WebSphere Application Server

Writing Enterprise Beans in WebSphere

Version 3.5

SC09-4431-02

WebSphere Application Server

Writing Enterprise Beans in WebSphere

Version 3.5

SC09-4431-02

Note

Before using this information and the product it supports, be sure to read the general information under "Notices" on page 237.

Third Edition (June 2000)

This edition replaces SC09-4431-01.

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

Contents

Figures

Tables

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 15 explains the main concepts associated with enterprise beans.
- "Chapter 3. Tools for developing and deploying enterprise beans in the EJB server (AE) environment" on page 29 explains how to set up and use the tools contained in the EJB server (AE) environment. It also discusses the major steps in developing and deploying enterprise beans in that environment. The EJB server (AE) is the EJB server implementation available with the WebSphere Application Server Advanced Edition.
- "Chapter 4. Tools for developing and deploying enterprise beans in the EJB server (CB) environment" on page 49 explains how to set up and use the tools contained in the EJB server (CB) environment. It also discusses the major steps in developing and deploying enterprise beans in that environment. The EJB server (CB) is the EJB server implementation available with Component Broker as part of the WebSphere Application Server Enterprise Edition.
- "Chapter 5. Developing enterprise beans" on page 89 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 6. Enabling transactions and security in enterprise beans" on page 121 explains how to enable transactions in enterprise beans by using the appropriate deployment descriptor attributes.
- "Chapter 7. Developing EJB clients" on page 129 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 8. Developing servlets that use enterprise beans" on page 145 discusses the basic code required in a servlet that accesses an enterprise bean.
- "Chapter 9. More-advanced programming concepts for enterprise beans" on page 157 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. Example code provided with WebSphere Application Server" on page 225 describes the major example used throughout this book and the additional examples that are delivered with the various editions of WebSphere Application Server.
- "Appendix B. Using XML in enterprise beans" on page 229 describes the extensible markup language (XML) that can be used to create deployment descriptors for use with enterprise beans in WebSphere.
- "Appendix C. Extensions to the EJB Specification" on page 235 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:

- *Building Business Solutions with WebSphere*
- Component Broker *Problem Determination Guide*
- Component Broker *System Administration Guide*
- *Getting Started with TXSeries*
- *Getting Started with Advanced Edition*
- *Getting Started with Component Broker*
- *Component Broker Release Notes*

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.
`%`	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.

Table 1. Conventions used in this book (continued)

Convention	Meaning
|	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.)
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 waseedoc@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

In the past few years, 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 Advanced Edition and WebSphere Application Server Enterprise Edition 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.

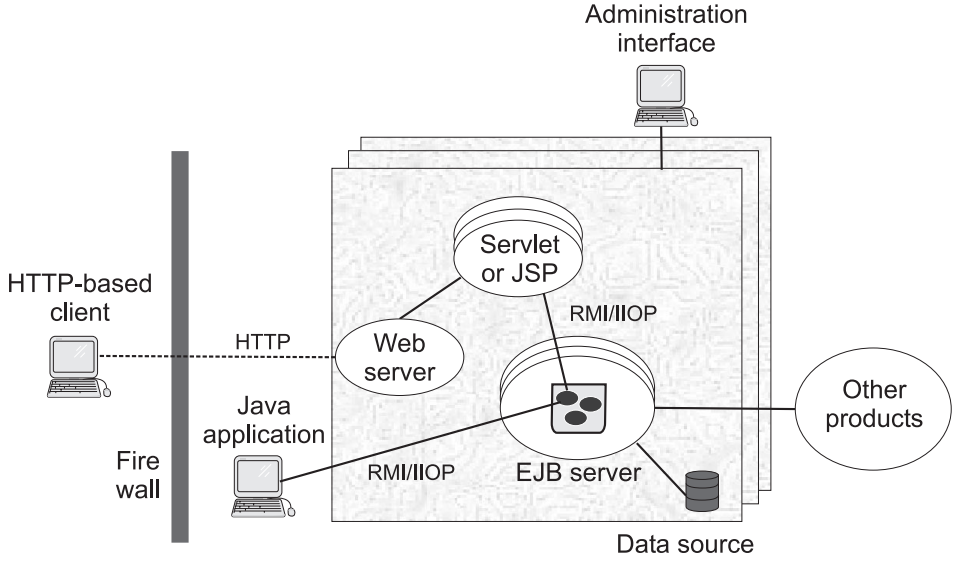

Figure 1. The components of the EJB environment

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 10.
- *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 11.
- The *administration interface*—The administrative interface allows you to manage the EJB server environment. For more information, see "The administration interface" on page 13.

The EJB server

The EJB server is the application server tier of WebSphere Application Server's three-tier architecture, connecting the client tier (Java servlets, applets, applications, and JSP) with the resource management tier (the data source). The WebSphere Application Server contains two types of EJB servers. If you have the Advanced Application Server, you get only one of these EJB servers; if you have the Enterprise Application Server, you get both. When referring generically to EJB servers, this documentation uses the phrase *EJB server*; when the documentation needs to refer specifically to one or the other, it uses the following terms:

- *EJB server (AE)*—The EJB server that comes with the Advanced Application Server. (Because Advanced Application Server is available as a part of Enterprise Application Server, this EJB server is also available with Enterprise Application Server.)

- *EJB server (CB)*—The EJB server that comes only with the Enterprise Application Server and is part of Component Broker (CB).

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.

The EJB server (CB) includes two standard types of containers: entity containers and session containers. As their names imply, these containers are specifically optimized to handle entity beans and session beans, respectively. The EJB server (AE) has one standard container that supports both entity and session beans.

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 a *deployment descriptor* that defines properties associated with the bean; the deployment descriptor and the other enterprise bean components are packaged in a file known as an *EJB JAR file*. For more information on deployment, see "Deploying an enterprise bean" on page 21.
- 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" on page 4.
- A workload management service that ensures that resources are used efficiently. For more information, see "The workload management service" on page 6.
- 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 6.
- 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 7.
- A transaction service that implements the transactional attributes in a bean's deployment descriptor. For more information, see "The transaction service" on page 7.

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 in the EJB server environment

There are some similarities between the security service in the two EJB server environments. In both EJB server environments, 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 19.

The major differences between the two security services are discussed in the following sections.

Security in the EJB server (AE) environment: In the EJB server (AE) environment, 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 in the EJB server (AE) environment, see the online help available with the WebSphere Administrative Console.

Security in the EJB server (CB) environment: In the EJB server (CB) environment, you must secure all the Component Broker name servers and applications servers in the network. Securing the name server on each server host prevents unauthorized access to the system objects (including name contexts used in the Component Broker namespace) in that server. Securing an application server prevents unauthorized access to the business objects for applications in that server.

To secure your name servers and application servers, you must do the following:

- Install and configure the Distributed Computing Environment (DCE) to provide authentication services to the servers. This allows secure access between servers.

- Configure key rings for clients and servers to provide authentication services to Java-based SSL clients.
- Configure authorization for access to business objects in the application service.
- Create a delegation policy to allow the application server to pass the requesting client principal to other servers.
- Configure credential mapping to provide access to any third tier system.
- Configure the qualities of protection to be used to protect messages that flow between clients and the application server.

The Component Broker *System Administration Guide* provides more detail about each of these tasks.

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 the EJB server (AE) environment and from within the Systems Management End User Interface for the EJB server (CB) environment. For more information on workload management, consult 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 15.

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.

In the EJB server (AE) environment, 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 10. The EJB server ensures that persistent data is always handled within the appropriate transactional context.

In the EJB server (CB) environment, the persistence service uses the following components to accomplish its task:

- The X/Open XA interface, which gives entity beans a standard interface to relational databases.
- The Object Management Group's (OMG) Object Transaction Service (OTS), which is also discussed in "Using transactions in the EJB server environment" on page 10.

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 on page 9, where each oval indicates work being done on a different machine, and each arrow indicates a remote method invocation (RMI).

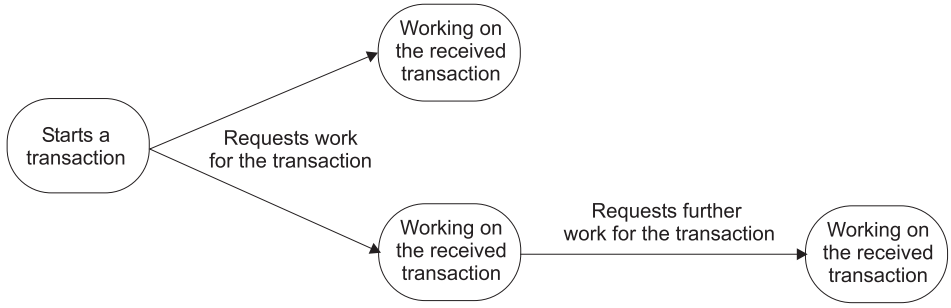

Figure 2. Example of a distributed transaction

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 121.

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 181.

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:

- The EJB server (AE) supports DB2®, Oracle, Sybase, and InstantDB.
- The EJB server (CB) supports DB2, Oracle, CICS®, IMS™, and MQSeries®.

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 157.

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. For the EJB server (CB), a Java applet can be used to directly interact with enterprise beans. For the EJB server (AE), a Java applet can be used only in combination with a servlet.

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 12.
- *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 (AE) 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.

– All EJB clients that access an EJB server (CB) and EJB clients that access an EJB server (AE) by using RMI (for example, Java applications) encounter the second security layer only. Like HTTP-based EJB clients, these EJB clients must authenticate with the security service.

  For more information, see "The security service" on page 4.

- *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 7. For information on managing transactions in a Java EJB client, see "Managing transactions in an EJB client" on page 139.

In the EJB server (CB) environment, an enterprise bean can also be accessed by EJB clients that use Microsoft ActiveX™, CORBA-based Java, and to a limited degree, CORBA-based C++. "More information on EJB clients specific to the EJB server (CB)" on page 141 provides additional information.

Note: In the EJB server (AE) environment, ActiveX and CORBA-based access to enterprise beans is not supported.

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 (CB) and EJB server (AE) each have their own administration tools:

- The EJB server (AE) uses the WebSphere Administrative Console. For more information on this interface, consult the online help available within the WebSphere Administrative Console.
- The EJB server (CB) uses the System Management End User Interface (SM EUI). For more information on this interface, see the Component Broker *System Administration Guide*.

You can also administer the EJB server (AE) 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 class*—This class encapsulates one or more variables that uniquely identify a specific entity bean instance. It also contains methods to create primary key objects and manipulate those objects.

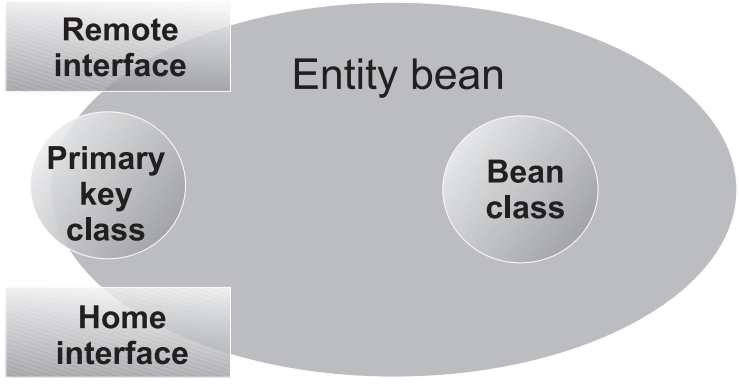

Figure 3. The components of an entity bean

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. You must use entity beans with BMP if you want to use a data source that is not supported by the EJB server. 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.

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 18:

- *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.

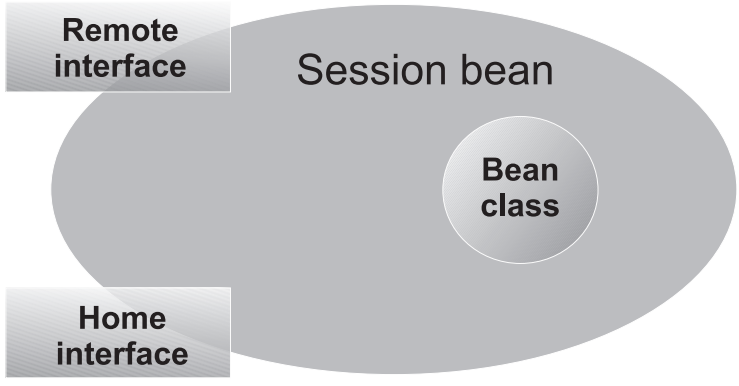

Figure 4. The components of a session bean

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, a session bean represents 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 104 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.

Packaging enterprise beans

The last step in the development of an enterprise bean is the creation of the deployment descriptor and the EJB JAR file. After the EJB JAR file is created, the enterprise bean can be deployed into the container of an EJB server.

The deployment descriptor

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

- *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. The values for this attribute are described in "Creating and getting a reference to a bean's EJB object" on page 131.
- *Transaction* attribute—Defines the transactional manner in which the container invokes a method. The values for this attribute are described in "Chapter 6. Enabling transactions and security in enterprise beans" on page 121.
- *Transaction isolation level* attribute—Defines the degree to which transactions are isolated from each other by the container. The values for this attribute are described in "Chapter 6. Enabling transactions and security in enterprise beans" on page 121.
- *Access control* attribute—Defines an access control entry that identifies users or roles that are permitted to access the methods in the enterprise bean. This value is not used by the WebSphere EJB servers.
- *RunAsMode* and *RunAsIdentity* attributes—The *RunAsMode* attribute defines the identity used to invoke the method. If a specific identity is required, the *RunAsIdentity* attribute is used to specify that identity. The *RunAsMode* attribute is used by the WebSphere EJB servers; the *RunAsIdentity* attribute is not. The values for the *RunAsMode* attribute are described in "Chapter 6. Enabling transactions and security in enterprise beans" on page 121.

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

- *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 102 or “Writing or selecting the primary key class (entity with BMP)” on page 172.
- *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 91.
- *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 117.

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

- *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 18.
- *Timeout* attribute—Defines the idle timeout value in seconds associated with this session bean.

Deployment descriptors can be created by using the tools within an integrated development environment (IDE) such as IBM VisualAge® for Java Enterprise Edition or by using the stand-alone tools contained in Websphere Application Server. For more information, see “Chapter 3. Tools for developing and deploying enterprise beans in the EJB server (AE) environment” on page 29 or “Chapter 4. Tools for developing and deploying enterprise beans in the EJB server (CB) environment” on page 49.

The EJB JAR file

The *EJB JAR file* is used to package enterprise beans; this file uses the standard Java archive file format. The EJB JAR file can be used to contain individual enterprise beans, multiple enterprise beans, and entire enterprise bean-based applications. For more information, see “Creating an EJB JAR file” on page 119.

An EJB JAR file can be created by using the tools within an integrated development environment (IDE) like IBM’s VisualAge for Java or by using the stand-alone tools contained in Websphere. For more information, see “Chapter 3. Tools for developing and deploying enterprise beans in the EJB server (AE) environment” on page 29.

Deploying an enterprise bean

When you deploy an enterprise bean, 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 shows a simplified version of a deployed entity bean.

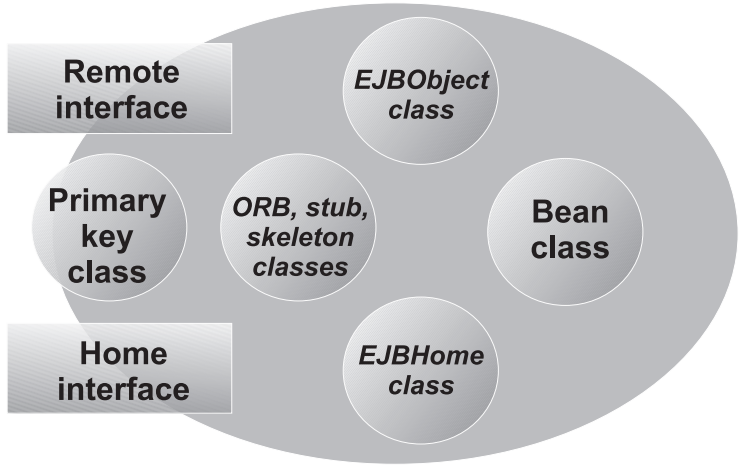

Figure 5. The major components of a deployed entity bean

You can deploy an enterprise bean with a variety of different tools. For more information, see "Chapter 3. Tools for developing and deploying enterprise beans in the EJB server (AE) environment" on page 29 or "Chapter 4. Tools for developing and deploying enterprise beans in the EJB server (CB) environment" on page 49.

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 22 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.

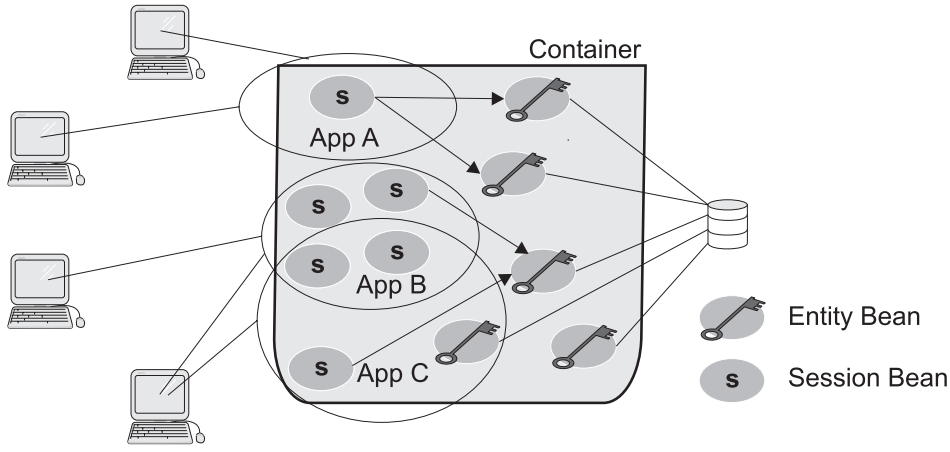

Figure 6. Conceptual view of EJB applications

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 5. Developing enterprise beans" on page 89.

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 139.

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 139.

Chapter 3. Tools for developing and deploying enterprise beans in the EJB server (AE) environment

There are two basic approaches to developing and deploying enterprise beans in the EJB server (AE) environment:

- 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. In the EJB server (AE) environment, use of VisualAge for Java is strongly recommended. 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 with EJB server (AE) tools" on page 30.

Before beginning development of enterprise beans in the EJB server (AE) environment, review the list of development restrictions contained in "Restrictions in the EJB server (AE) environment" on page 47.

Note: Deployment and use of enterprise beans for the EJB server (AE) environment must take place on the Microsoft Windows NT® operating system, the IBM AIX™ operating systems, or the Sun Microsystems Solaris® operating system.

For information on developing enterprise beans in the EJB server (CB) environment, see "Chapter 4. Tools for developing and deploying enterprise beans in the EJB server (CB) environment" on page 49.

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.

To access the enterprise bean development window, click the **EJB** tab from the Workbench. All enterprise beans must be part of an EJB group that is associated with a VisualAge for Java project. You must create a project and an EJB group before creating an enterprise bean.

After you have created an EJB group, you can add beans to the EJB group. This action brings up the SmartGuide for creating enterprise beans. The SmartGuide prompts you for the information needed to generate all of the components of an enterprise bean and much of the required code.

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 class for the EJB server (CB) environment or a finder helper interface for the EJB server (AE) environment, if required.
4. Define the deployment descriptor for the bean.
5. Package the bean components in an EJB JAR file.
6. Generate the deployment code for the bean.

VisualAge for Java contains a complete WebSphere Application Server runtime 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 with EJB server (AE) tools

If you have decided to develop enterprise beans *without* an IDE, you need at minimum the following low-level 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 Server **jetace** 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 (AE) environment. For more information, see "Installing and configuring the software for the EJB server (AE)" on page 31.

2. Set the CLASSPATH environment variable required by different components of the EJB server (AE) environment. For more information, see “Setting the CLASSPATH environment variable in the EJB server (AE) environment” on page 32.
3. Write and compile the components of the enterprise bean. For more information, see “Creating the components of an enterprise bean” on page 32.
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 (AE)” on page 33.
5. Create a deployment descriptor file and an EJB JAR file for the enterprise bean by using the **jetace** tool. For more information, see “Creating a deployment descriptor and an EJB JAR file” on page 33.
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 47.
7. Deploy the enterprise bean by using the WebSphere Administrative Console. For more information, see the online help available with the WebSphere Administrative Console.
8. Install the enterprise beans into an EJB server (AE) and start the server by using the WebSphere Administrative Console. For more information, see the online help available with the WebSphere Administrative Console.

Installing and configuring the software for the EJB server (AE)

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 (AE):

- 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
  - 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 Advanced application server *Getting Started* document.

Setting the CLASSPATH environment variable in the EJB server (AE) 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 the listed tasks:

- Developing an enterprise bean that does *not* use another enterprise bean:
  - ejs.jar
  - ujc.jar
  - iioptools.jar
- Developing an enterprise bean that *does* use another enterprise bean:
  - ejs.jar
  - ujc.jar
  - iioptools.jar
  - *otherDeployedBean*.jar (the deployed JAR file containing the enterprise bean being used by this enterprise bean)
- Developing an EJB client:
  - ejs.jar
  - ujc.jar
  - iioptools.jar
  - servlet.jar (required by EJB clients that are servlets)
  - *otherDeployedBean*.jar (the deployed JAR file containing the enterprise bean being used by this EJB client)
- Running an EJB client:
  - ejs.jar
  - ujc.jar
  - servlet.jar (required by EJB clients that are servlets)
  - *otherDeployedBean*.jar (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 5. Developing enterprise beans” on page 89.

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, since 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 (AE)

For the EJB server (AE) 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*QueryString, 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.

If you define the findLargeAccounts method shown in Figure 24 on page 100, you must also create the AccountBeanFinderHelper interface shown in Figure 7.

```
...
public interface AccountBeanFinderHelper{
     String findLargeAccountsQueryString =
          "select * from ejb.accountbeantbl where balance > ?";
}
```

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

Creating a deployment descriptor and an EJB JAR file

The WebSphere Application Server **jetace** tool can be used to create an EJB JAR file for one or more enterprise beans and generate a deployment descriptor file for each enterprise bean. The resulting EJB JAR file contains each enterprise bean's class files and deployment descriptor and an EJB-compliant manifest file. The **jetace** tool is available in both the EJB server (AE) and the EJB server (CB) environments.

Note: Before using the **jetace** tool in the EJB server (AE) environment, ensure that the JAVA_HOME environment variable identifies the path to the

SDK installation directory. For example on Windows NT, if your SDK installation directory is C:\SDK, set this environment variable as follows:

```
C:\> set JAVA_HOME=C:\SDK
```

Before you create an EJB JAR file for one or more enterprise beans, you must do *one* of the following:

- Place all of the components of each enterprise bean into a single directory.
- Create a standard JAR file that contains the class and interface files of each enterprise bean by using the Java Archiving tool (**jar**). The following command, when run from the root directory of the Account bean's full package name, can be used to create the file AccountIn.jar with a default manifest file:

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

- Create a standard ZIP file that contains the class and interface files of each enterprise bean by using a tool like WinZip®.

Running the jetace tool

To run the **jetace** tool, type **jetace** on the command line. The window shown in Figure 8 is displayed.

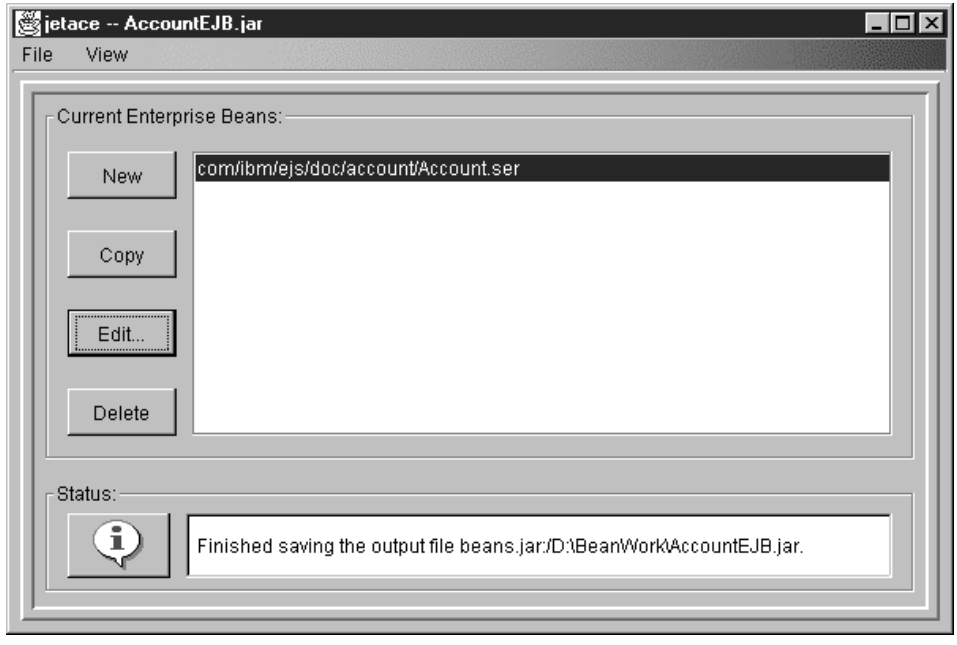

Figure 8. The initial window of jetace tool

To generate an EJB JAR file with the **jetace** tool, do the following:

1. Click the **File->Load** item, and select the JAR or ZIP file or the directory containing one or more enterprise beans. Use the **Browse** button to obtain the file or directory.

   Note: To specify the current directory as the input source, type an = (equals character) in the **File Name** field of the browser window and click **Open**.

   If you are creating a new EJB JAR file, click **New** and a default name for the deployment descriptor (for example, UNAMED_BEAN_1.ser) appears in the **Current Enterprise Beans** list box. (You can edit this name on any of the remaining tabbed pages of the **jetace** GUI by editing the **Deployed Name** field at the top of each tabbed page. This field is described in "Specifying the enterprise bean components and JNDI home name" on page 36.)

   If you are editing an existing EJB JAR file, the name of the deployment descriptor for each enterprise bean in the EJB JAR file is displayed in the **Current Enterprise Beans** list box, as shown in Figure 8 on page 34.

   - If you do not want to include a listed enterprise bean in the resulting EJB JAR file, highlight that enterprise bean's deployment descriptor and click **Delete**. This action removes the deployment descriptor from the list box.
   - If you want to create a duplicate of an enterprise bean, highlight its deployment descriptor and click **Copy**. This action adds a new default deployment descriptor to the list box. Copying can be useful if you want to create a deployment descriptor for one enterprise bean that is similar to the deployment descriptor of the copied bean. You must then edit the new deployment descriptor.

2. To create a new deployment descriptor or edit an existing one, highlight the deployment descriptor and press the **Edit** button. This action causes the **Basic** page to display. On this page, set or confirm the names of the deployment descriptor file, the enterprise bean class, the home interface, and the remote interface and specify the JNDI name of the enterprise bean. For information, see "Specifying the enterprise bean components and JNDI home name" on page 36.
3. Set the entity bean or session bean attributes for the enterprise bean's deployment descriptor on the **Entity** or **Session** page, respectively. For information on setting deployment descriptor attributes for entity beans, see "Setting the entity bean-specific attributes" on page 38. For information on setting deployment descriptor attributes for session beans, see "Setting the session bean-specific attributes" on page 40.

4. Set the transaction attributes for the enterprise bean's deployment descriptor on the **Transactions** page. For information, see "Setting transaction attributes" on page 41.
5. Set the security attributes for the enterprise bean's deployment descriptor on the **Security** page. For information, see "Setting security attributes" on page 43.
6. Set any environment variables to be associated with the enterprise bean on the **Environment** page. For information, see "Setting environment variables for an enterprise bean" on page 44.
7. Set any class dependencies to be associated with the enterprise bean on the **Dependencies** page. For information, see "Setting class dependencies for an enterprise bean" on page 46.
8. After you have set the appropriate deployment descriptor attributes for each enterprise bean, click **File->Save As** to create an EJB JAR file. (If desired, a ZIP file can be created instead of a JAR file.)

The **jetace** tool can also be used to read and generate an XML version of an enterprise bean's deployment descriptor. To read an XML file, click the **File->Read XML** item. To generate an XML file from an existing enterprise bean (after saving the output EJB JAR file) click the **File->Write XML** item.

The **jetace** tool can also be run from the command line to create an EJB JAR file. The syntax of this command follows, where *xmlFile* is the name of an XML file containing the enterprise bean's deployment descriptor:

```
% jetace -f xmlFile
```

Note: In the EJB server (AE) environment, use of the XML feature provided by the **jetace** tool is not recommended.

For more information on the syntax of the XML file required for this command, see "Appendix B. Using XML in enterprise beans" on page 229.

Specifying the enterprise bean components and JNDI home name

The **Basic** page is used to set the full pathname of the deployment descriptor file and the Java package name of the enterprise bean class, home interface, and remote interface and to set the enterprise bean's JNDI home name. To access this page, which is shown in Figure 9 on page 37, click the **Basic** tab.

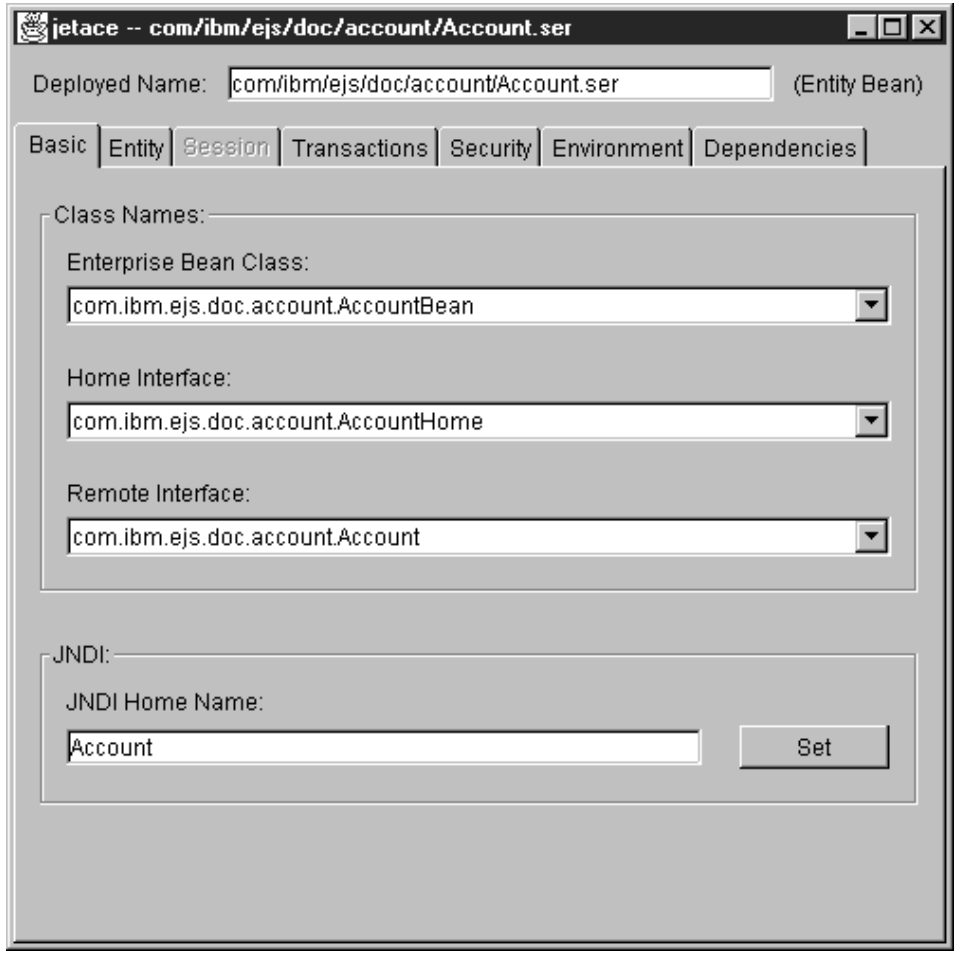

Figure 9. The Basic page of the jetace tool

In the Basic page, you must select or confirm values for the following fields:

- **Deployed Name**—The pathname of the deployment descriptor file to be created. It is recommended that this directory name match the full package name of the enterprise bean class. For the Account bean, the full name is `com/ibm/ejs/doc/account/Account.ser`.
- **Enterprise Bean Class**—Specify the full package name of the bean class. For the Account bean, the full name is `com.ibm.ejs.doc.account.AccountBean`.
- **Home Interface**—Specify the full package name of the bean's home interface. For the Account bean, the full name is `com.ibm.ejs.doc.account.AccountHome`.

- **Remote Interface**—Specify the full package name of the bean's remote interface. For the Account bean, the full name is `com.ibm.ejs.doc.account.Account`.
- **JNDI Home Name**—Specify the JNDI home name of the bean's home interface. This the name under which the enteprise bean's home interface is registered and therefore is the name that must be specified when an EJB client does a lookup of the home interface. For the Account bean, the JNDI home name is `Account`.

Setting the entity bean-specific attributes

To set the deployment descriptor attributes associated specifically with an entity bean, click the **Entity** tab in the **jetace** tool to display the **Entity** page shown in Figure 10 on page 39. This tab is disabled if the highlighted enterprise bean in the initial **jetace** window is a session bean.

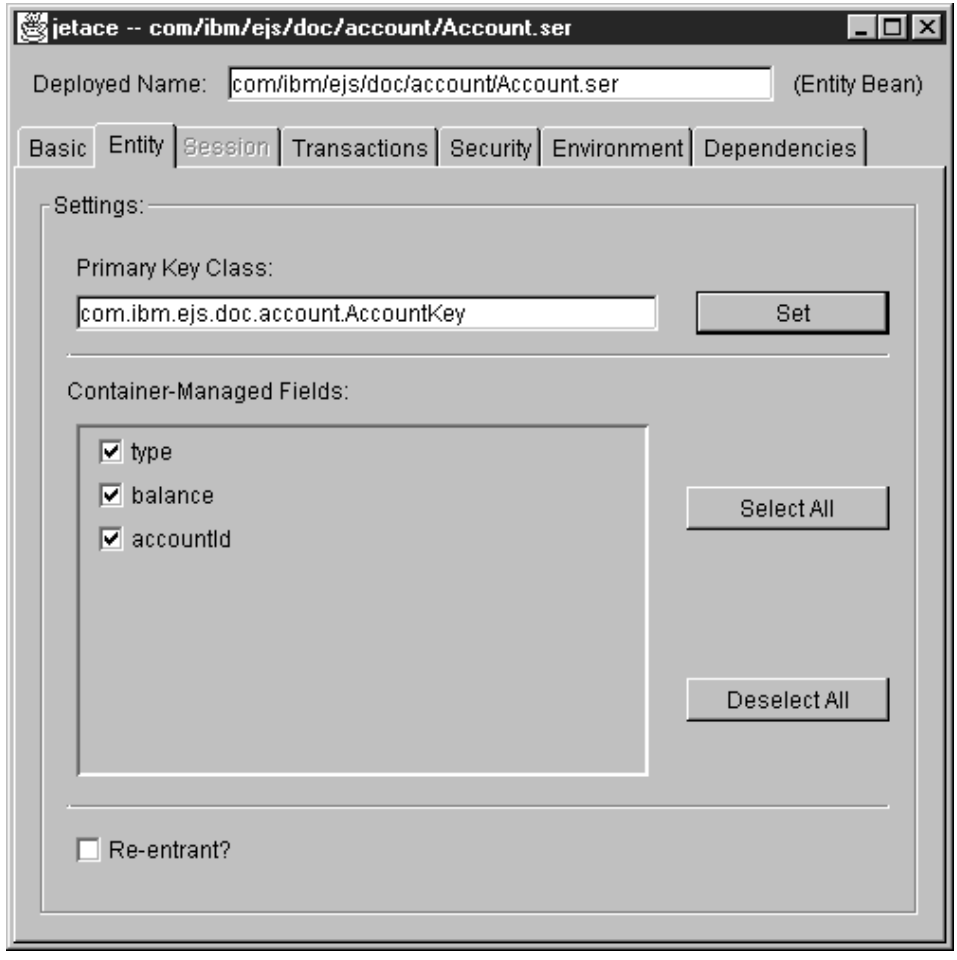

Figure 10. The Entity page of the jetace tool

In the **Entity** page, you must select or confirm values for the following fields:

- **Primary Key Class**—Specify the full package name of the bean's primary key class. For the example Account bean, the full name is com.ibm.ejs.doc.account.AccountKey.
- **Container-Managed Fields**—Check the check boxes of the variables in the bean class for which the container needs to handle persistence management. This is required for entity beans with CMP only, and must *not* be done for entity beans with BMP. For the Account bean, the type, balance, and accountId variables are container managed, so each box is checked.
- **Re-entrant?**—Check this check box if the bean is reentrant. By default, an entity bean is not reentrant. If an instance of a non-reentrant entity bean is executing a client request in a transaction context and it receives another request using the same transaction context, the EJB container throws the

java.rmi.RemoteException exception to the second request. Since a container cannot distinguish between a legal loopback call from another bean and an illegal concurrent call from another client or client thread, a client must take care to prevent concurrent calls to a reentrant bean. The example Account bean is *not* reentrant.

Setting the session bean-specific attributes

To set the deployment descriptor attributes associated specifically with a session bean, click the **Session** tab in the **jetace** tool to display the **Session** page shown in Figure 11. This tab is disabled if the highlighted enterprise bean in the initial **jetace** window is an entity bean.

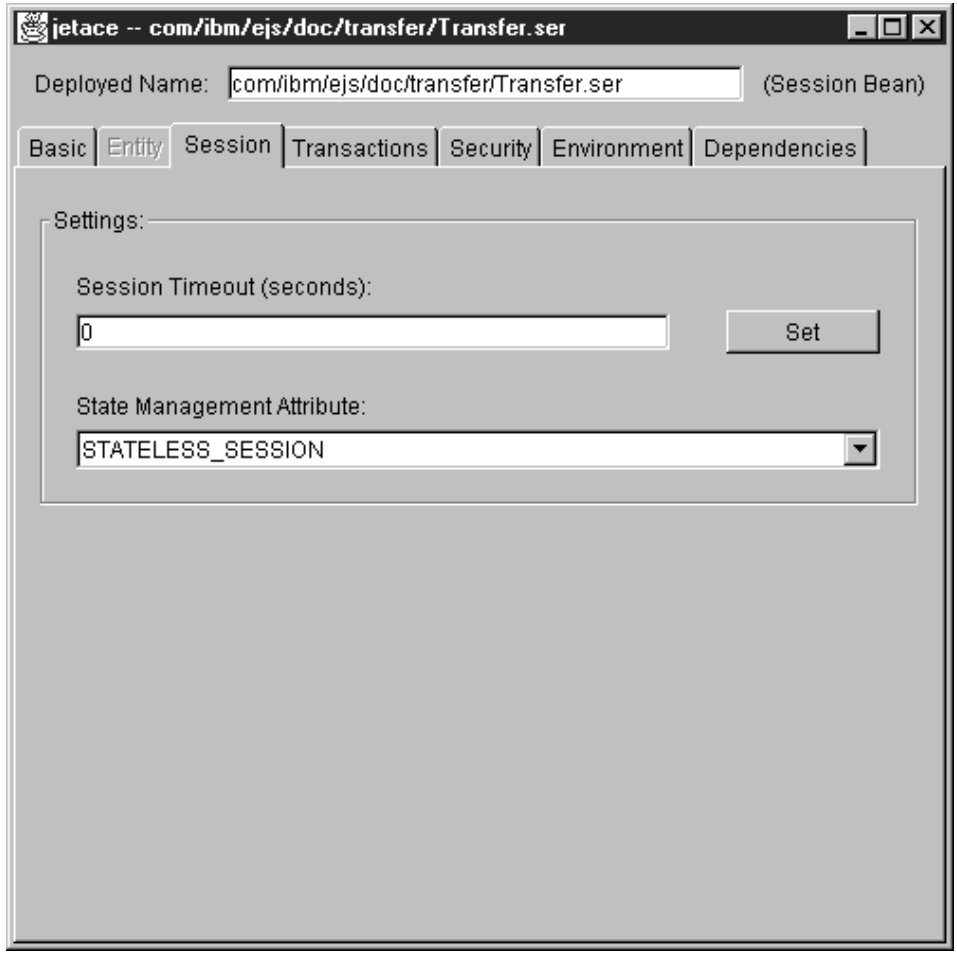

Figure 11. The Session page of the jetace tool

On the **Session** page, you must select or confirm values for the following fields:

- **Session Timeout (seconds)**—Specify the idle timeout value for this bean in seconds; a `0` (zero) indicates that idle bean instances timeout after the maximum allowable timeout period has elapsed (by default in the EJB server (AE), this timeout is 600 seconds or 10 minutes). For the Transfer bean, the value is left at `0` to indicate that the default timeout is used.

  Note: In the EJB server (CB) environment, this attribute is not used.
- **State Management Attribute**—Specify whether the bean is stateless or stateful. The example Transfer bean is `STATELESS_SESSION`. For more information, see "Stateless versus stateful session beans" on page 18.

Setting transaction attributes

The **Transactions** page is used to set the transaction and transaction isolation level attributes for all of the methods in an enterprise bean and for individual methods in an enterprise bean. If an attribute is set for an individual method, that attribute overrides the default attribute value set for the enterprise bean as a whole.

Note: In the EJB server (CB), the transactional attribute can be set only for the bean as a whole; the transaction attribute cannot be set on individual methods in a bean.

To access the **Transaction** page, click the **Transactions** tab in the **jetace** tool. Figure 12 on page 42 shows an example of this page.

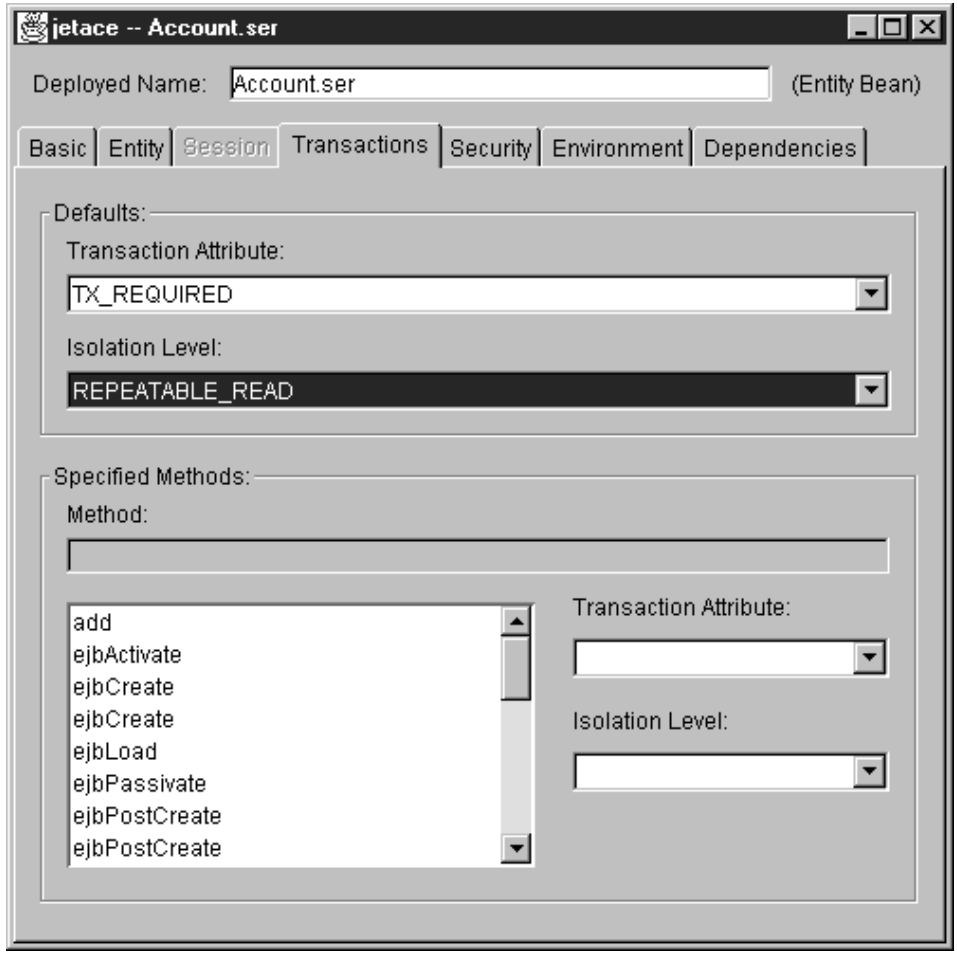

Figure 12. The Transactions page of the jetace tool

On the **Transactions** page, you must select or confirm values for the following fields in the **Defaults** group box:

- **Transaction Attribute**—Set a value for the transaction attribute. The values for this attribute are described in "Chapter 6. Enabling transactions and security in enterprise beans" on page 121. For the Account bean, the value `TX_MANDATORY` is used because the methods in this bean must be associated with an existing transaction when invoked; as a result, the Transfer bean must use the value that begins a new transaction or passes on an existing one.
- **Isolation Level**—Set a value for the transaction isolation level attribute. The values for this attribute are described in "Chapter 6. Enabling transactions and security in enterprise beans" on page 121. For the Account bean, the value `REPEATABLE_READ` is used.

If necessary, you can also set these attributes on individual methods by highlighting the appropriate method and setting one or both of the attributes in the **Specified Methods** group box.

Setting security attributes

The **Security** page is used to set the security attributes for all of the methods in an enterprise bean and for individual methods in an enterprise bean. If an attribute is set for an individual method, that attribute overrides the default attribute value set for the enterprise bean as a whole.

To access the **Security** page, click the **Security** tab in the **jetace** tool. Figure 13 shows an example of this page.

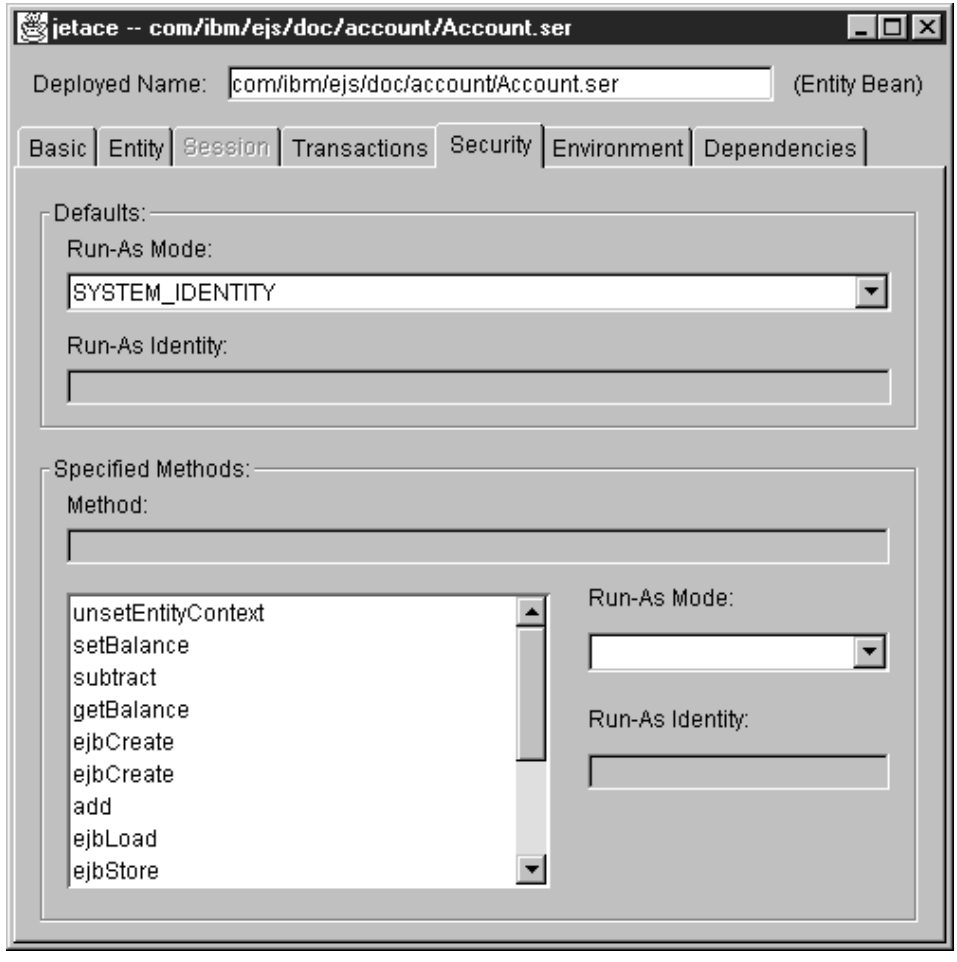

Figure 13. The Security page of the jetace tool

On the **Security** page, you must select or confirm values for the **Run-As Mode** field in the **Defaults** group box. This field must be set to one of the values described in "Setting the security attribute in the deployment descriptor" on page 126. The *run-as identity* attribute is not used by WebSphere EJB servers, so you cannot set the value for the corresponding field in the **jetace** tool.

If necessary, you can also set the *run-as mode* attribute on individual methods by highlighting the appropriate method and setting the attribute in the **Specified Methods** group box.

Setting environment variables for an enterprise bean

The **Environment** page is used to associate environment variables (and their corresponding values) with an enterprise bean. To access the **Environment** page, click the **Environment** tab in the **jetace** tool. Figure 14 on page 45 shows an example of this page.

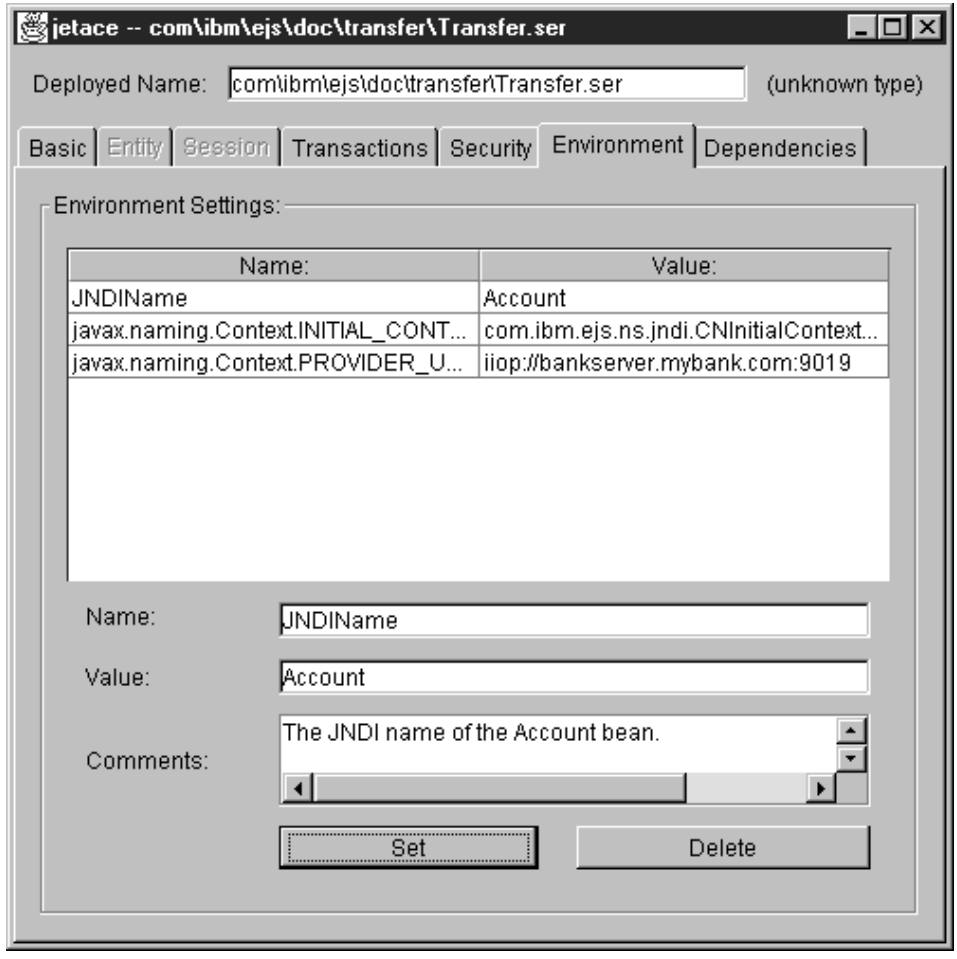

Figure 14. The Environment page of the jetace tool

To set an environment variable to its value, specify the environment variable name in the **Name** field and specify the environment variables value in the **Value** field. If desired, use the **Comment** field to further identify the environment variable. Press the **Set** button to set the value. To delete an environment variable, highlight the variable in the **Environment Settings** window and press the **Delete** button.

For the example Transfer bean, the following environment variables are required:

- JNDIName—The JNDI name of the Account bean, which is accessed by the Transfer bean. For more information, see Figure 9 on page 37.

- javax.naming.Context.INITIAL_CONTEXT_FACTORY—The name of the initial context factory used by the Transfer bean to look up the JNDI name of the Account bean
- javax.naming.Context.PROVIDER_URL—The location of the naming service used by the Transfer bean to look up the JNDI name of the Account bean.

For more information on how these environment variables are used by the Transfer bean, see “Implementing the ejbCreate methods” on page 108.

Setting class dependencies for an enterprise bean

The **Dependencies** page is used to specify classes on which the enterprise bean depends. To access the **Dependencies** page, click the **Dependencies** tab in the **jetace** tool. Figure 15 shows an example of this page.

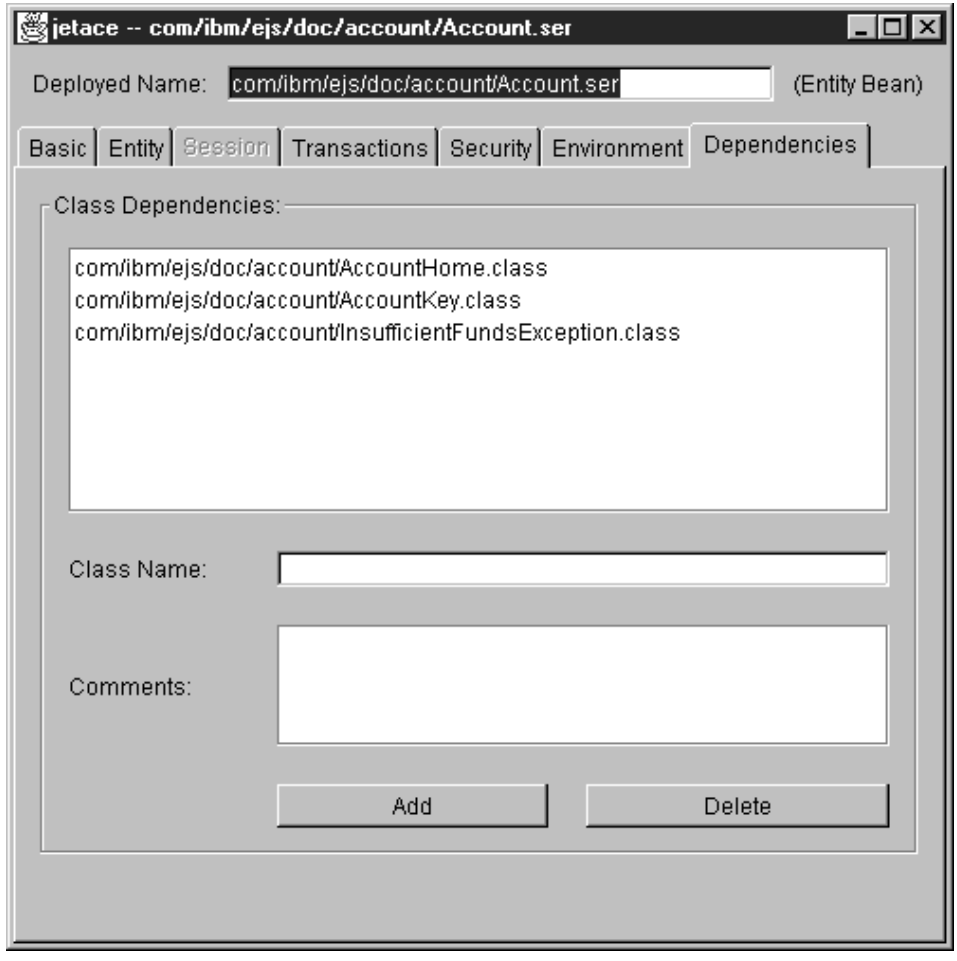

Figure 15. The Dependencies page of the jetace tool

Generally, the **jetace** tool discovers class dependencies automatically and sets them here. If there are other class dependencies required by an enterprise bean, you must set them here by entering the fully-qualified Java class name in the **Classname** field. If desired, use the **Comment** field to further identify the dependency. Press the **Add** button to set the value. To remove a dependency, highlight it in the **Class Dependencies** window and press the **Delete** button.

For the example Account bean, the **jetace** tool set the dependencies shown in Figure 15 on page 46.

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. If you are not using VisualAge for Java, it is strongly recommended that you use the WebSphere Administrative Console to automatically create database tables for CMP entity beans. The console names the database schema and table ejb.*beanName*beantbl, where *beanName* is the name of the enterprise bean (for example, ejb.accountbeantbl).

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.

Restrictions in the EJB server (AE) environment

The following restrictions apply when developing enterprise beans for the EJB server (AE) environment:

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

- The *run-as identity* and *access control* deployment descriptor attributes are not used.

Chapter 4. Tools for developing and deploying enterprise beans in the EJB server (CB) environment

The following are the basic approaches to developing and deploying enterprise beans in the EJB server (CB) environment:

- You can use the tools available in the Java Software Development Kit (SDK) and WebSphere Application Server, Enterprise Edition. For more information, see "Developing and deploying enterprise beans with EJB server (CB) tools".
- You can use one of the available integrated development environments (IDEs) such as IBM VisualAge for Java. IDE tools automatically generate significant parts of the enterprise bean code and contain integrated tools for packaging and testing enterprise beans. For more information, see "Using VisualAge for Java" on page 29.
- You can create an enterprise bean from an existing CICS or Information Management System (IMS) application by using the **PAOToEJB** tool. The application must be mapped into a procedural adapter object (PAO) before this tool is used. For more information, see "Creating an enterprise bean from an existing CICS or IMS application" on page 81.
- You can create an enterprise bean that communicates with IBM MQSeries by using the **mqaaejb** tool. For more information, see "Creating an enterprise bean that communicates with MQSeries" on page 82.

Before beginning development of enterprise beans in the EJB server (CB) environment, review the list of development restrictions contained in "Restrictions in the EJB server (CB) environment" on page 84.

Note: Deployment and use of enterprise beans for the EJB server (CB) environment must take place on the Microsoft Windows NT operating system, the IBM AIX operating systems, or the Sun Solaris operating system.

For information on developing and deploying enterprise beans in the EJB server (AE) environment, see "Chapter 3. Tools for developing and deploying enterprise beans in the EJB server (AE) environment" on page 29.

Developing and deploying enterprise beans with EJB server (CB) tools

You need the following tools to develop and deploy enterprise beans for the EJB server (CB):

- 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 following tools available in the WebSphere Application Server, Enterprise Edition:
  - **jetace**, which enables you to create or update an EJB JAR file for one or more enterprise beans; this includes the creation of the enterprise bean's deployment descriptor, which instructs the EJB server on how to properly manage the enterprise bean.
  - Object Builder, which is the recommended tool for deploying enterprise beans. Use of this tool is not documented in this book. For more information on using Object Builder to deploy enterprise beans, see the Component Broker *Application Development Tools Guide*.
  - **cbejb**, which works with Object Builder to create and compile the necessary files needed by the EJB server (CB) to manage an enterprise bean. The **cbejb** tool looks inside the EJB JAR file to examine the EJB home and EJB object classes and the deployment descriptors. The **cbejb** tool generates a model that Object Builder uses to create the necessary deployment library files. The output of this process is a set of server-side and client-side JAR and library files.
  - **ejbbind**, which binds an enterprise bean's Java Naming and Directory Interface (JNDI) home name (found in its deployment descriptor) to a factory in an EJB server (CB). This tool is deprecated for servers running on the AIX, Windows NT, and Solaris platforms.
  - **appbind**, which allows enterprise bean deployers to create an application-specific naming context and associate it with a selected factory finder so that the EJB home lookup operations are resolved with that factory finder. This tool is available only on the AIX, Windows NT, and Solaris platforms and only be applied to servers installed on any of those platforms.

This section describes the steps you must follow to manually develop and deploy enterprise beans by using the EJB server (CB) tools. The following tasks are involved:

1. Ensure that you have the prerequisite software to develop and deploy enterprise beans in the EJB server (CB). For more information, see "Prerequisite software for the EJB server (CB)" on page 51.
2. Set the CLASSPATH environment variable required by different components of the EJB server (CB) environment. For more information, see "Setting the CLASSPATH environment variable in the EJB server (CB) environment" on page 51.
3. Write and compile the components of the enterprise bean. For more information, see "Creating the components of an enterprise bean" on page 52.

4. Create a finder helper class 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 (CB)” on page 53.
5. Use the **jetace** tool to create an EJB JAR file to contain the enterprise bean. For more information, see “Creating a deployment descriptor and an EJB JAR file” on page 33.
6. Deploy the enterprise bean by using the **cbejb** command. For more information, see “Deploying an enterprise bean” on page 56.
7. Build a data object (DO) implementation for use by the enterprise bean by using Object Builder (This step is part of the deployment process). For more information, see “Building a data object during CMP entity bean deployment” on page 62.
8. Install the deployed enterprise bean and configure its EJB server (CB). For more information, see “Installing an enterprise bean and configuring its EJB server (CB)” on page 72.
9. Bind the JNDI name of the enterprise bean into the JNDI namespace by using the **ejbbind** tool. (This step is not necessary on the AIX, Windows NT, or Solaris platforms.) For more information, see “Binding the JNDI name of an enterprise bean into the JNDI namespace” on page 73.
10. Start the EJB server (CB). For more information see the Component Broker *System Administration Guide*.

Prerequisite software for the EJB server (CB)

Note: Any items marked *PAO only* are needed only if you intend to use the **PAOToEJB** tool and need the CICS- or IMS-related support.

You must configure the tools provided with the EJB server (CB) environment; however, before you can configure the tools, you must ensure that you have installed and configured the following prerequisite software products contained in the Enterprise Application Server:

- CB Server
- CB Tools (including the Object Builder, VisualAge Component Development toolkit, samples, the Server SDK, and (*PAO only*) CICS and IMS Application Adapter SDK
- (*PAO only*) CICS/IMS Application run time
- (*PAO only*) CICS/IMS Application client

Setting the CLASSPATH environment variable in the EJB server (CB) environment

To do any of the tasks listed below, make sure that the classes.zip file contained in the Java Development Kit is included in the CLASSPATH

environment variable. In addition, make sure that the following files are identified by the CLASSPATH environment variable to perform the associated task:

- Developing an enterprise bean or an EJB client: no additional files.
- Deploying an EJB JAR file:
  - somojor.zip
  - The EJB JAR file being deployed and any JAR or ZIP files on which it depends
- Running an EJB server (CB) managing an enterprise bean named *beanName*. These JAR files are automatically added to the CLASSPATH environment variable.
  - *beanName*S.jar
  - The EJB JAR file used to create *beanName*S.jar and any JAR or ZIP files on which it depends
- Running a pure Java EJB client using an enterprise bean named *beanName*:
  - *beanName*C.jar
  - somojor.zip
- Running an EJB server (CB) that contains an enterprise bean named *clientBeanName* that accesses another enterprise bean named *beanName* as a client. These JAR files are automatically added to the CLASSPATH environment variable.
  - *clientBeanName*S.jar
  - The EJB JAR file used to create *clientBeanName*S.jar and any JAR or ZIP files on which it depends
  - *beanName*C.jar

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 of the EJB specification. These components are described in “Chapter 5. Developing enterprise beans” on page 89.

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, and the bean’s remote interface.

After you have properly coded these components, use the Java compiler to create the corresponding Java class files. For example, since the components of the example Account bean are stored in a specific directory, you can compile the bean components 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 (CB)

In the EJB server (CB), finder logic is contained in a finder helper class. The enterprise bean deployer must implement the finder helper class before deploying the enterprise bean and then specify the name of the class with the -finderhelper option of the **cbejb** tool.

For each specialized finder method in the home interface (other than the findByPrimaryKey method), the finder helper class must have a corresponding method with the same name and parameter types. When an EJB client invokes a specialized finder method, the generated CB home that implements the enterprise bean's home interface invokes the corresponding finder helper method to determine what to return to the EJB client.

The finder helper class must also have a constructor that takes a single argument of type com.ibm.IManagedClient.IHome. When the CB home instantiates the finder helper class, the CB home passes a reference to itself to the finder helper constructor. This allows the finder helper to invoke methods on the CB home within the implementation of the finder helper methods, which is particularly useful when the CB home is an IQueryableIterableHome because the finder helper can narrow the IHome object passed to the constructor and invoke query service methods on the CB home.

The names of the entity bean's container-managed fields are mapped to interface definition language (IDL) attributes of the same name, except that an underscore (_) is appended, in the business object (BO) interface, the CB key class, and the CB copy helper class. These names are mapped exactly to IDL attributes in the DO interface. For example, in the AccountBean class, the *accountId* variable is mapped to *accountId_* in the BO interface, the CB key class, and the CB copy helper class, but is mapped to *accountId* in the DO interface.

This renaming is necessary, and relevant to finder helper classes implemented by using the Component Broker Query Service, because the entity bean's remote interface can also have a property named *accountId* (of potentially a different type) that must also be exposed through the BO interface. If that is the case, then a query over the BO attribute *accountId* is done in object space, whereas a query over the BO attribute *accountId_* is done directly against the underlying data source, which is typically more efficient.

If a home interface's specialized finder method returns a single entity bean, then the corresponding method in the finder helper class must return the

java.lang.Object type. When invoked, the finder helper method can return the EJB object, the CB key object, or the entity bean's primary key object. If the finder helper method returns a CB object or a primary key object, the CB home determines the corresponding EJB object to return to the EJB client.

If a home interface's specialized finder method returns a java.util.Enumeration type, the corresponding finder helper method must also return java.util.Enumeration. When invoked, the finder helper method can return an Enumeration of EJB objects, CB key objects, enterprise bean primary key objects, or a heterogeneous mix of one or more of the three. The CB home then constructs a serializable Enumeration object containing the corresponding EJB objects, which is returned to the EJB client.

An optional base class, named com.ibm.ejb.cb.runtime.FinderHelperBase, is provided with the EJB server (CB) environment to assist in the development of a finder helper class. This class encapsulates the Component Broker Query Service, so that the deployer does not need to write any CB-specific code. The FinderHelperBase base class contains an evaluate method, which takes an Object-Oriented Structured Query Language (OOSQL) predicate as a parameter and returns an Enumeration of objects that meet the conditions of the query. (The evaluate method calls the IQueryableIterableHome.evaluate method, then iterates through the returned IIterator object to construct an Enumeration object containing the search results. This method throws a javax.ejb.FinderException if any errors occur; the finder helper class does not need to catch this exception; instead, the class can pass it on to the EJB client.)

A utility class, named **com.ibm.ejb.cb.emit.cb.FinderHelperGenerator** (contained in the developEJB.jar file), is also provided to further assist the deployer in the development of a finder helper class. This utility takes the name of an entity bean's home interface and generates a Java source file containing a class that extends com.ibm.ejb.cb.runtime.FinderHelperBase and that contains skeleton methods for each specialized finder method in the home interface. In addition, each finder helper method (with a corresponding finder method that returns an Enumeration object) contains code to invoke the evaluate method.

By using the **FinderHelperGenerator** utility, the deployer can easily implement the finder helper class. You can use a batch file to run the utility. For example, to generate a finder helper class for the example AccountHome interface, enter the following command:

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

This command generates the finder helper class shown in Figure 16 on page 55.

```
...
public class AccountFinderHelper extends FinderHelperBase {
     ...
     AccountFinderHelper(IManagedClient.IHome iHome) {
          ...
     }
     public Enumeration findLargeAccounts(float amount) {
          return evaluate("Place SQL String here");
     }
 }
```

Figure 16. Code example: Generated AccountFinderHelper class for the EJB server (CB)

To enable the helper class for use in a deployed enterprise bean, the deployer makes a few simple edits to the parameters of the evaluate invocations. For example, for the AccountFinderHelper class, the `"Place SQL String here"` String is replaced with `"balance_>" + amount` as shown in Figure 17. The generated finder helper class can be used only with an enterprise bean that is deployed to have a queryable home by using the -queryable option of the **cbejb** tool.

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

Figure 17. Code example: Completed AccountFinderHelper class for the EJB server (CB)

Using lazy enumeration

The Enumeration returned by the evaluate method is called *eager*, because all the enterprise bean references that match the query are brought into memory and stored in the enumeration before being passed from the server to the client. If the number of references returned by the query is large, the deployer can use *lazy* enumeration; that is, it incrementally fetches more enterprise bean references only when the client calls the nextElement method on the Enumeration.

To use lazy enumeration, change the call to the evaluate method in the FinderHelper to a call to the lazyEvaluate method. A transaction must already be started before the home's finder method is called. The caller must not call the nextElement method on the Enumeration after the completion of the transaction.

At configuration time, the System Management End User Interface must be used to enable the settings for lazy Enumerations. Refer to “Configuring systems management to enable lazy enumeration” on page 75

Creating an EJB JAR file for an enterprise bean

Once the bean components are built, the next step is package them into an EJB JAR file. The **jetace** tool is used to create an EJB JAR file for one or more enterprise beans. For more information on creating an EJB JAR file, see “Creating a deployment descriptor and an EJB JAR file” on page 33.

Deploying an enterprise bean

During deployment, a deployed JAR file is generated from an EJB JAR file. Use the **cbejb** tool to deploy enterprise beans in the EJB server (CB) environment. The deployed JAR file contains classes required by the EJB server. The **cbejb** tool also generates the data definition language (DDL) file used during installation of the enterprise bean into the EJB server (CB).

If you want to use an enterprise bean on a different machine from the one on which it was developed (and on which you ran **cbejb**), follow the guidelines for installing applications in the Component Broker document entitled *System Administration Guide*. If an enterprise bean uses additional files (such as other JAR files) that need to be copied with the enterprise bean, specify these files in the properties notebook of the application (*not* the family).

The **cbejb** tool has the following syntax:

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

The *ejb-jarFile* parameter is required; it must be the first argument and it must specify a valid EJB JAR file as described in “Creating an EJB JAR file” on page 119. If the -ob option is used, it must come second on the command line. The other options can be specified in any order. The *beanNames* argument is a list of one or more fully qualified enterprise bean names delimited by colons (:) (for example, com.ibm.ejs.doc.transfer.Transfer:com.ibm.ejs.doc.account.Account). For the enterprise bean name, specify either the bean’s remote interface name or the

name of its deployment descriptor. If the *beanNames* argument is not specified for a particular option, then the effect of that option is applied to all enterprise beans in the EJB JAR file for which the option is valid.

Note: The relative file name of the JAR files specified by the *ejb-jarFile* variable and by the two *deployed-jarFile* variables must be different from each other. JAR file names that have the same relative file names but different paths are not valid.

The rest of the command parameters are optional and can be specified in any order. For explanation purposes, the options can be grouped by function into three general categories:

- Deployment options, which govern the generation and compilation of code.
- Storage options, which govern persistent storage.
- Execution options, which govern the run time environment.

The -rsp option does not fit into these categories. This option allows you to create a file containing some or all of the other options and their values (except the *ejb-jarFile* parameter). You can then submit the file to the **cbejb** command. This allows the common setting to be saved and makes commands easier to issue.

- Deployment options
  - -ob *projDir* — Specifies the relative or full path of the project directory in which the generated files are stored. If this option is not specified, the current working directory is used as the project directory.
  - Compilation modifiers — By default, the **cbejb** tool does the following for each enterprise bean contained in the EJB JAR file:
    1. Generate and import XML.
    2. Generate code—Creates a DDL file, makefile, and other source files for each enterprise bean contained in the EJB JAR file. These files are placed in the specified project directory.
    3. Compile and link—Invokes the generated makefile to compile an application. Each application file is placed in the specified project directory. While the Dynamic Link Libraries (DLLs) are being linked, numerous duplicate symbol warnings appear; these warnings are harmless and can be ignored.

    The following command options modify the default compilation behavior:

    - -nm — Suppresses the XML-processing step.
    - -ng — Suppresses the code-generation step.
    - -nc — Suppresses the compilation-and-linking step.

- -cc — Removes previously compiled and linked code by invoking the generated makefile to remove non-source files. This option must be used if you specify either of these combinations:
  - -ng -nc
  - -nm -ng -nc

– -bean *beanNames* — Identifies the enterprise beans in the EJB JAR file to be deployed. By default, all enterprise beans in the EJB JAR file are deployed. To deploy multiple enterprise beans, delimit the bean names with a : (colon). For example, `Account:Transfer`.

– -platform — Specifies the platform for which to generate code. This also sets the deployment platform in the Object Builder tool, but it does not set the platform for viewing, generating, or applying development constraints. You must set these manually by using the choices on the **Platform** menu.

– -guisg — Directs the tool to present the Object Builder graphical user interface (GUI), which enables the tool to collect options from the user rather than from the command line.

– -usecurdopo — Directs the tool to use the current mapping between the data object and the persistent object in the existing model rather than bringing up the Object Builder interface to build a mapping. Use this option when redeploying beans for which a satisfactory mapping already exists. The deployment will proceed automatically.

  When you first deploy CMP entity beans, you must *not* use this option. The tool will then build the default mapping between the data and persistent objects and, if you specify the -guisg option, launch the Object Builder interface.

– -nousraction — Directs the tool to use only the information on the command line after building the mapping between data objects and persistent objects. Otherwise, if you have also specified the -guisg option, the tool prompts you for the next action.

– -polymorphichome — Specifies the beans that use polymorphic home interfaces.

– -queryable — Directs the tool to generate a queryable CB home object. This option can be used only for entity beans with CMP that store their persistent data in a relational database. This option must be used if the finder helper class, which is used to implement the finder methods in a CMP entity bean, uses the CB query service. This option must *not* be used if an entity bean uses CICS or IMS to store its persistent data.

  By default, the interface definition language (IDL) interface of an enterprise bean's CB home extends the IManagedClient::IHome class, and the home implementation extends the IManagedAdvancedServer::ISpecializedHome class. An IDL interface of a queryable home extends the

IManagedAdvancedClient::IQueryableIterableHome class, and the home implementation extends the IManagedAdvancedServer::ISpecializedQueryableIterableHome class.

In addition, the generated BO interface is marked as queryable. For queryable homes, the EJB client programming model remains unchanged; however, a Common Object Request Broker Architecture (CORBA) EJB client can treat the EJB home as an IManagedAdvancedClient::IQueryableIterableHome object.

For more information on queryable homes, see the *Advanced Programming Guide*.

- Storage options
  - -dbname *DBName* — Specifies the name of the database for beans with CMP.
  - Database choices—The default database for persistent storage of container-managed beans is DB2 version 5.2 with embedded SQL. You can override this default by using:
    - -cacheddb2v52 — Identifies entity beans with CMP that require DB2 version 5.2 used with the Cache Service to store persistent data.
    - -cacheddb2v61 — Identifies entity beans with CMP that require DB2 version 6.1 used with the Cache Service to store persistent data.
    - -db2v61 — Identifies entity beans with CMP that require DB2 version 6.1 used with embedded SQL to store persistent data.
    - -oracle — Identifies entity beans with CMP that require Oracle to store persistent data. If you specify this option, you must also use the -queryable option.
    - -informix — Identifies entity beans with CMP that require Informix to store persistent data. A given transaction cannot access more than one Informix database from a CB server. To access two Informix databases in one transaction, you must access each from a different CB server. If you specify this option, you must also use the -queryable option.
    - -jdbcaa — Identifies entity beans with BMP that require JDBC to store persistent data. This option enables the beans to join distributed transactions by allowing the bean implementation to connect to the Transaction Service. Beans with BMP that do not use this option will handle transactions in an implementation-dependent manner.
  - -hod — Identifies entity beans with CMP that use Host-on Demand (HOD) to store persistent data. These beans will use the Session Service. This option must *not* be used for enterprise beans generated from the **PAOToEJB** tool.
  - -eci — Identifies entity beans with CMP that use the external call interface (ECI) to store persistent data. These beans will use the Session Service. This option must *not* be used for enterprise beans generated from the **PAOToEJB** tool.

- -appc — Identifies entity beans with CMP that use advanced program-to-program communications (APPC) to store persistent data. These beans will use the Transaction Service. This option must *not* be used for enterprise beans generated from the **PAOToEJB** tool.
- -exci — Identifies entity beans with CMP that use the EXCI to store persistent data. These beans will use the Transaction Service. This option must *not* be used for enterprise beans generated from the **PAOToEJB** tool.
- -otma — Identifies entity beans with CMP that use the OTMA to store persistent data. These beans will use the Transaction Service. This option must *not* be used for enterprise beans generated from the **PAOToEJB** tool.
- -ccf — Identifies entity beans with CMP that use the SAP interface, which is a common connector framework (CCF) back end. These beans will use the Transaction Service.

• Execution options
- -family *familyName* — Specifies the application family name to be generated. By default, this name is set to the name of the EJB JAR file with the word Family appended. This option can be specified more than once, as long as the values are unique.
- -finderHelper *finderHelperClassName remoteInterface*— Specifies the finder helper class name (*finderHelperClassName*) and remote interface name (*remoteInterface*) for entity beans with CMP. If unspecified, it is assumed that no finder helper class is provided by the deployer. This option can be specified more than once, as long as the values are unique. For more information on finder helper classes, see “Defining finder methods” on page 100.
- -usewstringindo — Directs the tool to map the container-managed fields of an entity bean to the wstring IDL type (rather than the string type) on the DO. It is preferable to map to the string IDL type if the data source contains single-byte character data; it is preferable to map to the wstring IDL type if the data source contains double-byte or Unicode character data.
- -workloadmanaged — Directs the tool to configure a CMP entity bean or a stateless session bean into a workload managing container and with a workload managed home interface. For a BMP entity bean or a stateful session bean it directs the tool to configure the bean only with a workload managed home interface.
- -clientdep *deployed-jarFile* — Specifies the name of a dependent JAR required by an EJB client that uses the enterprise bean being deployed. You must specify the full path of the file. To create multiple client JAR files, you must specify this option for each JAR file. This option can be specified more than once, as long as the values are unique.

- -serverdep *deployed-jarFile* — Specifies the name of a dependent JAR required by the EJB server (CB) that runs the deployed enterprise bean. You must specify the full path of the file. To create multiple dependent JAR files, you must specify this option for each JAR file. This option can also be used to identify existing JAR files that contain classes required by the enterprise bean being deployed; when this is done, the EJB server's CLASSPATH environment variable is automatically updated to include this specified JAR file. This option can be specified more than once, as long as the values are unique.
- -sentinel *sentinelValue* — Specifies an value for a Java type or container-managed field for the deployed beans. If you set a value for a Java type, do not put spaces around the = (equals) sign.
- -strbehavior — Specifies how the tool should determine the behavior of the strings for a container-managed string fields in deployed beans. The corba value indicates that strings should be handled as CORBA strings; the strip value directs the tool to remove trailing spaces from strings.

For session beans or entity beans with BMP, the code generation process runs without additional user intervention. For entity beans with CMP, the Object Builder GUI is displayed during execution of the command, and you must create a DO implementation to manage the entity bean's persistent data. For more information, see "Building a data object during CMP entity bean deployment" on page 62.

The **cbejb** tool deploys enterprise beans by generating extensible markup language (XML) files and importing those files into Object Builder. If the XML import fails, you can view any error messages generated by Object Builder in the import_model.log file located in the project directory.

If your CLASSPATH environment variable is too long, the **cbejb** command file fails. If this happens, shorten your CLASSPATH by removing any unnecessary files.

The **cbejb** tool generates the following files for an EJB JAR file containing an enterprise bean named Account:

- AccountS.jar and (*Windows NT*) AccountS.dll *or* (*AIX* or *Solaris*) libAccountS.so—The files required by the EJB server (CB) that contains this enterprise bean. The AccountS.jar file contains the code generated from the Account EJB JAR file. The AccountS.dll and libAccountS.so files contain the required C++ classes.

  (*Windows NT*) To run the Account enterprise bean in an EJB server (CB), the AccountS.jar file must be defined in the server's CLASSPATH environment variable, and the AccountS.dll file must be defined in the server's PATH environment variable. Typically, the System Management End User

Interface (SM EUI) sets these environment variables during installation of the deployed enterprise bean into an EJB server (CB).

(*AIX* or *Solaris*) To run the Account enterprise bean in an EJB server (CB), the AccountS.jar file must be defined in the server's CLASSPATH environment variable, and the libAccountS.so file must be defined in the server's LD_LIBRARY_PATH environment variable. Typically, the SM EUI sets these environment variables during installation of the deployed enterprise bean into an EJB server (CB).

- AccountC.jar—The file required by an EJB client, including enterprise beans that access other enterprise beans. This JAR file contains everything in the original EJB JAR file except the enterprise bean implementation class. To use the Account enterprise bean, a Java EJB client must have the AccountC.jar and the IBM Java ORB defined in its CLASSPATH environment variable.
- (*PAO only*) *paotoejbName*.jar—This file is created by the **PAOToEJB** tool and is used to wrap an existing procedural adapter object (PAO) in an enterprise bean.
- EJBAccountFamily.DDL—This file is used during installation of the Account family into an EJB server (CB) to update the database used by the SM EUI. Its name is composed of the EJB JAR file name with the string Family.DDL appended.

Building a data object during CMP entity bean deployment

When deploying an entity bean with CMP in the EJB server (CB), you must create a DO implementation by using Component Broker's Object Builder. This DO implementation manages the entity bean's persistent data.

To build a DO implementation, you must map the entity bean's container-managed fields to the appropriate data source as described in "Guidelines for mapping the container-managed fields to a data source". Then, you must do one of the following:

- Use an existing DB2 or Oracle database to store the bean's persistent data; for more information, see "Using an existing DB2 or Oracle data source to store persistent data" on page 65.
- Use an existing CICS or IMS application to store the bean's persistent data; for more information, see "Using an existing CICS or IMS application to store persistent data" on page 67.
- Define a new DB2 or Oracle database to store the bean's persistent data; for more information, see "Defining a new DB2 or Oracle database to store persistent data" on page 69.

Guidelines for mapping the container-managed fields to a data source

When you deploy enterprise beans with the **cbejb** tool, a Component Broker DO IDL interface is created. The IDL attributes of this interface correspond to the entity bean's container-managed fields. You must then define the DO

implementation by using Object Builder to map the DO attributes to the attributes of a Persistent Object (PO) or Procedural Adapter Object (PAO), which correspond to the data types found in the data source.

This section contains information on how the **cbejb** tool maps the container-managed fields of entity beans to DO IDL attributes, and how the enterprise bean deployer maps DO IDL attributes to the entity bean's data source. These guidelines apply whether you are using an existing data source (also known as meet-in-the-middle deployment) or defining a new one (also known as top-down deployment).

- EJBObject or EJBHome variables—Objects of classes that implement the EJBObject or EJBHome interface map to the Object IDL type. At run time, this DO attribute contains the CORBA proxy for the EJBObject or EJBHome object. The CB EJB run time automatically converts between the EJBObject or EJBHome object (stored in the bean's container-managed field) and the CORBA::Object attribute (stored in the C++ DO). It is possible to deploy container-managed beans that have container-managed fields of the same type, for example, a linked list implementation where each node of the list is a container-managed bean that has a reference to the next node. It is also possible to have circular references in a container-managed field, for example, a container-managed Bean A can have a container-managed field of type Bean B, which in turn has a container-managed field of type Bean A. When defining the DO-to-PO mapping in Object Builder, you can use either a predefined Component Broker mapping of CORBA::Object to the data source, or implement a C++ DO-to-PO mapping helper (in the standard Component Broker way) to invoke methods on the C++ proxy to obtain the persistent data. For more information on creating a C++ DO-to-PO mapping, see the Component Broker *Programming Guide*.

  Note: Although Component Broker allows an entity bean's container-managed fields to be EJBObject or EJBHome objects, the Enterprise JavaBeans 1.0 specification does not.
- Primary key variables—Do not map an enterprise bean's primary key variables to the SQL type long varchar in a DB2 or an Oracle database. Instead, use either a varchar or a char type and set the length appropriately.
- java.lang.String variables—Objects of this class are mapped to a DO IDL attribute of type string or wstring, depending on the command-line options used when the entity bean was deployed by using the **cbejb** tool (see "Deploying an enterprise bean" on page 56). By default, a variable of type java.lang.String is mapped to a DO IDL attribute of type string; however, the -usewstringindo option of the **cbejb** tool can be used to map java.lang.String variables to DO IDL attributes of type wstring. (Mapping some of a bean's String fields to the IDL string type and others to the IDL wstring type is not supported.) It is preferable to map to the string IDL

type if the data source contains single-byte character data; it is preferable to map to the wstring IDL type if the data source contains double-byte or Unicode character data.

- java.io.Serializable variables—Objects of classes that implement this interface are mapped to a DO IDL attribute of type ByteString (which is a typedef for sequence of octet defined in the IManagedClient.idl file). The EJB server (CB) automatically converts serializable objects (stored in the entity bean's container-managed fields) to the C++ sequence of octets containing the serialized form of the object (stored in the DO). Use the Component Broker default DO-to-PO mapping for ByteString to store the serialized object directly in the data source.

  Unless you implement a C++ DO-to-PO mapping helper that passes the C++ ByteString to a Java implementation by way of the interlanguage object model (IOM), it is not possible to manipulate the serialized Java object contained in a ByteString from within a C++ DO implementation. Therefore, if you are doing top-down enterprise bean development and you don't want to store a serialized Java object in the data source, it is recommended that you avoid defining container-managed fields of type Serializable. Instead, make the Serializable variable a nonpersistent variable, define primitive type container-managed fields to capture the state of the Serializable variable, and convert between the Serializable variable and the primitive variable in the ejbLoad and ejbStore methods of the enterprise bean.
- Array variables—These variables are mapped to a DO IDL sequence of the corresponding type in the same way that the individual types are mapped to DO IDL attributes. For example, an array of the java.lang.String class is mapped to a DO IDL attribute that is a sequence of type string (or a sequence of type wstring, if the -usewstringindo option of the **cbejb** tool is used). The EJB server (CB) automatically converts between the array (stored in the entity bean's container-managed fields) and the C++ sequence (stored in the DO). You can store the entire sequence in the data source as a whole, or you can write a C++ DO-to-PO mapping helper (in the standard Component Broker way) to iterate through the sequence and store individual elements in the data source separately. For more information on creating a C++ DO-to-PO mapping, see the Component Broker *Programming Guide*.
- Date/Time fields—The **cbejb** tool maps container-managed fields of type java.util.Date and its subclasses (java.sql.Date, java.sql.Time, java.sql.Timestamp *only*) differently from other Serializable fields. The following mapping rules are used:
  - java.util.Date: ISO-formatted timestamp string (*yyyy-mm-dd-hh.mm.ss.mmmmmm*)
  - java.sql.Date: ISO-formatted date string (*yyyy-mm-dd*)
  - java.sql.Time: ISO-formatted time string (*hh.mm.ss*)

– java.sql.Timestamp: ISO-formatted timestamp string (*yyyy-mm-dd-hh.mm.ss.mmmmmm*)

Therefore a container-managed field of one of the above types should be mapped to either a string or a database-specific date/time field that can take an ISO-formatted string as input. (For example, both DB2 and Oracle Date/Time/Timestamp column types can take ISO strings as input values.) If a deployer chooses to map a Date/Time container-managed field to something other than the types mentioned above, then a special data mapping code should be written in the DO implementation. The mapping code must be able to convert an ISO-formatted string to a backend-specific type and vice versa.

The java.sql.Timestamp class has a precision of nanoseconds, whereas ISO timestamp format has a precision of microseconds. Therefore, precision is compromised (by rounding nanoseconds to nearest microseconds) when a Timestamp CMP field is mapped. Users should be particularly aware of this when they use the java.sql.Timestamp class as one of the attributes of bean's primary key.

While mapping java.sql.Date to ISO Date format, the time field values are ignored. Similarly while mapping java.sql.Time to ISO Time, the date field values are ignored.

Note: For DB2 only: If an existing database outputs date/time in a non-ISO format, then the deployer must rebind DB2 packages using the ″DATETIME ISO″ option.

Using an existing DB2 or Oracle data source to store persistent data

To use an existing DB2 or Oracle database to store a CMP entity bean's persistent data, follow these steps. The end result is a PO with attributes that correspond to the items in the database schema.

1. When Object Builder starts, it presents the Open Project dialog. Choose the location of the project directory for your enterprise bean and click **Finish**.
2. To import an existing relational database schema, click **DBA-Defined Schemas** and right-click the appropriate database type.
   a. On the pop-up menu, click **Import** and **SQL**.
   b. On the **Import SQL** dialog box, click **Find** and browse for your SQL file.
   c. Double-click your SQL file.
   d. Change the name in the **Database Name** text field from `Database` to the actual name of the database.
   e. Select the appropriate database type and click **Finish**.
3. To create a persistent object (PO) from the database schema, expand **DBA-Defined Schemas** and expand your group.

a. Highlight your schema and then right-click it to display a pop-up menu. Click **Add->Persistent Object**.

b. On the **Names and Attributes** dialog box, accept the defaults and click **Finish**.

4. Create a DO implementation as follows:

   a. Expand the **User-Defined DOs**, expand the **DO File** (for example CBAccountDO), expand the **DO Interface** (for example, com_ibm_ejs_doc_account_AccountDO), and select the **DO Implementation**.

   b. On the **DO Implementation** pop-up menu, select **Properties**.

   c. On the **Name and Platform** page, select the **Deployment Platform** (for example, NT, AIX, or Solaris) and click **Next**.

   d. On the **Behavior** page, make the appropriate selections and click **Next**:

      - *For DB2*, select `BOIM with any Key` for **Environment**, select `Embedded SQL` for **Form of Persistent Behavior and Implementation**, select `Delegating` for **Data Access Pattern**, and select `Home name and key` for **Handle for Storing Pointers**.
      - *For Oracle*, select `BOIM with any Key` for **Environment**, select `Oracle Caching services` for **Form of Persistent Behavior and Implementation**, select `Delegating` for **Data Access Pattern**, and select `Home name and key` for **Handle for Storing Pointers**.

   e. On the **Implementation Inheritance** page, make the appropriate selections for the parent class and click **Next**:

      - *For DB2*, select `IRDBIMExtLocalToServer::IDataObject`
      - *For Oracle*, select `IRDBIMExtLocalToServer::ICachingServiceDataObject`

   f. Accept the defaults for the **Attributes**, **Methods**, and **Key and Copy Helper** pages by clicking **Next** on each page.

   g. On the **Associated Persistent Objects** page, click **Add Another**. Accept the default for the instance name (iPO) and select the correct type. Click **Next**.

   h. On the **Attribute Mapping** page, map the container-managed fields of the entity bean to the corresponding items in the database schema. Object Builder creates default mappings for the data object attributes for which it can identify corresponding persistent object attributes. The default mapping is generally suitable for everything except for the primary key variable, which you must map to a varchar or char type rather than a long varchar type. For more information, see "Guidelines for mapping the container-managed fields to a data source" on page 62. After you finish mapping the attributes, click **Finish**.

   i. *Oracle only*. When mapping an entity bean with CMP to an Oracle database, expand the **Container Definition** folder and right-click the

EJB container. From the pop-up menu, click **Properties**. In the wizard, click **Next** until you reach the **Data Access Patterns**; on that page, check the **Cache Service** checkbox and click **Finish**.

j. Exit from Object Builder by clicking **File->Exit**; save any changes if prompted.

k. Create the database specified by the **Database** text field and use the SQL file specified by the **Schema File** text field to create a database table. For more information on creating a database and database table with an SQL file, consult your DB2 or Oracle documentation. The SQL file can be found in the following directory, where *projDir* is the project directory created by the **cbejb** tool:

   - *On Windows NT*, *projDir*\Working\NT
   - *On AIX*, *projDir*/Working/AIX
   - *On Solaris*, *projDir*/Working/Solaris

Using an existing CICS or IMS application to store persistent data

To use CICS or IMS for Persistent Adaptor Object (PAO) storage, following these instructions. Note that if the persistent store uses a CICS or IMS application (by way of a PAO), only application data is used; the methods on the CICS or IMS application are pushdown methods, which run application-specific logic rather than storing and loading data.

The following prerequisites must be met to map an entity bean with CMP to an existing CICS or IMS application:

- The entity bean's transaction attribute must be set to TX_MANDATORY if you want to map the bean to a HOD- or ECI-based application. The transaction attribute must be set to either the TX_MANDATORY or TX_REQUIRED if you want to map it to an APPC-based application.
- The existing CICS or IMS application must be represented as a procedural adapter object (PAO). See the *Procedural Application Adaptor Development Guide* for more information on creating PAOs.
- The PAO class files must be specified in the CLASSPATH environment variable.
- The entity bean must implement all enterprise bean logic; the only remaining requirement is to map the entity bean's container-managed fields to the PAO. Pushdown methods on the PAO cannot be utilized from the enterprise bean. (PAO pushdown methods can be used from an entity bean with CMP generated by using the **PAOToEJB** tool as described in "Creating an enterprise bean from an existing CICS or IMS application" on page 81.)
- The **cbejb** tool must be run as follows, where the *ejb-jarFile* is the EJB JAR file containing the entity bean:

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

For a description of the **cbejb** tool's syntax, see "Deploying an enterprise bean" on page 56.

If you have met the prerequisites, use Object Builder to create the mapping between the entity bean and the CICS or IMS application:

1. When Object Builder starts, it presents the Open Project dialog. Choose the location of the project directory for your enterprise bean and click **Finish**.
2. From the main menu, click **Platform** and then **Target**. Uncheck the 390 platform.
3. Click **User-Defined PA Schemas** and right-click the selection.
4. From the pop-up menu, click **Import** and then **Bean**. On the **Import Bean** dialog box, type the class name of the PAO bean and click **Next**.
5. Select the appropriate connector type and click **Next**.
6. Select the primary key attribute name from the **Properties** list.
7. Click >> to move the primary key to the **Key Attributes** list and click **Finish**.
8. *For HOD and ECI only*, do the following for both the MO and the HomeMO:
   a. In the **Tasks and Object** panel, expand the **User-Defined Business Objects**, expand the object, and expand the object's BO. From the MO file's pop-up menu, click **Properties**.
   b. Change the **Service to use** property from `Transaction Service` to `Session Service`.
9. Create a DO implementation as follows:
   a. On the **Tasks and Object** panel, expand the **User-Defined DOs**, expand the **DO File** from the menu, and click the **DO Interface**.
   b. On the **DO Interface** pop-up menu, select **Add Implementation**.
   c. On the **Behavior** page, select `BOIM with any Key` for **Environment**, select `Procedural Adapters` for **Form of Persistent Behavior and Implementation**, select `Delegating` for **Data Access Patterns**, and select `Default` for **Handle for Storing Pointers**. Click **Next**.
   d. Click **Next** on the **Implementation Inheritance** page, the **Attributes** page, the **Methods** page, and the **Key and Copy Helper** page.
   e. On the **Associated Persistent Object** page, click **Add Another**, verify that the PO that you previously created is selected, and click **Next**.
   f. On the **Attribute Mapping** page, designate how the container-managed fields of the entity bean correspond to the items in the existing PAO. This designation is done by defining a mapping between the attributes of the DO (which match the entity bean's container-managed fields) to

the attributes of the PO (which match the existing PAO). In the **Attributes** list, there is a DO attribute corresponding to each of the bean's container-managed fields.

For each DO attribute in the **Attributes** list, right-click the attribute and click **Primitive** from the menu. From the **Persistent Object Attribute** drop-down menu, select the PO attribute (the item from the existing database schema) that corresponds to the DO attribute. For more information, see "Guidelines for mapping the container-managed fields to a data source" on page 62. After you have processed all container-managed fields, click **Next**.

g. On the **Methods Mapping** page, for each method in the list of **Special Framework Methods**, right-click a method and click **Add Mapping**. From the **Persistent Object Method** drop-down menu, select the PO method with the same name as the selected DO method. If there are more methods than available mappings, map methods to similarly named methods. For example, map update to update(). After you have processed all of the methods, click **Finish**.

h. Expand the **Container Definition** folder and right-click the EJB container. From the pop-up menu, click **Properties**. In the wizard, click **Next** until you reach the **Data Access Patterns** page.

i. On the **Data Access Patterns** page, select one of the following items and then click **Next**:

- *For HOD or ECI*, select `Use PAA Session services`.
- *For APPC*, select `Use PAA Transaction services`.

j. On the **Service Details** page, do the following and then click **Next**:

- *For HOD or ECI*, select `Throw an exception and abandon the call` for **Behavior for Methods Called Outside a Transaction**; define a connection name, for example, `MY_PAA_Connection`; select `Host on Demand` or `ECI connection`, respectively, for the **Type of connection**.
- *For APPC*, select `Throw an exception and abandon the call` for enterprise beans with the TX_MANDATORY transaction attribute, or select `Start a new transaction and complete the call` for enterprise beans with the TX_REQUIRED transaction attribute.

k. Select `Caching` for **Business Object**.

l. Select `Delegating` for **Data Object**.

m. Click **Finish**.

10. Exit from Object Builder by clicking **File->Exit**; save any changes if prompted.

Defining a new DB2 or Oracle database to store persistent data

When you use a top-down development approach to enterprise bean development, enterprise bean deployment must occur in three phases:

1. Define the database schema, map the container-managed fields of the entity bean with CMP to the database schema, and generate the code to encapsulate this mapping. For more information, see “Mapping the database schema”.
2. Create the database and database tables. For more information, see “Creating the database and database table”.
3. Compile the code generated in phase 1; compilation fails if the database and database tables do not exist.

Mapping the database schema: After you have defined the manner in which the entity bean maps to a database, create the mapping by running the **cbejb** tool with the -nc option to prevent automatic compilation after code generation. For example, to create a mapping for an Account bean stored in an EJB JAR file named EJBAccount.jar, enter the following command:

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

Note: If the database being used to store the persistent data is either Oracle or DB2, those options must also be specified.

Creating the database and database table: Follow these instructions to create a database and database table by using the Object Builder GUI:

1. When Object Builder starts, it presents the Open Project dialog. Choose the location of the project directory for your enterprise bean and click **Finish**.
2. Create a DO implementation as follows:
   a. Expand the **User-Defined DOs**, expand the **DO File** from the menu, and click the **DO Interface**.
   b. On the **DO Interface** pop-up menu, select **Add Implementation**. If the implementation is already present, you can modify it by selecting the implementation, invoking the pop-up menu, and selecting **Properties**.
   c. On the **Name and Platform** page, select the platform and click **Next**.
   d. On the **Behavior** page, make the appropriate selections and click **Next**:
      - *For DB2*: select `BOIM with any Key` for **Environment**, select `Embedded SQL` for **Form of Persistent Behavior and Implementation**, select `Delegating` for **Data Access Pattern**, and select `Home name and key` for **Handle for Storing Pointers**.
      - *For Oracle*: select `BOIM with any Key` for **Environment**, select `Oracle Caching services` for **Form of Persistent Behavior and Implementation**, select `Delegating` for **Data Access Pattern**, and select `Home name and key` for **Handle for Storing Pointers**.
   e. On the **Implementation Inheritance** page, make the appropriate selections for the parent class and click **Next**:
      - *For DB2*, select `IRDBIMExtLocalToServer::IDataObject`

- *For Oracle*, select `RDBIMExtLocalToServer::ICachingServiceDataObject`
- *For CICS or IMS PAO*, select `IRDBIMExtLocalToServer::IDataObject`

f. Accept the defaults for the **Attributes**, **Methods**, and **Key and Copy Helper** pages by clicking **Next** on each page.

g. On the **Associated Persistent Objects** page, click **Add Another**. Accept the default for the instance name (iPO) and select the correct type. Click **Next**.

h. On the **Attribute Mapping** page, map the container-managed fields of the entity bean to the corresponding items in the database schema. The default mapping is generally suitable for everything except for the primary key variable, which you must map to a varchar or char type rather than a long varchar type. Object Builder creates default mappings for the data object attributes for which it can identify corresponding persistent object attributes. For more information, see "Guidelines for mapping the container-managed fields to a data source" on page 62. After you finish mapping the attributes, click **Finish**.

i. *Oracle only*. When mapping an entity bean with CMP to an Oracle database, expand the **Container Definition** folder and right-click the EJB container. From the pop-up menu, click **Properties**. In the wizard, click **Next** until you reach the **Data Access Patterns**; on that page, check the **Cache Service** checkbox and click **Finish**.

j. Exit from Object Builder by clicking **File->Exit**; save any changes if prompted.

k. Create the database specified by the **Database** text field and use the SQL file specified by the **Schema File** text field to create a database table. For more information on creating a database and database table with an SQL file, consult your DB2 or Oracle documentation. The SQL file can be found in the following directory, where *projDir* is the project directory created by the **cbejb** tool:

- *On Windows NT*, *projDir*\Working\NT
- *On AIX*, *projDir*/Working/AIX
- *On Solaris*, *projDir*/Working/Solaris

Compiling the generated code: After both the database and database table are created, compile the enterprise bean code by using the following commands:

- **On Windows NT.**

```
# cd projDir\Working\NT
# nmake -f all.mak
```

- **On AIX.**

```
# cd projDir/Working/AIX
# make -f all.mak
```

- **On Solaris.** Transfer the code that was generated on Windows NT or AIX, then use the following commands.

```
# cd projDir/Working/Solaris
# make -f all.mak
```

Installing an enterprise bean and configuring its EJB server (CB)

Follow these steps to install an enterprise bean and configure the resulting EJB server (CB):

1. (*Entity bean with CMP using DB2 only*) Use the bind file, which Object Builder generates as a side effect of using the **cbejb** tool, to bind the enterprise bean to the database (for example, `db2 bind AccountTblPO.bnd`).
2. Using the SM EUI, install the application generated by **cbejb**. In general, this installation is the same as installing a Component Broker application generated by Object Builder:
   a. Load the application into a host image.
   b. Add the application to a configuration.
   c. Associate the EJB application with a server group or server. (If the server group or server does not already exist, you must create it.)
   d. (*Entity bean with CMP only*) Associate the entity bean's data source (DB2, Oracle, CICS, or IMS PAA) with the EJB application:
      - DB2: associate the DB2 services (iDB2IMServices) with the EJB server.
      - Oracle: associate the Oracle services (iOAAServices) with the EJB server.
      - CICS or IMS PAA: associate the PAA services (iPAAServices) with the EJB server.
   e. Configure the EJB server (CB) with a host.
   f. Set the ORB request timeout for both clients and servers to 300 seconds.
   g. If the EJB server requires Java Virtual Machine (JVM) properties to be set, edit the JVM properties. Do this in the server model instead of the server image. For instance, if the enterprise bean performs a JNDI lookup to access other enterprise beans, the server hosting the enterprise bean must have its JVM properties set to include values for JNDI properties.
   h. Activate the EJB server configuration.
   i. Start the EJB server.

Binding the JNDI name of an enterprise bean into the JNDI namespace

Note: This section does not apply to servers running on the AIX, Windows NT, or Solaris platforms.

An enterprise bean's JNDI home name is defined within its deployment descriptor as described in "The deployment descriptor" on page 19. This name is used by EJB clients (including other enterprise beans) to find the home interface of an enterprise bean.

The **ejbbind** tool locates the CB home that implements the enterprise bean's EJBHome interface in the Component Broker namespace. It also rebinds the home name into the namespace, using the JNDI home name specified in the enterprise bean's deployment descriptor. This binding enables an EJB client to look up the EJB home by using the JNDI name specified in the bean's deployment descriptor. An enterprise bean can be bound on a different machine from the one on which the bean was deployed.

The subtree of the Component Broker namespace in which the JNDI name is bound can be controlled by the command-line options used with the **ejbbind** tool. The manner in which the name is bound (the subtree chosen) affects the JNDI name that EJB clients must use to look up the enterprise bean's EJB home and also affects the visibility of the enterprise bean's EJB home. Specifically, the JNDI name can be bound in one of the following ways:

- The JNDI name can be bound into the local root. Under this binding approach, EJB clients use the JNDI name in the enterprise bean's deployment descriptor. The approach restricts the visibility of the EJB home to EJB clients using the same name server (the same bootstrap host) and can cause collisions with other names in the tree.
- The JNDI name can be bound into the host name tree (at host/resources/factories/EJBHomes). Under this binding approach, EJB clients must prefix the string `host/resources/factories/EJBHomes` to the JNDI name given in the bean's deployment descriptor. This approach minimizes collisions with other names in the tree, but restricts visibility of the enterprise bean home to clients using the same name server.
- The JNDI name can be bound into the workgroup name tree (at workgroup/resources/factories/EJBHomes). Under this binding approach, EJB clients must prefix the string `workgroup/resources/factories/EJBHomes` to the JNDI name given in the enterprise bean's deployment descriptor, and the EJB home is visible to all EJB clients using a name server that belongs to the same preferred workgroup.
- The JNDI name can be bound into the cell name tree (at cell/resources/factories/EJBHomes). Under this binding approach, EJB

clients must prefix cell/resources/factories/EJBHomes to the JNDI name in the bean's deployment descriptor, and the EJB home is visible throughout the cell.

Before running the **ejbbind** tool, do the following:

- Deploy your enterprise bean for Component Broker by using the **cbejb** tool. For more information, see "Deploying an enterprise bean" on page 56.
- Install the Component Broker application that **cbejb** tool generates, and configure it on a specific EJB server (CB) by using the SM EUI. For more information, see "Installing an enterprise bean and configuring its EJB server (CB)" on page 72.
- Start the CBConnector Service and a name server, if they are not already running. For more information, see the Component Broker *System Administration Guide*.
- Activate the configuration containing the EJB server (CB) that runs the application.
- Determine the IP address (the bootstrap host name) and port number (the bootstrap port) of the machine running the name server.

Invoke the **ejbbind** command with the following syntax:

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

The *ejb-jarFile* is the fully-qualified path name of the EJB JAR file containing the enterprise bean to be bound or unbound. The optional *beanParm* argument is used to bind a single enterprise bean in the EJB JAR file; you can identify this bean by supplying a fully qualified name (for example, com.ibm.ejs.doc.account.Account, where Account is the bean name) or the name of the enterprise bean's deployment descriptor file without the .ser extension. If an enterprise bean has multiple deployment descriptors in the same EJB JAR file, you must supply the deployment descriptor file name rather than the enterprise bean name.

When no options are specified, the JNDI name is bound into the local root's name tree, using the local host and port 900 for the bootstrap host (the name server).

The other options do the following:

- -f — Force the bind, even if the JNDI name is already bound in the namespace; this option is not valid with the unbind command options.
- -BindLocalRoot — Bind the JNDI name into the local root's name tree.
- -BindHost — Bind the JNDI name into the host name tree.

- -BindWorkgroup — Bind the JNDI name into the workgroup name tree.
- -BindCell — Bind the JNDI name into the cell name tree.
- -BindAllTrees — Bind the JNDI name into the host, the workgroup, and the cell name trees.
- -ORBInitialHost *hostName* — Identify the bootstrap host (the default is the local host).
- -ORBInitiallPort *portNumber* — Identify the bootstrap port (the default is port 900).
- -u — Unbind the JNDI name; this option is not valid with bind command options.
- -UnbindLocalRoot — Unbind the JNDI name from the local root's name tree.
- -UnbindHost — Unbind the JNDI name from the host name tree.
- -UnbindWorkgroup — Unbind the JNDI name from the workgroup name tree.
- -UnbindCell — Unbind the JNDI name from the cell name tree.
- -UnbindAllTrees — Unbind the JNDI name from the host, the workgroup, and the cell name trees.

If the command is successful, it issues a message similar to the following:

```
Name AccountHome was bound to CB Home
```

You must run the **ejbbind** tool again if any of the following occurs:

- You modify the JNDI name of an enterprise bean. You can modify the JNDI name by using the **jetace** tool. For more information, see "Creating a deployment descriptor and an EJB JAR file" on page 33.
- You reconfigure Component Broker. In this case, you must rebind every enterprise bean served by this configuration.
- You move the enterprise bean to a different EJB server (CB) or a different machine.

Configuring systems management to enable lazy enumeration

To enable lazy enumeration (see "Creating finder logic in the EJB server (CB)" on page 53), follow these steps:

1. From the System Management End User Interface (SM EUI), go to the View menu, and set the View Level to Control.
2. Expand **Host Images**
3. Expand the name of your host.
4. Expand **Server Images**.
5. Expand the name of your server.
6. Expand **Container Images**.

7. Right-click **iIteratorSysObjsNoPRef**. From the pop-up menu, select **Properties**. Change the following properties:
   - Change the **Default transaction policy** to `throwException`.
   - Change the **Memory management policy** to `passivate at end of transaction`.

The transaction policy ensures that the caller starts a transaction. The memory management policy ensures that the lazy enumerations are passivated when the transaction completes.

Resolving to EJB homes using lifecycle services in CBConnector

Note: This section applies only to servers running on the AIX, Windows NT, or Solaris platforms.

When an EJB client performs a simple JNDI lookup, a 1-to-1 mapping is made between the name and the particular EJB home instance. In a distributed environment, this model can be limiting. In such an environment, for example, there may be many EJB homes supporting the same type of enterprise bean. It is better to have an approach that does not require an application to request a specific instance of that home. In addition, as changes are made to the system, it is important that applications not have to be changed or redeployed to specify a different instance of an EJB home. The CBConnector LifeCycle Service provides a level of indirection and abstraction that allows the application to request a home that is within a particular scope of location within the distributed environment, yet be isolated from the specifics of the exact configuration of the environment. For more info on lifecycle factory finders, see the LifeCycle section in the *Advanced Programming Guide*.

Using CBConnector, a JNDI context can be associated with a LifeCycle Service factory finder so that the associated factory finder is used to resolve EJB home lookup operations from the context. Contexts such as these enable deployers of EJB applications to take advantage of the power of factory finders in a manner that is transparent to clients of these applications.

To resolve EJB home lookups with factory finders, the application deployer can use pre-defined default application contexts associated with the various CBConnector-supplied default factory finders or use the **appbind** tool to create application-specific contexts and associate them with any given factory finder. For more information on each approach, see “Default context-to-finder associations” on page 77 and “Application-specific contexts and the appbind tool” on page 78.

Note: Default application contexts and application-specific contexts eliminate the need for the **ejbbind** tool, which creates a simple 1-to-1 mapping of a JNDI name and an EJB home instance. Clients must use one of the default initial context factories or an application-specific context factory generated by the **appbind** tool.

Default context-to-finder associations

There are several default factory finders built into CBConnector, each of which searches particular scopes of location when finding a factory. When an EJB application is deployed on a CBConnector server, the EJB homes for the application are bound in the LifeCycle repository using the names for the EJB homes as specified by the deployment descriptors contained in the application's EJB jar file. A factory finder can find any EJB home within the scope of its particular search rules.

An EJB client can use a particular built-in CBConnector default factory finder simply by using the initial context factory that corresponds to that factory finder. The initial context returned by the context factory will use its corresponding factory finder to resolve EJB home lookup requests.

Contexts returned by the following initial context factories:

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

resolve EJB home lookup operations with the corresponding factory finders:

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

9. cell / resources / factory-finders / host-scope
10. cell / resources / factory-finders / *server*-server-scope
11. cell / resources / factory-finders / *server*-server-scope-widened

Server-based context factories can only be used by a client that is running as a CBConnector server, in which case, *server* is the name of the CBConnector server.

Default context factories can only be used by client applications that issue fully qualified EJB home lookups. If a client traverses to a subcontext and then performs a partially qualified EJB home lookup, you must run the **appbind** tool to create an application-specific context with home subcontexts and to generate an application-specific initial context factory. For more information, see "Application-specific contexts and the appbind tool".

Application-specific contexts and the appbind tool

If a CBConnector-supplied default factory finder is being used to locate an EJB home, CBConnector supplies a default mapping between application contexts and default factory finders (for more information, see "Default context-to-finder associations" on page 77). For added flexibility, an enterprise bean deployer can create an application-specific context with optional EJB home subcontexts and associate it with any factory finder. The factory finder association can be changed at a later time if desired. To isolate clients from the actual context name, the enterprise bean deployer generates an initial context factory for the application-specific context by using the **appbind** tool.

The **appbind** tool allows deployers to create an application-specific naming context and associate it with a selected factory finder so that lookup operations are resolved with that factory finder. These application-specific contexts are designed to be initial JNDI contexts for EJB clients so that JNDI lookup calls on EJB homes are transparently resolved with the associated factory finder. The **appbind** tool enables users to create, modify, and delete such application-specific contexts. Note that the application's EJB home instances are not actually bound under the application-specific context. Instead,they are bound to the LifeCycle repository. The associated factory finder will resolve the EJB home lookups using the lifecycle rules defined for it.

All application-specific contexts must have one of the following context name stems:

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

depending on whether a scope of host, workgroup, or cell is specified when the context is created.

By default, the factory finder host/resources/factory-finders/host-scope-widened is associated with an application-specific context created with the **appbind** tool. However, the user can specify another factory finder. The factory finder can be one of the other default factory finders, one created by an administrator using System Management, or one created by an application program you write. For more information, see the LifeCycle section in the *Advanced Programming Guide*.

Under an application-specific context, subcontexts for EJB home names optionally can be created. For example, if the name for a home is com/mycom/myapp/MyHome, the subcontext com/mycom/myapp can be created. These subcontexts provide additional transparency to the client. They allow a client to traverse the JNDI name space from the application-specific context down to any subcontext that corresponds to a non-leaf component of an EJB home name. The factory finder associated with the application-specific context is also used to resolve EJB home lookup operations from these subcontexts. The **appbind** tool creates a subcontext for each home name in the deployment descriptors within a specified EJB JAR file.

The **appbind** tool can optionally create a Java source file for an initial context factory for the application-specific context being created. This initial context factory can be used as the initial context factory by clients. The **appbind** tool also allows the user to override the default bootstrap host to use for ORB initialization.

Invoke the **appbind** tool with the following syntax:

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

The context being bound or unbound is specified with the required -name option, where *contextName* is the name of the JNDI application-specific context to bind or unbind. All application context names are relative to one of the following context name stems

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

depending on whether a scope of host, workgroup, or cell was specified. (See the -host, -workgroup, and -cell options below.)

A bind operation is performed unless the -u option is specified, in which case, an unbind operation is performed. If a bind operation is performed on an existing context, the current factory finder association is added or replaced. The context cannot be a child or parent of a context which already has a factory finder association.

The other options do the following:

- -u—This flag is used to perform an unbind operation. An unbind operation unbinds the context specified with the -name option and the -sc option, if specified. If the -sc option is specified, only the subcontexts corresponding to the JNDI home names in the JAR's deployment descriptors are removed. If the -sc option is not used, the context specified by the -name option and all of its subcontexts are unbound. To help keep the name tree manageable, once a context or subcontext is unbound, parent contexts are recursively unbound up to the context name stem (see the -name option above) or until a non-empty parent is encountered.
- -sc—This option is used to specify subcontexts, where file *jarFileName* is the name of an EJB JAR file that contains deployment descriptors with EJB home names. Each of the EJB home names, not including the leaf-name component, is treated as a subcontext name. For example, if the name for a home is com/mycom/myapp/MyHome, the subcontext name is com/mycom/myapp.

  When binding, the subcontext names are created under the application-specific context specified by the -name flag. When unbinding, the contexts which are unbound are restricted to the subcontext names identified by the JAR file. Whether binding or unbinding, other subcontexts are not affected.
- -host, -workgroup, -cell—These flags control the scope of the application context being bound or unbound. Each scope has a corresponding context name stem, as described in the -name flag section above. The -host, -workgroup, and -cell flags specify a scope of host, workgroup, or cell, respectively, for the context. The default scope is host scope. Only one scope can be specified per bind or unbind operation.
- -factoryfinder—This option is used to specify which factory finder to associate with the application-specific context being bound, where *factoryFinderPath* is the name of the factory finder. The default factory finder is host/resources/factory-finders/host-scope-widened.

  This option does not apply to unbind operations.
- -genctxfactory—Typically, when an application-specific context is bound, it is desirable to have an initial context factory for the application-specific context. This option directs the **appbind** tool to create a Java source file for an initial context factory, where *factoryClassName* is the fully-qualified class name of the context factory. All package prefix subdirectories are created, if necessary. If the source file already exists, it is replaced. The file and its

containing subdirectories are created relative to the directory specified with the -o option or, by default, relative to the current directory.

This option does not apply to unbind operations.

- -o—This option is used to specify the target directory for the initial context factory file (see the -genctxfactory option), where *targetDir* is the directory path (not including package prefix directories). The default target directory is the current directory.

  This option does not apply to unbind operations.

  If the -o option is used, use of the -genctxfactory flag is required.
- -boothost—This option is used to override the default host and port used for ORB initialization, where *bootstrapHostUrl* is the URL of the bootstrap host. The bootstrap host URL has the form

  `iiop:// hostName [: portNumber]`

Creating an enterprise bean from an existing CICS or IMS application

You can create an enterprise bean from an existing CICS or IMS application by using the **PAOToEJB** tool. The application must be mapped into a PAO prior to creating the enterprise bean. For more information on creating PAOs, see the Component Broker document entitled *Procedural Application Adaptor Development Guide* and the VisualAge for Java, Enterprise Edition documentation.

The **PAOToEJB** tool runs independently of the other tools described in this chapter. To create an enterprise bean from a PAO class, do the following:

1. Change to the directory where your PAO class file exists.
2. Add the PAO class file's directory, or the JAR file containing the class, to your CLASSPATH environment variable.
3. Invoke the **PAOToEJB** command with the following syntax:

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

   The *ejbName* argument is optional and specifies the enterprise bean's name (for example, Account). If this name is not supplied, the enterprise bean is named by using the short name of the PAO class. The *paoClass* argument is required and specifies the fully qualified Java name of the PAO class without the .class extension; the PAO class is always a subclass of com.ibm.ivj.eab.paa.EntityProceduralAdapterObject. You must also specify one of the following options:

   - -hod —This indicates that the PAO class is for Host On-Demand (HOD). HOD is a browser-based 3270 telnet connection.
   - -eci —This indicates that the PAO class is for External Call Interface (ECI). ECI is a proprietary protocol that provides a remote procedure call (RPC)-like interface into CICS.

- -appc —This indicates that the PAO class is for advanced program-to-program communications (APPC), which is the System Network Architecture (SNA) for LU 6.2 communications.

Note: EJB clients that access entity beans with CMP that use HOD or ECI for CICS or IMS applications must begin a transaction before invoking a method on these entity beans. This is necessary because these types of entity beans must use the TX_MANDATORY transaction attribute.

4. If the *paoClass* is part of a Java package, then you must create the corresponding directory structure and move the generated Java files into this directory.
5. Compile the Java source files of the newly created enterprise bean:

   ```
   javac ejbName*.java
   ```

6. Place the compiled class components of the enterprise bean into a JAR or ZIP file and use the **jetace** tool to create an EJB JAR file for the bean, as described in "Creating a deployment descriptor and an EJB JAR file" on page 33.
7. Deploy the EJB JAR file by using the **cbejb** tool as described in "Deploying an enterprise bean" on page 56.

Creating an enterprise bean that communicates with MQSeries

Component Broker contains tools for developing BOs that send or receive MQSeries messages. It also allows access to MQSeries queues within distributed transactions. The EJB server (CB) builds on this MQSeries support and allows you to create an enterprise bean that wraps an MQSeries-based BO.

The MQSeries EJB support enables an EJB client application to indirectly interact with MQSeries through an EJB client interface. Both the Component Broker support for MQSeries BOs and the EJB support described here require you to modify the DO implementation generated by Object Builder. The main difference between these two supported approaches is that when Component Broker MQSeries-based BOs are built, the MQSeries message content is specified through Object Builder, whereas the EJB support requires the MQSeries message content to be specified in a Java properties file.

For more information on the MQSeries support in Component Broker, see the *MQSeries Application Adaptor Development Guide* document.

The **mqaaejb** tool generates a session bean that wraps a Component Broker BO based on the MQSeries Application Adaptor. The resulting session bean implementation is specific to the EJB server (CB) and is not portable to other

EJB servers. To deploy the generated session bean, use the **cbejb** tool. The **mqaaejb** tool runs independently of other EJB server (CB) tools.

To create a session bean for a particular MQSeries queue, do the following:

1. Create a Java properties file that contains these items:
   - The message type specification—The property name must be messageType, and its value must be either Inbound, Outbound, or InOut. If InOut is chosen, a pair of enterprise beans, instead of a single one, are created to accommodate paired inbound and outbound message queues. Here is an example of this specification:

     ```
     messageType=Inbound
     ```

   - A list of message field specifications—For each message field, the property name is the field name, and the property value is the field type. Here is an example of this specification:

     ```
     bankName=java.lang.String
     accountNumber=int
     ```

     Note: Java class names in the type specifications must be the fully qualified package name.

2. Run the **mqaaejb** command with the following syntax:

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

   The -f and -n options are required. The *propertiesFile* specifies the name of the properties file created in Step 1, and the *baseBeanName* argument specifies the base name of the enterprise bean or beans to be generated. For example, if the base name is Account and the properties file specifies that it is for both an inbound and an outbound message, then the **mqaaejb** command generates session beans, related interfaces, and artifacts with the following names:

   AccountInboundBean

   AccountEJBObject

   AccountInboundEJBHome

   AccountOutboundBean

   AccountOutboundEJBObject

   AccountOutboundEJBHome

   AccountMsgTemplate

   The -p option specifies the package name of the enterprise bean; if not specified, the package name defaults to `mytest.ejb.mqaa`.

Unless the -i option or the -o and -c options are specified, the **mqaaejb** command makes a mark for the the **cbejb** command; later, when the the **cbejb** command is run over the beans, it generates the required backing message BOs for the session beans. If you have already created and tested MQSeries Application Adaptor-based BOs (following the procedure described in the *MQSeries Application Adaptor Development Guide*), you now need only wrap them in session beans. You can specify the names of these BOs and the Copy object to the **mqaaejb** command. The **mqaaejb** command then creates session beans that use the specified BOs. The names of these objects must be fully qualified. For example:

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

You still must specify the base bean name with the -n option independently of the existing BOs. You also must provide a properties file; the message format specified in this file must be consistent with the existing BOs. The correct mapping between the C++ field types in the BOs and the Java types in the properties file can be established by referring to the IDL C++/Java binding documentation.

The following items are generated in the working directory on successful completion of the **mqaaejb** command:

- The Java source files (and the corresponding compiled class files) that compose the enterprise bean in the subdirectory corresponding to the package name.
- A JAR file containing the Java source files and compiled files that compose the enterprise bean.
- An XML file containing the enterprise bean's deployment descriptor.

3. Run the **jetace** tool as follows to generate an EJB JAR file for the enterprise bean:

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

4. Run the **cbejb** tool to deploy the enterprise bean contained in the EJB JAR file. For more information, see "Deploying an enterprise bean" on page 56. When the **cbejb** command is complete, unless you are using existing BOs, you possibly need to follow the steps in the *MQSeries Application Adaptor Development Guide* to modify the DO implantation.

Restrictions in the EJB server (CB) environment

The following restrictions apply when developing enterprise beans for the EJB server (CB) environment:

- Unqualified interface and exception names cannot be duplicated in enterprise beans. For example, the com.ibm.ejs.doc.account.Account interface must not be reused in a package named

com.ibm.ejs.doc.bank.Account. This restriction is necessary because the EJB server (CB) tools generate enterprise bean support files that use the unqualified name only.

- Container-managed fields in entity beans must be valid for use in CORBA IDL files. Specifically, the variable names must use ISO Latin-1 characters; they must *not* begin with an underscore character (_), they must *not* contain the dollar character ($), and they must *not* be CORBA keywords. Variables that have the same name but different cases are not allowed. (For example, you cannot use the following variables in the same class: *accountId* and *AccountId*. For more information on CORBA IDL, consult a CORBA programming guide.

  Also, container-managed fields in entity beans must be valid Java types, but they *cannot* be of type ejb.javax.Handle or an array of type EJBObject or EJBHome. Furthermore, container-managed fields of type String (or arrays of type String) *cannot* be set to null at run time because these types map to CORBA IDL type string or wstring, which are prohibited by CORBA from having null values.
- The use of underscores (_) in the names of user-defined interfaces and exception classes is discouraged.
- Method names in the remote interface must *not* match method names in the Component Broker Managed Object Framework (that is, methods in the IManagedServer::IManagedObjectWithCachedDataObject, CosStream::Streamable, CosLifeCycle::LifeCycleObject, and CosObjectIdentity::IdentifiableObject interfaces). For more information on the Managed Object Framework, see the Component Broker *Programming Guide*. In addition, do not use underscores (_) at the end of property or method names; this restriction prevents name collision with queryable attributes in BO interfaces that correspond to container-managed fields.
- 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.
- The javax.ejb.SessionSynchronization interface is *not* supported.
- Entity beans with BMP that use Java Database Connectivity (JDBC) to access a database cannot participate in distributed transactions because the environment does not support XA-enabled JDBC. In addition, a BMP entity bean that uses JDBC to access a DB2 database must not be run in the same server process as a CMP entity bean that uses DB2 or in the same server process as an ordinary CB BO that uses DB2. Similarly, a BMP entity bean that uses JDBC to access an Oracle database must not be run in the same server process as a CMP entity bean (or other CB BO) that uses Oracle.

- The variables of the primary key class of a BMP entity bean must be public.
- The *run-as identity* and *access control* deployment descriptor attributes are not used.
- The remove method inherited by an enterprise bean's remote interface (from the javax.ejb.EJBObject interface) does not throw the javax.ejb.RemoveException exception, even if the enterprise bean's corresponding ejbRemove() method throws this exception. This restriction is necessary because of the name conflict between the remove method and the CORBA CosLifeCycle::LifeCycleObject::remove method, which is inherited by all Component Broker managed objects.
- Single-threaded access to enterprise beans is enforced only if a bean's transaction attribute is set to either TX_NOT_SUPPORTED or TX_BEAN_MANAGED. For other enterprise beans, access from different transactions is serialized, but serialized access from different threads running under the same transaction is not enforced. Illegal callbacks for enterprise beans deployed with the TX_NOT_SUPPORTED or TX_BEAN_MANAGED transaction attribute result in a java.rmi.RemoteException exception being thrown to the EJB client.
- The session bean timeout attribute is *not* supported.
- The transaction attribute can be set only for the bean as a whole; the transaction attribute cannot be set on individual methods in a bean.
- If a stateful session bean has the TX_BEAN_MANAGED transaction attribute value, a method that begins a transaction must also complete that transaction (commit or roll back the transaction). In other words, a transaction cannot span multiple methods in a stateful session bean when used in the EJB server (CB) environment.
- The TX_MANDATORY transaction attribute value must be used in entity beans with container-managed persistence (CMP) that use HOD or ECI to access CICS or IMS applications. As a result, EJB clients that access these entity beans must do so within a client-initiated one-phase commit transaction (CB session service).
- The TX_NOT_SUPPORTED transaction attribute value is not supported for entity beans with CMP, because these beans must be accessed within a transaction.
- The TX_REQUIRES_NEW transaction attribute is *not* supported.
- The TX_SUPPORTS transaction attribute *can* be used in entity beans with CMP; however, EJB clients that access these beans must do so within a client-initiated transaction.
- The transaction isolation level attribute is *not* supported.
- When using the com.ibm.ejb.cb.runtime.CBCtxFactory context factory, any of the default initial context factories (see "Default context-to-finder associations" on page 77), or an application-specific initial context factory generated by the **appbind** tool (see "Application-specific contexts and the

appbind tool" on page 78), the javax.naming.Context.list and javax.naming.Context.listBindings methods can return no more than 1000 elements in the javax.naming.NamingEnumeration object.

- C++ CORBA-based EJB clients are restricted to invoking methods that do *not* use parameters that are arrays or that are of the java.io.Serializable type or the java.lang.String type. This restriction effectively prohibits these EJB clients from accessing entity beans directly because primary key classes must be serializable. The String and array types in the remote or home interface are mapped to IDL value types to allow null values to be passed between a Java EJB client and an enterprise bean. CORBA C++ EJB clients cannot invoke the javax.ejb.EJBHome.remove and javax.ejb.EJBObject.getHandle methods because these methods contain Serializable parameters. EJB clients cannot be built with Microsoft Visual C++®.

Chapter 5. 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 157.

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 225.

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.

Note: In the EJB server (CB) environment, do not duplicate unqualified interface and exception names in enterprise beans. For example, the com.ibm.ejs.doc.account.Account interface must not be reused in a package named com.ibm.ejs.doc.bank.Account. This restriction is necessary because the EJB server (CB) tools generate enterprise bean support files that use the unqualified name only.

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 157.

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 98.
- The enterprise bean's remote interface. For more information, see "Writing the remote interface (entity with CMP)" on page 101.
- The enterprise bean's primary key class. For more information, see "Writing the primary key class (entity with CMP)" on page 102.

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 96.
- It must define instance variables that correspond to persistent data associated with the enterprise bean. For more information, see "Defining variables" on page 91.
- 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 93.
- 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 94.

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

Figure 18. 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.

Note: In the EJB server (CB) environment, container-managed fields in entity beans must be valid for use in CORBA IDL files. Specifically, the variable names must use ISO Latin-1 characters, they must *not* begin with an underscore character (_), they must *not* contain the dollar character ($), and they must *not* be CORBA keywords. Variables that have the same name but different cases are not allowed. (For example,

you cannot use the following variables in the same class: *accountId* and *AccountId*. For more information on CORBA IDL, consult a CORBA programming guide.

Also, container-managed fields in entity beans must be valid Java types, but they *cannot* be of type ejb.javax.Handle or an array of type EJBObject or EJBHome. Furthermore, container-managed fields of type String (or arrays of type String) *cannot* be set to null at run time because these types map to CORBA IDL type string or wstring, which are prohibited by CORBA from having null values.

The AccountBean class contains three container-managed fields (shown in Figure 19):

- *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 19. 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 94).

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 102, 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 20 on page 94 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 25 on page 102. 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: In the EJB server (CB) environment, use of underscores (_) in the names of user-defined interfaces and exception classes is discouraged.

```
...
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 20. Code example: The business methods of the AccountBean class

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 157.

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

- It must be public and return void.
- 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 117.
- 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 java.rmi.RemoteException exception and the javax.ejb.CreateException exception.

Figure 21 on page 96 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 void ejbCreate(AccountKey key) {
          ejbCreate(key, 1, 0.0f);
     }
     ...
     public void ejbCreate(AccountKey key, int type, float initialBalance)
     throws RemoteException {
          accountId = key.accountId;
          type = type;
          balance = initialBalance;
     }
     ...
     public void ejbPostCreate(AccountKey key) throws RemoteException {
          ejbPostCreate(key, 1, 0);
     }
     ...
     public void ejbPostCreate(AccountKey key, int type, float initialBalance) { }
     ...
}
```

Figure 21. 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 26.) All of these methods must be public and return void, and they can throw the java.rmi.RemoteException exception.

- 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 9. More-advanced programming concepts for enterprise beans" on page 157.

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 22 on page 98, 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 RemoteException { }
     ...
     public void ejbLoad () throws RemoteException { }
     ...
     public void ejbPassivate() throws RemoteException { }
     ...
     public void ejbRemove() throws RemoteException { }
     ...
     public void ejbStore () throws RemoteException { }
     ...
     public void setEntityContext(EntityContext ctx) throws RemoteException {
          entityContext = ctx;
     }
     ...
     public void unsetEntityContext() throws RemoteException {
          entityContext = null;
     }
}
```

Figure 22. 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 116 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 100.
- 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 117. In addition, each method’s throws clause must include the java.rmi.RemoteException exception class.

Figure 23 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 23. 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 23 on page 99).
- 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 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 or the java.util.Enumeration interface (when a finder method can return more than one EJB object).
- 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 24. 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 24. 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 (AE)" on page 33 or "Creating finder logic in the EJB server (CB)" on page 53.

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 116 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 117.
- Each method's throws clause must include the java.rmi.RemoteException exception class.

Note: In the EJB server (CB) environment, do not use method names in the remote interface that match method names in the Component Broker Managed Object Framework (that is, methods in the IManagedServer::IManagedObjectWithCachedDataObject, CosStream::Streamable, CosLifeCycle::LifeCycleObject, and CosObjectIdentity::IdentifiableObject interfaces). For more information on the Managed Object Framework, see the Component Broker *Programming Guide*. In addition, do not use underscores (_) at the end of property or method names; this restriction prevents name collision with queryable attributes in business object interfaces that correspond to container-managed fields.

Figure 25 on page 102 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 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 25. 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.

The primary key class is used to encapsulate 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.

A primary key class is used to create and manage the primary key for an EJB object. The primary key class must meet the following requirements:

- It must be public and it must be serializable. For more information, see "The java.io.Serializable and java.rmi.Remote interfaces" on page 117.
- 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: For the EJB server (AE) environment, 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 26 shows the primary key class for the Account enterprise bean. In effect, this class acts as a wrapper around the primitive long variable *accountId*. The hashCode method for the AccountKey class simply invokes the corresponding hashCode method in the java.lang.Long class after creating a temporary Long object by using the value of the *accountId* variable. In addition to the default constructor, the AccountKey class also defines a constructor that sets the value of the primary key variable to a specified long.

More complicated enterprise beans are likely to have composite primary keys, with multiple instance variables representing the primary key.

```
...
import java.io.*;
public class AccountKey implements Serializable {
     public long accountId;
     ...
     public AccountKey() {
          super();
     }
     ...
     public AccountKey(long accountId) {
          this.accountId = accountId;
     }
     ...
     // EJB server (AE)-specific method
     public boolean equals(Object o) {
          if (o instanceof AccountKey) {
               AccountKey otherKey = (AccountKey) o;
               return (((accountId == otherKey.accountId)));
          }
          else {
               return false;
          }
     }
     ...
     // EJB server (AE)-specific method
     public int hashCode() {
          return ((new Long(accountId).hashCode()));
     }
}
```

Figure 26. Code example: The AccountKey primary key class

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 113.
- The enterprise bean's remote interface. For more information, see "Writing the remote interface (session)" on page 115.

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 106.
- 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 108.
- 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 112.

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.

Note: The SessionSynchronization interface is *not* supported in the EJB server (CB) environment.

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 27 on page 106 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.ResurceBundle;
import java.util.ListResourceBundle;
import javax.ejb.*;
import java.lang.*;
import javax.naming.*;
import com.ibm.ejs.doc.account.*;
...
public class TransferBean implements SessionBean {
     ...
     private SessionContext mySessionCtx = null;
     private InitialContext initialContext = null;
     private AccountHome accountHome = null;
     private Account fromAccount = null;
     private Account toAccount = null;
     ...
     public void ejbActivate() throws RemoteException { }
     ...
     public void ejbCreate() throws RemoteException {
          ...
     }
     ...
     public void ejbPassivate() throws RemoteException { }
     ...
     public void ejbRemove() throws RemoteException { }
     ...
     public float getBalance(long acctId) throws FinderException,
          RemoteException {
          ...
     }
     ...
     public void setSessionContext(javax.ejb.SessionContext ctx)
          throws java.rmi.RemoteException {
          ...
     }
     ...
     public void transferFunds(long fromAcctId, long toAcctId, float amount)
          throws java.rmi.RemoteException {
          ...
     }
}
```

Figure 27. 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 28 on page 108 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 98. Obtaining the EJB home object is discussed in "Implementing the ejbCreate methods" on page 108.

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

Figure 28. 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. A stateless session bean must have only one ejbCreate method, which must return void and contain no arguments; a stateful session bean can have multiple ejbCreate methods.

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

Each ejbCreate method must meet the following requirements:

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

Figure 29 on page 110 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 131.

```
...
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 RemoteException {
          // 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 29. 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

("Setting environment variables for an enterprise bean" on page 44 shows an example of the **jetace** page required to set these variables.)

Figure 30 on page 111 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 30. Code example: The getProviderURL method

Getting the reference to the home object: After constructing the InitialContext object (*initialContext*), 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 131.

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

Figure 31. Code example: Creating the AccountHome object in the ejbCreate method of the TransferBean class

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, return void, and throw the java.rmi.RemoteException exception.

- 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 181.

  Note: In the EJB server (CB) environment, the isCallerInRole and getCallerIdentity methods inherited from the javax.ejb.EJBContext interface are not supported.

As shown in Figure 32, 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 RemoteException { }
     ...
     public void ejbPassivate() throws RemoteException { }
     ...
     public void ejbRemove() throws RemoteException { }
     ...
     public void setSessionContext(SessionContext ctx) throws RemoteException {
          mySessionCtx = ctx;
     }
     ...
}
```

Figure 32. 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 116 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 108. 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 117. In addition, each method's throws clause must include the java.rmi.RemoteException exception class.

Figure 33 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 33. 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 116 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 117.
- Each method's throws clause must include the java.rmi.RemoteException exception class.

Figure 34 on page 116 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 34. 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 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 (AE) enforces single-threaded access to all enterprise beans. Illegal callbacks result in a java.rmi.RemoteException exception being thrown to the EJB client.

The EJB server (CB) enforces single-threaded access to enterprise beans only if their transaction attribute is set to either TX_NOT_SUPPORTED or

TX_BEAN_MANAGED. For other enterprise beans, access from different transactions is serialized, but serialized access from different threads running under the same transaction is not enforced. For enterprise beans deployed with the transaction attribute value of TX_NOT_SUPPORTED or TX_BEAN_MANAGED, illegal callbacks result in a RemoteException exception being thrown to the EJB client.

Packaging enterprise beans

There are three tasks involved in packaging an enterprise bean:

- Making the components of the bean part of the same Java package. For more information, see "Making bean components part of a Java package".
- Creating a deployment descriptor for the bean. For more information, see "Creating the deployment descriptor file".
- Creating an EJB JAR file. For more information, see "Creating an EJB JAR file" on page 119.

If you develop enterprise beans in an IDE, these packaging 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 package beans in the EJB server (AE) programming environment, see "Chapter 3. Tools for developing and deploying enterprise beans in the EJB server (AE) environment" on page 29.
- For more information on the tools used to package beans in the EJB server (CB) programming environment, see "Chapter 4. Tools for developing and deploying enterprise beans in the EJB server (CB) environment" on page 49.

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 the deployment descriptor file

The deployment descriptor file provides instructions for the container on how to handle a particular bean. A standard deployment descriptor must support the attributes described in "The deployment descriptor" on page 19. The deployment descriptor is stored in a special file that contains a serialized

instance of a javax.ejb.deployment.EntityDescriptor object for an entity bean or a javax.ejb.deployment.SessionDescriptor object for a session bean.

To create a deployment descriptor, you can use either the **jetace** tool provided with WebSphere Application Server or the mechanism built into an integrated development environment (IDE) that supports enterprise bean development (for example, VisualAge for Java). You can also create a deployment descriptor programmatically, though this approach is not discussed in this documentation.

Creating an EJB JAR file

A JAR file for an enterprise bean, known as an EJB JAR file, must contain the following components:

- 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.
- The deployment descriptor file for the enterprise bean.
- The manifest file that describes the content of the EJB JAR file.

Manifest files are organized into sections that are separated by blank lines; each section corresponds to a file stored in the JAR file. The manifest file must be named META-INF/MANIFEST.MF. Each section contains one or more tag-value pairs with the syntax *tag*:*value*. The section corresponding to the deployment descriptor file for each enterprise bean in an EJB JAR file must contain the following headers:

```
Name: deploymentDescriptorFile
Enterprise-Bean: True
```

Figure 35 on page 120 shows the first two sections of the manifest file for the Account bean's EJB JAR file. Although not shown in the example, the manifest file contains a section for each of the class files—Account.class, AccountBean.class, AccountKey.class, and AccountBeanFinderHelper.class—and any other files in the EJB JAR file. However, only the section associated with the deployment descriptor file contains the enterprise bean-specific headers.

```
Manifest-Version: 1.0
Name: com/ibm/ejs/doc/account/Account.ser
Enterprise-Bean: true
Digest-Algorithms: SHA MD5
SHA-Digest: xhCUGthNU+Kds5X3xol5q7Mz9JI=
MD5-Digest: t40IcinyAjss0hrM1dpY0A==
Name: com/ibm/ejs/doc/account/AccountHome.class
Digest-Algorithms: SHA MD5
SHA-Digest: 02pfJv1buUu0FqpFwwERUstsNIg=
MD5-Digest: a8auOXob9ryPgbcnpFvzpQ==
...
```

Figure 35. Code example: Fragment of the manifest file for the Account EJB JAR file

To build an EJB JAR file, you can use either the **jetace** tool provided with WebSphere Application Server or the mechanism built into an integrated development environment (IDE) that supports enterprise bean development (for example, VisualAge for Java). Both of these tools automatically can create the deployment descriptor, appropriately format a manifest file, and create an EJB JAR file to contain one or more enterprise beans.

Chapter 6. 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* TX_BEAN_MANAGED and set the transaction isolation level attribute. To enable bean-managed transactions, you must set the transaction attribute to TX_BEAN_MANAGED 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 181.

  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 139.
- For security, only 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 126.

These attributes, like the other deployment descriptor attributes, are set by using one of the tools available with either the EJB server (AE) or the EJB server (CB). For more information, see "Chapter 3. Tools for developing and deploying enterprise beans in the EJB server (AE) environment" on page 29 or "Chapter 4. Tools for developing and deploying enterprise beans in the EJB server (CB) environment" on page 49.

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 each enterprise bean, 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 181.

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 139.

The EJB specification defines two attributes in a standard deployment descriptor that 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. “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. “Setting the transaction isolation level attribute” on page 124 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 can be set for the bean as a whole and for individual methods in a bean.

Note: The EJB server (CB) does not support the setting of the transaction attribute for individual enterprise bean methods; the transaction attribute can be set only for the entire bean.

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

TX_BEAN_MANAGED

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 181.

In the EJB server (CB) environment, if a stateful session bean has this attribute value, a method that begins a transaction must also complete that transaction (commit or roll back the transaction). In other words, a transaction cannot span multiple methods in a stateful session bean when used in the EJB server (CB) environment.

TX_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 TX_BEAN_MANAGED value or use the TX_REQUIRED or TX_REQUIRES_NEW 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 139.

In the EJB server (CB) environment, this attribute value must be used in entity beans with container-managed persistence (CMP) that use Host On-Demand (HOD) or the External Call Interface (ECI) to access CICS or IMS applications.

TX_REQUIRED

Directs the container to invoke the bean method within a transaction context. If a client invokes a bean method from within a transaction context, the container invokes the bean method within the client transaction context. If a client invokes a bean method 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.

TX_REQUIRES_NEW

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

The EJB server (CB) does *not* support this attribute value.

TX_SUPPORTS

Directs the container to invoke the bean method within a transaction context if the client invokes the bean method within a transaction. If the client invokes the bean method without a transaction context, 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.

In the EJB server (CB) environment, entity beans with CMP must be accessed within a transaction. If an entity bean with CMP uses this

transaction attribute, the EJB client must initiate a transaction before invoking a method on the entity bean.

TX_NOT_SUPPORTED

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

In the EJB server (CB) environment, entity beans with CMP must be accessed within a transaction. Therefore, this attribute value is not supported in entity beans with CMP in the EJB server (CB) environment.

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

Transaction attribute	Client transaction context	Bean transaction context
TX_MANDATORY	No transaction	Not allowed
	Client transaction	Client transaction
TX_NOT_SUPPORTED	No transaction	No transaction
	Client transaction	No transaction
TX_REQUIRES_NEW	No transaction	New transaction
	Client transaction	New transaction
TX_REQUIRED	No transaction	New transaction
	Client transaction	Client transaction
TX_SUPPORTS	No transaction	No transaction
	Client transaction	Client transaction

Setting the transaction isolation level attribute

Note: The EJB server (CB) does not support the transaction isolation level attribute.

The transaction isolation level determines how strongly one transaction is isolated from another. This attribute can be set for the enterprise bean as a whole and 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:

TRANSACTION_SERIALIZABLE
This level prohibits all of the following types of reads:

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

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

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

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

These isolation levels correspond to the isolation levels defined in the Java Database Connectivity (JDBC) java.sql.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 TRANSACTION_SERIALIZABLE value protects data by eliminating concurrent access to it. Conversely, the TRANSACTION_READ_UNCOMMITTED 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.

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:

CLIENT_IDENTITY
: The security service makes no changes to the principal's credential properties.

SYSTEM_IDENTITY
: The security service alters the principal's credential properties to match the credential properties associated with the EJB server.

SPECIFIED_IDENTITY
: The security service attempts to match the principal's credential properties with the identity of any application with which the enterprise bean is associated. If successful, the security service alters the principal's credential properties to match the credential properties of the application.

The *run-as identity* and *access control* attributes are not used in the EJB server environments.

Chapter 7. 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 8. Developing servlets that use enterprise beans” on page 145.
- 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 5. Developing enterprise beans” on page 89.

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 225.

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 131.
- Handle invalid EJB objects when using session beans. For more information, see “Handling an invalid EJB object for a session bean” on page 137.
- 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 139.

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 139.

Note: In the EJB server (CB) environment, an enterprise bean can also be accessed by a Java applet, an ActiveX client, a CORBA-based Java client, and to a limited degree, by a C++ CORBA client. The Travel example briefly described in “Information about the examples described in the documentation” on page 225 illustrates some of these types of clients. “More information on EJB clients specific to the EJB server (CB)” on page 141 provides additional information about EJB clients that use ActiveX and CORBA-based Java and C++.

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 (AE) environment” on page 32 or “Setting the CLASSPATH environment variable in the EJB server (CB) environment” on page 51. You can install needed files on your client machine by doing a WebSphere Application Server installation on the machine, selecting 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 1.2.x install.

Figure 36 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 36. 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" on page 132.
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 136.

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 155 , 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 108. 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:///`. In the EJB server (AE), 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.

- In the EJB server (AE) environment, this property must be set to `com.ibm.ejs.ns.jndi.CNInitialContextFactory`.
- In the EJB server (CB) environment, this property must be set to `com.ibm.ejb.cb.runtime.CBCtxFactory`. When using this context factory, the javax.naming.Context.list and

javax.naming.Context.listBindings methods can return no more than 1000 elements in the javax.naming.NamingEnumeration object.

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 134.

Creating an InitialContext object

Figure 37 on page 134 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 225).

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 137. 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 37. 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 38 on page 135. 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:

- When performing a JNDI lookup on an enterprise bean deployed in an EJB server (AE; CB on AIX, Windows NT, or Solaris platforms), only the JNDI name specified in the deployment descriptor is used.

- When performing a JNDI lookup on an enterprise bean deployed in an EJB server (CB on platforms other than AIX, Windows NT, and Solaris), the JNDI home name passed to the lookup method is the JNDI name specified in the enterprise bean's deployment descriptor with a CB-specific prefix attached. The content of this prefix depends on where in the Component Broker namespace the system administrator bound the EJB home (by using the **ejbbind** tool).

  If the system administrator binds the EJB home in the host name tree of a specific bootstrap host, then the JNDI name prefix will be `host/resources/factories/EJBHomes`. If the system administrator binds the EJB home in a workgroup name tree, then the JNDI name prefix will be `workgroup/resources/factories/EJBHomes`, and the EJB client must belong to the same preferred workgroup. If the system administrator binds the EJB home in the cell name tree, then the JNDI name prefix is `cell/resources/factories/EJBHomes`.

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

Figure 38. Code example: Creating the EJBHome object

Migration considerations for creating an EJB home object

If you are migrating existing applications from WebSphere Application Server 2.x to 3.x, you will need to make a change in how home objects are narrowed. In the 2.x environment, each class has its own Helper whose name follows the ClassNameHelper format. In the 3.x environment, however, the PortableRemoteObject class must be used for narrowing all classes, instead of a specific Helper for each class.

For example, a 2.x version of the example TransferApplication would contain lines similar to those in Figure 39, where the narrowing is done using the TransferHomeHelper class that is specific for TransferHome.

```
. . .
java.lang.Object homeObject = ivjInitContext.lookup(
     bundle.getString("transferName"));
transferHome = TransferHomeHelper.narrow(
     (org.omg.CORBA.Object)homeObject);
. . .
```

Figure 39. Code example: Narrowing the home object in WebSphere Application Server 2.x

In order to migrate to a 3.x version of the example TransferApplication, change the second line to use the narrow method of the PortableRemoteObject class, supplying as parameters the object to be narrowed and its class. This is shown in the code sample in Figure 40. (The complete example code for creating an EJB home object is discussed in "Creating EJB home object" on page 134.)

```
. . .
java.lang.Object homeObject = ivjInitContext.lookup(
     bundle.getString("transferName"));
transferHome = (TransferHome)javax.rmi.PortableRemoteObject.narrow(
     (org.omg.CORBA.Object)homeObject, TransferHome.class);
. . .
```

Figure 40. Code example: Narrowing the home object in WebSphere Application Server 3.x

Creating an EJB object

After the EJB home object is created, it is used to create the EJB object. Figure 41 on page 137 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 41. 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 131. For the example TransferApplication client, this code is contained in the createTransfer method.

Figure 42 on page 138 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 42. 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 session EJB object, the EJB client must remove that object. Removing unneeded session EJB objects can prevent memory leaks in the EJB server. 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 131, the TransferApplication contains only one reference to a Transfer EJB object that is created when the application is initialized.

Figure 43 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 43. 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 TX_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 44 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 `jta/usertransaction`.

```
...
javax.transaction.*;
...
// Use JNDI to locate the UserTransaction object
Context initialContext = new InitialContext();
UserTransaction tranContext = (
     UserTransaction)initialContext.lookup("jta/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 44. Code example: Managing transactions in an EJB client

More information on EJB clients specific to the EJB server (CB)

When developing EJB clients for the EJB server (CB) environment, you can develop the following types of clients:

- Microsoft ActiveX clients. For some general information, see “EJB clients that use ActiveX” on page 142.
- CORBA-based Java and C++ clients. For some general information, see “C++ and Java EJB clients that use a CORBA interface” on page 142.

- Clients using the Component Broker Session Service. For some general information, see “Clients using the Component Broker Session Service” on page 143.

For more information on developing these types of clients, see the IBM Redbook entitled *IBM Component Broker Connector Overview*, form number SG24-2022-02.

EJB clients that use ActiveX

If you write your EJB client as a component that adheres to the JavaBeans™ Specification, you can use the JavaBeans bridge to run the EJB client as an ActiveX control. An EJB client of this type must provide a no-argument constructor, it must implement the java.io.Serializable interface, and it must have a readObject and a writeObject method, if applicable.

If your EJB client is also an applet, you must not perform your JNDI initialization as part of object construction. Rather, perform JNDI initialization in the applet’s start method. The JavaBeans bridge must create an instance of your EJB client so that it can introspect it and make the necessary stubs to create the ActiveX proxy for it. You must delay the JNDI connections until the user can specify the necessary properties by way of the ActiveX property sheet.

C++ and Java EJB clients that use a CORBA interface

Typically, Java EJB clients are written to use the enterprise bean’s RMI interface. However, you can also write a Java-based EJB client that uses the enterprise bean’s CORBA interface. To generate Java/CORBA bindings for an enterprise bean deployed in a Component Broker server, use the -idl option of the Java **rmic** command to generate an interface definition language (IDL) file from the enterprise bean’s remote and home interfaces.

Then, use the IDL file as input to Component Broker’s IDL compiler (com.ibm.idltoJava.Compile). You must not generate Java/CORBA bindings from the IDL file generated by the **cbejb** deployment tool, because this IDL file incorporates special interface-name and method-signature mangling that allows the IDL file to be used to generate C++ bindings for enterprise beans without requiring C++ implementations of the Java Serializable types used by the bean.

You can develop EJB clients that use C++ CORBA; however, these clients are restricted to invoking methods that do *not* use parameters that are arrays or that are of the java.io.Serializable type or the java.lang.String type. This restriction effectively prohibits C++ EJB clients from accessing entity beans directly because primary key classes must be serializable. The String and array types in the remote or home interface are mapped to IDL value types to allow null values to be passed between a Java EJB client and an enterprise bean. CORBA C++ EJB clients cannot invoke the javax.ejb.EJBHome.remove

and javax.ejb.EJBObject.getHandle methods because these methods contain Serializable parameters. EJB clients of this type cannot be built with Microsoft Visual C++.

To generate the CORBA C++ bindings for an enterprise bean, run the Component Broker **idlc** tool on the IDL file generated from the enterprise bean by the **cbejb** tool. Do not generate CORBA C++ bindings by using the IDL file generated by the Java **rmic** command, because this IDL file contains nested modules that can be re-opened, and these are not supported by the Component Broker C++ bindings due to the lack of namespace support in the C++ compiler.

Clients using the Component Broker Session Service

In addition to the Transaction Service, Component Broker also provides a Session Service for the Procedural Application Adaptor (PAA) that enables the use of backend systems such as CICS and IMS. Since the JTA does not have a Session Service, it is not possible to use JNDI to look up a handle to the service in an EJB client. In this case, the EJB client must act as an ordinary CB Java client.

The normal lookup procedure for a CB Java client is to use the CORBA resolve_initial_references method. In this case, the CORBA object to look up is named SessionCurrent.

Before you can call the resolve_initial_references method, the ORB needs to be properly initialized for the CB runtime environment. The initialization method depends on whether or not you are using VisualAge for Java access beans in the CB environment. If you are using access beans, then the ORB must be manually initialized. ORB initialization in access beans is done in a ″lazy″ fashion. That is, initialization is not done until the first remote method is invoked. However, because a session must be started before that method is called, the ORB initialization must be done manually. The example code in Figure 45 shows this initialization.

```
String[] CBargs = null;
CBargs = new String[6];
CBargs[0] = "-ORBBootstrapHost";
// substitute your bootstrap host name
CBargs[1] = "cbs3.rchland.ibm.com";
CBargs[2] = "-ORBBootstrapPort";
CBargs[3] = "900";
CBargs[4] = "-ORBClass";
CBargs[5] = "com.ibm.CORBA.iiop.ORB";
com.ibm.CBCUtil.CBSeriesGlobal.Initialize(CBargs);
```

Figure 45. Code example: Initializing the ORB (if using access beans)

If you are not using access beans, initialization code is not necessary. The ORB is properly initialized during the creation of the InitialContext object with the appropriate properties. For example, your client code should already contain lines similar to those in Figure 46. This code is used to find the service, look up the home object, narrow the home object, and create the proxy object (tasks automatically done if an access bean is being used).

```
Properties properties = new Properties();
properties.put(javax.naming.Context.PROVIDER_URL, "iiop:///");
// CB Factory Name
properties.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
"com.ibm.ejb.cb.runtime.CBCtxFactory");
Context ctx = new InitialContext(properties);
```

Figure 46. Code example: Creating the InitialContext object (if not using access beans)

After the ORB is initialized (either automatically or manually), you must use CB-specific APIs for creating and using the sessionCurrent object. You must include code similar to the example code in Figure 47.

```
org.omg.CORBA.Object orbCurrent = null;
com.ibm.ISessions.Current sessionCurrent = null;
...
orbCurrent = com.ibm.CBCUtil.CBSeriesGlobal.orb().resolve_initial_references(
"ISessions::Current");
sessionCurrent = com.ibm.ISessions.CurrentHelper.narrow(orbCurrent);
sessionCurrent.beginSession("myApp");
...
// commit
sessionCurrent.endSession(com.ibm.ISessions.EndMode.EndModeCheckPoint, true);
```

Figure 47. Code example: Creating and using the sessionCurrent object

For more information on using the resolve_initial_references method, see the Component Broker *Programming Guide*.

Chapter 8. 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. In addition, the IBM HTTP Web server is available with both the Advanced Application Server and the Enterprise Application Server. For more information, consult the *Getting Started with Advanced Edition* document.

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 225.

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 48 on page 146 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 48. 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 49 on page 147 shows what create.html looks like on a browser.

Figure 49. 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 50 on page 148 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 50. Code example: The CreateAccount class

The servlet's instance variables

Figure 51 on page 149 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 131.

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 51.

```
...
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 51. 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 52 on page 150. 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 131.

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

Figure 52. Code example: The init method of the CreateAccount servlet

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 53 on page 152 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 53. 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 54 on page 153. 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 54. 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 55 on page 154 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 55. 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 56 on page 155.

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 56. 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 9. 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 89.

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)".
- The enterprise bean's home interface. For more information, see "Writing the home interface (entity with BMP)" on page 168.
- The enterprise bean's remote interface. For more information, see "Writing the remote interface (entity with BMP)" on page 170.

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 172.

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 166.
- It must define instance variables that correspond to persistent data associated with the enterprise bean. For more information, see "Defining instance variables" on page 159.
- 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 161.
- 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 173.
- 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 161.
- 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 163.

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 57 on page 159 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 57. 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 58. 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 connections in the EJB server (CB) environment" on page 174 or "Managing database connections in the EJB server (AE) environment" on page 177.

For more information on how to set an enterprise bean's environment variables, refer to "Setting environment variables for an enterprise bean" on page 44.

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 20 on page 94.

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 173.

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 59 on page 163 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 59. 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.

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 173.

Figure 60 on page 165 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 60 on page 165. 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 173.

```
public AccountBMKey ejbFindByPrimaryKey (AccountBMKey key) throws FinderException,
     RemoteException {
     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 60. Code example: The ejbFindByPrimaryKey method of the AccountBMBean class

Figure 61 on page 166 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 173.)

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 RemoteException {
     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 61. 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 116 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 170.
- 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 117. In addition, each method's throws clause must include the java.rmi.RemoteException exception class.

Figure 62 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 62. 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 23 on page 99).
- 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 or the java.util.Enumeration interface (when a finder method can return more than one EJB object).
- 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 62 on page 169. 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 163.

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 116 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 117.
- Each method's throws clause must include the java.rmi.RemoteException exception class.

Figure 63 on page 172 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 63. 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 CMP, you must write a distinct primary key class. In an entity bean with BMP, 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 117.

The example AccountBM bean uses a primary key class that is identical to the AccountKey class contained in the Account bean shown in Figure 26 on page 103, with the exception that the key class is named AccountBMKey.

Note: For the EJB server (AE) environment, the primary key class of an entity bean with BMP does not need to implement the hashCode and equals method. However, the primary key class can contain these methods if you require their functionality. 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 code required to manage database connections varies across the EJB server implementations:

- The EJB server (CB) uses JDBC 1.0 to manage database connections directly. For more information on the EJB server (CB), see "Managing connections in the EJB server (CB) environment" on page 174.
- The EJB server (AE) 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 on the EJB server (AE), see "Managing database connections in the EJB server (AE) environment" on page 177.

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. If you are using the EJB server (CB), which does not contain a connection manager, this approach is probably the best one.
- 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. If you are using the EJB server (AE), which contains a connection manager, 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. These methods must be coded differently based on which EJB server environment you use:

- The code required to make the AccountBM bean work with the connection manager in the EJB server (CB) is shown in "Managing connections in the EJB server (CB) environment".
- The code required to make the AccountBM bean work with the connection manager in the EJB server (AE) is shown in "Managing database connections in the EJB server (AE) environment" on page 177.

The code required to manipulate data in a database is identical for both EJB server environments. For more information, see "Manipulating data in a database" on page 180.

Managing connections in the EJB server (CB) environment

In the EJB server (CB) environment, the standard java.sql.DriverManager interface is used to load and register a database driver and to get and release connections to the database.

Loading and registering a data source

The example AccountBM bean uses an IBM DB2 relational database to store its persistent data. To interact with DB2, the example bean must load one of the available JDBC drivers. Figure 64 on page 175 shows the code required to load the driver class. The value of the *driverName* variable is obtained by the getEnvProps method, which accesses a corresponding environment variable in the deployed enterprise bean.

The Class.forName method loads and registers the driver class. The AccountBM bean loads the driver in its setEntityContext method, ensuring that every instance of the bean has immediate access to the driver after creating the bean instance and establishing the bean's context.

Note: In the EJB server (CB) environment, entity beans with BMP that use JDBC to access a database cannot participate in distributed transactions because the environment does not support XA-enabled JDBC. In addition, a BMP entity bean that uses JDBC to access a DB2 database must not be run in the same server process as a CMP entity bean that uses DB2 or in the same server process as an ordinary CB business object that uses DB2. Similarly, a BMP entity bean that uses JDBC to access an Oracle database must not be run in the same server process as a CMP entity bean (or other CB business object) that uses Oracle.

```
public void setEntityContext(EntityContext ctx)
     throws RemoteException {
     entityContext = ctx;
     try {
          getEnvProps();
          // Load the applet driver for DB2
          Class.forName(driverName);
     } catch (Exception e) {
          ...
     }
}
```

Figure 64. Code example: Loading and registering a JDBC driver in the setEntityContext method

Creating and closing a connection to a database

After loading and registering a database driver, the BMP entity bean must get a connection to the database. When it no longer needs that connection, the BMP entity bean must close the connection.

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.

The makeConnection method is invoked when a new database connection is required. It invokes the static method java.sql.DriverManager.getConnection and passes the DB2 URL value defined in the *jdbcUrl* variable (and described in “Defining instance variables” on page 159). The getConnection method is overloaded; the method shown here only uses the database URL, other versions require the URL and the database user ID or the URL, database user ID, and the user password.

```
# import java.sql.*;
...
private void checkConnection() throws RemoteException {
     if (jdbcConn == null) {
          makeConnection();
     }
     return;
}
...
private void makeConnection() throws RemoteException {
     ...
     try {
          // Open database connection
          jdbcConn = DriverManager.getConnection(jdbcUrl);
     } catch(Exception e) { // Could not get database connection
          ...
     }
}
```

Figure 65. Code example: The checkConnection and makeConnection methods of the AccountBMBean class

Entity beans with BMP must also drop database connections when a particular bean instance no longer requires it. The AccountBMBean class contains a dropConnection method to handle this task. To drop the database connection, the dropConnection method does the following:

1. Invokes the commit method on the connection object (*jdbcConn*), to drop any locks held on the database.
2. Invokes the close method on the connection object to close the connection.
3. Sets the connection object reference to null.

```
private void dropConnection() {
     try {
          // Close and delete jdbcConn
          jdbcConn.commit();
     } catch (Exception e) {
          // Could not commit transactions to database
          ...
     } finally {
          jdbcConn.close();
          jdbcConn = null;
     }
}
```

Figure 66. Code example: The dropConnection method of the AccountBMBean class

Managing database connections in the EJB server (AE) environment

In the EJB server (AE) 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 (AE). The connection manager creates a pool of database connections, which are allocated and deallocated to individual entity beans as needed.

Note: The example code contained in this section cannot be found in the AccountBMBean, which manages database connections by using the DriverManager interface described in "Managing connections in the EJB server (CB) environment" on page 174. This section shows the code that is required if the AccountBM bean were rewritten to use the DataSource interface.

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 67 on page 178 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 159.

```
...
# 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 RemoteException {
     entityContext = ctx;
           try {
               getEnvProps();
               ds = initContext.lookup("jdbc/sample");
          } catch (NamingException e) {
               ...
          }
     }
...
```

Figure 67. 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 68 on page 179).

```
private void checkConnection() throws RemoteException {
     if (jdbcConn == null) {
          makeConnection();
     }
     return;
}
...
private void makeConnection() throws RemoteException {
     ...
     try {
          // Open database connection
          jdbcConn = ds.getConnection();
     } catch(Exception e) { // Could not get database connection
          ...
     }
}
```

Figure 68. 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 69):

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 69. 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 70, 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, RemoteException {
     // 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 70. 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. Figure 71 on page 181 provides an example of how the data in a ResultSet is manipulated in the ejbLoad method of the AccountBMBean class.

```
public void ejbLoad () throws RemoteException {
     // 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 71. 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 6. Enabling transactions and security in enterprise beans" on page 121.

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 TX_BEAN_MANAGED, you indicate to the container that the bean is an active participant in transactions.

Note: In the EJB server (AE) environment, the value TX_BEAN_MANAGED 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.)

Note: In the EJB server (CB) environment, a stateful session bean that implements the TX_BEAN_MANAGED attribute must begin and complete a transaction within the scope of a single method.

Figure 72 on page 183 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 RemoteException {
          mySessionCtx = ctx;
     }
     ...
     public float doSomething(long arg1) throws FinderException, RemoteException {
          UserTransaction userTran = mySessionCtx.getUserTransaction();
          ...
          // User userTran object to call transaction methods
          userTran.begin();
         // Do transactional work
          ...
          userTran.commit();
          ...
     }
     ...
}
```

Figure 72. 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 10. WebSphere Programming Model Extensions

This section discusses the two 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 196.

These packages are available as part of WebSphere Application Server Advanced Edition and Enterprise Edition. They are general-purpose utilities, designed to provide common functions in a reusable way. Although these two 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*.

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 188.
- 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 189.
- 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 73 on page 187 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 73. 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 192.

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 73 on page 187). The constructors can simply pass arguments to the constructor in the DistributedException class by using super methods, as illustrated in Figure 74 on page 190.

```
...
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 74. 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 75 on page 191 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 75. 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 73 on page 187). 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 76 on page 192 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 76. 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 77 on page 194 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 77. 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 78 shows the use of the instanceof method to test for exception chaining.

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

Figure 78. 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 79 shows the use of the MyDistributedException(Throwable) constructor.

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

Figure 79. 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 80 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 80. 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 198.
- Classes and interfaces for determining where the command is run. For more information, see "Facilities for setting and determining targets" on page 199.
- Classes defining package-specific exceptions. For more information, see "Exceptions in the command package" on page 200.

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 81 on page 198 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 81. 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 82 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 82. 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 83 on page 199 shows the structure of an implementation class for the interface in Figure 82.

```
...
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 83. 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 220.
- An enterprise bean (either a session bean or an entity bean). Figure 84 on page 200 shows the structure of the remote interface and enterprise bean class for an entity bean that implements the CommandTarget interface.

```
...
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 84. 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 215.

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 185.

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 203. 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 220.

Figure 85 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 85. 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 86 shows the structure of the ModifyCheckingAccountCmdImpl class.

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

Figure 86. 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 87 on page 204 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 87. 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 88 on page 205 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 215.)

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 97 on page 214). 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 213.)

```
...
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 88. Code example: Constructors in the ModifyCheckingAccountCmdImpl class

Figure 89 on page 206 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 89. 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 90 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 90. 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 91 on page 208 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 91. 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 92 on page 209 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 92. 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 93 on page 210 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 93. Code example: Variables and constructor in the ModifyCheckingAccountCompensatorCmd class

Figure 94 on page 211 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 94. 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 95 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 216.) 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 95. 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 96 on page 213 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 96. Code example: Using the ModifyCheckingAccountCompensator command

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 220.) 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 97 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 amout) throws RemoteException;
    float deposit (int amout) throws RemoteException;
    String getAccountName() throws RemoteException;

    float getBalance() throws RemoteException;
    float setBalance(float amount) throws RemoteException;

    float withdrawal (float amout) throws RemoteException, Exception;
    float withdrawal (int amout) throws RemoteException, Exception;
}
```

Figure 97. 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 98 on page 215 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 98. 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 201.) 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 99 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 99. Code example: The TargetPolicyDefault class

Setting the command target: The ModifyCheckingAccountImpl class provides two command constructors (see Figure 88 on page 205). 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 95 on page 212 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 100 uses the same constructor to set the target explicitly. This example differs from Figure 95 on page 212 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 100. 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 101 on page 218 illustrates this technique. This example compares with Figure 95 on page 212 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 101. 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 102 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 95 on page 212, 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 102. 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 103 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 103. 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 88 on page 205). 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 104 on page 220 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 104. 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 202 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 105 on page 221 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 105. 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 106.

```
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 106. 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 107 on page 223 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 107. 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 108 on page 224. 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 108. 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 use 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.

Appendix A. Example code provided with WebSphere Application Server

This appendix contains information on the example code provided with the WebSphere Application Server for both Advanced Edition and Enterprise Edition.

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 89.
- 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 157.
- 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 104.
- 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 8. Developing servlets that use enterprise beans” on page 145.

- 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 7. Developing EJB clients” on page 129.
- 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 in the EJB server (AE) environment

Table 3 provides a summary of the enterprise bean-specific examples provided with the EJB server (AE).

Table 3. Examples available with the EJB server (AE)

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.

Information about other examples in the EJB server (CB) environment

Table 4 provides a summary of the enterprise bean-specific examples provided with the EJB server (CB). or more information about these examples, see the README file that accompanies each example.

Table 4. Examples available with the EJB server (CB)

Name	Bean types	EJB client types	Additional information
Hello	Stateless session	Java application.	Very simple example of a session bean.
Calculator	Stateful session	Java applet, ActiveX control	Demonstrates maintaining state information in a session bean.
Card Game	Stateful session, CMP entity	Java applet, ActiveX control	Demonstrates a session bean selecting entity beans using a variety of finder methods on the entity's home.
Travel	Stateful session, BMP entity, CMP entity	Java applet, ActiveX control	Demonstrates client-side transactions. Enterprise bean uses a PAA as a data source. One enterprise bean accesses another bean.

Appendix B. Using XML in enterprise beans

Note: In the EJB server (AE) environment, use of the XML feature described here is not recommended.

This appendix contains instructions for creating deployment descriptors for enterprise beans by using the extensible markup language (XML). This appendix does not contain general information on creating or using XML; for more information on XML, consult a commercially available book.

An XML file, which is a standard ASCII file, can be created manually or by using the graphical user interface (GUI) of the **jetace** tool. Once created, the XML file can be used to create an EJB JAR file from the command line by using the **jetace** tool. For more information, see "Creating a deployment descriptor and an EJB JAR file" on page 33.

An XML-based deployment descriptor must contain the following major components:

- Standard header and EJB JAR tags. For more information, see "Creating the standard header and EJB JAR tags".
- The input file and output file tags. For more information, see "Creating the input file and output file tags" on page 230.
- Session bean or entity bean tag, depending on the type of bean for which the deployment descriptor is being generated. An XML file can contain instructions for generating an EJB JAR file with multiple enterprise beans of all types. For more information, see "Creating the entity bean tags" on page 230 and "Creating the session bean tags" on page 231.
- The tags used by all enterprise beans. For more information, see "Creating tags used by all enterprise beans" on page 232.

Creating the standard header and EJB JAR tags

Every XML-based deployment descriptor must have the standard header tag, which defines the XML version and the standalone status of the XML file. For enterprise beans, these properties must be set to the values shown in Figure 109 on page 230. Except for the header tag, which must be the first tag in the file, the remaining content of the XML file must be enclosed in opening and closing EJB JAR tags.

```
<?xml version='1.0' standalone='yes' ?>
<ejb-JAR>
<!-- Content of the XML file -->
...
</ejb-JAR>
```

Figure 109. Code example: The standard header and EJB JAR tags

Creating the input file and output file tags

The input file tag identifies the JAR or ZIP file or the directory containing the required components of one or more enterprise beans. The output file tag identifies the EJB JAR file to be created; by default a JAR file is created, but you can force the creation or a ZIP file by adding a .zip extension to the output file name. These components are described in “Creating an EJB JAR file” on page 119. The input and output files for the example Account bean are shown in Figure 110.

```
<?xml version='1.0' standalone='yes' ?>
<ejb-JAR>
<input-file>AccountIn.jar</input-file>
<output-file>Account.jar</output-file>
...
</ejb-JAR>
```

Figure 110. Code example: The input file and output file tags

Creating the entity bean tags

If you are creating a deployment descriptor for an entity bean, you must use an entity bean tag. The entity bean open tag must contain a dname attribute, which must be set to the fully qualified name of the deployment descriptor associated with the entity bean.

Between the open and close entity bean tags, you must create the following entity bean-specific attribute tags:

- <primary-key> — Identifies the fully qualified name of the primary key class for this entity bean.
- <re-entrant> — Specifies whether the entity bean is re-entrant. This tag must contain a value attribute, which must be set to either `true` (re-entrant) or `false` (not re-entrant).
- <container-managed> — Identifies the persistent variables in a CMP entity bean that are container managed. You must use a separate tag for each persistent variable.

In addition to the entity bean-specific tags, you must create the tags required by all enterprise beans described in “Creating tags used by all enterprise beans” on page 232.

Figure 111 shows the entity bean-specific tags for the example Account bean.

```
<?xml version='1.0' standalone='yes' ?>
<ejb-JAR>
<input-file>AccountIn.jar</input-file>
<output-file>Account.jar</output-file>
...
<entity-bean dname="com/ibm/ejs/doc/account/Account.ser">
<primary-key>com.ibm.ejs.doc.account.AccountKey</primary-key>
<re-entrant value=false/>
<container-managed>accountId</container-managed>
<container-managed>type</container-managed>
<container-managed>balance</container-managed>
<!--Other tags used by all enterprise beans--!>
...
</entity-bean>
...
</ejb-JAR>
```

Figure 111. Code example: The entity bean-specific tags

Creating the session bean tags

If you are creating a deployment descriptor for an session bean, you must use a session bean tag. The session bean open tag must contain a dname attribute, which must be set to the fully qualified name of the deployment descriptor associated with the session bean. Between the open and close session bean tags, you must also create the following session bean attribute tags:

- <session-timeout> — Defines the idle timeout in seconds associated with the session bean.
- <state-management> — Identifies the type of session bean: `STATELESS_SESSION` or `STATEFUL_SESSION`.

In addition to the session bean-specific tags, you must create the tags required by all enterprise beans described in “Creating tags used by all enterprise beans” on page 232.

Figure 112 on page 232 shows the session bean tags for the example Transfer bean.

```
<?xml version='1.0' standalone='yes' ?>
<ejb-JAR>
<input-file>TransferIn.jar</input-file>
<output-file>Transfer.jar</output-file>
...
<session-bean dname="com/ibm/ejs/doc/transfer/Transfer.ser">
<session-timeout>0<\session-timeout>
<state-management>STATELESS_SESSION<\state-management>
<!--Other tags used by all enterprise beans--!>
...
</session-bean>
...
</ejb-JAR>
```

Figure 112. Code example: The session bean-specific tags

Creating tags used by all enterprise beans

The following tags are used by all types of enterprise beans. These tags must be placed between the appropriate set of opening and closing session or entity bean tags in addition to the tags that are specific to those types of beans.

- <remote-interface> — Identifies the fully qualified name of the enterprise bean's remote interface.
- <enterprise-bean> — Identifies the fully qualified name of the enterprise bean's bean class.
- <JNDI-name> — Identifies the JNDI home name of the enterprise bean.
- <transaction-attr> — Defines the transaction attribute for the entire enterprise bean. This attribute can also be set for an individual bean method. The valid values are TX_MANDATORY, TX_NOT_SUPPORTED, TX_REQUIRES_NEW, TX_REQUIRED, TX_SUPPORTS, and TX_BEAN_MANAGED. For more information on the meaning of and restrictions on these values, see "Setting the transaction attribute" on page 122.
- <isolation-level> — Defines the transactional isolation level attribute for the entire enterprise bean. This attribute can also be set for an individual bean method. The valid values, which must be set by using a value attribute within the open tag, are SERIALIZABLE, REPEATABLE_READ, READ_COMMITTED, and READ_UNCOMMITTED. For more information on the meaning of and restrictions on these values, see "Setting the transaction isolation level attribute" on page 124.
- <run-as-mode> — Defines the run-as mode attribute for the entire enterprise bean. This attribute can also be set for an individual bean method. The valid values, which must be set by using a value attribute within the open tag, are CLIENT_IDENTITY, SYSTEM_IDENTITY, and

SPECIFIED_IDENTITY. For more information on the meaning of these values, see “Setting the security attribute in the deployment descriptor” on page 126.

- <run-as-id> — Defines the run-as identity attribute for the entire enterprise bean. This attribute can also be set for an individual bean method. This attribute is not used with the EJB server environments contained in WebSphere Application Server.
- <method-control> — Identifies individual bean methods with transaction or security attributes that are different from the attribute values for the entire bean.
- <dependency> — Identifies the fully qualified names of classes on which this enterprise bean is dependent.
- <env-setting> — Identifies environment variables (and their values) required by the enterprise bean. The environment variable name is specified with a name attribute, while the environment variable value is placed between the open and close tags.

Figure 113 shows the enterprise bean tags for the example Transfer bean. A similar set is required by the Account bean.

```
<?xml version='1.0' standalone='yes' ?>
<ejb-JAR>
<input-file>TransferIn.jar</input-file>
<output-file>Transfer.jar</output-file>
...
<session-bean dname="com/ibm/ejs/doc/transfer/Transfer.ser">
<!--Session bean-specific tags --!>
...
<remote-interface>com.ibm.ejs.doc.transfer.Transfer</remote-interface>
<enterprise-bean>com.ibm.ejs.doc.transfer.TransferBean</enterprise-bean>
<JNDI-name>Transfer </JNDI-name>
<transaction-attr value="TX_REQUIRED"/>
<isolation-level value="SERIALIZABLE"/>
<run-as-mode value="CLIENT_IDENTITY"/>
<dependency>com/ibm/ejs/doc/account/InsufficientFundsException.class</dependency>
...
<env-setting name="ACCOUNT_NAME">Account<env-setting>
...
</session-bean>
...
</ejb-JAR>
```

Figure 113. Code example: The tags used for all enterprise beans

If you want to override the enterprise bean-wide transaction or security attribute for particular method in that bean, you must use the <method-control> tag. Between the open and close tags, you must identify the method with the <method-name> tag and the method’s parameter types by using the <parameter> tag. In addition, the following tags can used to

identify those attribute values that are different in the method from the enterprise bean as a whole: <transaction-attr>, <isolation-level>, <run-as-mode>, and <run-as-id>.

For example, the XML shown in Figure 114 is required to override the transaction attribute of the Transfer bean (TX_REQUIRED) in the getBalance method to TX_SUPPORTED. Because only the transaction attribute is overridden, the method automatically inherits the values of the <isolation-level> and <run-as-mode> tags from the Transfer bean.

```
<?xml version='1.0' standalone='yes' ?>
<ejb-JAR>
<input-file>TransferIn.jar</input-file>
<output-file>Transfer.jar</output-file>
...
<session-bean dname="com/ibm/ejs/doc/transfer/Transfer.ser">
<!--Session bean-specific tags --!>
...
<transaction-attr value="TX_REQUIRED"/>
<isolation-level value="SERIALIZABLE"/>
<run-as-mode value="CLIENT_IDENTITY"/>
...
<method-control>
<method-name>getBalance</method-name>
<parameter>long</parameter>
<transaction-attr value="TX_SUPPORTED"/>
</method-control>
</session-bean>
...
</ejb-JAR>
```

Figure 114. Code example: Method-specific tags

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

Note: This extension is supported only in the EJB server (AE) environment.

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:

AFS
AIX
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
IMS
MQSeries
MVS/ESA
OS/2
OS/390
OS/400
PowerPC
RISC System/6000
RS/6000
S/390
Transarc
TXSeries
VSE/ESA
VTAM
VisualAge
WebSphere

Microsoft, Windows, Windows NT, and the Windows 95 logo are trademarks of Microsoft Corporation in the United States and/or other countries.

Oracle and Oracle8 are registered trademarks of the Oracle Corporation in the United States and/or other countries.

UNIX is a registered trademark of The Open Group in the United States and/or other countries licensed exclusively through X/Open Company Limited.

OSF and Open Software Foundation are registered trademarks of the Open Software Foundation, Inc.

* HP-UX is a Hewlett-Packard branded product. HP, Hewlett-Packard, and HP-UX are registered trademarks of Hewlett-Packard Company.

Orbix is a registered trademark and OrbixWeb is a trademark of IONA Technologies Ltd.

Sun, SunLink, Solaris, SunOS, Java, all Java-based trademarks and logos, NFS, and Sun Microsystems are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and/or 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.

Index

D

E

F

R

S

T

U

V

W

X

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC09-4431-02

Spine information:

WebSphere Writing Enterprise Beans in WebSphere Version 3.5 SC09-4431-02