isolation levels.

Setting the security attribute in the deployment descriptor

When an EJB client invokes a method on an enterprise bean, the user context of the client principal is encapsulated in a CORBA Current object, which contains credential properties for the principal. The Current object is passed among the participants in the method invocation as required to complete the method.

The security service uses the credential information to determine the permissions that a principal has on various resources. At appropriate points,

the security service determines if the principal is authorized to use a particular resource based on the principal's permissions.

If the method invocation is authorized, the security service does the following with the principal's credential properties based on the value of the *run-as mode* attribute of the enterprise bean. If a specific identity is required, the *RunAsIdentity* attribute is used to specify that identity.

Identity of Caller
The security service makes no changes to the principal's credential properties.

Identity of EJB Server
The security service alters the principal's credential properties to match the credential properties associated with the EJB server.

Identity Assigned to Specified Role
A security principal that has been assigned to the specified role is used for the execution of the bean's methods. This association is part of the application binding where the role is associated with a user ID and password of a user who is granted that role.

Chapter 6. Developing EJB clients

An enterprise bean can be accessed by all of the following types of EJB clients in both EJB server environments:

- Java servlets. For more information about writing Java servlets that use enterprise beans, see "Chapter 7. Developing servlets that use enterprise beans" on page 91.
- Java Server Pages (JSP). For more information about writing JSP, consult a commercially available book.
- Java applications that use remote method invocation (RMI). For more information on writing Java applications, consult a commercially available book.
- Other enterprise beans. For example, the Transfer session bean acts as a client to the Account bean, as described in "Chapter 4. Developing enterprise beans" on page 33.

It is recommended that you avoid accessing EJB entity beans from client or servlet code. Instead, wrap and access EJB entity beans from EJB session beans. This improves performance in two ways:

- It reduces the number of remote method calls. When the client application accesses the entity bean directly, each getter method is a remote call. A wrapping session bean can access the entity bean locally, and collect the data in a structure, which it returns by value.
- It provides an outer transaction context for the EJB entity bean. An entity bean synchronizes its state with its underlying data store at the completion of each transaction. When the client application accesses the entity bean directly, each getter method becomes a complete transaction. A store and a load action follow each method. When the session bean wraps the entity bean to provide an outer transaction context, the entity bean synchronizes its state when the outer session bean reaches a transaction boundary.

Except for the basic programming tasks described in this chapter, creating a Java servlet, JSP, or Java application that is a client to an enterprise bean is not very different from designing standard versions of these types of Java programs. This chapter assumes that you understand the basics of writing a Java servlet, a Java application, or a JSP file.

Except where noted, all of the code described in this chapter is taken from the example Java application named TransferApplication. This Java application

and the other EJB clients available with the documentation example code are explained in "Information about the examples described in the documentation" on page 197.

To access and manipulate an enterprise bean in any of the Java-based EJB client types listed previously, the EJB client must do the following:

- Import the Java packages required for naming, remote method invocation (RMI), and enterprise bean interaction.
- Get a reference to an instance of the bean's EJB object by using the Java Naming and Directory Interface (JNDI). For more information, see "Creating and getting a reference to a bean's EJB object" on page 79.
- Handle invalid EJB objects when using session beans. For more information, see "Handling an invalid EJB object for a session bean" on page 84.
- Remove session EJB objects when they are no longer required or remove entity EJB objects when the associated data in the data source must be removed. For more information, see "Removing a bean's EJB object" on page 86.

In addition, an EJB client can participate in the transactions associated with enterprise beans used by the client. For more information, see "Managing transactions in an EJB client" on page 86.

Importing required Java packages

Although the Java packages required for any particular EJB client vary, the following packages are required by all EJB clients:

- java.rmi — This package contains most of the classes required for remote method invocation (RMI).
- javax.rmi — This package contains the PortableRemoteObject class required to get a reference to an EJB object.
- java.util — This package contains various Java utility classes, such as Properties, Hashtable, and Enumeration used in a variety of ways throughout all enterprise beans and EJB clients.
- javax.ejb — This package contains the classes and interfaces defined in the EJB specification.
- javax.naming — The package contains the classes and interfaces defined in the Java Naming and Directory Interface (JNDI) specification and is used by clients to get references to EJB objects.
- The package or packages containing the enterprise beans with which the client interacts.

The Java client object request broker (ORB), which is automatically initialized in EJB clients, does not support dynamic download of implementation

bytecode from the server to the client. As a result, all classes required by the EJB client at runtime must be available from the files and directories identified in the client's CLASSPATH environment variable. For information on the JAR files required by EJB clients, see "Setting the CLASSPATH environment variable in the EJB server environment" on page 29. You can install needed files on your client machine by doing a WebSphere Application Server installation on the machine. Select the **Developer's Client Files** option. You also need to make sure that the ioser and ioserx executable files are accessible on your client machine; these files are normally part of the Java install. If you are using a Windows System, make sure that EJB clients can locate the ioser.dll library file at run time.

Figure 26 shows the import statements for the example Java application com.ibm.ejs.doc.client.TransferApplication. In addition to the required Java packages mentioned previously, the example application imports the com.ibm.ejs.doc.transfer package because the application communicates with a Transfer bean. The example application also imports the InsufficientFundsException class contained in the same package as the Account bean.

```
...
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 {
     ...
}
```

Figure 26. Code example: The import statements for the Java application TransferApplication

Creating and getting a reference to a bean's EJB object

To invoke a bean's business methods, a client must create or find an EJB object for that bean. After the client has created or found this object, it can invoke methods on it in the standard way.

To create or find an instance of a bean's EJB object, the client must do the following:

1. Locate and create an EJB home object for that bean. For more information, see "Locating and creating an EJB home object".
2. Use the EJB home object to create or (for entity beans only) find an instance of the bean's EJB object. For more information, see "Creating an EJB object" on page 83.

The TransferApplication client contains one reference to a Transfer EJB object, which the application uses to invoke all of the methods on the Transfer bean. When using session beans in Java applications, it is a good idea to make the reference to the EJB object a class-level variable rather than a variable that is local to a method. This allows your EJB client to repeatedly invoke methods on the same EJB object rather than having to create a new object each time the client invokes a session bean method. As discussed in "Threading issues" on page 101, this approach is not recommended for servlets, which must be designed to handle multiple threads.

Locating and creating an EJB home object

JNDI is used to find the name of an EJB home object. The properties that an EJB client uses to initialize JNDI and find an EJB home object vary across EJB server implementations. To make an enterprise bean more portable between EJB server implementations, it is recommended that you externalize these properties in environment variables, properties files, or resource bundles rather than hard code them into your enterprise bean or EJB client code.

The example Transfer bean uses environment variables as discussed in "Implementing the ejbCreate methods" on page 56. The TransferApplication uses a resource bundle contained in the com.ibm.ejs.doc.client.ClientResourceBundle.class file.

To initialize a JNDI name service, an EJB client must set the appropriate values for the following JNDI properties:

javax.naming.Context.PROVIDER_URL

This property specifies the host name and port of the name server used by the EJB client. The property value must have the following format: iiop://*hostname*:*port*, where *hostname* is the IP address or hostname of the machine on which the name server runs and *port* is the port number on which the name server listens.

For example, the property value `iiop://bankserver.mybank.com:9019` directs an EJB client to look for a name server on the host named bankserver.mybank.com listening on port 9019. The property value `iiop://bankserver.mybank.com` directs an EJB client to look for a name server on the host named bankserver.mybank.com at port number 900. The property value `iiop:///` directs an EJB client to look for a name server on the local host listening on port 900. If not specified, this property defaults to the local host and port number 900,

which is the same as specifying `iiop:///`. The port number used by the name service can be changed by using the administrative interface.

javax.naming.Context.INITIAL_CONTEXT_FACTORY
This property identifies the actual name service that the EJB client must use. This property must be set to `com.ibm.ejs.ns.jndi.CNInitialContextFactory`.

Locating an EJB home object is a two-step process:

1. Create a javax.naming.InitialContext object. For more information, see "Creating an InitialContext object".
2. Use the InitialContext object to create the EJB home object. For more information, see "Creating EJB home object" on page 82.

Creating an InitialContext object

Figure 27 on page 82 shows the code required to create the InitialContext object. To create this object, construct a java.util.Properties object, add values to the Properties object, and then pass the object as the argument to the InitialContext constructor. In the TransferApplication, the value of each property is obtained from the resource bundle class named com.ibm.ejs.doc.client.ClientResourceBundle, which stores all of the locale-specific variables required by the TransferApplication. (This class also stores the variables used by the other EJB clients contained in the documentation example, described in "Information about the examples described in the documentation" on page 197).

The resource bundle class is instantiated by calling the ResourceBundle.getBundle method. The values of variables within the resource bundle class are extracted by calling the getString method on the *bundle* object.

The createTransfer method of the TransferApplication can be called multiple times as explained in "Handling an invalid EJB object for a session bean" on page 84. However, after the InitialContext object is created once, it remains good for the life of the client session. Therefore, the code required to create the InitialContext object is placed within an if statement that determines if the reference to the InitialContext object is null. If the reference is null, the InitialContext object is created; otherwise, the reference can be reused on subsequent creations of the EJB object.

```
...
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;
}
```

Figure 27. Code example: Creating the InitialContext object

Creating EJB home object

After the InitialContext object (*ivjInitContext)* is created, the application uses it to create the EJB home object, as shown in Figure 28 on page 83. This creation is accomplished by invoking the lookup method, which takes the JNDI name of the enterprise bean in String form and returns a java.lang.Object object. The JNDI name specified in the deployment descriptor is used.

The example TransferApplication gets the JNDI name of the Transfer bean from the ClientResourceBundle class.

After an object is returned by the lookup method, the static method javax.rmi.PortableRemoteObject.narrow is used to obtain an EJB home object for the specified enterprise bean. The narrow method takes two parameters: the object to be narrowed and the class of the EJB home object to be returned by the narrow method. The object returned by the javax.rmi.PortableRemoteObject.narrow method is cast to the class associated with the home interface.

```
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(
                                        homeObject, TransferHome.class);
     } catch (Exception e)  { // Error getting the home interface
          ...
     }
     ...
     // Create a new Transfer object to return
     ...
     return transfer;
}
```

Figure 28. Code example: Creating the EJBHome object

Creating an EJB object

After the EJB home object is created, it is used to create the EJB object. Figure 29 on page 84 shows the code required to create the EJB object by using the EJB home object. A create method is invoked to create an EJB object or (for entity beans only) a finder method is invoked to find an existing EJB object. Because the Transfer bean is a stateless session bean, the only choice is the default create method.

```
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;
}
```

Figure 29. Code example: Creating the EJB object

Handling an invalid EJB object for a session bean

Because session beans are ephemeral, the client cannot depend on a session bean's EJB object to remain valid. A reference to an EJB object for a session bean can become invalid if the EJB server fails or is restarted or if the session bean times out due to inactivity. (The reference to an entity bean's EJB object is always valid until that object is removed.) Therefore, the client of a session bean must contain code to handle a situation in which the EJB object becomes invalid.

An EJB client can determine if an EJB object is valid by placing all method invocations that use the reference inside of a try/catch block that specifically catches the java.rmi.NoSuchObjectException, in addition to any other exceptions that the method needs to handle. The EJB client can then invoke the code to handle this exception.

You determine how to handle an invalid EJB object. The example TransferApplication creates a new Transfer EJB object if the one it is currently using becomes invalid.

The code to create a new EJB object when the old one becomes invalid is the same code used to create the original EJB object and is described in "Creating and getting a reference to a bean's EJB object" on page 79. For the example TransferApplication client, this code is contained in the createTransfer method.

Figure 30 on page 85 shows the code used to create the new EJB object in the getBalance method of the example TransferApplication. The getBalance

method contains the local boolean variable *sessionGood*, which is used to specify the validity of the EJB object referenced by the variable *ivjTransfer*. The *sessionGood* variable is also used to determine when to break out of the do-while loop.

The *sessionGood* variable is initialized to false because the *ivjTransfer* can reference an invalid EJB object when the getBalance method is called. If the *ivjTransfer* reference is valid, the TransferApplication invokes the Transfer bean's getBalance method and returns the balance. If the *ivjTransfer* reference is invalid, the NoSuchObjectException is caught, the TransferApplication's createTransfer method is called to create a new Transfer EJB object reference, and the *sessionGood* variable is set to false so that the do-while loop is repeated with the new valid EJB object. To prevent an infinite loop, the *sessionGood* variable is set to true when any other exception is thrown.

```
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;
}
```

Figure 30. Code example: Refreshing the EJB object reference for a session bean

Removing a bean's EJB object

When an EJB client no longer needs a stateful session EJB object, the EJB client should remove that object. Instances of stateful session beans have affinity to specific clients. They will remain in the container until they are explicitly removed by the client, or removed by the container when they time out. Meanwhile, the container might need to passivate inactive stateful session beans to disk. This requires overhead for the container and impacts performance of the application. If the passivated session bean is subsequently required by the application, the container activates it by restoring it from disk. By explicitly removing stateful session beans when finished with them, applications can decrease the need for passivation and minimize container overhead.

You remove entity EJB objects *only* when you want to remove the information in the data source with which the entity EJB object is associated.

To remove an EJB object, invoke the remove method on the object. As discussed in "Creating and getting a reference to a bean's EJB object" on page 79, the TransferApplication contains only one reference to a Transfer EJB object that is created when the application is initialized.

Figure 31 shows how the example Transfer EJB object is removed in the TransferApplication in the killApp method. To parallel the creation of the Transfer EJB object when the TransferApplication is initialized, the application removes the final EJB object associated with *ivjTransfer* reference right before closing the application's GUI window. The killApp method closes the window by invoking the dispose method on itself.

```
...
private void killApp() {
     try {
          ivjTransfer.remove();
          this.dispose();
          System.exit(0);     } catch (Throwable ivjExc) {
          ...
     }
}
```

Figure 31. Code example: Removing a session EJB object

Managing transactions in an EJB client

In general, it is practical to design your enterprise beans so that all transaction management is handled at the enterprise bean level. In a strict three-tier, distributed application, this is not always possible or even desirable. However, because the middle tier of an EJB application can include two subcomponents—session beans and entity beans—it is much easier to design

the transactional management completely within the application server tier. Of course, the resource manager tier must also be designed to support transactions.

Note: EJB clients that access entity beans with CMP that use Host On-Demand (HOD) or the External Call Interface (ECI) for CICS or IMS applications must begin a transaction before invoking a method on these entity beans. This restriction is required because these types of entity beans must use the Mandatory transaction attribute.

Nevertheless, it is still possible to program an EJB client (that is not an enterprise bean) to participate in transactions for those specialized situations that require it. To participate in a transaction, the EJB client must do the following:

1. Obtain a reference to the javax.transaction.UserTransaction interface by using JNDI as defined in the Java Transaction Application Programming Interface (JTA).
2. Use the object reference to invoke any of the following methods:
   - begin—Begins a transaction. This method takes no arguments and returns void.
   - commit—Attempts to commit a transaction; assuming that nothing causes the transaction to be rolled back, successful completion of this method commits the transaction. This method takes no arguments and returns void.
   - getStatus—Returns the status of the referenced transaction. This method takes no arguments and returns int; if no transaction is associated with the reference, STATUS_NO_TRANSACTION is returned. The following are the valid return values for this method:
     - STATUS_ACTIVE—Indicates that transaction processing is still in progress.
     - STATUS_COMMITTED—Indicates that a transaction has been committed and the effects of the transaction have been made permanent.
     - STATUS_COMMITTING—Indicates that a transaction is in the process of committing (that is, the transaction has started committing but has not completed the process).
     - STATUS_MARKED_ROLLBACK—Indicates that a transaction is marked to be rolled back.
     - STATUS_NO_TRANSACTION—Indicates that a transaction does not exist in the current transaction context.
     - STATUS_PREPARED—Indicates that a transaction has been prepared but not completed.

  - STATUS_PREPARING—Indicates that a transaction is in the process of preparing (that is, the transaction has started preparing but has not completed the process).
  - STATUS_ROLLEDBACK—Indicates that a transaction has been rolled back.
  - STATUS_ROLLING_BACK—Indicates that a transaction is in the process of rolling back (that is, the transaction has started rolling back but has not completed the process).
  - STATUS_UNKNOWN—Indicates that the status of a transaction is unknown.
- rollback—Rolls back the referenced transaction. This method takes no arguments and returns void.
- setRollbackOnly—Specifies that the only possible outcome of the transaction is for it to be rolled back. This method takes no arguments and returns void.
- setTransactionTimeout—Sets the timeout (in seconds) associated with the transaction. If some transaction participant has not specifically set this value, a default timeout is used. This method takes a number of seconds (as type int) and returns void.

Figure 32 on page 89 provides an example of an EJB client creating a reference to a UserTransaction object and then using that object to set the transaction timeout, begin a transaction, and attempt to commit the transaction. (The source code for this example is *not* available with the example code provided with this document.) Notice that the client does a simple type cast of the lookup result, rather than invoking a narrow method as required with other JNDI lookups. In both EJB server environments, the JNDI name of the UserTransaction interface is `java:comp/UserTransaction`.

```
...
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();
```

Figure 32. Code example: Managing transactions in an EJB client

Chapter 7. Developing servlets that use enterprise beans

A servlet is a Java application that enables users to access Web server functionality. To use servlets, a Web server is required. The WebSphere Application Server plugs into a number of commonly used Web servers. The IBM HTTP Server with the Advanced Application Server. For more information, consult the Advanced Edition InfoCenter.

Java servlets can be combined with enterprise beans to create powerful EJB applications. This chapter describes how to use enterprise beans within a servlet. The example CreateAccount servlet, which uses the example Account bean, is used to illustrate the concepts discussed in this chapter. The example servlet and enterprise bean discussed in this chapter are explained in "Information about the examples described in the documentation" on page 197.

An overview of standard servlet methods

Usually, a servlet is invoked from an HTML form on the user's browser. The first time the servlet is invoked, the servlet's init method is run to perform any initializations required at startup. For the first and all subsequent invocations of the servlet, the doGet method (or, alternatively, the doPost method) is run. Within the doGet method (or the doPost method), the servlet gets the information provided by the user on the HTML form and uses that information to perform work on the server and access server resources.

The servlet then prepares a response and sends the response back to the user. After a servlet is loaded, it can handle multiple simultaneous user requests. Multiple request threads can invoke the doGet (or doPost) method at the same time, so the servlet needs to be made thread safe.

When a servlet shuts down, the destroy method of the servlet is run in order to perform any needed shutdown processing.

Writing an HTML page that embeds a servlet

Figure 33 on page 92 shows the HTML file (named create.html) used to invoke the CreateAccount servlet. The HTML form is used to specify the account number for the new account, its type (checking or savings), and its initial balance. The request is passed to the doGet method of the servlet, where the servlet is identified with its full Java package name, as shown in the example.

```
<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>
```

Figure 33. Code example: Content of the create.html file used to access the CreateAccount servlet

The HTML response from the servlet is designed to produce a display identical to create.html, enabling the user to continue creating new accounts. Figure 34 on page 93 shows what create.html looks like on a browser.

Figure 34. The initial form and output of the CreateAccount servlet

Developing the servlet

This section discusses the basic code required by a servlet that interacts with an enterprise bean. Figure 35 on page 94 shows the basic outline of the code that makes up the CreateAccount servlet. As shown in the example, the CreateAccount servlet extends the javax.servlet.http.HttpServlet class and implements an init method and a doGet method.

```
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. ---
      ...
}
```

Figure 35. Code example: The CreateAccount class

The servlet's instance variables

Figure 36 on page 95 shows the instance variables used in the CreateAccount servlet. The *nameService*, *accountName*, and *providerUrl* variables are used to specify the property values required during JNDI lookup. These values are obtained from the ClientResourceBundle class as described in "Creating and getting a reference to a bean's EJB object" on page 79.

The CreateAccount class also initializes the string constants that are used to create the HTML response sent back to the user. (Only three of these variables are shown, but there are many of them). The init method in the CreateAccount servlet provides a way to read strings from a resource bundle to override these US English defaults in order to provide a response in a different national language.

The instance variable *accountHome* is used by all client requests to create a new Account bean instance. The *accountHome* variable is initialized in the init method as shown in Figure 36.

```
...
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;
     ...

}
```

Figure 36. Code example: The instance variables of the CreateAccount class

The servlet's init method

The init method of the CreateAccount servlet is shown in Figure 37 on page 96. The init method is run once, the first time a request is processed by the servlet, after the servlet is started. Typically, the init method is used to do any one-time initializations for a servlet. For example, the default US English strings used in preparing the HTML response can be replaced with another national language.

The init method is also the best place to initialize the value of references to the home interface of any enterprise beans used by the servlet. In the CreateAccount's init method, the *accountHome* variable is initialized to reference the EJB home object of the Account bean.

As in other types of EJB clients, the properties required to do a JNDI lookup are specific to the EJB implementation. Therefore, these properties are externalized in a properties file or a resource bundle class. For more information on these properties, see "Creating and getting a reference to a bean's EJB object" on page 79.

Note that in the CreateAccount servlet, a HashTable object is used to store the properties required to do a JNDI lookup whereas a Properties object is used in

the TransferApplication. Both of these classes are valid for storing these properties.

```
// 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(
                         homeObject, AccountHome.class);
     }
     // Determine cause of failure.
     catch (NamingException e) {
          ...
      }
      catch (Exception e) {
          ...
     }
}
```

Figure 37. Code example: The init method of the CreateAccount servlet

Note: Although the init method is a good place to obtain references to EJB home objects, it is not a good place to create enterprise beans or access

other beans that might be protected with WebSphere security. Depending upon the authorization policy on the protected objects, creating or accessing these objects from within the init method could fail for authentication or authorization reasons because they were not accessed with the proper security credentials.

Creating or accessing protected objects should be done after the init method, in one of the servlet's do*XXX* methods.

The servlet's doGet method

The doGet method is invoked for every servlet request. In the CreateAccount servlet, the method does the following tasks to manage user input. These tasks are fairly standard for this method:

- Read the user input from the HTML form and decide if the input is valid—for example, whether the user entered a valid number for an initial balance.
- Perform the initializations required for each request.

Figure 38 on page 98 shows the parts of the doGet method that handle user input. Note that the *req* variable is used to read the user input from the HTML form. The *req* variable is a javax.servlet.http.HttpServletRequest object passed as one of the arguments to the doGet method.

```
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. ---
     ...
}
```

Figure 38. Code example: The doGet method of the CreateAccount servlet

Creating an enterprise bean

If the user input is valid, the doGet method attempts to create a new account based on the user input as shown in Figure 39 on page 99. Besides the

initialization of the home object reference in the init method, this is the only other piece of code that is specific to the use of enterprise beans in a servlet.

```
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. ---
     ...
 }
```

Figure 39. Code example: Creating an enterprise bean in the doGet method

Determining the content of the user response

Next, the doGet method prepares a response message to be sent to the user. There are three possible responses:

- The user input was not valid.
- The user input was valid, but the account was not created for some reason.
- The account was created successfully. If the previous two errors do not occur, this response is prepared.

Figure 40 on page 100 shows the code used by the servlet to determine which response to send to the user. If no errors are encountered, then the response indicates success.

```
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. ---
     ...
 }
```

Figure 40. Code example: Determining a user response in the doGet method

Sending the user response

With the type of response determined, the doGet method then prepares the full HTML response and sends it to the user's browser, incorporating the appropriate message. Relevant parts of the full HTML response are shown in Figure 41 on page 101.

The *res* variable is used to pass the response back to the user. This variable is an HttpServletResponse object passed as an argument to the doGet method.

The response code shown here mixes both display (HTML) and content in one servlet. You can separate the display and the content by using JavaServer Pages (JSP). A JSP allows the display and content to be developed and maintained separately.

```
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>");
}
```

Figure 41. Code example: Responding to the user in the doGet method

Threading issues

Except for the instance variable required to get a reference to the Account bean's home interface and to support multiple languages (which remain unchanged for all user requests), all other variables used in the CreateAccount servlet are local to the doGet method. Each request thread has its own set of local variables, so the servlet can handle simultaneous user requests.

As a result, the CreateAccount servlet is thread safe. By taking a similar approach to servlet design, you can also make your servlets thread safe.

Chapter 8. More-advanced programming concepts for enterprise beans

This chapter discusses some of the more advanced programming concepts associated with developing and using enterprise beans. It includes information on developing entity beans with bean-managed persistence (BMP), writing the code required by a BMP bean to interact with a database, and developing session beans that directly participate in transactions.

Developing entity beans with BMP

In an entity bean with container-managed persistence (CMP), the container handles the interactions between the enterprise bean and the data source. In an entity bean with bean-managed persistence (BMP), the enterprise bean must contain all of the code required for the interactions between the enterprise bean and the data source. For this reason, developing an entity bean with CMP is simpler than developing an entity bean with BMP. However, you must use BMP if any of the following is true about an entity bean:

- The bean's persistent data is stored in more than one data source.
- The bean's persistent data is stored in a data source that is not supported by the EJB server that you are using.

This section examines the development of entity beans with BMP. For information on the tasks required to develop an entity bean with CMP, see "Developing entity beans with CMP" on page 33.

Every entity bean must contain the following basic parts:

- The enterprise bean class. For more information, see "Writing the enterprise bean class (entity with BMP)" on page 104.
- The enterprise bean's home interface. For more information, see "Writing the home interface (entity with BMP)" on page 114.
- The enterprise bean's remote interface. For more information, see "Writing the remote interface (entity with BMP)" on page 117.

In an entity bean with BMP, you can create your own primary key class or use an existing class for the primary key. For more information, see "Writing or selecting the primary key class (entity with BMP)" on page 118.

Writing the enterprise bean class (entity with BMP)

In an entity bean with BMP, the bean class defines and implements the business methods of the enterprise bean, defines and implements the methods used to create instances of the enterprise bean, and implements the methods invoked by the container to move the bean through different stages in the bean's life cycle.

By convention, the enterprise bean class is named *Name*Bean, where *Name* is the name you assign to the enterprise bean. The enterprise bean class for the example AccountBM enterprise bean is named AccountBMBean.

Every entity bean class with BMP must meet the following requirements:

- It must be public, it must *not* be abstract, and it must implement the javax.ejb.EntityBean interface. For more information, see "Implementing the EntityBean interface" on page 112.
- It must define instance variables that correspond to persistent data associated with the enterprise bean. For more information, see "Defining instance variables" on page 105.
- It must implement the business methods used to access and manipulate the data associated with the enterprise bean. For more information, see "Implementing the business methods" on page 107.
- It must contain code for getting connections to, interacting with, and releasing connections to the data source (or sources) used to store the persistent data. For more information, see "Using a database with a BMP entity bean" on page 119.
- It must define and implement an ejbCreate method for each way in which the enterprise bean can be instantiated. It can, but is not required to, define and implement a corresponding ejbPostCreate method for each ejbCreate method. For more information, see "Implementing the ejbCreate and ejbPostCreate methods" on page 107.
- It must implement the ejbFindByPrimaryKey method that takes a primary key and determines if it is valid and unique. It can also define and implement additional finder methods as required. For more information, see "Implementing the ejbFindByPrimaryKey and other ejbFind methods" on page 109.

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

Figure 42 on page 105 shows the import statements and class declaration for the example AccountBM enterprise bean.

```
...
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 {
   ...
}
```

Figure 42. Code example: The AccountBMBean class

Defining instance variables

An entity bean class can contain both persistent and nonpersistent instance variables; however, static variables are not supported in enterprise beans unless they are also final (that is, they are constants). Persistent variables are stored in a database. Unlike the persistent variables in a CMP entity bean class, the persistent variables in a BMP entity bean class can be private.

Nonpersistent variables are *not* stored in a database and are temporary. Nonpersistent variables must be used with caution and must not be used to maintain the state of an EJB client between method invocations. This restriction is necessary because nonpersistent variables cannot be relied on to remain the same between method invocations outside of a transaction because other EJB clients can change these variables or they can be lost when the entity bean is passivated.

The AccountBMBean class contains three instance variables that represent persistent data associated with the AccountBM enterprise bean:

- *accountId*, which identifies the account ID associated with an account
- *type*, which identifies the account type as either savings (1) or checking (2)
- *balance*, which identifies the current balance of the account

The AccountBMBean class contains several nonpersistent instance variables including the following:

- *entityContext*, which identifies the entity context of each instance of an AccountBM enterprise bean. The entity context can be used to get a reference to the EJB object currently associated with the bean instance and to get the primary key object associated with that EJB object.
- *jdbcUrl*, which encapsulates the database universal resource locator (URL) used to connect to the data source. This variable must have the following format: *dbAPI*:*databaseType*:*databaseName*. For example, to specify a database

named sample in an IBM DB2 database with the Java Database Connectivity (JDBC) API, the argument is `jdbc:db2:sample`.

- *driverName*, which encapsulates the database driver class required to connect to the database.
- *DBLogin*, which identifies the database user ID required to connect to the database.
- *DBPassword*, which identifies password for the specified user ID (*DBLogin*) required to connect to the database.
- *tableName*, which identifies the database table name in which the bean's persistent data is stored.
- *jdbcConn*, which encapsulates a Java Database Connectivity (JDBC) connection to a data source within a java.sql.Connection object.

```
...
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;
      ...
}
```

Figure 43. Code example: The instance variables of the AccountBMBean class

To make the AccountBM bean more portable between databases and database drivers, the database-specific variables (*jdbcUrl*, *driverName*, *DBLogin*, *DBPassword*, and *tableName*) are set by retrieving corresponding environment variables contained in the enterprise bean. The values of these variables are retrieved by the getEnvProps method, which is implemented in the AccountBMBean class and invoked when the setEntityContext method is called. For more information, see "Managing database connections in the EJB server environment" on page 120.

Although Figure 43 shows database access compatible with version 1.0 of the JDBC specification, you can also perform database accesses that are compatible with version 2.0 of the JDBC specification. An administrator binds a javax.sql.DataSource reference (which encapsulates the information that was

formerly stored in the jdbcURL and driverName variables) into the JNDI namespace. The entity bean with BMP does the following to get a java.sql.Connection:

```
DataSource ds = (dataSource)initialContext.lookup("java:comp/env/jdbc/MyDataSource");
Connection con = ds.getConnection();
```

where *MyDataSource* is the name the administrator assigned to the datasource.

Implementing the business methods

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

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

There is no difference between the business methods defined in the AccountBMBean bean class and those defined in the CMP bean class AccountBean shown in Figure 10 on page 38.

Implementing the ejbCreate and ejbPostCreate methods

You must define and implement an ejbCreate method for each way in which you want a new instance of an enterprise bean to be created. For each ejbCreate method, you can also define a corresponding ejbPostCreate method. Each ejbCreate method must correspond to a create method in the EJB home interface.

Like the business methods of the bean class, the ejbCreate and ejbPostCreate methods cannot be invoked directly by the client. Instead, the client invokes the create method of the enterprise bean's home interface by using the EJB home object, and the container invokes the ejbCreate method followed by the ejbPostCreate method.

Unlike the method in an entity bean with CMP, the ejbCreate method in an entity bean with BMP must contain all of the code required to insert the bean's persistent data into the data source. This requirement means that the ejbCreate method must get a connection to the data source (if one is not already available to the bean instance) and insert the values of the bean's variables into the appropriate fields in the data source.

Each ejbCreate method in an entity bean with BMP must meet the following requirements:

- It must be public and return the bean's primary key class.
- Its arguments and return type must be valid for Java remote method invocation (RMI).
- It must contain the code required to insert the values of the persistent variables into the data source. For more information, see "Using a database with a BMP entity bean" on page 119.

Each ejbPostCreate method must be public, return void, and have the same arguments as the matching ejbCreate method.

If necessary, both the ejbCreate method and the ejbPostCreate method can throw the java.rmi.RemoteException exception, the javax.ejb.CreateException exception, the javax.ejb.DuplicateKeyException exception, and any user-defined exceptions.

Figure 44 on page 109 shows the two ejbCreate methods required by the example AccountBMBean bean class. No ejbPostCreate methods are required.

As in the AccountBean class, the first ejbCreate method calls the second ejbCreate method; the latter handles all of the interaction with the data source. The second method initializes the bean's instance variables and then ensures that it has a valid connection to the data source by invoking the checkConnection method. The method then creates, prepares, and executes an SQL INSERT call on the data source. If the INSERT call is executed correctly, and only one row is inserted into the data source, the method returns an object of the bean's primary key class.

```
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;
}
```

Figure 44. Code example: The ejbCreate methods of the AccountBMBean class

Implementing the ejbFindByPrimaryKey and other ejbFind methods

At a minimum, each entity bean with BMP must define and implement the ejbFindByPrimaryKey method that takes a primary key and determines if it is valid and unique for an instance of an enterprise bean; if the primary key is valid and unique, it returns the primary key. An entity bean can also define and implement other finder methods to find enterprise bean instances. All finder methods can throw the javax.ejb.FinderException exception to indicate an application-level error. Finder methods designed to find a single bean can also throw the javax.ejb.ObjectNotFoundException exception, a subclass of the FinderException class. Finder methods designed to return multiple beans should not use the ObjectNotFoundException to indicate that no suitable beans were found; instead, such methods should return empty return values. Throwing the java.rmi.RemoteException exception is deprecated; see "Standard application exceptions for entity beans" on page 38 for more information.

Like the business methods of the bean class, the ejbFind methods cannot be invoked directly by the client. Instead, the client invokes a finder method on the enterprise bean's home interface by using the EJB home object, and the container invokes the corresponding ejbFind method. The container invokes an ejbFind method by using a generic instance of that entity bean in the pooled state.

Because the container uses an instance of an entity bean in the pooled state to invoke an ejbFind method, the method must do the following:

1. Get a connection to the data source (or sources).
2. Query the data source for records that match specifications of the finder method.
3. Drop the connection to the data source (or sources).

For more information on these data source tasks, see "Using a database with a BMP entity bean" on page 119.

Figure 45 on page 111 shows the ejbFindByPrimaryKey method of the example AccountBMBean class. The ejbFindByPrimaryKey method gets a connection to its data source by calling the makeConnection method shown in Figure 45 on page 111. It then creates and invokes an SQL SELECT statement on the data source by using the specified primary key.

If one and only one record is found, the method returns the primary key passed to it in the argument. If no records are found or multiple records are found, the method throws the FinderException. Before determining whether to return the primary key or throw the FinderException, the method drops its connection to the data source by calling the dropConnection method described in "Using a database with a BMP entity bean" on page 119.

```
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));
      }
}
```

Figure 45. Code example: The ejbFindByPrimaryKey method of the AccountBMBean class

Figure 46 on page 112 shows the ejbFindLargeAccounts method of the example AccountBMBean class. The ejbFindLargeAccounts method also gets a connection to its data source by calling the makeConnection method and drops the connection by using the dropConnection method. The SQL SELECT statement is also very similar to that used by the ejbFindByPrimaryKey method. (For more information on these data source tasks and methods, see "Using a database with a BMP entity bean" on page 119.)

While the ejbFindByPrimaryKey method needs to return only one primary key, the ejbFindLargeAccounts method can be expected to return zero or more

primary keys in an Enumeration object. To return an enumeration of primary keys, the ejbFindLargeAccounts method does the following:

1. It uses a while loop to examine the result set (*sqlResults*) returned by the executeQuery method.
2. It inserts each primary key in the result set into a hash table named *resultTable* by wrapping the returned account ID in a Long object and then in an AccountBMKey object. (The Long object, *memberId*, is used as the hash table's index.)
3. It invokes the elements method on the hash table to obtain the enumeration of primary keys, which it then returns.

```
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();
     }
}
```

Figure 46. Code example: The ejbFindLargeAccounts method of the AccountBMBean class

Implementing the EntityBean interface

Each entity bean class must implement the methods inherited from the javax.ejb.EntityBean interface. The container invokes these methods to move the bean through different stages in the bean's life cycle. Unlike an entity bean

with CMP, in an entity bean with BMP, these methods must contain all of the code for the required interaction with the data source (or sources) used by the bean to store its persistent data.

- ejbActivate—This method is invoked by the container when the container selects an entity bean instance from the instance pool and assigns that instance to a specific existing EJB object. This method must contain the code required to activate the enterprise bean instance by getting a connection to the data source and using the bean's javax.ejb.EntityContext class to obtain the primary key in the corresponding EJB object.

  In the example AccountBMBean class, the ejbActivate method obtains the bean instance's account ID, sets the value of the *accountId* variable, and invokes the checkConnection method to ensure that it has a valid connection to the data source.
- ejbLoad—This method is invoked by the container to synchronize an entity bean's persistent variables with the corresponding data in the data source. (That is, the values of the fields in the data source are loaded into the persistent variables in the corresponding enterprise bean instance.) This method must contain the code required to load the values from the data source and assign those values to the bean's instance variables.

  In the example AccountBMBean class, the ejbLoad method obtains the bean instance's account ID, sets the value of the *accountId* variable, invokes the checkConnection method to ensure that it has a valid connection to the data source, constructs and executes an SQL SELECT statement, and sets the values of the *type* and *balance* variables to match the values retrieved from the data source.
- ejbPassivate—This method is invoked by the container to disassociate an entity bean instance from its EJB object and place the enterprise bean instance in the instance pool. This method must contain the code required to "passivate" or deactivate an enterprise bean instance. Usually, this passivation simply means dropping the connection to the data source.

  In the example AccountBMBean class, the ejbPassivate method invokes the dropConnection method to drop the connection to the data source.
- ejbRemove—This method is invoked by the container when a client invokes the remove method inherited by the enterprise bean's home interface (from the javax.ejb.EJBHome interface) or remote interface (from the javax.ejb.EJBObject interface). This method must contain the code required to remove an enterprise bean's persistent data from the data source. This method can throw the javax.ejb.RemoveException exception if removal of an enterprise bean instance is not permitted. Usually, removal involves deleting the bean instance's data from the data source and then dropping the bean instance's connection to the data source.

  In the example AccountBMBean class, the ejbRemove method invokes the checkConnection method to ensure that it has a valid connection to the data

source, constructs and executes an SQL DELETE statement, and invokes the dropConnection method to drop the connection to the data source.

- setEntityContext—This method is invoked by the container to pass a reference to the javax.ejb.EntityContext interface to an enterprise bean instance. This method must contain any code required to store a reference to a context.

  In the example AccountBMBean class, the setEntityContext method sets the value of the *entityContext* variable to the value passed to it by the container.
- ejbStore—This method is invoked by the container when the container needs to synchronize the data in the data source with the values of the persistent variables in an enterprise bean instance. (That is, the values of the variables in the enterprise bean instance are copied to the data source, overwriting the previous values.) This method must contain the code required to overwrite the data in the data source with the corresponding values in the enterprise bean instance.

  In the example AccountBMBean class, the ejbStore method invokes the checkConnection method to ensure that it has a valid connection to the data source and constructs and executes an SQL UPDATE statement.
- unsetEntityContext—This method is invoked by the container, before an enterprise bean instance is removed, to free up any resources associated with the enterprise bean instance. This is the last method called prior to removing an enterprise bean instance.

  In the example AccountBMBean class, the unsetEntityContext method sets the value of the *entityContext* variable to null.

Writing the home interface (entity with BMP)

An entity bean's home interface defines the methods used by EJB clients to create new instances of the bean, find and remove existing instances, and obtain metadata about an instance. The home interface is defined by the enterprise bean developer and implemented in the EJB home class created by the container during enterprise bean deployment. The container makes the home interface accessible to clients through the Java Naming and Directory Interface (JNDI).

By convention, the home interface is named *Name*Home, where *Name* is the name you assign to the enterprise bean. For example, the AccountBM enterprise bean's home interface is named AccountBMHome.

Every home interface for an entity bean with BMP must meet the following requirements:

- It must extend the javax.ejb.EJBHome interface. The home interface inherits several methods from the javax.ejb.EJBHome interface. See "The javax.ejb.EJBHome interface" on page 65 for information on these methods.

- Each method in the interface must be either a create method, which corresponds to an ejbCreate method (and possibly an ejbPostCreate method) in the enterprise bean class, or a finder method, which corresponds to an ejbFind method in the enterprise bean class. For more information, see "Defining create methods" and "Defining finder methods" on page 116.
- The parameters and return value of each method defined in the home interface must be valid for Java RMI. For more information, see "The java.io.Serializable and java.rmi.Remote interfaces" on page 66. In addition, each method's throws clause must include the java.rmi.RemoteException exception class.

Figure 47 shows the relevant parts of the definition of the home interface (AccountBMHome) for the example AccountBM bean. This interface defines two abstract create methods: the first creates an AccountBM object by using an associated AccountBMKey object, the second creates an AccountBM object by using an associated AccountBMKey object and specifying an account type and an initial balance. The interface defines the required findByPrimaryKey method and the findLargeAccounts method.

```
...
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;
}
```

Figure 47. Code example: The AccountBMHome home interface

Defining create methods

A create method is used by a client to create an enterprise bean instance and insert the data associated with that instance into the data source. Each create method must be named create and it must have the same number and types of arguments as a corresponding ejbCreate method in the enterprise bean class. (The ejbCreate method can itself have a corresponding ejbPostCreate method.) The return types of the create method and its corresponding ejbCreate method are always different.

Each create method must meet the following requirements:

- It must be named create.
- It must return the type of the enterprise bean's remote interface. For example, the return type for the create methods in the AccountBMHome interface is AccountBM (as shown in Figure 13 on page 44).
- It must have a throws clause that includes the java.rmi.RemoteException exception, the javax.ejb.CreateException exception, and all of the exceptions defined in the throws clause of the corresponding ejbCreate and ejbPostCreate methods.

Defining finder methods

A finder method is used to find one or more existing entity EJB objects. Each finder method must be named find*Name*, where *Name* further describes the finder method's purpose.

At a minimum, each home interface must define the findByPrimaryKey method that enables a client to locate an EJB object by using the primary key only. The findByPrimaryKey method has one argument, an object of the bean's primary key class, and returns the type of the bean's remote interface.

Every other finder method must meet the following requirements:

- It must return the type of the enterprise bean's remote interface, the java.util.Enumeration interface, or the java.util.Collection interface (when a finder method can return more than one EJB object or an EJB collection).
- It must have a throws clause that includes the java.rmi.RemoteException and javax.ejb.FinderException exception classes.

Although every entity bean must contain only the default finder method, you can write additional ones if needed. For example, the AccountBM bean's home interface defines the findLargeAccounts method to find objects that encapsulate accounts with balances of more than a specified dollar amount, as shown in Figure 47 on page 115. Because this finder method can be expected to return a reference to more than one EJB object, its return type is java.util.Enumeration.

Unlike the implementation in an entity bean with CMP, in an entity bean with BMP, the bean developer must fully implement the ejbFindByPrimaryKey method that corresponds to the findByPrimaryKey method. In addition, the bean developer must write each additional ejbFind method corresponding to the finder methods defined in the home interface. The implementation of the ejbFind methods in the AccountBMBean class is discussed in "Implementing the ejbFindByPrimaryKey and other ejbFind methods" on page 109.

Writing the remote interface (entity with BMP)

An entity bean's remote interface provides access to the business methods available in the bean class. It also provides methods to remove an EJB object associated with a bean instance and to obtain the bean instance's home interface, object handle, and primary key. The remote interface is defined by the EJB developer and implemented in the EJB object class created by the container during enterprise bean deployment.

By convention, the remote interface is named *Name*, where *Name* is the name you assign to the enterprise bean. For example, the AccountBM enterprise bean's remote interface is named AccountBM.

Every remote interface must meet the following requirements:

- It must extend the javax.ejb.EJBObject interface. The remote interface inherits several methods from the javax.ejb.EJBObject interface. See "Methods inherited from javax.ejb.EJBObject" on page 65 for information on these methods.
- It must define a corresponding business method for every business method implemented in the enterprise bean class.
- The parameters and return value of each method defined in the interface must be valid for Java RMI. For more information, see "The java.io.Serializable and java.rmi.Remote interfaces" on page 66.
- Each method's throws clause must include the java.rmi.RemoteException exception class.

Figure 48 on page 118 shows the relevant parts of the definition of the remote interface (AccountBM) for the example AccountBM enterprise bean. This interface defines four methods for displaying and manipulating the account balance that exactly match the business methods implemented in the AccountBMBean class.

All of the business methods throw the java.rmi.RemoteException exception class. In addition, the subtract method must throw the user-defined exception com.ibm.ejs.doc.account.InsufficientFundsException because the corresponding method in the bean class throws this exception. Furthermore, any client that calls this method must either handle the exception or pass it on by throwing it.

```
...
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;
}
```

Figure 48. Code example: The AccountBM remote interface

Writing or selecting the primary key class (entity with BMP)

Every entity EJB object has a unique identity within a container that is defined by a combination of the object's home interface name and its primary key, the latter of which is assigned to the object at creation. If two EJB objects have the same identity, they are considered identical.

The primary key class is used to encapsulate an EJB object's primary key. In an entity bean (with BMP or CMP), you can write a distinct primary key class or you can use an existing class as the primary key class, as long as that class is serializable. For more information, see "The java.io.Serializable and java.rmi.Remote interfaces" on page 66.

The example AccountBM bean uses a primary key class that is identical to the AccountKey class contained in the Account bean shown in Figure 16 on page 49, with the exception that the key class is named AccountBMKey.

Note: The primary key class of an entity bean with BMP must implement the hashCode and equals method. In addition, the variables that make up the primary key must be public.

The java.lang.Long class is also a good candidate for a primary key class for the AccountBM bean.

Using a database with a BMP entity bean

In an entity bean with BMP, each ejbFind method and all of the life cycle methods (ejbActivate, ejbCreate, ejbLoad, ejbPassivate, and ejbStore) must interact with the data source (or sources) used by the bean to maintain its persistent data. To interact with a supported database, the BMP entity bean must contain the code to manage database connections and to manipulate the data in the database.

The EJB server uses a set of specialized beans to encapsulate information about databases and an IBM-specific interface to JDBC to allow entity bean interaction with a connection manager. For more information, see "Managing database connections in the EJB server environment" on page 120

In general, there are three approaches to getting and releasing connections to databases:

- The bean can get a database connection in the setEntityContext method and release it in the unsetEntityContext method. This approach is the easiest for the enterprise bean developer to implement. However, without a connection manager, this approach is not viable because under it bean instances hold onto database connections even when they are not in use (that is, when the bean instance is passivated). Even with a connection manager, this approach does not scale well.
- The bean can get a database connection in the ejbActivate and ejbCreate methods, get and release a database connection in each ejbFind method, and release the database connection in the ejbPassivate and ejbRemove methods. This approach is somewhat more difficult to implement, but it ensures that only those bean instances that are activated have connections to the database.
- The bean can get and release a database connection in each method that requires a connection: ejbActivate, ejbCreate, ejbFind, ejbLoad, and ejbStore. This approach is more difficult to implement than the first approach, but is no more difficult than the second approach. This approach is the most efficient in terms of connection use and also the most scalable.

The example AccountBM bean, uses the second approach described in the preceding text. The AccountBMBean class contains two methods for making a connection to the DB2 database, checkConnection and makeConnection, and one method to drop connections: dropConnection.The code required to make the AccountBM bean work with the connection manager is shown in "Managing database connections in the EJB server environment" on page 120

The code required to manipulate data in a database is described in "Manipulating data in a database" on page 123.

Managing database connections in the EJB server environment

In the EJB server environment, the administrator creates a specialized set of entity beans that encapsulate information about the database and the database driver. These specialized entity beans are created by using the WebSphere Administrative Console.

An entity bean that requires access to a database must use JNDI to create a reference to an EJB object associated with the right database bean instance. The entity bean can then use the IBM-specific interface (named com.ibm.db2.jdbc.app.stdext.javax.sql.DataSource) to get and release connections to the database.

The DataSource interface enables the entity bean to transparently interact with the connection manager of the EJB server. The connection manager creates a pool of database connections, which are allocated and deallocated to individual entity beans as needed.

Getting an EJB object reference to a data source bean instance

Before a BMP entity bean can get a connection to a database, the entity bean must perform a JNDI lookup on the data source entity bean associated with the database used to store the BMP entity bean's persistent data. Figure 49 on page 121 shows the code required to create an InitialContext object and then get an EJB object reference to a database bean instance. The JNDI name of the database bean is defined by the administrator; it is recommended that the JNDI naming convention be followed when defining this name. The value of the required database-specific variables are obtained by the getEnvProps method, which accesses the corresponding environment variables from the deployed enterprise bean.

Because the connection manager creates and drops the actual database connections and simply allocates and deallocates these connections as required, there is no need for the BMP entity bean to load and register the database driver. Therefore, there is no need to define the *driverName* and *jdbcUrl* variables discussed in "Defining instance variables" on page 105.

```
...
# 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) {
               ...
           }
     }
...
```

Figure 49. Code example: Getting an EJB object reference to a data source bean instance in the setEntityContext method (rewritten to use DataSource)

Allocating and deallocating a connection to a database

After creating an EJB object reference for the appropriate database bean instance, that object reference is used to get and release connections to the corresponding database. Unlike when using the DriverManager interface, when using the DataSource interface, the BMP entity bean does not really create and close data connections; instead, the connection manager allocates and deallocates connections as required by the entity bean. Nevertheless, a BMP entity bean must still contain code to send allocation and deallocation requests to the connection manager.

In the AccountBMBean class, the checkConnection method is called within other bean class methods that require a database connection, but for which it can be assumed that a connection already exists. This method checks to make sure that the connection is still available by checking if the *jdbcConn* variable is set to null. If the variable is null, the makeConnection method is invoked to get the connection (that is a connection allocation request is sent to the connection manager).

The makeConnection method is invoked when a database connection is required. It invokes the getConnection method on the data source object. The getConnection method is overloaded: it can take either a user ID and password or no arguments, in which case the user ID and password are implicitly set to null (this version is used in Figure 50 on page 122).

```
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
          ...
     }
}
```

Figure 50. Code example: The checkConnection and makeConnection methods of the AccountBMBean class (rewritten to use DataSource)

Entity beans with BMP must also release database connections when a particular bean instance no longer requires it (that is, they must send a deallocation request to the connection manager). The AccountBMBean class contains a dropConnection method to handle this task. To release the database connection, the dropConnection method does the following (as shown in Figure 51):

1. Invokes the close method on the connection object to tell the connection manager that the connection is no longer needed.
2. Sets the connection object reference to null.

Putting the close method inside a try/catch/finally block ensures that the connection object reference is always set to null even if the close method fails for some reason. Nothing is done in the catch block because the connection manager must clean up idle connections; this is not the job of the enterprise bean code.

```
private void dropConnection() {
     try {
          // Close the connection
          jdbcConn.close();
     catch (SQLException ex) {
          // Do nothing
      } finally {
          jdbcConn = null;
     }
}
```

Figure 51. Code example: The dropConnection method of the AccountBMBean class (rewritten to use DataSource)

Manipulating data in a database

After an instance of a BMP entity bean obtains a connection to its database, it can read and write data. The AccountBMBean class communicates with the DB2 database by constructing and executing Java Structured Query Language (JSQL) calls by using the java.sql.PreparedStatement interface.

As shown in Figure 52, the SQL call is created as a String (*sqlString*). The String variable is passed to the java.sql.Connection.prepareStatement method; and the values of each variable in the SQL call are set by using the various setter methods of the PreparedStatement class. (The variables are substituted for the question marks in the *sqlString* variable.) Invoking the PreparedStatement.executeUpdate method executes the SQL call.

```
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
          ...
     }
     ...
}
```

Figure 52. Code example: Constructing and executing an SQL update call in an ejbCreate method

The executeUpdate method is called to insert or update data in a database; the executeQuery method is called to get data from a database. When data is retrieved from a database, the executeQuery method returns a java.sql.ResultSet object, which must be examined and manipulated using the methods of that class.

Note: To improve scalability and performance, you do not need to call PreparedStatement for each database update. Instead, you can cache the results of the first PreparedStatement call.

Figure 53 provides an example of how the data in a ResultSet is manipulated in the ejbLoad method of the AccountBMBean class.

```
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.
          ...
     }
}
```

Figure 53. Code example: Manipulating a ResultSet object in the ejbLoad method

Using bean-managed transactions

In most situations, an enterprise bean can depend on the container to manage transactions within the bean. In these situations, all you need to do is set the appropriate transactional properties in the deployment descriptor as described in "Chapter 5. Enabling transactions and security in enterprise beans" on page 69.

Under certain circumstances, however, it can be necessary to have an enterprise bean participate directly in transactions. By setting the *transaction* attribute in an enterprise bean's deployment descriptor to BeanManaged, you indicate to the container that the bean is an active participant in transactions.

Note: The value BeanManaged is not a valid value for the *transaction* deployment descriptor attribute in entity beans. In other words, entity beans cannot manage transactions.

When writing the code required by an enterprise bean to manage its own transactions, remember the following basic rules:

- An instance of a stateless session bean *cannot* reuse the same transaction context across multiple methods called by an EJB client. Therefore, it is recommended that the transaction context be a local variable to each method that requires a transaction context.
- An instance of a stateful session bean can reuse the same transaction context across multiple methods called by an EJB client. Therefore, make

the transaction context an instance variable or a local method variable at your discretion. (When a transaction spans multiple methods, you can use the javax.ejb.SessionSynchronization interface to synchronize the conversational state with the transaction.)

Figure 54 on page 126 shows the standard code required to obtain an object encapsulating the transaction context. There are three basics steps involved:

1. The enterprise bean class must set the value of the javax.ejb.SessionContext object reference in the setSessionContext method.
2. A javax.transaction.UserTransaction object is created by calling the getUserTransaction method on the SessionContext object reference.
3. The UserTransaction object is used to participate in the transaction by calling transaction methods such as begin and commit as needed. If a enterprise bean begins a transaction, it must also complete that transaction either by invoking the commit method or the rollback method.

   Note: In both EJB servers, the getUserTransaction method of the javax.ejb.EJBContext interface (which is inherited by the SessionContext interface) returns an object of type javax.transaction.UserTransaction rather than type javax.jts.UserTransaction. While this is a deviation from the 1.0 version of the EJB Specification, the 1.1 version of the EJB Specification requires that the getUserTransaction method return an object of type javax.transaction.UserTransaction and drops the requirement to return objects of type javax.jts.UserTransaction.

```
...
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();
          ...
     }
     ...
}
```

Figure 54. Code example: Getting an object that encapsulates a transaction context

The following methods are available with the UserTransaction interface:

- begin—Begins a transaction. This method takes no arguments and returns void.
- commit—Attempts to commit a transaction; assuming that nothing causes the transaction to be rolled back, successful completion of this method commits the transaction. This method takes no arguments and returns void.
- getStatus—Returns the status of the referenced transaction. This method takes no arguments and returns int; if no transaction is associated with the reference, STATUS_NO_TRANSACTION is returned. The following are the valid return values for this method:
  - STATUS_ACTIVE—Indicates that transaction processing is still in progress.
  - STATUS_COMMITTED—Indicates that a transaction has been committed and the effects of the transaction have been made permanent.
  - STATUS_COMMITTING—Indicates that a transaction is in the process of committing (that is, the transaction has started committing but has not completed the process).
  - STATUS_MARKED_ROLLBACK—Indicates that a transaction is marked to be rolled back.
  - STATUS_NO_TRANSACTION—Indicates that a transaction does not exist in the current transaction context.

  - STATUS_PREPARED—Indicates that a transaction has been prepared but not completed.
  - STATUS_PREPARING—Indicates that a transaction is in the process of preparing (that is, the transaction has started preparing but has not completed the process).
  - STATUS_ROLLEDBACK—Indicates that a transaction has been rolled back.
  - STATUS_ROLLING_BACK—Indicates that a transaction is in the process of rolling back (that is, the transaction has started rolling back but has not completed the process).
  - STATUS_UNKNOWN—Indicates that the status of a transaction is unknown.
- rollback—Rolls back the referenced transaction. This method takes no arguments and returns void.
- setRollbackOnly—Specifies that the only possible outcome of the transaction is rollback. This method takes no arguments and returns void.
- setTransactionTimeout—Sets the timeout (in seconds) associated with the transaction. If some transaction participant has not specifically set this value, a default timeout is used. This method takes a number of seconds (as type int) and returns void.

Chapter 9. WebSphere Programming Model Extensions

This section discusses facilities that are provided as part of the Programming Model Extensions in WebSphere Application Server:

- The exception-chaining package, which can be used by distributed applications to capture a sequence of exceptions. For more information, see "The distributed-exception package".
- The command package, which can be used by distributed applications to reduce the number of remote invocations they must make. For more information, see "The command package" on page 140.
- The localizable-text package, which can be used by distributed applications spanning locales to deliver output in a user-specified language. For more information, see "The localizable-text package" on page 170.

The exception-chaining and command packages are available as part of WebSphere Application Server Advanced Edition and Enterprise Edition; the localizable-text package is available as part of WebSphere Application Server Advanced Edition. All three packages are general-purpose utilities, designed to provide common functions in a reusable way. Although these facilities are described in the context of enterprise beans, they are available to any WebSphere Application Server Java application. They are not restricted to use with enterprise beans.

The distributed-exception package

Distributed applications require a strategy for exception handling. As applications become more complex and are used by more participants, handling exceptions becomes problematic. To capture the information contained in every exception, methods have to rethrow every exception they catch. If every method adopts this approach, the number of exceptions can become unmanageable, and the code itself becomes less maintainable. Furthermore, if a new method introduces a new exception, all existing methods that call the new method have to be modified to handle the new exception. Trying to explicitly manage every possible exception in a complex application quickly becomes intractable.

In order to keep the number of exceptions manageable, some programmers adopt a strategy in which methods catch all exceptions in a single clause and throw one exception in response. This reduces the number of exceptions each method must recognize, but it also means that the information about the originating exception is lost. This loss of information can be desirable, for

example, when you wish to hide implementation details from end users. However, this strategy can make applications much more difficult to debug.

The distributed-exception package provides a facility that allows you to build chains of exceptions. An *exception chain* encapsulates the stack of previous exceptions. With an exception chain, you can throw one exception in response to another without discarding the previous exceptions, so you can manage the number of exceptions without losing the information they carry. Exceptions that support chaining are called *distributed exceptions*.

Distributed exceptions are packaged in the ras.jar file, which must be included in the application's CLASSPATH variable.

Overview

Support for chaining distributed exceptions is provided by the com.ibm.websphere.exception Java package. The following classes and interfaces make up this package:

- DistributedException—This class provides access to the methods on the DistributedExceptionInfo object. It acts as the root class for exceptions in a distributed application. For more information, see "The DistributedException class".
- DistributedExceptionEnabled—This interface allows exceptions that cannot inherit from the DistributedException class to be used in exception chains, so that exceptions based on predefined exceptions can be captured. For more information, see "The DistributedExceptionEnabled interface" on page 132 .
- DistributedExceptionInfo—This class encapsulates the work necessary for distributed exceptions. An exception class that extends the DistributedException class automatically gets access to this class. A class that implements the DistributedExceptionEnabled interface must explicitly declare a DistributedExceptionInfo attribute. For more information, see "The DistributedExceptionInfo class" on page 133.
- ExceptionInstantiationException—This class defines the exception that is thrown if an exception chain cannot be created. This exception is instantiated internally, but you can catch and re-throw it.

This section provides a general description of the interfaces and classes in the exception-chaining package.

The DistributedException class

The DistributedException class provides the root exception for exception hierarchies defined by applications. With this class, you build chains of exceptions by saving a caught exception and bundling it into the new exception to be thrown. This way, the information about the old exception is forwarded along with the new exception. The class declares six constructors; Figure 55 on page 131 shows the signatures for these constructors. When your

exception is a subclass of the DistributedException class, you must provide corresponding constructors in your exception class.

```
...
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
    ...
}
```

Figure 55. Code example: Constructors for the DistributedException class

The class also provides methods for extracting exceptions from the chain and querying the chain. These methods include:

- getMessage—This method returns the message string associated with the current exception.
- getPreviousException—This method returns the preceding exception in a chain as a Throwable object. If there are no previous exceptions, it returns null.
- getOriginalException—This method returns the original exception in a chain as a Throwable object. If there is no prior exception, it returns null.
- getException—This method returns the most recent instance of the named exception from the chain as a Throwable object. If there are no instances present, it returns null.
- getExceptionInfo—This method returns the DistributedExceptionInfo object for the exception.
- printStackTrace—These methods print the stack trace for the current exception, which includes the stack traces of all previous exceptions in the chain.

Localization support: Support for localized messages is provided by two of the constructors for distributed exceptions. These constructors take arguments representing a resource bundle, a resource key, a default message, and the set of replacement strings for variables in the message. A resource bundle is a collection of resources or resource names representing information associated with a specific locale. Resource bundles are provided as either a subclass of the ResourceBundle class or in a properties file. The resource key indicates which resource in the bundle to retrieve. The default message is returned if either the name of the resource bundle or the key is null or invalid.

The DistributedExceptionEnabled interface

Use the DistributedExceptionEnabled interface to create distributed exceptions when your exception cannot extend the DistributedException class. Because Java does not permit multiple inheritance, you cannot extend multiple exception classes. If you are extending an existing exception class, for example, javax.ejb.CreateException, you cannot also extend the DistributedException class. To allow your new exception class to chain other exceptions, you must implement the DistributedExceptionEnabled interface instead.

The DistributedExceptionEnabled interface declares eight methods you must implement in your exception class:

- getMessage—This method returns the message string associated with the current exception.
- getPreviousException—This method returns the preceding exception in a chain as a Throwable object. If there are no previous exceptions, it returns null.
- getOriginalException—This method returns the original exception in a chain as a Throwable object. If there is no prior exception, it returns null.
- getException—This method returns the most recent instance of the named exception from the chain as a Throwable object. If there are no instances present, it returns null.
- getExceptionInfo—This method returns the DistributedExceptionInfo object for the exception.
- printStackTrace—These methods print the stack trace for the current exception, which includes the stack traces of all previous exceptions in the chain.
- printSuperStackTrace—This method is used by a DistributedExceptionInfo object to retrieve and save the current stack trace.

When implementing the DistributedExceptionEnabled interface, you must declare a DistributedExceptionInfo attribute. This attribute provides implementations for most of these methods, so implementing them in your exception class consists of calling the corresponding methods on the

DistributedExceptionInfo object. For more information, see "Implementing the methods from the DistributedExceptionEnabled interface" on page 136.

The DistributedExceptionInfo class

The DistributedExceptionInfo class provides the functionality required for distributed exceptions. It must be used by any exception that implements the DistributedExceptionEnabled interface (which includes the DistributedException class). A DistributedExceptionInfo object contains the exception itself, and it provides constructors for creating exception chains and methods for retrieving the information within those chains. It also provides the underlying methods for managing chained exceptions.

Extending the DistributedException class

The DistributedException class provides the root exception for exception hierarchies defined by applications. The class also provides methods for extracting exceptions from the chain and querying the chain. You must provide constructors corresponding to the constructors in the DistributedException class (see Figure 55 on page 131). The constructors can simply pass arguments to the constructor in the DistributedException class by using super methods, as illustrated in Figure 56 on page 134.

```
...
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);
    }
}
```

Figure 56. Code example: Constructors in an exception class that extends the DistributedException class

Implementing the DistributedExceptionEnabled interface

Use the DistributedExceptionEnabled interface to create distributed exceptions when your exception cannot extend the DistributedException class. To allow your new exception class to be chained, you must implement the DistributedExceptionEnabled interface instead. Figure 57 on page 135 shows the structure of an exception class that extends the existing javax.ejb.CreateException class and implements the DistributedExceptionEnabled interface. The class also declares the required DistributedExceptionInfo object.

```
...
import javax.ejb.*;
import com.ibm.websphere.exception.*;
public class AccountCreateException extends CreateException
implements DistributedExceptionEnabled
{
    DistributedExceptionInfo exceptionInfo = null;
    // Constructors
    ...
    // Methods from the DistributedExceptionEnabled interface
    ...
}
```

Figure 57. Code example: The structure of an exception class that implements the DistributedExceptionEnabled interface

Implementing the constructors for the exception class

The exception-chaining package supports six different ways of creating instances of exception classes (see Figure 55 on page 131). When you write an exception class by implementing the DistributedExceptionEnabled interface, you must implement these constructors. In each one, you must use the DistributedExceptionInfo object to capture the information for chaining the exception. Figure 58 on page 136 shows standard implementations for the six constructors.

```
...
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
    ...
}
```

Figure 58. Code example: Constructors for an exception class that implements the DistributedExceptionEnabled interface

Implementing the methods from the DistributedExceptionEnabled interface

The DistributedExceptionInfo object provides implementations for most of the methods in the DistributedExceptionEnabled interface, so you can implement the required methods in your exception class by calling the corresponding methods on the DistributedExceptionInfo object. Figure 59 on page 138 illustrates this technique. The only two methods that do not involve calling a

corresponding method on the DistributedExceptionInfo object are the getExceptionInfo method, which returns the object, and the printSuperStackTrace method, which calls the super.printStackTrace method.

```
...
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;
    }
}
```

Figure 59. Code example: Implementations of the methods in the DistributedExceptionEnabled interface

Using distributed exceptions

Defining a distributed exception gives you the ability to chain exceptions together. The DistributedExceptionInfo class provides methods for adding information to an exception chain and for extracting information from the chain. This section illustrates the use of distributed exceptions.

Catching distributed exceptions

You can catch exceptions that extend the DistributedException class or implement the DistributedExceptionEnabled interface separately. You can also test a caught exception to see if it has implemented the DistributedExceptionEnabled interface. If it has, you can treat it as any other distributed exception. Figure 60 shows the use of the instanceof method to test for exception chaining.

```
....
try {
    someMethod();
}
catch (Exception e) {
    ...
    if (e instanceof DistributedExceptionEnabled) {
            ...
    }
...
```

Figure 60. Code example: Testing for an exception that implements the DistributedExceptionEnabled interface

Adding an exception to a chain

To add an exception to a chain, you must call one of the constructors for your distributed-exception class. This captures the previous exception information and packages it with the new exception. Figure 61 shows the use of the MyDistributedException(Throwable) constructor.

```
void someMethod() throws MyDistributedException {
    try {
        someOtherMethod();
    }
    catch (DistributedExceptionEnabled e) {
        throw new MyDistributedException(e);
    }
    ...
}...
```

Figure 61. Code example: Adding an exception to a chain

Retrieving information from a chain

Chained exceptions allow you to retrieve information about prior exceptions in the chain. For example, the getPreviousException, getOriginalException,

and getException(String) methods allow you to retrieve specific exceptions from the chain. You can retrieve the message associated with the current exception by calling the getMessage method. You can also get information about the entire chain by calling one of the printStackTrace methods. Figure 62 illustrates calling the getPreviousException and getOriginalException methods.

```
...
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();
        ...
        }
    }
}
...
```

Figure 62. Code example: Extracting exceptions from a chain

The command package

Distributed applications are defined by the ability to utilize remote resources as if they were local, but this remote work affects the performance of distributed applications. Distributed applications can improve performance by using remote calls sparingly. For example, if a server does several tasks for a client, the application can run more quickly if the client bundles requests

together, reducing the number of individual remote calls. The command package provides a mechanism for collecting sets of requests to be submitted as a unit.

In addition to giving you a way to reduce the number of remote invocations a client makes, the command package provides a generic way of making requests. A client instantiates the command, sets its input data, and tells it to run. The command infrastructure determines the target server and passes a copy of the command to it. The server runs the command, sets any output data, and copies it back to the client. The package provides a common way to issue a command, locally or remotely, and independently of the server's implementation. Any server (an enterprise bean, a Java Database Connectivity (JDBC) server, a servlet, and so on) can be a target of a command if the server supports Java access to its resources and provides a way to copy the command between the client's Java Virtual Machine (JVM) and its own JVM.

Overview

The command facility is implemented in the com.ibm.websphere.command Java package. The classes and interfaces in the command package fall into four general categories:

- Interfaces for creating commands. For more information, see "Facilities for creating commands".
- Classes and interfaces for implementing commands. For more information, see "Facilities for implementing commands" on page 142.
- Classes and interfaces for determining where the command is run. For more information, see "Facilities for setting and determining targets" on page 143.
- Classes defining package-specific exceptions. For more information, see "Exceptions in the command package" on page 144.

This section provides a general description of the interfaces and classes in the command package.

Facilities for creating commands

The Command interface specifies the most basic aspects of a command. This interface is extended by both the TargetableCommand interface and the CompensableCommand interface, which offer additional features. To create commands for applications, you must:

- Define an interface that extends one or more of interfaces in the command package.
- Provide an implementation class for your interface.

In practice, most commands implement the TargetableCommand interface, which allows the command to be executed remotely. Figure 63 on page 142 shows the structure of a command interface for a targetable command.

```
...
import com.ibm.websphere.command.*;
public interface MySimpleCommand extends TargetableCommand {
      // Declare application methods here
}
```

Figure 63. Code example: The structure of an interface for a targetable command

The CompensableCommand interface allows the association of one command with another that can undo the work of the first. Compensable commands also typically implement the TargetableCommand interface. Figure 64 shows the structure of a command interface for a targetable, compensable command.

```
...
import com.ibm.websphere.command.*;
public interface MyCommand extends TargetableCommand, CompensableCommand {
      // Declare application methods here
}
```

Figure 64. Code example: The structure of an interface for a targetable, compensable command

Facilities for implementing commands

Commands are implemented by extending the class TargetableCommandImpl, which implements the TargetableCommand interface. The TargetableCommandImpl class is an abstract class that provides some implementations for some of the methods in the TargetableCommand interface (for example, setting return values) and declares additional methods that the application itself must implement (for example, how to execute the command).

You implement your command interface by writing a class that extends the TargetableCommandImpl class and implements your command interface. This class contains the code for the methods in your interface, the methods inherited from extended interfaces (the TargetableCommand and CompensableCommand interfaces), and the required (abstract) methods in the TargetableCommandImpl class. You can also override the default implementations of other methods provided in the TargetableCommandImpl class. Figure 65 on page 143 shows the structure of an implementation class for the interface in Figure 64.

```
...
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
     ...
}
```

Figure 65. Code example: The structure of an implementation class for a command interface

Facilities for setting and determining targets

The object that is the target of a TargetableCommand must implement the CommandTarget interface. This object can be an actual server-side object, like an entity bean, or it can be a client-side adapter for a server. The implementor of the CommandTarget interface is responsible for ensuring the proper execution of a command in the desired target server environment. This typically requires the following steps:

1. Copying the command to the target server by using a server-specific protocol.
2. Running the command in the server.
3. Copying the executed command from the target server to the client by using a server-specific protocol.

Common ways to implement the CommandTarget interface include:

- A local target, which runs in the client's JVM.
- A client-side adapter for a server. For an example that implements the target as a client-side adapter, see "Writing a command target (client-side adapter)" on page 166.
- An enterprise bean (either a session bean or an entity bean). Figure 66 on page 144 shows the structure of the remote interface and enterprise bean class for an entity bean that implements the CommandTarget interface. An enterprise bean is provided with WebSphere that can be deployed and used as a CommandTarget. See "Using the WebSphere EJBCommandTarget bean as a command target" on page 157.

```
...
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
     ...
}
```

Figure 66. Code example: The structure of a command-target entity bean

Since targetable commands can be run remotely in another JVM, the command package provides mechanisms for determining where to run the command. A *target policy* associates a command with a target and is specified through the TargetPolicy interface. You can design customized target policies by implementing this interface, or you can use the provided TargetPolicyDefault class. For more information, see "Targets and target policies" on page 161.

Exceptions in the command package

The command package defines a set of exception classes. The CommandException class extends the DistributedException class and acts as the base class for the additional command-related exceptions: UnauthorizedAccessException, UnsetInputPropertiesException, and UnavailableCompensableCommandException. Applications can extend the CommandException class to define additional exceptions, as well.

Although the CommandException class extends the DistributedException class, you do not have to import the distributed-exception package, com.ibm.websphere.exception, unless you need to use the features of the DistributedException class in your application. For more information on distributed exceptions, see "The distributed-exception package" on page 129.

Writing command interfaces

To write a command interface, you extend one or more of the three interfaces included in the command package. The base interface for all commands is the

Command interface. This interface provides only the client-side interface for generic commands and declares three basic methods:

- isReadyToCallExecute—This method is called on the client side before the command is passed to the server for execution.
- execute—This method passes the command to the target and returns any data.
- reset—This method reverts any output properties to the values they had before the execute method was called so that the object can be reused.

The implementation class for your interface must contain implementations for the isReadyToCallExecute and reset methods. The execute method is implemented for you elsewhere; for more information, see "Implementing command interfaces" on page 147. Most commands do not extend the Command interface directly but use one of the provided extensions: the TargetableCommand interface and the CompensableCommand interface.

The TargetableCommand interface

The TargetableCommand interface extends the Command interface and provides for remote execution of commands. Most commands will be targetable commands. The TargetableCommand interface declares several additional methods:

- setCommandTarget—This method allows you to specify the target object to a command.
- setCommandTargetName—This method allows you to specify the target by name to a command.
- getCommandTarget—This method returns the target object of the command.
- getCommandTargetName—This method returns the name of the target object of the command.
- hasOutputProperties—This method indicates whether or not the command has output that must be copied back to the client. (The implementation class also provides a method, setHasOutputProperties, for setting the output of this method. By default, hasOutputProperties returns true.)
- setOutputProperties—This method saves output values from the command for return to the client.
- performExecute—This method encapsulates the application-specific work. It is called for you by the execute method declared in the Command interface.

With the exception of the performExecute method, which you must implement, all of these methods are implemented in the TargetableCommandImpl class. This class also implements the execute method declared in the Command interface.

The CompensableCommand interface

The CompensableCommand interface also extends the Command interface. A compensable command is one that has another command (a compensator)

associated with it, so that the work of the first can be undone by the compensator. For example, a command that attempts to make an airline reservation followed by a hotel reservation can offer a compensating command that allows the user to cancel the airline reservation if the hotel reservation cannot be made.

The CompensableCommand interface declares one method:

- getCompensatingCommand—This methods returns the command that can be used to undo the effects of the original command.

To create a compensable command, you write an interface that extends the CompensableCommand interface. Such interfaces typically extend the TargetableCommand interface as well. You must implement the getCompensatingCommand method in the implementation class for your interface. You must also implement the compensating command.

The example application

The example used throughout the remainder of this discussion uses an entity bean with container-managed persistence (CMP) called CheckingAccountBean, which allows a client to deposit money, withdraw money, set a balance, get a balance, and retrieve the name on the account. This entity bean also accepts commands from the client. The code examples illustrate the command-related programming. For a servlet-based example, see "Writing a command target (client-side adapter)" on page 166.

Figure 67 shows the interface for the ModifyCheckingAccountCmd command. This command is both targetable and compensable, so the interface extends both TargetableCommand and CompensableCommand interfaces.

```
...
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();
      ...
}
```

Figure 67. Code example: The ModifyCheckingAccountCmd interface

Implementing command interfaces

The command package provides a class, TargetableCommandImpl, that implements all of the methods in the TargetableCommand interface except the performExecute method. It also implements the execute method from the Command interface. To implement an application's command interface, you must write a class that extends the TargetableCommandImpl class and implements your command interface. Figure 68 shows the structure of the ModifyCheckingAccountCmdImpl class.

```
...
public class ModifyCheckingAccountCmdImpl extends TargetableCommandImpl
implements ModifyCheckingAccountCmd
{
    // Variables
    ...
    // Methods
    ...
}
```

Figure 68. Code example: The structure of the ModifyCheckingAccountCmdImpl class

The class must declare any variables and implement these methods:

- Any methods you defined in your command interface.
- The isReadyToCallExecute and reset methods from the Command interface.
- The performExecute method from the TargetableCommand interface.
- The getCompensatingCommand method from the CompensableCommand interface, if your command is compensable. You must also implement the compensating command.

You can also override the nonfinal implementations provided in the TargetableCommandImpl class. The most likely candidate for reimplementation is the setOutputProperties method, since the default implementation does not save final, transient, or static fields.

Defining instance and class variables

The ModifyCheckingAccountCmdImpl class declares the variables used by the methods in the class, including the remote interface of the CheckingAccount entity bean; the variables used to capture operations on the checking account (balances and amounts); and a compensating command. Figure 69 on page 148 shows the variables used by the ModifyCheckingAccountCmd command.

```
...
public class ModifyCheckingAccountCmdImpl extends TargetableCommandImpl
implements ModifyCheckingAccountCmd
{
    // Variables
    public float balance;
    public float amount;
    public float oldBalance;
    public CheckingAccount checkingAccount;
    public ModifyCheckingAccountCompensatorCmd
                        modifyCheckingAccountCompensatorCmd;
    ...
}
```

Figure 69. Code example: The variables in the ModifyCheckingAccountCmdImpl class

Implementing command-specific methods

The ModifyCheckingAccountCmd interface defines several command-specific methods in addition to extending other interfaces in the command package. These command-specific methods are implemented in the ModifyCheckingAccountCmdImpl class.

You must provide a way to instantiate the command. The command package does not specify the mechanism, so you can choose the technique most appropriate for your application. The fastest and most efficient technique is to use constructors. The most flexible technique is to use a factory. Also, since commands are implemented internally as JavaBeans components, you can use the standard Beans.instantiate method. The ModifyCheckingAccountCmd command uses constructors.

Figure 70 on page 149 shows the two constructors for the command. The difference between them is that the first uses the default target policy for determining the target of the command and the second allows you to specify a custom policy. (For more information on targets and target policies, see "Targets and target policies" on page 161.)

Both constructors take a CommandTarget object as an argument and cast it to the CheckingAccount type. The CheckingAccount interface extends both the CommandTarget interface and the EJBObject (see Figure 80 on page 160). The resulting checkingAccount object routes the command to the desired server by using the bean's remote interface. (For more information on CommandTarget objects, see "Writing a command target (server)" on page 159.)

```
...
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);
    }
    ...
}
```

Figure 70. Code example: Constructors in the ModifyCheckingAccountCmdImpl class

Figure 71 on page 150 shows the implementation of the command-specific methods:

- setBalance—This method sets the balance of the account.
- getAmount—This method returns the amount of a deposit or withdrawal.
- getOldBalance, getBalance—These methods capture the balance before and after an operation.
- getCmdTargetPolicy—This method retrieves the current target policy.
- setCheckingAccount, getCheckingAccount—These methods set and retrieve the current checking account.

```
...
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;
    }
    ...
}
```

Figure 71. Code example: Command-specific methods in the ModifyCheckingAccountCmdImpl class

The ModifyCheckingAccountCmd command operates on a checking account. Because commands are implemented as JavaBeans components, you manage input and output properties of commands using the standard JavaBeans techniques. For example, initialize input properties with set methods (like setCheckingAccount) and retrieve output properties with get methods (like getCheckingAccount). The get methods do not work until after the command's execute method has been called.

Implementing methods from the Command interface

The Command interface declares two methods, isReadyToCallExecute and reset, that must be implemented by the application programmer. Figure 72 shows the implementations for the ModifyCheckingAccountCmd command. The implementation of the isReadyToCallExecute method ensures that the checkingAccount variable is set. The reset method sets all of the variables back to starting values.

```
...
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();
    }
   ...
}
```

Figure 72. Code example: Methods from the Command interface in the ModifyCheckingAccountCmdImpl class

Implementing methods from the TargetableCommand interface

The TargetableCommand interface declares one method, performExecute, that must be implemented by the application programmer. Figure 73 on page 152 shows the implementation for the ModifyCheckingAccountCmd command. The implementation of the performExecute method does the following:

- Saves the current balance (so the command can be undone by a compensator command)
- Calculates the new balance
- Sets the current balance to the new balance
- Ensures that the hasOutputProperties method returns true so that the values are returned to the client

In addition, the ModifyCheckingAccountCmdImpl class overrides the default implementation of the setOutputProperties method.

```
...
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.");
        }
    }
    ...
}
```

Figure 73. Code example: Methods from the TargetableCommand interface in the ModifyCheckingAccountCmdImpl class

Implementing the CompensableCommand interface

The CompensableCommand interface declares one method, getCompensatingCommand, that must be implemented by the application programmer. Figure 74 on page 153 shows the implementation for the ModifyCheckingAccountCmd command. The implementation simply returns an instance of the ModifyCheckingAccountCompensatorCmd command associated with the current command.

```
...
public class ModifyCheckingAccountCmdImpl extends TargetableCommandImpl
implements ModifyCheckingAccountCmd
{
    ...
    // Method from CompensableCommand interface
    public Command getCompensatingCommand() throws CommandException {
        modifyCheckingAccountCompensatorCmd =
            new ModifyCheckingAccountCompensatorCmd(this);
        return (Command)modifyCheckingAccountCompensatorCmd;
    }
}
```

Figure 74. Code example: Method from the CompensableCommand interface in the ModifyCheckingAccountCmdImpl class

Writing the compensating command: An application that uses a compensable command requires two separate commands: the primary command (declared as a CompensableCommand) and the compensating command. In the example application, the primary command is declared in the ModifyCheckingAccountCmd interface and implemented in the ModifyCheckingAccountCmdImpl class. Because this command is also a compensable command, there is a second command associated with it that is designed to undo its work. When you create a compensable command, you also have to write the compensating command.

Writing a compensating command can require exactly the same steps as writing the original command: writing the interface and providing an implementation class. In some cases, it may be simpler. For example, the command to compensate for the ModifyCheckingAccountCmd does not require any methods beyond those defined for the original command, so it does not need an interface. The compensating command, called ModifyCheckingAccountCompensatorCmd, simply needs to be implemented in a class that extends the TargetableCommandImpl class. This class must:

- Provide a way to instantiate the command; the example uses a constructor
- Implement the three required methods:
  - isReadyToCallExecute and reset—both from the Command interface
  - performExecute—from the TargetableCommand interface

Figure 75 on page 154 shows the structure of the implementation class, its variables (references to the original command and to the relevant checking account), and the constructor. The constructor simply instantiates the references to the primary command and account.

```
...
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
    ....
}
```

Figure 75. Code example: Variables and constructor in the ModifyCheckingAccountCompensatorCmd class

Figure 76 on page 155 shows the implementation of the inherited methods. The implementation of the isReadyToCallExecute method ensures that the checkingAccount variable has been instantiated.

The performExecute method verifies that the actual checking-account balance is consistent with what the original command returns. If so, it replaces the current balance with the previously stored balance by using the ModifyCheckingAccountCmd command. Finally, it saves the most-recent balances in case the compensating command needs to be undone. The reset method has no work to do.

```
...
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() {}
}
```

Figure 76. Code example: Methods in ModifyCheckingAccountCompensatorCmd class

Using a command

To use a command, the client creates an instance of the command and calls the command's execute method. Depending on the command, calling other methods can be necessary. The specifics will vary with the application.

In the example application, the server is the CheckingAccountBean, an entity enterprise bean. In order to use this enterprise bean, the client gets a reference to the bean's home interface. The client then uses the reference to the home

interface and one of the bean's finder methods to obtain a reference to the bean's remote interface. If there is no appropriate bean, the client can create one using a create method on the home interface. All of this work is standard enterprise bean programming covered elsewhere in this document.

Figure 77 illustrates the use of the ModifyCheckingAccountCmd command. This work takes place after an appropriate CheckingAccount bean has been found or created. The code instantiates a command, setting the input values by using one of the constructors defined for the command. The null argument indicates that the command should look up the server using the default target policy, and 1000 is the amount the command attempts to add to the balance of the checking account. (For more information on how the command package uses defaults to determine the target of a command, see "The default target policy" on page 162.) After the command is instantiated, the code calls the setCheckingAccount method to identify the account to be modified. Finally, the execute method on the command is called.

```
{
    ...
    CheckingAccount checkingAccount
    ...
    try {
        ModifyCheckingAccountCmd cmd =
            new ModifyCheckingAccountCmdImpl(null, 1000);
        cmd.setCheckingAccount(checkingAccount);
        cmd.execute();
    }
    catch (Exception e) {
       System.out.println(e.getMessage());
    }
    ...
}
```

Figure 77. Code example: Using the ModifyCheckingAccountCmd command

Using a compensating command

To use a compensating command, you must retrieve the compensator associated with the primary command and call its execute method. Figure 78 on page 157 shows the code used to run the original command and to give the user the option of undoing the work by running the compensating command.

```
{
    ...
    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());
    }
    ...
}
```

Figure 78. Code example: Using the ModifyCheckingAccountCompensator command

Using the WebSphere EJBCommandTarget bean as a command target

WebSphere ships a CommandTarget enterprise bean to allow administrators to execute a command in a designated server without providing their own implementation of CommandTarget. The EJBCommandTarget class, along with the EJBCommandTarget bean (CommandServerSessionBean), are located in the EJBCommandTarget.jar file in the lib directory under the WebSphere installation directory. This is a deployed jar file. You can use this JAR file in a new application or add it into an existing application.

The EJBCommandTarget class serves as a wrapper for a CommandTarget bean. CommandServerSessionBean is the WebSphere implementation of this CommandTarget bean. A command developer can set this EJBCommandTarget object into the Command. Figure 79 on page 158 shows an example.

```
EJBCommandTarget target = new EJBCommandTarget();

MyCommand cmd = new MyCommandImpl(Arguments...);
cmd.setCommandTarget(target);
cmd.execute();
```

Figure 79. Code example: Using an EJBCommandTarget bean

In this example, the client creates a MyCommand object. It is then executed in the application server. When the execute method is performed, the target (EJBCommandTarget) looks up the CommandServerSessionHome from the InitialContext and executes the executeCommand method on the CommandServerSessionBean. The EJBCommandTarget object ensures that there is only one CommandServerSessionBean per object to avoid extra naming lookup.

An EJBCommandTarget object can be created using four different constructors:

- EJBCommandTarget(″MyNamingServerName″, ″PortNumber″, ″JNDIName″)
- EJBCommandTarget(InitialContext,″JNDIName″)
- EJBCommandTarget(″JNDIName″)
- EJBCommandTarget()

The first constructor allows the application to specify the naming server name and the port. The JNDI name of the CommandServerSessionBean can also be specified. The EJBCommandTarget constructs a provider URL of ″iiop://MyNamingServerName:PortNumber″ and looks up the CommandServerSessionBean with the given JNDI name. If null values are passed in for any of the parameters the WebSphere defaults for server and port and a default JNDI name of CommandServerSession are used.

The second constructor allows the application to specify its own initial context. The EJBCommandTarget object then uses this initial context to look up the CommandServerSession bean with the specified JNDI name.

The third constructor allows the application to set up the naming server (the provider URL) in property files.

The default constructor uses the default values for the provider URL and default JNDI name for the CommandServerSession bean (CommandServerSession).

You do not need to use the EJBCommandTarget class. You can instead create your own custom target policy that uses the EJBCommandTarget bean

(CommandServerSessionBean). The EJBCommandTarget object is a convenience class and attempts to address most usage scenarios

Writing a command target (server)

In order to accept commands, a server must implement the CommandTarget interface and its single method, executeCommand.

The example application implements the CommandTarget interface in an enterprise bean. (For a servlet-based example, see "Writing a command target (client-side adapter)" on page 166.) The target enterprise bean can be a session bean or an entity bean. You can write a target enterprise bean that forwards commands to a specific server, such as another entity bean. In this case, all commands directed at a specific target go through the target enterprise bean. You can also write a target enterprise bean that does the work of the command locally.

Make an enterprise bean the target of a command by:

- Extending the CommandTarget interface when you define the bean's remote interface, which must also extend the EJBObject interface
- Implementing the CommandTarget interface when you implement the bean class, which must also implement either the SessionBean or EntityBean interface

The target of the example application is an enterprise bean called CheckingAccountBean. This bean's remote interface, CheckingAccount, extends the CommandTarget interface in addition to the EJBObject interface. The methods declared in the remote interface are independent of those used by the command. The executeCommand is declared in neither the bean's home nor remote interfaces. Figure 80 on page 160 shows the CheckingAccount interface.

```
...
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;
}
```

Figure 80. Code example: The remote interface for the CheckingAccount entity bean, also a command target

The enterprise bean class, CheckingAccountBean, implements the EntityBean interface as well as the CommandTarget interface. The class contains the business logic for the methods in the remote interface, the necessary life-cycle methods (ejbActivate, ejbStore, and so on), and the executeCommand declared by the CommandTarget interface. The executeCommand method is the only command-specific code in the enterprise bean class. It attempts to run the performExecute method on the command and throws a CommandException if an error occurs. If the performExecute method runs successfully, the executeCommand method uses the hasOutputProperties method to determine if there are output properties that must be returned. If the command has output properties, the method returns the command object to the client. Figure 81 on page 161 shows the relevant parts of the CheckingAccountBean class.

```
...
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;
    }
}
```

Figure 81. Code example: The bean class for the CheckingAccount entity bean, also a command target

Targets and target policies

A targetable command extends the TargetableCommand interface, which allows the client to direct a command to a particular server. The TargetableCommand interface (and the TargetableCommandImpl class) provide two ways for a client to specify a target: the setCommandTarget and setCommandTargetName methods. (These methods were introduced in "The TargetableCommand interface" on page 145.) The setCommandTarget methods allows the client to set the target object directly on the command. The setCommandTargetName method allows the client to refer to the server by name; this approach is useful when the client is not directly aware of server objects. A targetable command also has corresponding getCommandTarget and getCommandTargetName methods.

The command package needs to be able to identify the target of a command. Because there is more than one way to specify the target and because different

applications can have different requirements, the command package does not specify a selection algorithm. Instead, it provides a TargetPolicy interface with one method, getCommandTarget, and a default implementation. This allows applications to devise custom algorithms for determining the target of a command when appropriate.

The default target policy

The command package provides a default implementation of the TargetPolicy interface in the TargetPolicyDefault class. If you use this default implementation, the command determines the target by looking through an ordered sequence of four options:

1. The CommandTarget value
2. The CommandTargetName value
3. A registered mapping of a target for a specific command
4. A defined default target

If it finds no target, it returns null.

The TargetPolicyDefault class provides methods for managing the assignment of commands with targets (registerCommand, unregisterCommand, and listMappings), and a method for setting a default name for the target (setDefaultTargetName). The default target name is com.ibm.websphere.command.LocalTarget, where LocalTarget is a class that runs the command's performExecute method locally. Figure 82 shows the relevant variables and the methods in the TargetPolicyDefault class.

```
...
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) {
        ... }
}
```

Figure 82. Code example: The TargetPolicyDefault class

Setting the command target: The ModifyCheckingAccountImpl class provides two command constructors (see Figure 70 on page 149). One of them takes a command target as an argument and implicitly uses the default target

policy to locate the target. The constructor used in Figure 77 on page 156 passes a null target, so that the default target policy traverses its choices and eventually finds the default target name, LocalTarget.

The example in Figure 83 uses the same constructor to set the target explicitly. This example differs from Figure 77 on page 156 as follows:

- The command target is set to the checking account rather than null. The default target policy starts to traverse its choices and finds the target in the first place it looks.
- It does not have to call the setCheckingAccount method to indicate the account on which the command should operate; the constructor uses the target variable as both the target and the account.

```
{
    ...
    CheckingAccount checkingAccount
    ....
    try {
        ModifyCheckingAccountCmd cmd =
            new ModifyCheckingAccountCmdImpl(checkingAccount, 1000);
        cmd.execute();
    }
    catch (Exception e) {
       System.out.println(e.getMessage());
    }
    ...
}
```

Figure 83. Code example: Identifying a target with CommandTarget

Setting the command target name: If a client needs to set the target of the command by name, it can use the command's setCommandTargetName method. Figure 84 on page 164 illustrates this technique. This example compares with Figure 77 on page 156 as follows:

- Both explicitly set the command target in the constructor to null.
- Both use the setCheckingAccount method to indicate the account on which the command should operate.
- This example sets the target name explicitly by using the setCommandTargetName method. When the default target policy traverses its choices, it finds a null for the first choice and a name for the second.

```
{
    ...
    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());
    }
    ...
}
```

Figure 84. Code example: Identifying a target with CommandTargetName

Mapping the command to a target name: The default target policy also permits commands to be registered with targets. Mapping a command to a target is an administrative task that most appropriately done through a configuration tool. The WebSphere Application Server administrative console does not yet support the configuration of mappings between commands and targets. Applications that require support for the registration of commands with targets must supply the tools to manage the mappings. These tools can be visual interfaces or command-line tools.

Figure 85 shows the registration of a command with a target. The names of the command class and the target are explicit in the code, but in practice, these values would come from fields in a user interface or arguments to a command-line tool. If a program creates a command as shown in Figure 77 on page 156, with a null for the target, when the default target policy traverses its choices, it finds a null for the first and second choices and a mapping for the third.

```
{
    ...
   targetPolicy.registerCommand(
        "com.ibm.sfc.cmd.test.ModifyCheckingAccountImpl",
        "com.ibm.sfc.cmd.test.CheckingAccountBean");
    ...
}
```

Figure 85. Code example: Mapping a command to a target in an external application

Customizing target policies

You can define custom target policies by implementing the TargetPolicy interface and providing a getCommandTarget method appropriate for your

application. The TargetableCommandImpl class provides setTargetPolicy and getTargetPolicy methods for managing custom target policies.

So far, the target of all the commands has been a checking-account entity bean. Suppose that someone introduces a session enterprise bean (MySessionBean) that can also act as a command target. Figure 86 shows a simple custom policy that sets the target of every command to MySessionBean.

```
...
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();
        }
    }
}
```

Figure 86. Code example: Creating a custom target policy

Since commands are implemented as JavaBeans components, using custom target policies requires importing the java.beans package and writing some elementary JavaBeans code. Also, your custom target-policy class must also implement the java.io.Serializable interface.

Using a custom target policy: The ModifyCheckingAccountImpl class provides two command constructors (see Figure 70 on page 149). One of them implicitly uses the default target policy; the other takes a target policy object as an argument, which allows you to use a custom target policy. The example in Figure 87 on page 166 uses the second constructor, passing a null target and a custom target policy, so that the custom policy is used to determine the target. After the command is executed, the code uses the reset method to return the target policy to the default.

```
{
    ...
    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());
    }
}
```

Figure 87. Code example: Using a custom target policy

Writing a command target (client-side adapter)

Commands can be used with any Java application, but the means of sending the command from the client to the server varies. The application described in "The example application" on page 146 used enterprise beans. The example in this section shows how you can send a command to a servlet over the HTTP protocol.

In this example, the client implements the CommandTarget interface locally. Figure 88 on page 167 shows the structure of the client-side class; it implements the CommandTarget interface by implementing the executeCommand method.

```
...
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() {
         ... }
}
```

Figure 88. Code example: The structure of a client-side adapter for a target

The main method in the client-side adapter constructs and intializes the CommandTarget object, as shown in Figure 89.

```
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
    ...
}
```

Figure 89. Code example: Instantiating the client-side adapter

Implementing a client-side adapter

The CommandTarget interface declares one method, executeCommand, which the client implements. The executeCommand method takes a TargetableCommand object as input; it also returns a TargetableCommand.

Figure 90 shows the implementation of the method used in the client-side adapter. This implementation does the following:

- Serializes the command it receives
- Creates an HTTP connection to the servlet
- Creates input and output streams, to handle the command as it is sent to the server and returned
- Places the command on the output stream
- Sends the command to the server
- Retrieves the returned command from the input stream
- Returns the returned command to the caller of the executeCommand method

```
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
    ....
}
```

Figure 90. Code example: A client-side implementation of the executeCommand method

Running the command in the servlet

The servlet that runs the command is shown in Figure 91. The service method retrieves the command from the input stream and runs the performExecute method on the command. The resulting object, with any output properties that must be returned to the client, is placed on the output stream and sent back to the client.

```
...
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();
        }
    }
}
```

Figure 91. Code example: Running the command in the servlet

In this example, the target invokes the performExecute method on the command, but this is not always necessary. In some applications, it can be preferable to implement the work of the command locally. For example, the command can be used only to send input data, so that the target retrieves the data from the command and runs a local database procedure based on the input. You must decide the appropriate way to use commands in your application.

The localizable-text package

Overview

Users of distributed applications can come from widely varying areas; they can speak different languages, represent dates and times in regionally specific ways, and use different currencies. An application intended to be used by such an audience must either force them all to use the same interface (for example, an English-based interface), or it can be written in such a way that it can be configured to the linguistic conventions of the users, so English-speaking users can use the English interface but French-speaking users can interact with the application through a French interface.

An application that can present information to users in formats that abide by the users' linguistic conventions is said to be *localizable:* the application can be configured to interact with users from different localities in linguistically appropriate ways. In a localized application, a user in one region sees error messages, output, and interface elements (like menu options) in the requested language. Additionally, other elements that are not strictly linguistic, like date and time formats and currencies, are presented in the appropriate style for users in the specified region. A user in another region sees output in the conventional language or format for that region.

Historically, the creation of localizable applications has been restricted to large corporations writing complex systems. The strategies for writing localizable code, collectively called *internationalization techniques*, have traditionally been expensive and difficult to implement, so they have been applied only to major development efforts. However, given the rise in distributed computing and in use of the World Wide Web, application developers have been pressured to make a much wider variety of applications localizable. This requires making internationalization—the techniques for writing localizable programs—much more accessible to application developers. The WebSphere localizable-text package is a set of Java classes and interfaces that can be used by WebSphere application developers to localize distributed WebSphere applications easily. Language catalogs for distributed WebSphere applications can be stored centrally, so the catalogs can be maintained and administered efficiently.

Writing localizable programs

In a nonlocalizable application, parts of the application that a user sees are unalterably coded into the application. For example, a routine that prints an error message simply prints a string, probably in English, to a file or the console. A localizable program adds a layer of abstraction into the design. Instead of going simply from error condition to output string, a localizable program represents error messages with some language-neutral information; in the simplest case, each error condition corresponds to a key. In order to print a usable error string for the user, the application looks up the key in the configured message catalog. A message catalog is a list of keys with corresponding strings. Different message catalogs provide the strings in different languages. The application looks up the key in the appropriate catalog, retrieves the corresponding error message in the desired language, and prints this string for the user.

The technique of localization can be used for far more than translating error messages. For example, by using keys to represent each element—button, label, menu item, and so forth—in a graphical user interface and by providing a message catalog containing translations of the button names, labels, and menu items, the graphical interface can be automatically translated into multiple languages. In addition, extending support to additional languages requires providing message catalogs for those languages; the application itself requires no modification.

Localization of an application is driven by two variables, the time zone and the locale. The time zone variable indicates how to compute the local time as an offset from a standard time like Greenwich Mean Time. The locale is a collection of information that indicates a geographic, political, or cultural region. It provides information on language, currency, and the conventions for presenting information like dates, and in a localizable program, the locale also indicates the message catalog from which an application retrieves messages. A time zone can cover many locales, and a single locale can span time zones. With both time zone and locale, the date, time, currency, and language for users in a specific region can be determined.

Identifying localizable text: To write a localizable application, an application developer must determine which aspects of the application need to be translatable. These are typically the parts of an application a user must read and understand. Application developers must consider the parts of an application with which all users directly interact, like the application's interface, and the parts serving more specialized purposes, like messages in log files. Good candidates for localization include:

- Elements in graphical user interfaces
  - Title bars for windows

  - Menu names, and the items on the menus (for example, ″select File → Open″)
  - Labels on buttons (for example, ″click the OK button″)
  - Instructions directing users to fill in fields (for example, ″enter the account number″)
  - Any other elements that users must read
- Prompts in command-line interfaces
- Output from the program
  - Responses to user input
  - Error messages
  - Text returned when exceptions are thrown
  - Other status messages (warnings, audit messages, and others)

After identifying each element of the application to be localized, application developers must assign a unique key to each element and provide a message catalog for each language to be supported. Each message catalog consists of keys and the corresponding language-specific strings. The key, therefore, is the link between the program and the message catalog; the program internally refers to localizable elements by key and uses the message catalog to generate the output seen by the user. Translated strings are generated by calling the format method on a LocalizableTextFormatter object, which represents a key and a resource bundle (a set of message catalogs). The locale setting of the program determines the message catalog in which to search for the key.

Creating message catalogs: After identifying each element to be localized, message catalogs must be created for each language to be supported. These catalogs, which are implemented as Java resource bundles, can be created in two ways, either as subclasses of the ResourceBundle class or as Java properties files. Resource bundles have a variety of uses in Java; for message catalogs, the properties-file approach is more common. If properties files are used, support for languages to be added or removed without modifying the application code, and catalogs can be prepared by people without programming expertise.

A message catalog implemented in a properties file consists of a line for each key, where a key identifies a localizable element. Each line in the file has the following structure:

```
key = String corresponding to the key
```

For example, a grapical user interface for a banking system can have a pull-down menu to be used for selecting a type of account, like savings or checking. The label for the pull-down menu and the account types on the menu are good choices for localization. There are three elements that require keys: the label for the account menu and the two items on the menu. If the

keys are accountString, savingsString, and checkingString, the English properties file associates each with an English string.

```
accountString = Accounts
savingsString = Savings
checkingString = Checking
...
```

Figure 92. Three elements in an English message catalog

In the German properties files, each key is given a corresponding German value.

```
accountString = Konten
savingsString = Sparkonto
checkingString = Girokonto
...
```

Figure 93. Three elements in a German message catalog

Properties files can be added for any other needed languages, as well.

Naming the properties files: To enable resolution to a specific properties file, Java specifies naming conventions for the properties files in a resource bundle: *resourceBundleName_localeID*.properties

Each file takes a fixed extension, .properties. The set of files making up the resource bundle is given a collective name; for a simple banking application, an obvious name, like BankingResources, suffices for the resource bundle. Each file is given the name of the resource bundle with a locale identifier; the specific value of the locale ID varies with the locale. These are used internally by the Java.util.ResourceBundle class to match files in a resource bundle to combinations of locale and time-zone settings. The details of the algorithm vary with the release of the JDK; see your Java documentation for information specific to your installation.

In the banking application, typical files in the BankingResources resource bundle include BankingResources_en.properties for the English message catalog and BankingResources_de.properties for the German catalog. Additionally, a default catalog, BankingResources.properties, is provided for use when the requested catalog cannot be found. The default catalog is often the English-language catalog.

Resource bundles containing message catalogs for use with localizable text need to be installed only on the systems where the formatting of strings is

actually performed. The resource bundles are typically placed in an application's JAR file. See "WebSphere support" on page 175 for more information.

Localization support in WebSphere and Java

The Java package com.ibm.websphere.i18n.localizabletext contains the classes and interfaces constituting the localizable-text package. This package makes extensive use of the internationalization and localization features of the Java language; programmers using the WebSphere localizable-text package must understand the underlying Java support, which are not documented in any detail here.

Java support: The WebSphere localizable-text package relies primarily on the following Java components:

- java.util.Locale
- java.util.TimeZone
- java.util.ResourceBundle
- java.text.MessageFormat

This list is not exhaustive. WebSphere and these Java classes can also use related Java classes, but the related classes—for example, java.util.Calendar—are typically special-purposes classes. This section briefly describes only the primary classes.

Locale: A Locale object in Java encapsulates a language and a geographic region, for example, the java.util.Locale.US object contains locale information for the United States. An application that specifies a locale can then take advantage of the locale-sensitive formatters built into the Java language. These formatters, in the java.text package, handle the presentation of numbers, currency values, dates, and times.

TimeZone: A TimeZone object in Java encapsulates a representation of the time and provides methods for tasks like reporting the time and accommodating seasonal time shifts. Applications use the time zone to determine the local date and time.

ResourceBundle: A resource bundle is a named collection of resources—information used by the application, for example, strings, fonts, and images—used by a specific locale. The ResourceBundle class allows an application to retrieve the named resource bundle appropriate to the locale. Resource bundles are used to hold the messages catalogs, as described in "Writing localizable programs" on page 171. Resource bundles can be implemented in two ways, either as subclasses of the ResourceBundle class or as Java properties files.

MessageFormat: The MessageFormat class can be used to construct strings based on parameters. As a simple example, suppose a localized application represents a particular error condition with a numeric key. When the application reports the error condition, it uses a message formatter to convert the numeric key into a meaningful string. The message formatter constructs the output string by looking up the code (the parameter) in an appropriate resource bundle and retrieving the corresponding string from the message catalog. Additional parameters—for example, another key representing the program module—can also be used in assembling the output message.

WebSphere support: The WebSphere localizable-text package wraps the Java support and extends it for efficient and simple use in a distributed environment. The primary class used by application programmers is the LocalizableTextFormatter class. Objects of this class are created, typically in server programs, but clients can also create them. LocalizableTextFormatter objects are created for specific resource-bundle names and keys. Client programs that receive LocalizableTextFormatter objects call the object's format method. This method uses the locale of the client application to retrieve the appropriate resource bundle and assemble the locale-specific message based on the key.

For example, suppose that a WebSphere client-server application supports both French and English locales; the server is using an English locale and the client, a French locale. The server creates two resource bundles, one for English and one for French. When the client makes a request that triggers a message, the server creates a LocalizableTextFormatter object containing the name of the resource bundle and the key for the message, and passes the object back to the client.

When the client receives the LocalizableTextFormatter object, it calls the object's format method, which returns the message corresponding to the key from the French resource bundle. The format method retrieves the client's locale and, using the locale and name of the resource bundle, determines the resource bundle corresponding to the locale. (If the client has set an English locale, calling the format method results in the retrieval of an English message.) The formatting of the message is transparent to the client.

In this simple client-server example, the resource bundles reside centrally with the server. The client machine does not have to install them. Part of what the WebSphere localizable-text package provides is the infrastructure to support centralized catalogs. WebSphere uses an enterprise bean, a stateless session bean provided with the localizable-text package, to access the message catalogs. When the client calls the format method on the LocalizableTextFormatter object, the following events occur internally:

1. The client application sets the time zone and locale values in the LocalizableTextFormatter object, either by passing them explicitly or through defaults.
2. A call, LocalizableTextFormatterEJBFinder, is made to retrieve a reference to the formatting enterprise bean.
3. Information from the LocalizableTextFormatter object, including the client's time zone and locale, is sent to the formatting bean.
4. The formatting bean uses the name of the resource bundle, the message key, the time zone, and the locale to assemble the language-specific message.
5. The enterprise bean returns the formatted message to the client.
6. The formatted message is inserted into the LocalizableTextFormatter object and returned by the format method.

A call to a LocalizableTextFormatter.format method requires at most one remote invocation, to contact the formatting enterprise bean. However, the LocalizableTextFormatter object can optionally cache formatted messages, eliminating the formatting call for subsequent uses. It also allows the application to set a fallback string; this means the application can still return a readable string even if it cannot access a message catalog to retrieve the language-specific string. Additionally, the resource bundles can be stored locally. The localizable-text package provides a static variable that indicates whether the bundles are stored locally (LocalizableConfiguration.LOCAL) or remotely (LocalizableConfiguration.REMOTE), but the setting of this variable applies to all applications running within a Java Virtual Machine (JVM).

The LocalizableTextFormatter class: The LocalizableTextFormatter class, found in the package com.ibm.websphere.i18n.localizabletext, is the primary programming interface for using the localizable-text package. Objects of this class contain the information needed to create language-specific strings from keys and resource bundles.

Location of message catalogs and the ApplicationName value: Applications written with the WebSphere localizable-text package can store message catalogs locally or remotely. In a distributed environment, the use of remote, centrally stored catalogs is appropriate. All applications can use the same catalogs, and administration and maintenance of the catalogs are simplified; each component does not need to store and maintain copies of the message catalogs. Local formatting is useful in test situations and appropriate under some circumstances. In order to support both local and remote formatting, a LocalizableTextFormatter object must indicate the name of the formatting application. For example, when an application formats a message by using remote, centrally stored catalogs, the message is actually formatted by a simple enterprise bean (see "WebSphere support" on page 175 for more information). Although the localizable-text package contains the code to

automate looking up the enterprise bean and issuing a call to it, the application needs to know the name of the formatting enterprise bean. Several methods in the LocalizableTextFormatter class use a value described as *application name*; this refers to the name of the formatting application, which is not necessarily the name of the application in which the value is set.

Caching messages: The LocalizableTextFomatter object can optionally cache formatted messages so that they do not have to be reformatted when needed again. By default, caching is not used, but the LocalizableTextFormatter.setCacheSetting method can be used to enable caching. When caching is enabled and the LocalizableTextFormatter.format method is called, the method determines whether the message has already been formatted. If so, the cached message is returned. If the message is not found in the cache, the message is formatted and returned to the caller, and a copy of the message is cached for future use.

If caching is disabled after messages have been cached, those messages remain in the cache until the cache is cleared by a call to the LocalizableTextFormatter.clearCache method. The cache can be cleared at any time. The cache within a LocalizableTextFormatter object is automatically cleared when any of the following methods are called on the object:

- setResourceBundleName(String resourceBundleName)
- setPatternKey(String patternKey)
- setArguments(Object[] args)
- setApplicationName(String appName)

Fallback information: Under some circumstances, it can be impossible to format a message. The localizable-text package implements a fallback strategy, making it possible to get some information even if a message cannot be correctly formatted into the desired language. The LocalizableTextFomatter object can optionally store a fallback value for a message string, the time zone, and the locale. These can be ignored unless the LocalizableTextFormatter object throws an exception.

Application-specific variables: The localizable-text package provides native support for localization based on time zone and locale, but application developers can construct messages on the basis of other values as well. The localizable-text package provides an illustrative class, LocalizableTextDateTimeArgument, which reports the date and time. The date and time information is localized by using the locale and time-zone values, but the class also uses additional variables to determine how the output is presented. The date and time information can be requested in a variety of styles, from the fully detailed to the terse. In this example, the construction of message strings is driven by three variables: the locale, the time zone, and the

style. Applications can use any number of variables in addition to locale and time zone for constructing messages. See “Using optional arguments” on page 182 for more information.

Writing a localizable application

To develop a WebSphere application that uses localizable text, application developers must do the following:

- Determine the parts of the application to be localized.
  - Identify the application elements to be localized and assign each a key.
  - Create message catalogs for each language by associating a string with each key.

  These tasks were described previously. See “Identifying localizable text” on page 171 and “Creating message catalogs” on page 172 for more information.
- Assemble language-specific strings from keys, resource bundles, and other arguments.
  - Create a LocalizableTextFormatter object.
  - Set the values within the object for the key, the name of the resource bundle, the name of the remote formatting application, and any optional arguments.
  - Call the format method on the LocalizableTextObject, which returns the assembled string.

  This section describes these tasks.

Creating a LocalizableTextFormatter object

Server programs typically create LocalizableTextFormatter objects, which are sent to clients as the result of some operation; clients format the objects at the appropriate time. Less typically, clients can create LocalizableTextFormatter objects locally. To create a LocalizableTextFormatter object, applications use one of the constructors in the LocalizableTextFormatter class:

- LocalizableTextFormatter()
- LocalizableTextFormatter(String resourceBundleName, String patternKey, String appName)
- LocalizableTextFormatter(String resourceBundleName, String patternKey, String appName, Object[] args)

The LocalizableTextFormatter object must have values set for the name of the resource bundle, the key, the name of the formatting application, and for any optional values so the object can be formatted. The LocalizableTextFormatter object can be created and the values set in one step by using the constructor that takes the necessary arguments, or the object can be created and the values set in separate steps. Values are set by using methods on the

LocalizableTextFormatter object; for setting the values manually, rather than by using a constructor, use these methods:

- setResourceBundleName(String resourceBundleName)
- setPatternKey(String patternKey)
- setApplicationName(String appName)
- setArguments(Object[] args)

Note: When values in the array of optional arguments are set within a LocalizableTextFormatter object, they are copied into the object, not referenced. If an array variable holding a value is changed after the value has been copied into the LocalizableTextFormatter object, the value in the LocalizableTextFormatter object will not reflect the change unless it is also reset.

A LocalizableTextFormatter object also has methods that can be used to set values that cannot be set when the object is created, for example:

- To toggle the cache setting for the LocalizableTextFormatter object, use the setCacheSetting(boolean setting) method (See "Caching messages" on page 177 for more information.)
- To clear the cache, use the clearLocalizableTextFormatter method
- To set fallback values, use these methods:
  - setFallBackString
  - setFallBackLocale
  - setFallBackTimeZone

  (See "Fallback information" on page 177 for more information.)

Each of these set methods also has a corresponding get method for retrieving the value. The clearLocalizableTextFormatter method unsets all values, returning the LocalizableTextFormatter object to a blank state. After clearing the object, reuse the object by setting new values and calling the format method again.

Figure 94 on page 180 creates a LocalizableTextFormatter object by using the default constructor and uses methods on the new object to set values for the key, name of the resource bundle, name of the formatting application, and fallback string on the object.

```
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: ");
  ...
}
```

Figure 94. Code example: Creating a LocalizableTextFormatter object and setting values on it

Setting localization values

The application requesting a localized message can specify the locale and time zone for which the message is to be formatted, or the application can use the default values set for the JVM. For example, a graphical user interface can allow users to select the language in which to display the menus. A default value must be set, either in the environment or programmatically, so the menus can be generated when the application first starts, but users can then change the menu language to suit their needs. Figure 95 on page 181 illustrates how to change the locale used by the application based on the selection of a menu item.

```
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");
     ...
  }
  ...
}
```

Figure 95. Code example: Setting the locale programmatically

When an application calls a format method, it can specify no arguments, which causes the message to be formatted using the JVM's default values for locale and time zone, or a combination of locale and time zone can be specified to override the JVM's defaults. (See "Generating the localized text" for more information on the arguments to the format methods.)

Generating the localized text

After the LocalizableTextFormatter object has been created and the appropriate values set, the object can be formatted to generate the string appropriate to the locale and time zone. The format methods in the LocalizableTextFormatter class perform the work necessary to generate a string from a set of message keys and resource bundles, based on locale and time zone. The LocalizableTextFormatter class provides four format methods. Each format method returns the formatted message string. The methods take a combination of java.util.Locale and java.util.TimeZone objects and throw LocalizableException objects:

- String format();
- String format(locale);
- String format(timeZone);
- String format(locale, timeZone);

The format method with no arguments uses the locale and time-zone values set as defaults for the JVM. The other format methods can be used to override either or both of these values.

Figure 96 shows the creation of a localized string for the LocalizableTextFormatter object created in Figure 94 on page 180; formatting is based on the locale set in Figure 95 on page 181. If the formatting fails, the application retrieves and uses the fallback string instead of the localized string.

```
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);
  }
  ...
}
```

Figure 96. Code example: Formatting a LocalizableTextFormatter object

Using optional arguments

The localizable-text package allows users to specify an array of optional arguments in a LocalizableTextFormatter object. These optional arguments can greatly enhance the kinds of localization done in WebSphere applications. This section describes two ways in which applications can use the optional arguments:

- To assemble and format complex strings with variable substrings
- To customize the formatting of strings, taking variables other than locale and time zone into account

Assembling complex strings

All of the keys discussed so far have represented flat strings; during localization, a string in the appropriate language is substituted for the key. The localizable-text package also supports substitution into the strings, which can include variables as placeholders. For example, an application that needs to report that an operation on a specified account was successful must provide a string like "The operation on account *number* was successful"; the variable *number* is to be replaced by the actual account number. Without support for creating strings with variable pieces, each possible string would need its own key, or the strings would have to be built phrase by phrase.

Both of these approaches quickly become intractable if a variable can take many values or if a string has several variable components. Instead, the localizable text package supports substitution of variables in strings with optional arguments. A string in a message catalog uses integers in braces—for example, {0} or {1}—to represent variable components. Figure 97 shows an example from an English message catalog for a string with a single variable substitution. (The same key in message catalogs for other languages has a translation of this string with the variable in the appropriate location for the language.)

```
successfulTransaction = The operation on account {0} was successful.
```

Figure 97. A message-catalog entry with a variable substring

The values that are substituted into the string come from an array of optional arguments. One of the constructors for LocalizableTextFormatter objects takes an array of objects as an argument, and such an array of objects can be set within any LocalizableTextFormatter object. The array is used to hold values for variable parts of a string. When a format method is called on the object, the array is passed to the format method, which takes an element of the array and substitutes it into a placeholder with the matching index in the string. The value at index 0 in the array replaces the {0} variable in the string, the value at index 1 replaces {1}, and so forth.

Figure 98 on page 184 shows the creation of a single-element argument array and the creation and use of a LocalizableTextFormatter. The element in the argument array is the account number entered by the user. The LocalizableTextFormatter is created by using a constructor that takes the array of optional arguments; this can also be set directly by using the setArguments method on the LocalizableTextFormatter object. Later in the code, the application calls the format method. The format method automatically substitutes values from the array of arguments into the string returned from the appropriate message catalog.

```
public void updateAccount(String transactionType) {
  ...
  Object[] arg = { new String(this.accountNumber)};
  ...
  LocalizableTextFormatter successLTF =
     new LocalizableTextFormatter("BankingResources",
                                  "successfulTransaction",
                                  "BankingSample",
                                  arg);
  ...
  successLTF.format(this.applicationLocale);
  ...
}
```

Figure 98. Code example: Formatting a message with a variable substring

Nesting LocalizableTextFormatter objects: The ability to substitute variables into the strings in message catalogs adds a level of flexibility to the localizable-text package, but the additional flexibility is limited, at least in an international environment, unless the substituted arguments themselves can be localized. For example, if an application needs to report that an operation on a specific account was successful, a string like "The operation on account *number* was successful"—where the only variable is an account number—can be translated and used in message catalogs for multiple languages. A string in which a variable is also a string, for example, "The *type* operation on account *number* was successful"—where the new *type* variable takes values like "deposit" and "withdrawal"—cannot be as easily translated. The values assumed by the *type* variable also need to be localized.

Figure 99 shows a message string in an English catalog with two variables, one of which will be localized, and the keys for two possible values. (The second variable in the string, the account number, is simply a number that must be substituted into the string; it does not need to be localized.)

```
sucessfulTransaction = The {0} operation on account {1} was successful.
depositOpString = deposit
withdrawOpString = withdrawal
```

Figure 99. A message-catalog entry with two variable substrings

To support localization of substrings, the localizable-text package allows the nesting of LocalizableTextFormatter objects. This is done simply by inserting a LocalizableTextFormatter object into the array of arguments for another LocalizableTextFormatter. When the format method does the variable substitution, it formats any LocalizableTextFormatter objects as it substitutes array elements for variables. This allows substrings to be formatted independently of the string in which they are embedded.

Figure 100 modifies the example in Figure 98 on page 184 to format a message with a localizable substring. First, a LocalizableTextFormatter object for the localizable substring (referring to a deposit operation) is created. This object is inserted, along with the account-number information, into the array of arguments. The array of arguments is then used in constructing the LocalizableTextFormatter object for the complete string; when the format method is called, the embedded LocalizableTextFormatter object is formatted to replace the first variable, and the account number is substituted for the second variable.

```
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);
   ...
}
```

Figure 100. Code example: Formatting a message with a localizable variable substring

Customizing the behavior of a format method

The array of optional arguments can contain simple values, like an account number to be substituted into a formatted string, and other LocalizableTextFormatter objects, representing localizable substrings to be substituted into a larger formatted string. These techniques are described in "Assembling complex strings" on page 182. In addition, the optional-argument array can contain objects of user-defined classes.

User-defined classes used as optional arguments provide application-specific format methods, which programmers can use to perform localization on the basis of any number of values, not just locale and time zone. These user-defined classes need to be available only on the systems where they are constructed and inserted into LocalizableTextFormatter objects and where the actual formatting is done; client applications do not need to install these classes.

The localizable-text package provides an example of such a user-defined class in the LocalizableTextDateTimeArgument class. This class allows date and time information to be selectively formatted according to the style values

defined in the java.text.DateFormat class and according to the constants defined by the LocalizableTextDateTimeArgument class.

The DateFormat styles determine how information is reported about a date. For example, when the DateFormat.FULL style is chosen, the twenty-second day of February in 2000 is represented in English as *Tuesday, February 22, 2000*. When the DateFormat.SHORT style is used, the same date is represented as *2/22/00*. The valid values are:

- DateFormat.FULL
- DateFormat.LONG
- DateFormat.MEDIUM
- DateFormat.SHORT
- DateFormat.DEFAULT

The LocalizableTextDateTimeArgument class defines constants that can be used to request only date or time information, or both, either in date-time order or in time-date order. The defined values are:

- LocalizableTextDateTimeArgument.TIME
- LocalizableTextDateTimeArgument.DATE
- LocalizableTextDateTimeArgument.TIMEANDDATE
- LocalizableTextDateTimeArgument.DATEANDTIME

An object of a user-defined class like the LocalizableTextDateTimeArgument class can be set in the optional-argument array of a LocalizableTextFormatter object, and when the LocalizableTextFormatter object attempts to format the user-defined object, it calls the format method on that object. That format method, written by the application developer, can do whatever is appropriate with the application-specific values. In the case of the LocalizableTextDateTimeArgument class, the format method determines if date, time, or both are required, formats them according to the DateFormat value, and assembles them in the order requested in the LocalizableTextDateTimeArgument style. The date and time information are also affected by the locale and time-zone values, but the refinements in the formatting are accomplished by the DateFormat class and the user-defined values.

The string assembled from a user-defined class like the LocalizableTextDateTimeArgument class can then be substituted into a larger string, just as the return values of nested LocalizableTextFormatter objects can be. When writing such user-defined classes, it is helpful to think of them as specialized versions of the generic LocalizableTextFormatter class, and the way in which the LocalizableTextFormatter class is written provides a model for writing user-defined classes.

Structure of the LocalizableTextFormatter class: The LocalizableTextFormatter class is a general-purpose class for localizable text. It extends the java.lang.Object class and implements the java.io.Serializable interface and four localizable-text interfaces:

- LocalizableTextLTZ
- LocalizableTextL
- LocalizableTextTZ
- LocalizableText

Each of the localizable-text interfaces implemented by the LocalizableTextFormatter class implements the Localizable interface (which simply extends the Serializable interface) and defines a single format method:

- The LocalizableTextLTZ interface defines format(locale, timezone).
- The LocalizableTextL defines format(locale).
- The LocalizableTextTZ defines format(timezone).
- The LocalizableText defines format().

Because the LocalizableTextFormatter class implements all four of these interfaces, it must provide an implementation for each of these format methods.

Writing a user-defined class: A user-defined class must implement at least one of the localizable-text interfaces and its corresponding format method, as well as the Serializable interface. If the class implements more than one of the localizable-text interfaces and format methods, the order of evaluation of the interfaces is:

1. LocalizableTextLTZ
2. LocalizableTextL
3. LocalizableTextTZ
4. LocalizableText

For example, the LocalizableTextDateTimeArgument class implements only the LocalizableTextLTZ interface, as shown in Figure 101 on page 188.

```
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
{
   ...
}
```

Figure 101. Code example: The structure of the LocalizableTextDateTimeArgument class

A user-defined class must contain a constructor and an implementation of the format methods as defined in the localizable-text interfaces that the class implements. It can also contain other methods as needed. The LocalizableTextDateTimeArgument class contains a constructor, a single format method, an equality method, a hash-code generator, and a string-conversion method.

```
...
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()
   { ... }
}
```

Figure 102. Code example: The methods in the LocalizableTextDateTimeArgument class

Each format method in the user-defined class can do whatever is appropriate for the application. In the LocalizableTextDateTimeArgument class, the format method (see Figure 103 on page 190 for the implementation) examines the setting of the date-time style set within the object, for example, DATEANDTIME. It then assembles the requested information in the requested order, according to the date-format value.

```
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;
}
```

Figure 103. Code example: The format method in the LocalizableTextDateTimeArgument class

An application can create a LocalizableTextDateTimeArgument object (or an object of any other user-defined class) and place it in the optional-argument array of a LocalizableTextFormatter object. When the LocalizableTextFormatter object reaches the user-defined object, it will attempt to format it by calling

the object's format method. The returned string is then substituted for a variable as the LocalizableTextFormatter processes each element in the array of optional arguments.

Deploying the formatter enterprise bean

The **LocalizableTextEJBDeploy** tool is used by the application deployer to create a deployed LocalizableText JAR file for the LocalizableText service. You must deploy the enterprise bean for each server per application where the service is to be run. There may be servers for which the LocalizableText service does not need to be installed. The same deployed JAR file can be included in several application Enterprise Archive (EAR) files, but additional steps are required when the EAR file is deployed. The application deployer must also make sure that the application resource bundles are added to the application EAR file as files. The server's CLASSPATH variable must be adjusted to include the deployed location of the EAR file. This is so that the resource bundles can be located on the host and server.

Setting up the tool

Before the **LocalizableTextEJBDeploy** tool can be used, the following conditions must be met:

- A JAR file called ltext.jar must exist in the lib directory under the WebSphere installation directory.
- A working directory has to exist for the tool to use. The location is passed to the tool.

Using the LocalizableTextEJBDeploy Tool

After the prerequisites for the tool have been met, the tool can be used to deploy formatting session beans. The tool requires values for five arguments:

```
LocalizableTextEJBDeploy -a <appName>
          -h <hostName>
          -i <installationDir>
          -s <serverName>
          -w <workingDir>
```

The required arguments, which can be specified in any order, follow:

- appName: The name of the formatting session bean. This name is used in LocalizableTextFormatter objects to specify where the actual formatting takes place. If a LocalizableTextFormatter object specifies a name that cannot be resolved, an exception is thrown by the format method.
- hostName: The name of the machine on which the formatting session bean is deployed. This value specified here is case sensitive on all platforms.
- installationDir: The location at which WebSphere Application Server is installed on the machine.
- serverName: The name of the WebSphere Application Server. If this argument is not specified, the value `Default Server` is used.

- workingDir: The name of the working directory for the tool to use.

After the tool is run, a deployed JAR file is located in the working directory specified to the tool. This JAR file can be included in the application EAR or WAR file.

Special considerations when deploying a LocalizableText enterprise bean: When the application is being deployed onto a host and server, during the deployment process you will be asked if you want to regenerate the deployment code for the LocalizableText enterprise bean. Do not redeploy the bean. If the bean is redeployed, the JNDI name will be wrong.

If more than one LocalizableText enterprise bean is deployed with an application, there are two ways to handle the situation.

- Run the **LocalizableTextEJBDeploy** tool for each host/server combination. The tool generates a unique JNDI name for each enterprise bean. Otherwise, even though the bean has been deployed on multiple hosts and servers, the JNDI name is not changed, and there is only one entry in the naming service.
- During the deployment of the application, change the JNDI name for the localizable-text bean should begin with `com/ibm/websphere/i18n/localizabletext/homes/`. This should be followed by the application and host names, the server name, and by the string `LocalizableTextEJBHome`, all separated by two underscores, as follows:

  ```
  <AppName>/<HostName>__<ServerName> __LocalizableTextEJBHome
  ```

Appendix A. Changes for version 1.1 of the EJB specification

WebSphere Application Server supports version 1.1 of the EJB specification. This appendix describes features that are new or have changed in version 1.1 and discusses migration issues for enterprise beans written to version 1.0 of the EJB specification.

New and updated features

The following enterprise bean features are new or have changed for version 1.1.

- Environmental dependencies for enterprise beans are now specified using entries in a JNDI naming context. An instance of an enterprise bean creates a javax.naming.InitialContext object by invoking the constructor with no arguments specified. It looks up the environment naming context by using the InitialContext object under the name java:comp/env.
- Primary keys are handled differently in version 1.1 of the EJB specification. Entity bean providers are not required to specify the primary key class for entity beans with container-managed persistence (CMP), enabling the deployer to select the primary key fields when the bean is deployed into a container.
- The deployment descriptor has enhanced support for application assembly.

Migrating from version 1.0 to version 1.1

From the client's perspective, enterprise beans written to version 1.1 of the EJB specification appear nearly identical to enterprise beans written to version 1.0 of the specification. However, the following EJB 1.1 changes do affect clients:

- Enterprise beans written to version 1.1 of the EJB specification are registered in a different part of the JNDI namespace. For example, a client can look up the initial context of a version 1.0 enterprise bean in JNDI by using the **initialContext.lookup** method as follows:

  ```
  initialContext.lookup("com/ibm/Hello")
  ```

  The JNDI lookup for the equivalent version 1.1 enterprise bean is:

  ```
  initialContext.lookup("java:comp/env/ejb/Hello")
  ```

- The UserTransaction object is obtained differently for enterprise beans written to version 1.1 of the EJB specification. Under version 1.0, it was obtained as:

```
initialContext.lookup("jta/UserTransaction")
```

Under version 1.1, it is obtained as:

```
initialContext.lookup("java:comp/UserTransaction")
```

- Because entity beans written to version 1.1 of the EJB specification now support primitive primary keys (instead of having to encapsulate them in a primary key class), the client needs to look up these primitive keys directly. For example, a client can look up a primitive key of the type java.lang.Integer as follows:

  ```
  accountHome.findByPrimaryKey(new Integer(5))
  ```

  Primary key classes are still supported, although their use for primitive data types is deprecated.

From the application developer's perspective, the following changes need to be made to make enterprise beans written to version 1.0 of the EJB specification compatible with version 1.1 of the specification.

- All deployment descriptors must be converted to the XML format specified in version 1.1 of the EJB specification.
- In general, enterprise beans written to version 1.0 of the EJB specification are compatible with version 1.1. However, you need to modify or recompile enterprise bean code in the following cases:
  - The return value of the ejbCreate method must be modified for all entity beans with CMP. The ejbCreate method is now required to return the same type as the primary key; the actual value returned must be null. These beans also must be recompiled. For more information, see "Implementing the ejbCreate and ejbPostCreate methods" on page 39
  - If the javax.jts.UserTransaction interface is used. This interface has been renamed to javax.transaction.UserTransaction. Enterprise beans that use this interface must be modified to use the new interface name. There have also been minor changes to the exceptions thrown by this interface.
  - If the getCallerIdentity or isCallerInRole methods of the javax.ejb.EJBContext interface are used. These methods were deprecated because the javax.security.Identity class is deprecated under the Java 2 platform.
  - If an entity bean uses the UserTransaction interface, which is not permitted under version 1.1 of the EJB specification.
  - If an entity bean whose finder methods do not define the FinderException in the methods' throws classes. Under version 1.1, the finder methods of entity beans must define this exception.

- If an entity bean uses the UserTransaction interface and implements the SessionSynchronization interface. Entity beans can neither use the UserTransaction interface nor implement the SessionSynchronization interface under version 1.1.
- If a stateless session bean implements the SessionSynchronization interface. Stateless session beans should not implement the SessionSynchronization interface under version 1.1.
- If an enterprise bean violates any of the new semantic restrictions defined in version 1.1 of the EJB specification.
- Throwing the javax.ejb.RemoteException exception from the bean implementations is deprecated in version 1.1. This exception should be replaced by the javax.ejb.EJBException or a more specific exception such as the javax.ejb.CreateException. The javax.ejb.EJBException inherits from the javax.ejb.RuntimeException and does not need to be explicitly declared in throws clauses.

  Declare the javax.ejb.RemoteException exception in the remote and home interfaces, as required by RMI. Throwing this exception directly by the bean implementation is deprecated. However, it can be thrown by the container due to a system exception or by mapping an exception thrown by the bean implementation.

Appendix B. Example code provided with WebSphere Application Server

This appendix contains information on the example code provided with the WebSphere Application Server.

Information about the examples described in the documentation

The example code discussed throughout this document is taken from a set of examples provided with the product. This set of examples is composed of the following main components:

- The Account entity bean, which models either a checking or savings bank account and maintains the balance in each account. An account ID is used to uniquely identify each instance of the bean class and to act as the primary key. The persistent data in this bean is container managed and consists of the following variables:
  - *accountId*—The account ID that uniquely identifies the account. This variable is of type long.
  - *type*—An integer that identifies the account as either a savings account (1) or a checking account (2). This variable is of type int.
  - *balance*—The current balance of the account. This variable is of type float.

  The major components of this bean are discussed in "Developing entity beans with CMP" on page 33.
- The AccountBM entity bean, which is nearly identical to the Account entity bean; however, the AccountBM bean implements bean-managed persistence. This bean is not used by any other enterprise bean, application, or servlet contained in the documentation example set. The major components of this bean are discussed in "Developing entity beans with BMP" on page 103.
- The Transfer session bean, which models a funds transfer session that involves moving a specified amount between two instances of an Account bean. The bean contains two methods: the transferFunds method transfers funds between two accounts, the getBalance method retrieves the balance for a specified account. The bean is stateless. The major components of this bean are discussed in "Developing session beans" on page 50.
- The CreateAccount servlet, which can be used to easily create new bank accounts (and corresponding Account bean instances) with the specified account ID, account type, and initial balance. Although this servlet is designed to make it easy for you to create accounts and demonstrate the other components in the example set, it also illustrates servlet interaction

with an entity bean. This servlet is discussed in "Chapter 7. Developing servlets that use enterprise beans" on page 91.

- The TransferApplication Java application, which provides a graphical user interface that was built with the abstract windowing toolkit (AWT). The application creates an instance of the Transfer session bean, which is then manipulated to transfer funds between two selected accounts or to get the balance for a specified account. The TransferApplication code implements many of the requirements for using enterprise beans in an EJB client. The parts of this application that are relevant to interacting with an enterprise bean are discussed in "Chapter 6. Developing EJB clients" on page 77.
- The TransferFunds servlet, which is a servlet version of the TransferApplication Java application. This servlet is provided so that you can compare the use of enterprise beans between a Java application and a Java servlet that basically are doing the same tasks. This document does not discuss this servlet in any detail.

Note: The example code in the documentation was written to be as simple as possible. The goal of these examples is to provide code that teaches the fundamental concepts of enterprise bean and EJB client development. It is not meant to provide an example of how a bank (or any similar company) possibly approaches the creation of a banking application. For example, the Account bean contains a *balance* variable that has a type of float. In a real banking application, you must not use a float type to keep records of money; however, using a class like java.math.BigDecimal or a currency-handling class within the examples would complicate them unnecessarily. Remember this as you examine these examples.

Information about other examples

Table 3 provides a summary of the enterprise bean-specific examples provided with the EJB server

Table 3. Examples available with the EJB server

Name	Bean types	EJB client types	Additional information
Hello	Stateless session	Java servlet	Very simple example of a session bean.
Increment	CMP entity	Java servlet	Very simple example of an entity bean.

Appendix C. Extensions to the EJB Specification

This appendix briefly discusses functional extensions to the EJB Specification that are available in the EJB server environments contained in WebSphere Application Server. These extensions are specific to WebSphere Application Server and use of these features is supported only with VisualAge for Java, Enterprise Edition. For information on implementing these features, consult your VisualAge for Java documentation.

Access beans

Access beans are Java components that adhere to the Sun Microsystems JavaBeans™ Specification and are meant to simplify development of EJB clients. An access bean adapts an enterprise bean to the JavaBeans programming model by hiding the home and remote interfaces from the access bean user (that is, an EJB client developer).

There are three types of access beans, which are listed in ascending order of complexity:

- Java bean wrapper—Of the three types of access beans, a Java bean wrapper is the simplest to create. It is designed to allow either a session or entity enterprise bean to be used like a standard Java bean and it hides the enterprise bean home and remote interfaces from you. Each Java bean wrapper that you create extends the com.ibm.ivj.ejb.access.AccessBean class.
- Copy helper—A copy helper access bean has all of the characteristics of a Java bean wrapper, but it also incorporates a single copy helper object that contains a local copy of attributes from a remote entity bean. A user program can retrieve the entity bean attributes from the local copy helper object that resides in the access bean, which eliminates the need to access the attributes from the remote entity bean.
- Rowset—A rowset access bean has all of characteristics of both the Java bean wrapper and copy helper access beans. However, instead of a single copy helper object, it contains multiple copy helper objects. Each copy helper object corresponds to a single enterprise bean instance.

VisualAge for Java provides a SmartGuide to assist you in creating or editing access beans.

Associations between enterprise beans

In the EJB server environment, an association is a relationship that exists between two CMP entity beans. There are three types of associations: one-to-one and one-to-many. In a one-to-one association, a CMP entity bean is associated with a single instance of another CMP entity bean. For example, an Employee bean could be associated with only a single instance of a Department bean, because an employee generally belongs only to a single department.

In a one-to-many association, a CMP entity bean is associated with multiple instances of another CMP entity bean. For example, a Department bean could be associated with multiple instances of an Employee bean, because most departments are made up of multiple employees.

The Association Editor is used to create or edit associations between CMP entity beans in VisualAge for Java.

Inheritance in enterprise beans

In Java, *inheritance* is the creation of a new class from an existing class or a new interface from an existing interface. The EJB server environment permits two forms of inheritance: standard class inheritance and EJB inheritance. In standard class inheritance, the home interface, remote interface, or enterprise bean class inherits properties and methods from base classes that are not themselves enterprise bean classes or interfaces.

In enterprise bean inheritance, by comparison, an enterprise bean inherits properties (such as CMP fields and association ends), methods, and method-level control descriptor attributes from another enterprise bean that resides in the same group.

VisualAge for Java provides a SmartGuide to assist you in implementing inheritance in enterprise beans.

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS DOCUMENT "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OR CONDITIONS OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will

be incorporated in new editions of the document. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

For Component Broker:
IBM Corporation
Department LZKS
11400 Burnet Road
Austin, TX 78758
U.S.A.

For TXSeries:
IBM Corporation
ATTN: Software Licensing
11 Stanwix Street
Pittsburgh, PA 15222
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples may include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks and service marks

The following terms are trademarks or registered trademarks of the IBM Corporation in the United States, other countries, or both:

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
SAA
SecureWay
TeamConnection
Transarc
TXSeries
VSE/ESA
VTAM
VisualAge
WebSphere

Domino, Lotus, and LotusScript are trademarks or registered trademarks of Lotus Development Corporation in the United States, other countries, or both.

Tivoli is a registered trademark of Tivoli Systems, Inc. in the United States, other countries, or both.

ActiveX, Microsoft, Visual Basic, Visual C++, Visual J++, Windows, Windows NT, and the Windows 95 logo are trademarks or registered trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Some of this documentation is based on material from Object Management Group bearing the following copyright notices:

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

This software contains RSA encryption code.

Java Compatible™ Enterprise Edition

Other company, product, and service names may be trademarks or service marks of others.

Index

A

ACID properties 6
administering
  WebSphere Application Server 11
  workload management service 5
afterBegin method 24
afterCompletion method 24
Application Assembly Tool 28, 30
atomicity 6
authentication 3
authorization 4

B

bean class
  entity beans (BMP) 14, 104
  entity beans (CMP) 14, 34
  session beans 15
  variables (entity with BMP) 105
  variables (entity with CMP) 35
bean-managed persistence 5, 15, 103
beforeCompletion method 24
business methods
  entity beans (BMP) 107
  entity beans (CMP) 36
  session beans 53

C

CDS (DCE) 6
CICS 9
CLASSPATH environment variable
  EJB server 29
clearLocalizableTextFormatter method 179
com.ibm.websphere.i18n.localizabletext package 174, 175
Command interface 141, 144, 151
CommandException class 144, 160
commands 140
  Command interface 141, 144, 151
  CommandException class 144, 160
  CommandTarget interface 143, 159, 160, 166, 167
  CompensableCommand interface 141, 145, 152
  DistributedException class 144
commands *(continued)*
  exception classes 144
  execute method 144
  executeCommand method 159, 160, 166, 167
  getCommandTarget method 145, 161, 164
  getCommandTargetName method 145, 161
  getCompensatingCommand method 145, 152
  getTargetPolicy method 164
  hasOutputProperties method 145, 160
  isReadyToCallExecute method 144, 151
  listMappings method 162
  LocalTarget class 162
  performExecute method 145, 151, 160, 169
  registerCommand method 162, 164
  reset method 144, 151
  setCommandTarget method 145, 161
  setCommandTargetName method 145, 161
  setDefaultTargetName method 162, 163
  setHasOutputProperties method 145
  setOutputProperties method 145, 147, 151
  setTargetPolicy method 164
  target 143, 159, 161, 166, 167, 169
  target (enterprise bean) 159
  target (servlet) 166, 167, 169
  target policy 143, 144, 161, 162, 163, 164, 165
  TargetableCommand interface 141, 142, 143, 145, 147, 151, 161, 167
  TargetableCommandImpl class 142, 147, 148, 164
  TargetPolicy interface 144, 162, 164
  TargetPolicyDefault class 144, 162
commands *(continued)*
  UnauthorizedAccessException class 144
  unregisterCommand method 162, 164
  UnsetInputPropertiesException class 144
  user-defined exception classes 144
CommandTarget interface 143, 159, 160, 166, 167
committing
  transactions 7, 88, 124
CompensableCommand interface 141, 145, 152
components
  EJB server 1
  entity beans 14
  entity beans (BMP) 103
  entity beans (CMP) 33
  session beans 15, 51
connection manager 120
connections (database)
  allocating 121
  deallocating 122
  entity beans (BMP) 119
  managing in EJB server 120
consistency 6
container-managed persistence 5, 15, 33
coordinators 8
create method
  entity beans 26
  entity beans (BMP) 107, 114, 115
  entity beans (CMP) 39, 43, 44
  session beans 23, 56, 62, 63
CreateException class 38, 40, 44, 63, 108, 115
creating
  deployment descriptors 30
  EJB home objects in EJB clients 82
  EJB modules 30, 67
  EJB objects in EJB clients 79
  enterprise beans in servlets 95, 98
creation state
  entity beans 25
  session beans 23

Current interface (CORBA) 74

D

data sources 9
databases 9
  allocating connections 121
  deallocating connections 122
  EJB object references 120
  EJB server 31
  getting connections 119
  manipulating data 123
DataSource interface 120, 122
DB2 database 9
DCE CDS 6
deploying
  enterprise beans 19, 27, 28, 66
deployment descriptors 17
  creating 30
  entity bean attributes 17
  environment variable attributes 56, 58
  security attributes 17, 69, 74
  session bean attributes 17
  transaction attributes 17, 69, 70, 72
destroy method (servlets) 91
developing
  EJB applications 20
  EJB clients 77, 91
  enterprise beans 27, 28, 33
  entity beans (BMP) 103
  entity beans (CMP) 33
  servlets with enterprise beans 91
  session beans 50, 51
distributed exceptions 129
  DistributedException class 130
  DistributedExceptionEnabled interface 130, 132
  DistributedExceptionInfo class 130, 133
  ExceptionInstantiationException class 130
  getException method 131, 132
  getExceptionInfo method 131, 132
  getMessage method 131, 132
  getOriginalException method 131, 132
  getPreviousException method 131, 132
  localization 132
  printStackTrace method 131, 132
  printSuperStackTrace method 132
distributed exceptions *(continued)*
  user-defined 133, 134, 135, 136, 139
distributed transactions 7
DistributedException class 130, 144
DistributedExceptionEnabled interface 130, 132
DistributedExceptionInfo class 130, 133
DNS 6
doGet method (servlets) 91, 97, 98, 99, 100
doPost method (servlets) 91
DuplicateKeyException 38
DuplicateKeyException class 40, 108
durability 6

E

EJB applications
  developing 20
  examples 21
EJB clients 9
  creating EJB object home objects 82
  creating EJB objects 79
  developing 77, 91
  managing transactions 86
  naming and communications 9
  removing EJB objects 86
  required Java packages 78
  security 9
  threads 9
  transactions 9
EJB home class 14, 15, 19, 43
EJB home objects 15, 19, 114
  creating in EJB clients 82
EJB JAR files 17
EJB modules 17
  creating 30, 67
  deployment descriptors 17
EJB object class 14, 15, 19, 63
EJB objects 15, 19
  creating in EJB clients 79
  invalid 84
  references to databases 120
  removing in EJB clients 86
EJB server 2
  CLASSPATH environment variable 29
  components 1
  databases 31
  example code 198
  finder helper interface 30
EJB server *(continued)*
  managing database connections 120
  prerequisite software 28
  services 2
  tools 2, 27, 28
ejbActivate method
  entity beans 26
  entity beans (BMP) 112
  entity beans (CMP) 41
  session beans 24, 61
ejbCreate method
  entity beans 26
  entity beans (BMP) 104, 107, 114, 115
  entity beans (CMP) 34, 39, 43, 44
  session beans 23, 51, 52, 56, 62, 63
EJBException class 38, 40, 41, 51, 54, 56
ejbFindByPrimaryKey method
  entity beans (BMP) 109
  entity beans (CMP) 45
  primary key 45
EJBHome interface 43, 62, 65, 114
ejbLoad method 26
  entity beans (BMP) 112
  entity beans (CMP) 41
EJBObject interface 46, 64, 65, 117
ejbPassivate method
  entity beans 26
  entity beans (BMP) 112
  entity beans (CMP) 41
  session beans 24, 61
ejbPostCreate method 26
  entity beans (BMP) 104, 107, 114, 115
  entity beans (CMP) 34, 39, 43, 44
ejbRemove method
  entity beans (BMP) 112
  entity beans (CMP) 41
  session beans 25, 61
ejbStore method 26
  entity beans (BMP) 112
  entity beans (CMP) 41
enterprise beans 13
  creating in servlets 95, 98
  deploying 19, 27, 28, 66
  developing 27, 28, 33
  EJB module 17
  life cycle 23
  managing transactions 124

enterprise beans *(continued)*
obtaining variable values 106, 120
packages (Java) 67
packaging 17
reentrancy 66
threads 66
using in servlets 91, 93
entity beans 13
bean class (BMP) 104
bean class (CMP) 34
business methods (BMP) 107
business methods (CMP) 36
components 14
components (BMP) 103
components (CMP) 33
creation state 25
deployment descriptor attributes 17
developing (BMP) 103
developing (CMP) 33
home interface (BMP) 114
home interface (CMP) 43
instance variables (BMP) 105
instance variables (CMP) 35
life cycle 25
pooled state 25
primary key class (BMP) 118
primary key class (CMP) 47
ready state 26
remote interface (BMP) 117
remote interface (CMP) 46
removal state 26
EntityBean interface 34, 41, 104, 112
Enumeration interface 45, 116
environment 60
environment naming context 60
environment variables
deployment descriptor attributes 56, 58
ephemeral processes 8
equals method 48
examples
documentation code 197
EJB applications 21
provided with EJB server 198
exception classes
CommandException 144, 160
CreateException 38, 40, 44, 63, 108, 115
DistributedException 144
DuplicateKeyException 38, 40, 108
EJBException 38, 40, 41, 51, 54, 56
exception classes *(continued)*
ExceptionInstantiationException 130 132
FinderException 38, 45, 54, 109, 110, 116
NoSuchObjectException 25, 85
ObjectNotFoundException 38, 109
RemoteException 40, 41, 43, 44, 45, 46, 54, 62, 63, 64, 108, 112, 114, 115, 116, 117
RemoveException 25, 38, 41, 112
RuntimeException 51
TransactionRequiredException 70
UnauthorizedAccessException 144
UnsetInputPropertiesException 144
user-defined 37, 47, 79, 117, 133, 134, 144
ExceptionInstantiationException class 130
exceptions
chaining 129
distributed 129
execute method 144
executeCommand method 159, 160, 166, 167

F

findByPrimaryKey method 45, 54, 116
entity beans (BMP) 114
entity beans (CMP) 43
finder helper interface 30
finder methods
entity beans (BMP) 109, 116
entity beans (CMP) 45
FinderException 38
FinderException class 45, 54, 109, 110, 116

G

getCommandTarget method 145, 161, 164
getCommandTargetName method 145, 161
getCompensatingCommand method 145, 152
getEJBHome method 65
getEJBMetaData method 65
getException method 131, 132
getExceptionInfo method 131, 132
getHandle method 65
getInitialContext method 56
getMessage method 131, 132
getOriginalException method 131, 132
getPreviousException method 131, 132
getPrimaryKey method 65
getTargetPolicy method 164

H

hashCode method 48
hasOutputProperties method 145, 160
home interface
entity beans (BMP) 14, 114
entity beans (CMP) 14, 43
finding with JNDI 81
session beans 15, 62
HTML
embedding servlets 91, 100
HTTP 9
HttpServlet class 93

I

IIOP 9
IMS 9
init method (servlets) 91, 95
INITIAL_CONTEXT_FACTORY property 56, 80
InitialContext interface 56, 81
initializing
servlets 95
instance variables
entity beans with BMP 105
entity beans with CMP 35
servlets 94
session beans 52
internationalization
techniques 171
isIdentical method 65
isolation 6, 72
isReadyToCallExecute method 144, 151

J

jar command 28
java.io package 66
java.jts package 70
java.lang package 51
java.rmi package 25, 40, 41, 43, 44, 45, 46, 54, 62, 63, 64, 66, 78, 85, 108, 112, 114, 115, 116, 117
java.sql package 119, 123
java.text.MessageFormat class 174, 175
java.util.Locale class 174
java.util package 45, 78, 116
java.util.ResourceBundle class 174
java.util.TimeZone class 174
javac command 28, 30

javax.ejb package 25, 34, 38, 40, 41, 43, 44, 45, 46, 51, 54, 56, 61, 62, 63, 64, 65, 78, 104, 108, 109, 110, 112, 114, 115, 116, 117
javax.naming package 56, 78, 80, 81
javax.rmi.PortableRemoteObject.narrow method 59, 83
javax.servlet.http package 93
javax.servlet package 93
javax.transaction package 8, 87, 125
JDBC 5, 119, 123
JNDI 6, 59, 80, 87
  finding home interfaces 81
  INITIAL_CONTEXT_FACTORY property 80
  PROVIDER_URL property 80
JSP 11, 100
JSQL 123
JTA 5, 87

L

LDAP 6
life cycle
  creation state (entity) 25
  creation state (session) 23
  enterprise beans 23
  entity beans 25
  pooled state (entity) 25
  pooled state (session) 24
  ready state (entity) 26
  ready state (session) 24
  removal state (entity) 26
  removal state (session) 25
  session beans 23
listMappings method 162
localizable text 170
  application analysis 171
  application-specific arguments 177
  assembling complex strings 182
  caching messages 177
  customized formatting 182, 185
  deploying formatter bean 191
  fallback information 177
  format methods 175, 176
  formatting application 176
  formatting details 175
  Java support 174
  java.text.MessageFormat class 174, 175
  java.util.Locale class 174
  java.util.ResourceBundle class 174
  java.util.TimeZone class 174
localizable text *(continued)*
  LocalizableConfiguration class 176
  LocalizableTextDateTimeArgument class 185
  LocalizableTextEJBDeploy tool 191
  LocalizableTextFormatter class 175, 176
  locating message catalogs 173, 176
  message catalogs 172
  naming message catalogs 173
  nested formatting 184
  optional arguments 182, 184, 185
  rationale 170
  resource bundles 172
  tasks 178
  techniques 171
  user-defined formatting 185
  variable substrings 182, 185
  variable substrings (localized) 184
  WebSphere support 174, 175
LocalizableConfiguration class 176
LocalizableException class 181
LocalizableText interface 187
  format method 187
  user-defined implementations 187
LocalizableTextDateTimeArgument class 185
LocalizableTextEJBDeploy tool 191
  prerequisites 191
  syntax 191
  usage 191
LocalizableTextFormatter class 175, 176
  application-specific arguments 177
  caching messages 177
  clearLocalizableTextFormatter method 179
  constructors 178
  fallback information 177, 181
  format methods 175, 176, 181, 187
  locale 181
  setApplicationName method 177, 179
  setArguments method 177, 179
  setCacheSetting method 179
  setFallBackLocale method 179
  setFallBackString method 179
LocalizableTextFormatter class *(continued)*
  setFallBackTimeZone method 179
  setPatternKey method 177, 179
  setResourceBundleName method 177, 179
  setting values 179, 180
  time zone 181
LocalizableTextLT interface 187
  format method 187
  user-defined implementations 187
LocalizableTextLTZ interface 187
  format method 187
  LocalizableTextDateTimeArgument class 187
  user-defined implementations 187
LocalizableTextZ interface 187
  format method 187
  user-defined implementations 187
localization 170
  application analysis 171
  rationale 170
  techniques 171
LocalTarget class 162
lookup method 59

M

managing
  database connections in EJB server 120
  transactions in EJB clients 86
  transactions in enterprise beans 124
message catalogs 172
  location 173
  naming 173
MQSeries 9

N

naming service 6
NoSuchObjectException class 25, 85

O

ObjectNotFoundException 38
ObjectNotFoundException class 109
Oracle database 9

P

packages (Java)
  enterprise beans 67
  required for EJB clients 78

packaging
enterprise beans 17
performExecute method 145, 151, 160, 169
persistence 15
persistence management service 5
pooled state
entity beans 25
session beans 24
prepare phase 8
PreparedStatement interface 123
primary key 14
and remove method 65
specifying at deployment 47
unknown 49
primary key class 14
entity beans (BMP) 118
entity beans (CMP) 47
principal contexts 74
printStackTrace method 131, 132
printSuperStackTrace method 132
Programming Model Extensions 129
command package 140
distributed-exception package 129
localizable-text package 170
PROVIDER_URL property 56, 80

R

ready state
entity beans 26
session beans 24
recoverable processes 8
reentrancy
in enterprise beans 66
refreshing
EJB objects for session beans 84
registerCommand method 162, 164
remote interface 47
entity beans (BMP) 14, 117
entity beans (CMP) 14, 46
session beans 15, 63
RemoteException class 40, 41, 43, 44, 45, 46, 54, 62, 63, 64, 66, 108, 112, 114, 115, 116, 117
removal state
entity beans 26
session beans 25
remove method 65
entity beans 26
invoking in EJB clients 86
session beans 25, 61
RemoveException 38
RemoveException class 25, 41, 112
removing
EJB objects in EJB clients 86
reset method 144, 151
resolution phase 8
resource bundles
in EJB JAR files 67
location 173
naming 173
obtaining variable values 36, 37, 80, 81, 94
ResultSet interface 123
RMI 9
valid parameters 66
rolling back
transactions 7, 88, 124
RuntimeException class 51

S

security 9
deployment descriptor attributes 17, 69, 74
security service 3
Serializable interface 66
services
naming 6
persistence 5
security 3
transaction 6
workload management 5
servlets
compared to JSP 100
creating enterprise beans 95, 98
embedding in HTML 91, 100
initializing 95
instance variables 94
making thread safe 101
processing user input 97, 99, 100
standard methods 91
using enterprise beans 91, 93
Web server requirements 11, 91
session beans 13
components 15, 51
creation state 23
deployment descriptor attributes 17
developing 50, 51
home interface 62
instance variables 52
life cycle 23
pooled state 24
ready state 24
remote interface 63
removal state 25
stateful 16, 52, 56, 62, 63, 124
session beans *(continued)*
stateless 16, 52, 56, 62, 63, 83, 124
SessionBean interface 51, 61
SessionSynchronization interface 52
setApplicationName method 177, 179
setArguments method 177, 179
setCacheSetting method 179
setCommandTarget method 145, 161
setCommandTargetName method 145, 161
setDefaultTargetName method 162, 163
setEntityContext method 25
entity beans (BMP) 112
entity beans (CMP) 41, 43
setFallBackLocale method 179
setFallBackString method 179
setFallBackTimeZone method 179
setHasOutputProperties method 145
setOutputProperties method 145, 147, 151
setPatternKey method 177, 179
setResourceBundleName method 177, 179
setSessionContext method 23, 61, 62
setTargetPolicy method 164
SQL Server 9
stateful session beans 16, 52, 56, 62, 63, 124
stateless session beans 16, 52, 56, 62, 63, 83, 124
static variables (restrictions) 35, 52, 105

T

target policy 143, 161, 162, 163, 164, 165
custom 164, 165
default 143, 161, 162, 163, 164
TargetableCommand interface 141, 142, 143, 145, 147, 151, 161, 167
TargetableCommandImpl class 142, 147, 148, 164
TargetPolicy interface 144, 162, 164
TargetPolicyDefault class 144, 162
threads 9
in enterprise beans 66
in servlets 101
tools
Application Assembly Tool 28

tools *(continued)*
EJB server 27, 28
VisualAge for Java 27
transaction service 6
TransactionRequiredException class 70
transactions 6, 9
bean managed 124
committing 7, 88, 124
coordinators 8
deployment descriptor attributes 17, 69, 70, 72
distributed 7
managing in EJB clients 86
managing in enterprise beans 124
prepare phase 8
resolution phase 8
rolling back 7, 88, 124
two-phase commit 8
two-phase commit 8

U

UnauthorizedAccessException class 144
unregisterCommand method 162, 164
unsetEntityContext method 26
entity beans (BMP) 112
entity beans (CMP) 41, 43
UnsetInputPropertiesException class 144
user contexts 74
user-defined exception classes 37, 47, 79, 117, 144
distributed exceptions 133, 134, 135, 136, 139
user-defined exceptions
in EJB JAR files 67
UserTransaction interface 8, 87, 125

V

variables
bean class (entity with BMP) 105
bean class (entity with CMP) 35
in servlets 94
obtaining values from enterprise beans 106, 120
variables *(continued)*
obtaining values from resource bundles 36, 37, 80, 81, 94
static (restrictions) 35, 52, 105
VisualAge for Java 27

W

Web servers
servlets and JSP 11, 91
WebSphere Administrative Console 11, 28, 31
WebSphere Application Server
administering 11
example code 197
WebSphere Programming Model Extensions 129
command package 140
distributed-exception package 129
localizable-text package 170
workload management service 5
administering 5