## Contents

### Chapter 1
**Preface**
- Organization of this manual: 1-1
- Which sections pertain to you?: 1-2
- Typographic conventions: 1-2
- Platform conventions: 1-3
- Where to find additional information: 1-3
  - EJB-related information: 1-3
  - CORBA-related information: 1-4
  - Contacting Inprise developer support: 1-4

### Chapter 2
**Overview**
- Enterprise JavaBeans architecture: 2-1
  - Enterprise bean components: 2-2
  - EJB roles: 2-4
  - EJB design patterns and naming conventions: 2-5
  - Enterprise JavaBean infrastructure: 2-6
  - Container: 2-7
- Why use Inprise’s EJB container?: 2-8
  - Complete and flexible EJB runtime: 2-8
- Inprise’s EJB Container is built on top of VisiBroker and RMI-IIOP: 2-8
- Inprise’s EJB container is CORBA: 2-9
- Container support for different enterprise beans: 2-9
  - Deployment support: 2-10
  - The transaction manager: 2-10
  - JDBD connection pool and transactional integration: 2-10
  - The naming service: 2-10
  - Support for security: 2-10
  - The Java database: 2-11
  - Container-managed persistence of entity beans: 2-11
  - Integration with other components: 2-11

### Chapter 3
**Quick start**
- Getting started with the examples: 3-1
  - Overview of the examples: 3-1
  - Building the examples: 3-2
  - Running the examples: 3-3
  - Basics of the EJB container and EJB tools: 3-5
    - Starting the EJB Container: 3-5
    - EJB Server and EJB Container concepts: 3-6
    - Diagnostic flags: 3-7
    - EJB tools: 3-8
- Stateless session bean example: 3-9
  - Write an Enterprise bean: 3-9
  - Write the client code: 3-13
  - Build the sort session bean and client: 3-13
  - Create the deployment descriptor: 3-14
  - Run the Sort example: 3-15

### Chapter 4
**Developing enterprise beans**
- Developing the enterprise bean:
  - Quick steps: 4-1
  - Using JBuilder: 4-2
  - Using an IDE other than JBuilder: 4-2
  - Enterprise bean development: 4-3
    - Enterprise bean class inheritance: 4-4
    - Remote interface: 4-4
    - Home interface: 4-5
    - Enterprise bean implementation: 4-10
    - Handles: 4-10
  - Programming restrictions: 4-11

### Chapter 5
**Writing enterprise bean clients**
- Client view of an enterprise bean:
  - Initializing the client: 5-1
  - Locating the home interface: 5-2
  - Obtaining the remote interface: 5-3
  - Invoking methods: 5-5
  - Removing bean instances: 5-6
  - Using a bean’s handle: 5-6
  - Managing transactions: 5-7
  - Getting information about an enterprise bean: 5-8
  - Support for JNDI: 5-9
    - EJB to CORBA mapping: 5-9
    - Mapping for distribution: 5-10
    - Mapping for naming: 5-11
    - Mapping for transaction: 5-12
    - Mapping for security: 5-13
## Chapter 6

**Writing session beans**

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview of a session bean</td>
<td>6-1</td>
</tr>
<tr>
<td>Life cycle of a stateful session bean</td>
<td>6-2</td>
</tr>
<tr>
<td>Life cycle of a stateless session bean</td>
<td>6-4</td>
</tr>
<tr>
<td>Developing a session bean</td>
<td>6-5</td>
</tr>
<tr>
<td>SessionBean interface</td>
<td>6-5</td>
</tr>
<tr>
<td>Session synchronization interface</td>
<td>6-6</td>
</tr>
<tr>
<td>Session bean implementation</td>
<td>6-7</td>
</tr>
<tr>
<td>The cart example</td>
<td>6-9</td>
</tr>
<tr>
<td>Stateful session beans and caching</td>
<td>6-9</td>
</tr>
<tr>
<td>The cart example’s files</td>
<td>6-9</td>
</tr>
<tr>
<td>Cart home interface</td>
<td>6-10</td>
</tr>
<tr>
<td>Cart remote interface</td>
<td>6-11</td>
</tr>
<tr>
<td>CartBean session bean</td>
<td>6-12</td>
</tr>
<tr>
<td>Item class</td>
<td>6-16</td>
</tr>
<tr>
<td>Exceptions</td>
<td>6-17</td>
</tr>
<tr>
<td>XML deployment descriptor file</td>
<td>6-18</td>
</tr>
</tbody>
</table>

## Chapter 7

**Writing entity beans**

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview of entity beans</td>
<td>7-1</td>
</tr>
<tr>
<td>Handling persistence</td>
<td>7-2</td>
</tr>
<tr>
<td>Bean-managed persistence</td>
<td>7-2</td>
</tr>
<tr>
<td>Container-managed persistence</td>
<td>7-3</td>
</tr>
<tr>
<td>Understanding the lifecycle of an entity bean</td>
<td>7-4</td>
</tr>
<tr>
<td>Implementing an entity bean</td>
<td>7-6</td>
</tr>
<tr>
<td>Entity bean interface</td>
<td>7-6</td>
</tr>
<tr>
<td>Entity bean methods</td>
<td>7-8</td>
</tr>
<tr>
<td>Method synchronization</td>
<td>7-10</td>
</tr>
<tr>
<td>Entity bean primary keys</td>
<td>7-11</td>
</tr>
<tr>
<td>Handling transactions with</td>
<td></td>
</tr>
<tr>
<td>optimistic concurrency</td>
<td>7-11</td>
</tr>
<tr>
<td>Bank entity bean example</td>
<td>7-12</td>
</tr>
<tr>
<td>Entity bean home interface</td>
<td>7-12</td>
</tr>
<tr>
<td>Entity bean remote interface</td>
<td>7-13</td>
</tr>
<tr>
<td>Entity bean with</td>
<td></td>
</tr>
<tr>
<td>container-managed persistence</td>
<td>7-14</td>
</tr>
<tr>
<td>Entity bean with</td>
<td></td>
</tr>
<tr>
<td>bean-managed persistence</td>
<td>7-15</td>
</tr>
<tr>
<td>Entity bean deployment descriptor</td>
<td>7-20</td>
</tr>
<tr>
<td>Deploying the bank enterprise beans</td>
<td>7-23</td>
</tr>
<tr>
<td>Using debug mode</td>
<td>7-23</td>
</tr>
<tr>
<td>An example using Oracle</td>
<td>7-23</td>
</tr>
</tbody>
</table>

## Chapter 8

**Transaction management**

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Understanding transactions</td>
<td>8-1</td>
</tr>
<tr>
<td>Characteristics of transactions</td>
<td>8-2</td>
</tr>
<tr>
<td>Transaction support</td>
<td>8-2</td>
</tr>
<tr>
<td>Transaction manager services</td>
<td>8-3</td>
</tr>
<tr>
<td>Enterprise beans and transactions</td>
<td>8-3</td>
</tr>
<tr>
<td>Understanding bean- and container-managed transactions</td>
<td>8-4</td>
</tr>
<tr>
<td>Transaction attributes</td>
<td>8-5</td>
</tr>
<tr>
<td>Local and global transactions</td>
<td>8-6</td>
</tr>
<tr>
<td>Using the transaction API</td>
<td>8-6</td>
</tr>
<tr>
<td>Handling transaction exceptions</td>
<td>8-7</td>
</tr>
<tr>
<td>System-level exceptions</td>
<td>8-7</td>
</tr>
<tr>
<td>Application-level exceptions</td>
<td>8-8</td>
</tr>
<tr>
<td>Handling application exceptions</td>
<td>8-8</td>
</tr>
<tr>
<td>JDBC support</td>
<td>8-10</td>
</tr>
<tr>
<td>Specifying DataSources</td>
<td>8-10</td>
</tr>
<tr>
<td>Handling database connections and pooling</td>
<td>8-11</td>
</tr>
<tr>
<td>Transaction isolation level</td>
<td>8-13</td>
</tr>
<tr>
<td>Distributed transactions</td>
<td>8-13</td>
</tr>
<tr>
<td>Two-phase commit</td>
<td>8-13</td>
</tr>
</tbody>
</table>

## Chapter 9

**Deploying Enterprise JavaBeans**

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deploying Enterprise JavaBeans:</td>
<td></td>
</tr>
<tr>
<td>Quick steps</td>
<td>9-1</td>
</tr>
<tr>
<td>Creating a deployment descriptor file</td>
<td>9-1</td>
</tr>
<tr>
<td>The role of the deployment descriptor</td>
<td>9-2</td>
</tr>
<tr>
<td>The types of information in the deployment descriptor</td>
<td>9-3</td>
</tr>
<tr>
<td>Inprise-specific information needed to deploy enterprise beans</td>
<td>9-5</td>
</tr>
<tr>
<td>Specifying the EJB’s runtime environment properties</td>
<td>9-10</td>
</tr>
<tr>
<td>Creating the EJB jar file</td>
<td>9-10</td>
</tr>
<tr>
<td>Deploying your EJB jar file to the container</td>
<td>9-11</td>
</tr>
</tbody>
</table>

## Chapter 10

**EJB tools**

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>java2iop</td>
<td>10-1</td>
</tr>
<tr>
<td>When do you use it?</td>
<td>10-1</td>
</tr>
<tr>
<td>Syntax</td>
<td>10-2</td>
</tr>
<tr>
<td>Options</td>
<td>10-2</td>
</tr>
<tr>
<td>Verify</td>
<td>10-2</td>
</tr>
<tr>
<td>When do you use it?</td>
<td>10-3</td>
</tr>
<tr>
<td>Syntax</td>
<td>10-3</td>
</tr>
</tbody>
</table>
The Inprise Application Server Enterprise JavaBeans Programmer’s Guide provides information on developing Enterprise JavaBeans, writing the necessary code for clients of the EJB container, and deploying enterprise beans.

This preface lists the contents of the Enterprise JavaBeans Programmer’s Guide. It also describes typographic and platform conventions used throughout the manual and provides references for more information about EJB and CORBA.

Organization of this manual

- Chapter 2, “Overview,”—provides an introduction to enterprise beans and the EJB architecture.
- Chapter 3, “Quick start,”—describes the basics for using the EJB Container and tools, plus gets you started running the examples. It walks you through an examples using stateless session beans.
- Chapter 4, “Developing enterprise beans,”—provides the details for developing enterprise beans.
- Chapter 5, “Writing enterprise bean clients,”—describes how to write code for clients of the EJB container.
- Chapter 6, “Writing session beans,”—describes how to write stateful and stateless session beans.
- Chapter 7, “Writing entity beans,”—describes how to write entity beans with bean-managed persistence and entity beans with container-managed persistence.
- Chapter 8, “Transaction management,”—describes the support for transactions and for JDBC.
- Chapter 9, “Deploying Enterprise JavaBeans,”—describes how to use the Inprise EJB deployment tools.
Which sections pertain to you?

- Chapter 10, “EJB tools,”—describes the tools for managing and administering enterprise beans.
- Chapter 10, “Analyzing the cart example’s output,”—shows the cart example in detail.
- Chapter 10, “EJB 1.0 support,”—explains the support for deployment descriptors that are compliant with the obsolete EJB 1.0 specification.
- “Index”

Which sections pertain to you?

The following table lists which sections pertain to the various audiences for this guide.

<table>
<thead>
<tr>
<th>Audience</th>
<th>Sections of Interest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Executives, evaluators</td>
<td>Chapter 1, “Preface,”, Chapter 2, “Overview”</td>
</tr>
<tr>
<td>Developers</td>
<td>Chapter 3, “Quick start,” and rest of the book</td>
</tr>
</tbody>
</table>

Typographic conventions

This manual uses the following conventions:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Used for</th>
</tr>
</thead>
<tbody>
<tr>
<td>boldface</td>
<td>Bold type indicates that syntax should be typed exactly as shown. For UNIX, used to indicate database names, filenames, and similar terms.</td>
</tr>
<tr>
<td>italics</td>
<td>Italics indicates information that the user or application provides, such as variables in syntax diagrams. Also used to introduce new terms.</td>
</tr>
<tr>
<td>computer</td>
<td>Computer typeface is used for sample command lines and code.</td>
</tr>
<tr>
<td>UPPERCASE</td>
<td>Uppercase letters indicate SQL statements and terms. For Windows, used to indicate database names, filenames, and similar terms.</td>
</tr>
<tr>
<td>[]</td>
<td>Brackets indicate optional items.</td>
</tr>
<tr>
<td>{}</td>
<td>Curly brackets are used in the more complex syntax statements to show a required item.</td>
</tr>
<tr>
<td>…</td>
<td>An ellipsis indicates the continuation of previous lines of code or that the previous argument can be repeated.</td>
</tr>
<tr>
<td></td>
<td>A vertical bar separates two mutually exclusive choices.</td>
</tr>
</tbody>
</table>
Platform conventions

This manual uses the following conventions to indicate that information is platform-specific:

- **Windows**: All Windows platforms including Windows 3.1, Windows NT, and Windows 95
- **WinNT**: Windows NT only
- **Win95**: Windows 95 only
- **Win98**: Windows 98 only
- **Win2000**: Windows 2000 only
- **UNIX**: All UNIX platforms
- **Solaris**: Solaris only
- **AIX**: AIX only
- **HP-UX**: HP-UX only
- **IRIX**: IRIX only
- **Digital UNIX**: Digital UNIX only

Where to find additional information

The documentation assumes familiarity with distributed computing in general and Sun’s Enterprise JavaBeans (EJB) specification in particular.

For more information about EJB and CORBA, see the sources which follow.

**EJB-related information**

To learn more about EJB, visit the Sun web site. To access the Sun EJB web site, go to the following URL: http://java.sun.com/products.ejb

Sun’s EJB web site provides the following:

- Sun’s Enterprise Java Beans (EJB) 1.1 specification
- Enterprise JavaBeans 1.0 CORBA Mapping
- A white paper entitled “Enterprise JavaBeans Technology Component Model for the Java Platform”
- An EJB list of frequently asked questions (FAQ)
- Other EJB-related documents
Where to find additional information

There are additional related documents on the Sun web site:

- JavaBeans. See http://java.sun.com/beans
- Java Naming and Directory Interface (JNDI).
  See http://java.sun.com/products/jndi
- Java Remote Method Invocation (RMI).
  See http://java.sun.com/products/jdk/rmi

CORBA-related information

The Object Management Group’s (OMG) web site also has several documents that will be helpful. To visit the Object Management Group’s (OMG) web site, go to the following URL: http://www.omg.org/

The OMG’s web site provides the following:

- CORBA 2.3/IIOP Specification
  See http://www.omg.org/
- Java to IDL Mapping
  See http://www.omg.org/corba/chapter.html#mijava
- OMG Object Transaction Service
  See http://www.omg.org/corba/sectrans.html
- ORB Portability Submission
  See http://www.omg.org/library/c2index.html

Contacting Inprise developer support

Inprise offers a variety of support options. These include free services on the Internet, where you can search our extensive information base and connect with other users of Inprise products. In addition, you can choose from several categories of telephone support, ranging from support on installation of the Inprise product to fee-based consultant-level support and detailed assistance.

For more information about Inprise’s developer support services, please see our web site at http://www.borland.com/devsupport, call Inprise Assist at 800-523-7070, or contact our Sales Department at 800-632-2864. For customers outside of the United States of America, please see our web site at http://www.borland.com/bww/intlcust.html.

When contacting support, be prepared to provide complete information about your environment, the version of the product you are using, and a detailed description of the problem.

For information about year 2000 issues and our products, see the following URL: http://www.borland.com/about/y2000/.
This chapter contains the following major topics:

- “Enterprise JavaBeans architecture” describes the architecture of the Enterprise JavaBeans (EJB) technology.
- “Why use Inprise’s EJB container?”

**Enterprise JavaBeans architecture**

The Enterprise JavaBeans architecture is a high-level component-based architecture for distributed business applications that uses the transaction system’s lower-level APIs. EJB simplifies the development, deployment, and execution of enterprise systems in the Java programming language. The Enterprise JavaBeans architecture is defined in a specification developed and edited by Sun Microsystems. Inprise’s EJB container is based on version 1.1 of the specification.

The Enterprise JavaBeans technology defines a set of reusable components called enterprise beans. You create a distributed application by coding the application’s business logic in these enterprise beans. Once the coding is completed, the enterprise beans are assembled into special files, with one or more enterprise beans per file, along with deployment parameters. Lastly, the enterprise beans are deployed onto a platform that runs an EJB container. Clients can locate an enterprise bean and create an instance of that bean through the enterprise bean’s home interface. Then, the client can invoke the business methods of the enterprise bean using the enterprise bean’s remote interface.

The EJB server manages EJB containers and functions as a bridge between the container and the underlying platform. It provides its EJB containers with access to the platform’s system services, such as database management and a transaction monitor, and to other existing enterprise applications.

All enterprise bean instances run within an EJB container. The container provides system-level services to its enterprise beans and controls their life cycle. Because the
container handles most system-level issues, the enterprise bean developer does not have to include this logic with the business methods of the enterprise bean. In general, containers handle such system-level issues as:

- **Security**—The deployment descriptor defines the clients that can access the different business methods. The container enforces this by permitting only authorized clients to invoke those methods to which they have access.

- **Remote connectivity**—The container manages the low-level communication issues for remote connectivity and hides these issues from the enterprise bean developer and the client. An enterprise bean developer writes the business methods as if they will be invoked on a local platform; the client is unaware that he or she is invoking a method that potentially must be reached remotely.

- **Life Cycle management**—Clients simply create instances of enterprise beans and (usually) remove these instances. However, the container manages these enterprise bean instances to maximize performance and memory usage. The container may do such things as inactivate and activate these enterprise bean instances, keep a pool of instances to share among clients, and so on.

- **Transaction management**—The deployment descriptor defines the transactional requirements of an enterprise bean. The container manages the complex issues of managing distributed transactions that potentially update databases spread across multiple platforms. The container keeps transactional data isolated, and it ensures that updates to all the databases occur successfully; otherwise, it rolls back all aspects of the transaction.

**Enterprise bean components**

Enterprise beans are components of distributed transaction-based enterprise applications. All enterprise beans share certain characteristics:

- **Enterprise beans contain business logic that operates on the enterprise’s data.**

- **The enterprise bean provider defines a client’s view of the enterprise bean (the remote interface to the bean), and this view is unaffected by the container or server on which the bean is deployed. As a result, enterprise beans do not require source code changes or recompilations when included in an assembled application.**

- **An EJB container creates and manages enterprise bean instances at runtime. The container also manages client access to the enterprise bean.**

- **Enterprise beans can be customized when deployed by editing their environment entries.**

- **Various system services, such as security and transaction attributes, are not part of an enterprise bean class and, thus, are managed by deployment and application assembly tools.**

There are two types of enterprise beans: session beans and entity beans.
Session beans

A session bean is an object that executes on behalf of a single client. The container creates the session bean instance in response to a remote task request from a client. A session bean has one client; in a sense, a session bean represents its client in the EJB server. Session beans can also be transaction-aware—they can update shared data in an underlying database but they do not directly represent the shared database data.

The life of a session bean is transient and relatively short-lived. Typically, the session bean lives for as long as its client maintains the session “conversation.” When the client terminates, the session bean is no longer associated to that client. A session bean is considered transient because the session bean instance is removed should the container crash, and the client must re-establish a new session object to continue.

A session bean typically maintains the state of the interaction or conversation with its client—that is, the session bean holds information about the client across method invocations and for the duration of the client session. A session bean that maintains its state is called a stateful session bean. When the client ends its interaction with the session bean, the session ends and the bean no longer maintains the state values.

A session bean may also be a stateless session bean. A stateless session bean does not maintain information or state for its client. A client may invoke a method of a stateless session bean to accomplish some objective, but the bean will hold values in its instance variables only for the duration of the method call. The stateless session bean does not retain these values (or state) when the method completes. Thus, all instances of stateless session beans are identical except when they are in the midst of a method invocation. As a result, stateless session beans can support multiple clients. The container can maintain a pool of stateless bean instances, and it can assign any instance to any client.

Entity beans

Entity beans provide an object view of data in a database. For example, an entity bean can model a row of a relational database table. Access to entity beans may be shared by more than one client—multiple clients can simultaneously access an entity bean. Entities access and update the underlying data within the context of a transaction so that data integrity is maintained.

An entity bean is considered to be long-lived and its state is persistent. It lives as long as the data remains in the database, rather than for the life of the application or server process. An entity bean survives the crash of the EJB container.

Entity bean persistence can be either container managed or bean managed. With container-managed persistence, the container handles the issues of ensuring the bean’s persistence. With bean-managed persistence, you must write the entity bean code to include the calls to the database.

Each entity bean is identified by a primary key, a unique object identifier. The primary key is usually the same primary key that identifies the piece of data in the database, such as a row within a table. The primary key enables the client to locate a particular piece of data.
EJB roles

The EJB architecture simplifies the development of complex business systems by dividing this process into six distinct roles, each with a specific task. These six roles address infrastructure services, application development, and deployment and operations issues.

The EJB architecture serves to simplify the application programmer’s job. With the EJB architecture, the container and server vendors take on many of the tasks that would ordinarily fall to application programmers in other environments. These vendors provide the system and platform level services that formerly had to be coded by application developers.

Infrastructure roles

An EJB Server Provider is typically a vendor with expertise in distributed infrastructures and services. The server provider implements a platform that facilitates the development of distributed applications and provides a runtime environment for these applications.

An EJB Container Provider is an expert in distributed systems, transactions, and security. The container provider provides deployment tools for the enterprise beans and runtime support for these deployed instances.

A container is a runtime system for one or multiple enterprise beans. It provides the glue between enterprise beans and the EJB server, including transaction, security, and network distribution management. A container is both prefabricated code and a tool that generates code specific for a particular enterprise bean. A container also provides tools for the deployment of an enterprise bean and a means for the container to monitor and manage the application.

Application Roles

An Enterprise Bean Provider is typically an expert in the application domain; for example, the bean provider may be an expert in the financial or telecommunications industry. The bean provider implements the business tasks, or the enterprise bean’s business methods, and defines the bean’s remote and home interfaces and the bean’s deployment descriptor. Because the container manages system-level tasks, the bean provider need not be concerned about the distribution, transaction, security, and other non-business-specific aspects of the application.

An Application Assembler is a domain expert who composes an application from various prefabricated building blocks—that is, enterprise beans—and adds other components such as GUI clients, applets, and servlets to complete the application. While composing an application, an assembler is only concerned with the interfaces to enterprise beans (the home and remote interfaces), but not with their implementation, and with the bean’s deployment descriptor. The result of the assembly process can be a complete application or another, more complex enterprise bean consisting of several enterprise beans.
Deployment and Operation Roles

A **Deployer** specializes in the operation environment for particular applications. The deployer adapts an application, composed of a number of enterprise beans, to a target operation environment by modifying the properties of the enterprise beans. For example, the deployer sets transaction and security policies by setting the appropriate properties in the deployment descriptor. The deployer also has the task of integrating the application with existing enterprise management software.

A **System Administrator** is concerned with a deployed application. The administrator configures and administers the enterprise computing and networking infrastructure, including the EJB server and container. The administrator monitors the running application and takes appropriate actions in the event that the application behaves abnormally. Typically, an administrator uses enterprise management tools that are connected to the application by the deployer through the hooks provided by the container.

A “traditional” application programmer now becomes an enterprise bean provider and, possibly, an application assembler. These tasks allow her or him to focus on the business problem and business logic. The deployer defines and sets the deployment policies when installing the enterprise bean. The complexity of implementing mechanisms for executing the deployment policies is delegated to specialized vendors. Although distributed applications remain complex, the application programmer’s job becomes easier because much of the complexity is addressed by EJB server and container providers.

The EJB specification achieves the aforementioned goals by introducing a number of predefined design patterns and naming conventions. This restricts the freedom in the application architecture, but allows the container and service providers to make assumptions about the application design and support them in an efficient manner.

EJB design patterns and naming conventions

There are three major approaches for building object-based, multi-tier, distributed systems: the stateless server approach, the session-oriented approach, and the persistent object approach.

The stateless server is an object that provides functionality through its operations but does not keep conversational state. When a client uses a stateless object, the client can not refer to state information provided in previous operations on the same object.

The session-oriented design creates an object in the middle tier, called a session, which acts as an agent for the client. Typically, the lifetime of a session object is determined by the client and the server program that is hosting it. A client can remove the object once it is finished with it. The server can time-out a session object, and when the server terminates, references to the session object may become invalid.

The persistent object design model wraps a certain piece of data (stored in a database) and provides operations to manipulate this data. Persistent objects are shared among multiple clients. The lifetime of a persistent object is determined by the lifetime of the data storage which contains the data.
The EJB specification defines these approaches as stateless session bean, stateful session bean, and entity bean. The session bean models the session-oriented design approach. The entity bean models persistent object design. There are defined interfaces and naming conventions for each, and these are explained in Chapter 4, “Developing enterprise beans.”

Subsequent chapters provide examples and explain how to use these different types of enterprise beans.

**Enterprise JavaBean infrastructure**

The container and server providers implement the EJB infrastructure. This infrastructure deals with distribution aspects, transaction management, and security aspects of an application. The EJB specification defines the requirements of an EJB infrastructure and the Java Application Programming Interface (API) for the various features; it does not prescribe the technology, platforms, protocols, and so forth to implement them.

Figure 2.1, “Enterprise beans, containers, and servers” illustrates the various requirements for the EJB infrastructure. The EJB infrastructure must provide communication channels to clients and other Enterprise beans. Although not required by the EJB specification, it is important that these communication channels be secure, especially when clients access remote Enterprise beans via the internet. The EJB infrastructure must also be able to enforce access control to Enterprise beans.

The persistence of the data contained by enterprise beans must be ensured. An EJB infrastructure must provide integration capabilities to existing systems and applications to be useful in an enterprise computing environment. All communication to and from a bean can be part of a distributed transaction that needs to be managed by the EJB infrastructure. For a successful deployment, the EJB infrastructure must provide hooks into a distributed application management tool.
Container

The container is probably the single most important concept in the enterprise beans approach because it provides the most benefit to the developer. Object-based middleware platforms like CORBA or Remote Method Invocation (RMI) free a distributed application developer from the networking aspects of the application by providing mechanisms for object location, data marshaling, and so forth. The concept of a container takes this idea a step further by simplifying other nontrivial aspects of a distributed application, such as security, transaction coordination, and data persistence.

Once an enterprise bean is ready for deployment, it is packed in a standard Java archive file, called an ejb-jar file. The ejb-jar file can contain one or multiple enterprise beans. For each enterprise bean, it contains its interfaces, classes, and a deployment descriptor.

The enterprise bean provider has to supply these things to the ejb-jar file:

- The enterprise bean’s remote interface to the enterprise bean. This interface specifies the application-specific methods that are available to a client.
- The home interface for the enterprise bean. The home interface enables a client to create and locate instances of the enterprise bean.
- The enterprise bean class itself, which is the implementation of the application semantics of the enterprise bean.
- The deployment descriptor. The deployer uses the deployment descriptor to provide declarative information not included in the code itself, such as policies and properties, that determine the execution of a particular enterprise bean. The EJB 1.0 specification defines the deployment descriptor as a serialized instance of the class `javax.ejb.deployment.SessionDescriptor` or `javax.ejb.deployment.EntityDescriptor`. This API has been deprecated in the EJB 1.1 version of the specification; instead, XML is used for defining deployment descriptors. Details on the deployment descriptor are given later in the chapter.

Deploying an enterprise bean means to install the ejb-jar file into a container. The installation process deals with the following issues:

- Ensuring that the different pieces of the enterprise bean work together.
- Registering the enterprise bean with a name service.
- Providing access to the enterprise bean through the EJB server’s communication system.
- Executing the transaction management and security policies.

There can be any number of different enterprise beans installed in a container. Besides the installation and execution of an enterprise bean, a container also provides tools for the deployment of an enterprise bean. Chapter 10, “EJB tools” explains the deployment tools provided by the Inprise EJB container.
Why use Inprise’s EJB container?

Inprise’s full-service EJB Container provides the following:

- Complete EJB 1.1 specification support.
- A robust container for running and managing EJBs.
- Ability to run a naming service, transaction service, and an embedded Java RDBMS for easy development and deployment.
- Examples to show you how to use EJBs and the EJB Container.

The EJB Container is the superior enterprise-class product for enterprise bean developers. Consider these advantages:

Complete and flexible EJB runtime

- EJB container is a complete implementation of the EJB 1.1 specification, including all optional functionality.
- Every enterprise bean object generated with the tools is simultaneously both an enterprise bean object and a CORBA object.
- The EJB Container can be deployed as a stand-alone, 100% pure Java service or as a fully-distributed deployment. This flexibility in deployment means that you can adjust the application’s scalability and availability based on your requirements.
- Unlike other EJB servers, Inprise’s EJB Container server has no restrictions on partitioning. Any number of enterprise beans can go into any number of containers running on any number of machines. Plus, support for distributed transactions protocol allows enterprise beans to be partitioned arbitrarily. Partitioning enables you to configure the application during deployment to optimize its overall performance.

Inprise’s EJB Container is built on top of VisiBroker and RMI-IIOP

- The EJB Container is built on top of Inprise’s VisiBroker. VisiBroker is a leading ORB that has proven itself in heavy stress conditions, both as part of vendors’ internal tests and, more importantly, in actual customer environments. VisiBroker supplies a wide range of mission-critical runtime features.
- To achieve superior scalability, VisiBroker uses a number of runtime optimizations: advanced connection multiplexing, connection pooling and management, plus thread pooling and management.
- Communication between clients and enterprise beans, among enterprise beans, and between enterprise beans and other CORBA-based applications is all done using IIOP by way of VisiBroker. VisiBroker is fully compliant with the CORBA 2.3 specification which specifies that RMI-over-IIOP must be implemented in terms of objects-by-value. RMI-over-IIOP must be implemented in terms of objects-by-value for true interoperability. This means that complex Java data types
Why use Inprise’s EJB container?

(such as dictionaries, vectors, and so on) are written into IIOP using the new IDL “value” types, as specified by CORBA 2.3. It is important that clients and servers agree on how to pass complex data types; otherwise, interoperability is jeopardized. Inprise’s EJB Container’s server is interoperable with any other server supporting RMI-over-IIOP.

- Security credentials are propagated by VisiBroker. This ensures that a client’s credentials are propagated from the client to the server.
- Transactional context is propagated by VisiBroker. This ensures that when a CORBA client begins a transaction and then accesses the EJB Container’s server, transactional context is propagated in the call to the server and the server uses this transaction context when making calls to various resources in its environment.
- Two-phased commit transactions are managed by Inprise’s Integrated Transaction Service (ITS). Two-phase commit is supported if the JDBC driver supports it. If the JDBC driver in use does not support it, then two-phase commit cannot be done.

Inprise’s EJB container is CORBA

Inprise’s java2iop compiler, as well as the EJB Container runtime, is CORBA compliant. The EJB Container understands RMI method calls used for EJB but it uses IDL definitions internally to store the interface definitions. Although the java2iop compiler takes the Java interfaces and generates stubs and skeletons from them, you can also generate IDL from your Java interfaces for use in other languages. To a CORBA client, the EJB Container is a CORBA server. The EJB Container tools are CORBA tools that are equally capable of handling EJB.

The Inprise EJB Container is based on JNDI over CosNaming and JTS/OTS. Together, these provide complete support for CORBA.

Container support for different enterprise beans

The principal feature of Inprise’s EJB container is that it supports and manages all types of enterprise beans, including:

- Stateless and stateful session enterprise beans.
- Entity beans with both container-managed and bean-managed persistence.

The EJB Container can contain one or more jar files. Each jar file can contain multiple enterprise beans. Each enterprise bean contains a deployment descriptor, an EJBHome interface, an EJBRemote interface, and the enterprise bean’s implementation code.

What differentiates Inprise’s EJB Container implementation is its ability to scale to support many concurrent clients. There are two aspects to the scalability:

- VisiBroker provides connection management. VisiBroker’s connection management allows a server to support more simultaneous clients than it has TCP connections, by closing TCP connections in LRU fashion.
Why use Inprise’s EJB container?

- The container is itself stateless with respect to clients. This means that the server does not allocate memory for each new session enterprise bean. Thus, the server can support an arbitrary number of stateless session enterprise beans.

Deployment support

The Inprise Container provides compilers for code generation. It also supplies a Verification tool to check the correctness of your beans before they are deployed. Complete EJB 1.1 JNDI naming and XML-based deployment descriptors are supported.

The transaction manager

The transaction manager provides transaction management, including distributed two-phase-commit support. This service complies with both the latest Java Transaction Service (JTS) specification from Sun, and the OTS specification from the OMG. Use the lighter-weight JTS during development or use Inprise’s more robust stand-alone transaction service (ITS) when you deploy and partition your application.

JDBC connection pool and transactional integration

All access to databases using JDBC are routed via Inprise’s implementation of a DataSource object. This DataSource object transparently pools connections to databases and associated JTS-managed global transactions with database transactions demarcated using JDBC.

The naming service

The Naming Service provides distributed naming support. This service complies with both Sun’s Java Naming and Directory Interface (JNDI) specification and the OMG’s CosNaming specification. Similar to the transaction service, you can use the lighter-weight naming service (JNS) during development or use the more robust stand-alone naming service (Inprise’s Naming Service) when you deploy and partition your application.

The JNDI provided with the EJB Container could sit on top of other JNDI implementations, enabling the use of other naming services like LDAP.

Support for security

Integration with Inprise Security Service based on CORBA is available as an add-on.
**The Java database**

The Java database provides a high-performance persistent storage of enterprise bean state in an all-Java relational database.

The Java database can either run in the same process as the EJB container or, to improve performance, it can run in its own process on a separate box. In fact, the Java database is pluggable, which means that the end-user can replace the all-Java version with a custom implementation. For example, one could implement a backing store which uses a commercial database to store session state or entity data, allowing for a highly available implementation.

**Container-managed persistence of entity beans**

Inprise provides a built-in Container-managed Persistence (CMP) engine that provides transparent object-relational (OR) mapping and persistence framework based on pure JDBC. Furthermore, third party tools can be plugged in via an open API.

**Integration with other components**

In a real-world deployment environment, the EJB Container’s four modules run with the following components:

- Clients connecting to the enterprise bean objects. These clients may consist of Java clients using enterprise bean interfaces, or CORBA clients using IDL interfaces. In Inprise’s EJB Container’s server, every enterprise bean object is simultaneously both an enterprise bean object and a CORBA object. The CORBA clients may be implemented in any language supporting CORBA, including C++, Java, and Delphi.

- Databases accessed from the enterprise bean objects. Typically, the enterprise bean objects are entity beans using either bean-managed or container-managed persistence.

- Back-end services including CORBA servers (implemented in C++, Java, Delphi, and so on, running on any CORBA-compliant ORB), other EJB servers (including another Inprise’s server or servers from other vendors which support IIOP), or other legacy services (such as ERP systems, main-frame applications, and so on).
This chapter includes the following major topics:

- Getting started with the examples
- Basics of the EJB container and EJB tools
- Stateless session bean example

## Getting started with the examples

The EJB Container provides a set of examples that walk you through most of the features provided by the release. Note that you must build the examples first before you can run them.

### Overview of the examples

The current release includes the following examples:

- **sort**—A stateless session bean which implements a merge-sort algorithm. This example demonstrates the basics of how you compile and run enterprise beans in the EJB container. The example also illustrates RMI-over-IIOP, and shows how objects are put in the JNDI naming service.

- **cart**—A stateful session bean which implements the canonical shopping cart. This example introduces transaction management through the deployment descriptor. This example also shows that the Inprise Container is backward-compatible with the EJB 1.0 deployment descriptor. It illustrates how to use the dd2xml utility to convert any EJB 1.0-compliant deployment descriptor to an EJB 1.1-compliant XML deployment descriptor.

- **pigs**—Two container-managed persistence entity beans which are related to each other via a one-to-one relationship and a one-to-many relationship. The two types
of entities are Containers and Contained entities. The Containers may contain any number of Contained entities. However, Contained entities are contained in at most one Container.

- bank—Two entity beans and a stateless session bean. The entity beans implement a bank account interface—one entity bean uses container-managed persistence and the other uses bean-managed persistence. The session bean implements a simple bank teller, which allows the user to transfer funds from one account to the other.

- data—This is an entity bean example that illustrates all the data types supported by the container managed persistence (CMP) engine. It enables you to observe the CMP engine’s tuned writing capabilities. This example is built on a container-managed entity bean that has one field for each data type supported by the default CMP engine. In a separate transaction for each operation, the client reads each entity bean field, writes new data to the field, then reads the field again to verify that the field was correctly modified. To observe the engine’s tuned writing capabilities, enable the EJBDebug flag and look at the database updates that are performed. Notice that read-only transactions do not update the database, and that the other transactions perform a tuned write and update the database minimally—they update only the single field that was modified in the given transaction.

- custom_cmp—A container-managed persistence entity bean example that illustrates data being “persisted” by a CORBA server. For this example, a custom implementation of the EJB Container’s CMP SPI is hooked into the container via the deployment descriptor. This CMP Manager delegates calls to a CORBA object, which acts as a transactional data store for the beans data.

Note: You might want to try the sort and cart examples first. They do not require an external database and are thus easier to run.

Note: We are adding new examples and refining existing ones. Please refer to the index HTML page on the examples for the most current information.

**Building the examples**

To build the examples, follow these two simple steps:

1. Set your environment variables. For information about setting environment variables, see the *Inprise Application Server Installation Guide*.
   
   - **CLASSPATH** variable—Both Windows and Solaris users set the **CLASSPATH** variable to dot (.). All required product JAR files are automatically picked up by the launcher. To run the examples that use an external database, the **CLASSPATH** variable must also contain a JDBC driver.
   
   - **PATH** variable—Solaris users should set the **PATH** variable to contain the location of the **vbj** launcher. The DLL is not required for Windows users.

2. Build the examples—For Solaris, go to the examples directory in the EJB Container distribution and type **make**. For Windows NT, type **make_all.bat**.
Makefile

The makefile automatically compiles Java code, runs the java2iiop compiler on the appropriate interfaces, generates a jar file, and then verifies it.

For example, here’s what the makefile does for the sort example:

1. Runs vbjc and compiles all the java code.
2. Runs the java2iiop compiler on the SortHome interface. The compiler generates CORBA (RMI over IIOP) stubs and skeletons for interfaces that you specify. The makefile does not specify the Sort interface. But because it specifies the SortHome interface, it loops through the entire structure and generates all the code upon which SortHome is dependent. In this case, it generates all needed classes.
3. Runs vbjc and compiles two classes—SortHomePOAInvokeHandler and SortPOAInvokeHandler—plus all the classes upon which these two are dependent.
4. Runs the standard JDK “jar” utility to generate an ejb-jar file for a Container to run.
5. Runs the EJB Container tool, VERIFY, to verify the ejb-jar file.

An ejb-jar file contains the following:
- The XML deployment descriptor file or files.
- Class files as byte code. It adds every file that ends with .class.

You can use the Java tool, jar, to look at what is in the jar file:

```
jar tvf beans.jar
```

See Chapter 9, “Deploying Enterprise JavaBeans” for more information on deployment.

Later, when you are creating your own enterprise beans, you can modify the make files and use them with any enterprise bean you create.

Running the examples

To run the examples:

1. Start a VisiBroker Smart Agent. This handles the initial bootstrap issues (for example, how the client locates the naming service, and so on). The Smart Agent is installed with the Application Server. To start a VisiBroker Smart Agent in console mode on a Windows machine:

   ```
   WinNT
   prompt% osagent -C
   ```

   On a Unix machine, start a VisiBroker SmartAgent in console mode as follows:

   ```
   UNIX
   % osagent &
   ```

2. Make sure you are in the directory specific to the example you want to run. For example, to run the sort example, change to the sort directory.
Getting started with the examples

3 Start the EJB server with the following vbjc command. This command actually starts the server and runs the pigs example:

```
prompt% vbj com.inprise.ejb.Container ejbcontainer pigs_beans.jar -jts -jns -jdb -jss
```

The flags (-jts, -jns, -jdb) start up the associated services. You use the -jdb flag to start up the database that comes with the EJB Container if the example uses entity beans. Use the -jss flag for examples with stateful session beans.

The bank example requires access to an external database. Start the EJB server for the bank example with the following command:

```
prompt% vbj -Djdbc.drivers=oracle.jdbc.driver.OracleDriver com.inprise.ejb.Container ejbcontainer bank_beans.jar -jts -jns
```

You should refer to the README file in the bank example directory for a complete explanation of how to build and run that example.

The data and custom_cmp examples require that the server be named “ejbcontainer.” Do not change the name of the server without making corresponding changes to the JDBC URL specified in the deployment descriptor.

The following flags are required for each example:

<table>
<thead>
<tr>
<th>Table 3.1</th>
<th>EJB server flags required for examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example</td>
<td>Required flags</td>
</tr>
<tr>
<td>sort</td>
<td>-jns</td>
</tr>
<tr>
<td>cart</td>
<td>-jns -jts -jss</td>
</tr>
<tr>
<td>bank</td>
<td>-jns -jts</td>
</tr>
<tr>
<td>pigs</td>
<td>-jns -jts -jdb</td>
</tr>
<tr>
<td>data</td>
<td>-jns -jts -jdb</td>
</tr>
<tr>
<td>custom_cmp</td>
<td>-jns -jts</td>
</tr>
</tbody>
</table>

4 Wait until the Container is ready. The following message appears:

```
Container [ejbcontainer] is ready.
```

5 Open another window and start the client. Each example has an associated client program. For example, the sort example client can be run as follows:

```
prompt% vbj SortClient
```

Debug mode

The first time you run the examples, you might want to turn debugging mode on.

To run the container/server with debugging mode enabled:

```
prompt% vbj -DEJBDebug com.inprise.ejb.Container ejbcontainer sort_beans.jar -jns
```

To run the client with debugging mode enabled:

```
prompt% vbj -DEJBDebug SortClient
```
**NotFoundException**

When starting up the client, if you get a **NotFoundException** Exception, check the following things:

- Make sure the Smart Agent is running.
- Make sure the container is running.
- Verify that you started the container in the same directory as you started the client.
- Verify that the environment variable OSAgent_PORT is the same when you start the OSAgent, the Container, and the client.
- Check the command lines.

---

**Basics of the EJB container and EJB tools**

You can start the EJB Container and its services using either a graphical user interface (GUI) or directly from the command line. The command line usage is described here.

**Starting the EJB Container**

To start the EJB Container (or server, if you prefer) and its services, and to set options, use the following command:

```plaintext
```

**Example**

```plaintext
prompt% vbj com.inprise.ejb.Container ejbcontainer sort_beans.jar -jts -jns -jdb -jss
```

**Options**

The following table lists and describes the options available when using the container.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vbj</td>
<td>This is the driver program for the VisiBroker for Java executables. <strong>vbj</strong> sets the OSAgent port from your environment variables. <strong>vbj</strong> may use java instead to start the JDK driver program, provided you set up the environment with the extra command line options.</td>
</tr>
<tr>
<td>com.inprise.ejb.Container</td>
<td>This is the Java main program for the Container.</td>
</tr>
<tr>
<td>server-name</td>
<td>This is the name of the server. (Currently, this is not used by clients; it is only used by the administration tools.)</td>
</tr>
<tr>
<td>ejb-jar-files</td>
<td>Specifies the names of any number of ejb-jar files, each containing one or more enterprise beans. You do have the option <strong>not</strong> to specify any jar files.</td>
</tr>
<tr>
<td>-jts</td>
<td>Starts an in-process implementation of JTS (e.g., OTS). If you do not specify this parameter, you must have an external OTS implementation (that is, ITS) running.</td>
</tr>
</tbody>
</table>
Basics of the EJB container and EJB tools

The EJB Server is a logical collection of “services,” such as a naming service, a transaction service, and so on, that are required by enterprise beans. These services may or may not be collocated in the same address space.

The EJB Container is a specialized service in which enterprise beans are deployed. This specialized service manages their lifecycle, transactions, security, naming, persistence, and so on, according to a specific contract and constrained model delineated by the EJB specification. To do this, the Container uses the generic services provided by the Server.

For example, the following command launches the EJB Container:

```
prompt% vbj com.inprise.ejb.Container test beans.jar -jns -jts -jdb
```

In actuality, with this command you launch a Container that itself starts up in-process the generic services. Logically, you can think of this as an EJB Server with four generic services (in the same address space), plus a fifth “special” service—the EJB Container—that hosts your enterprise beans in the file beans.jar.

In contrast, you could have issued the following set of commands:

```
% vbj com.inprise.ejb.Container test_NS -jns
% vbj com.inprise.ejb.Container test_TS -jns
% vbj com.inprise.ejb.Container test_DB -jdb
% vbj com.inprise.ejb.Container test beans.jar
% vbj com.inprise.ejb.Container test_replica beans.jar
```

You now have exactly the same application with a vastly distributed execution structure. Now your “EJB Server” is a logical collection of generic services and a replicated Container (presumably for fault tolerance).
Diagnostic flags

You can enable various types of diagnostics in either the EJB client or the EJB container. The following flags are most useful:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EJBDebug</td>
<td>Provides diagnostics for the following:</td>
</tr>
<tr>
<td></td>
<td>The EJB Container's EJB state machine</td>
</tr>
<tr>
<td></td>
<td>Message interceptors</td>
</tr>
<tr>
<td></td>
<td>The container-managed persistence</td>
</tr>
<tr>
<td>EJBExceptions</td>
<td>Prints out messages when exceptions are converted from RMI to CORBA. At this time, some information is lost when RMI exceptions are converted to CORBA exceptions, e.g., the stack trace and message are lost. This flag prints the stack trace and message to standard error.</td>
</tr>
<tr>
<td>EJBTimers</td>
<td>Turns on timer diagnostics, which allow the developer to monitor how the container is using the CPU.</td>
</tr>
<tr>
<td>EJBDetailTimers</td>
<td>Turns on timer diagnostics and prints all information generated by the EJBTimers flag, with the addition of method level timing information. This flag allows the developer to monitor how different methods of a bean are using the CPU resources. The console output for this flag requires you to widen your terminal to avoid wrapping of long lines.</td>
</tr>
<tr>
<td>EJBCopyArgs</td>
<td>Causes arguments to be copied in intra-bean in-process calls, which, by default, use pass-by-reference semantics. When this flag is enabled, intra-bean calls use pass-by-value semantics. Keep in mind that numerous enterprise beans will run significantly slower when using pass-by-value semantics. For example, the sort example may run five times slower because of the CPU time spent copying arguments.</td>
</tr>
</tbody>
</table>

Note: The flags are specified as defines to the Java VM. For example, to enable the EJBDebug flag in the previous example, use the command:

```
prompt% vbj -DEJBDebug com.inprise.ejb.Container test beans.jar -jts -jns -jdb
```

Using VisiBroker flags

As well as EJB and RMI specific flags, you can set VisiBroker for Java flags. The two most useful are ORBwarn and ORBdebug, shown in the following table:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vbroker.orb.warn</td>
<td>This flag can be set to a value of 2 or greater to turn on ORB error diagnostics.</td>
</tr>
<tr>
<td>vbroker.orb.debug</td>
<td>This flag can be set to True turn on verbose ORB diagnostics; set the flag to False to turn off verbose diagnostics.</td>
</tr>
<tr>
<td>vbroker.agent.port</td>
<td>This flag explicitly sets the OSAgent port.</td>
</tr>
</tbody>
</table>
This release provides a number of useful tools.

- **tools:deployment descriptor editor**—This tool enables user-friendly entry of deployment descriptor information, from which it generates the XML deployment descriptor file.

- **EJB code generator**—This tool is an enhanced version of the RMI-IIOP compiler. This tool generates the stubs required for the client to access the EJB server, plus it generates the skeletons and other server side code. Please see the Makefiles for details on how to invoke the compiler.

- **EJB code verifier**—This tool (Verify utility) looks at an EJB-JAR file and verifies that the code and deployment descriptor are in compliance with the EJB specification. For example, this tool verifies that the Bean implementation actually implements all of the methods in the Remote interface. The main routine for this tool is in `com.inprise.ejb.util.Verify`. The Verify utility performs its verification using the same DTD generated by the dd2xml tool (described below). By default, Verify validates the XML with the DTD, even if no DTD is present. You can turn this feature off by setting the system property `EJBValidatingParser` to "off", as follows: “EJBValidatingParser=off”.

- **EJB container shutdown utility**—This utility allows the container to be shutdown gracefully. The container will tolerate a hard shutdown, such as a kill signal or the machine cycling. However, when the container is restarted after a hard shutdown, it has to synchronize the jDataStore database, a process which can take more than ten seconds. Using the shutdown utility allows the container to restart significantly faster. The main routine for this tool is in `com.inprise.ejb.util.Shutdown`.

- **EJB 1.0 to EJB 1.1 converter**—This utility, dd2xml, converts a EJB 1.0 serialized deployment descriptor (or a set of descriptors) to an EJB 1.1 XML deployment descriptor. This tool can be used to migrate enterprise beans from other EJB vendors to the Inprise EJB Container. The dd2xml tool generates the DTD embedded within the XML deployment descriptor, which means that any standard XML editor can edit the generated XML deployment descriptor. The cart example walks through the process of taking a 1.0 enterprise bean and converting it to an 1.1 enterprise bean.

The standard jar utility from the JDK can be used to generate EJB Jar Files. Please see the examples for details.

To determine the usage of each tool, run the tool with no command line arguments. For example, running the command:

```
prompt% vbj com.inprise.ejb.util.Verify
```

provides the following usage:

```
Usage: vbj com.inprise.ejb.util.Verify <ejb-server-name> [host-name]
```

**Note**  
The EJB Container product ships with a GUI example that provides an alternative to using these command line tools.
Stateless session bean example

To introduce you to the EJB Container and enterprise beans, we start with a simple client/server application, called SortClient, that performs a sort routine. This sort example shows a stateless session bean which implements a merge-sort algorithm. The example demonstrates the basic steps to compile enterprise beans and run them in the Inprise EJB container. The example also demonstrates RMI-IIOP, and shows how objects are put in the JNDI naming service.

First, we explain the pieces of code written to implement the Sort session bean. The enterprise bean provider normally writes this code. Our sort example provides this code, so you really don’t need to write anything. Then, we show you how to compile and build the enterprise bean, deploy and package the application, build the client, and, lastly, run the client.

The following are the steps for developing this simple client/server application that uses an enterprise bean:

1. Write the code for the enterprise bean. This includes writing the enterprise bean code, and its remote and home interfaces.
2. Write the client code.
3. Compile and build the bean source code and the client.
4. Create the deployment descriptor inside a directory called META-INF.
5. Package the enterprise bean.
6. Deploy the enterprise bean to a JAR file.
7. Run the client.

Write an Enterprise bean

There are three components to all enterprise beans, and these components must be written by the enterprise bean provider. These components are:

- The implementation class—This is the enterprise bean class. It is the implementation of the business logic for the enterprise bean and the functionality defined in the home and remote interfaces. For the sort example, the implementation class is called SortBean.java.

- The Home interface—The home interface specifies operations for creating, finding, and removing enterprise beans. It plays a similar role to a factory object in the common object and CORBA design pattern. The home interface for our enterprise bean is called SortHome.java.

- The Remote interface—The remote interface specifies the business methods available to the client, and which are provided by the enterprise bean. The remote interface for the SortBean is called Sort.java.

You must have all three components to have an enterprise bean on which a client can invoke methods. A client application uses the bean’s home interface to locate the
remote interface for the enterprise bean. Once it has a reference to the remote interface, the client can invoke any of the methods declared in the remote interface. The client is unaware of whether the method is local or remote to his system. To the client, invoking an enterprise bean method is as simple as invoking any local method. The EJB container passes the method invocation from the remote interface to the actual enterprise bean instance, handling all necessary communication protocols, and passes the results back through the remote interface to the client.

Figure 3.1 shows how this might look when the SortClient client wants to invoke the `merge()` method of the SortBean session bean. Notice that SortClient first invokes the `create()` method on the SortHome home interface. SortHome returns a reference to the Sort remote interface. The client invokes the `merge()` method on the Sort remote interface, which passes the request to the underlying SortBean session bean.

**Figure 3.1** Client invoking methods on an enterprise bean

The sort enterprise bean for our example is a stateless session bean. Most session beans are stateful—the bean instance is created at the request of a client within a session, and the bean instance is dedicated to that client for the duration of the session. By definition, a stateful session bean maintains the client’s state for the life of the session.

A stateless session bean instance, on the other hand, is not dedicated to a particular client, nor does it retain any state for the client. When a client invokes a method of a stateless bean, the container assigns to the client an instance of the bean from a cached pool of bean instances. The bean instance maintains the state of the variables involved in the method operation until the method completes. At that point, the instance returns any state that results from the method, retains no state itself, and is no longer associated with the calling client. The container may or may not return the instance to the pool.

**Write the home interface**

The home interface specifies the methods to create, find, and remove the underlying enterprise bean instance. The home interface for a stateless session bean is simpler than for a stateful session bean or an entity bean. The SortHome home interface
Stateless session bean example

extends the `javax.ejb.EJBHome` interface and declares one `create()` method. The `create()` method takes no arguments and throws two exceptions: `RemoteException` and `CreateException`.

The `create()` method returns an object whose type is Sort; this is a reference to the remote interface for the SortBean. Keep in mind that a client does not call a bean instance’s methods directly; rather, the client invokes the methods that are exposed through the bean’s remote interface. The home interface `create()` method thus returns a reference to the bean’s remote interface.

Stateful session beans and entity beans, because they maintain client state, may have multiple methods for creating instances of the bean and they may pass initialization values to their `create()` methods. A stateless session bean, because it does not maintain client state, can have only one `create()` method, and that `create()` method must not have any arguments. The `remove()` method, which removes the bean instance from the available pool, is inherited from the EJBHome interface and you do not need to declare one in the SortHome home interface.

The home interface for SortHome is shown in Code sample 3.1:

**Code sample 3.1**  SortHome home interface

```java
// SortHome.java
public interface SortHome extends javax.ejb.EJBHome {
    Sort create() throws java.rmi.RemoteException, javax.ejb.CreateException;
}
```

Write the remote interface

The remote interface defines and makes public the enterprise bean’s methods that a client may invoke. The remote interface extends the `javax.ejb.EJBObject` interface.

Each method in the Sort remote interface must have a matching method declared in the SortBean implementation class—the methods must have the same name and identical signatures. In addition, methods in the Sort remote interface must throw a `java.rmi.RemoteException` and they must return valid RMI types.

The Sort remote interface declares two methods: `sort()` and `merge()`. Both throw `java.rmi.RemoteException` and both return `Vector` types. Code sample 3.2 shows the code for the remote interface:

**Code sample 3.2**  The sort remote interface

```java
// Sort.java
import java.util.Vector;

public interface Sort extends javax.ejb.EJBObject {
    Vector sort(Vector v, Compare c) throws java.rmi.RemoteException;
    Vector merge(Vector a, Vector b, Compare c) throws java.rmi.RemoteException;
}
```

Write the session bean implementation

Because SortBean is a session bean, it must implement the `javax.ejb.SessionBean` interface. Notice, too, that the SortBean class is declared as public.
Stateless session bean example

The SortBean session bean implementation class defines the business methods that a client might invoke. In our example, SortBean defines the implementations for the `sort()` and `merge()` methods. These are the two business methods that the client will invoke when running the sort/merge application. Notice that the signatures for these methods match the signatures for the `sort()` and `merge()` methods in the Sort remote interface.

A session bean implements an `ejbCreate()` method that matches each `create()` method in the home interface. The SortHome home interface defined one `create()` method with no parameters and which throws a `java.rmi.RemoteException`. The SortBean class implements an `ejbCreate()` method that has no parameters, and which also throws a `java.rmi.RemoteException`. However, unlike the SortHome `create()` method, which returned an object of type `Sort`, `ejbCreate()` returns `void`.

While a session bean may have a public constructor, SortBean does not. SortBean declares four methods in addition to its create and business methods. These methods are: `ejbRemove()`, `ejbPassivate()`, `ejbActivate()`, and `setSessionContext()`. The container handles implementations for these methods; SortBean needs only to declare them as public and set their return types to `void`.

Code sample 3.3 is the code from the SortBean session bean class. For clarity, we do not show the complete implementation of the `sort()` and `merge()` methods, nor do we show the debug statements in the code. You may view the complete source file for SortBean in the Sort example directory.

Code sample 3.3  SortBean session bean implementation

```java
// SortBean.java
import java.util.Vector;

public class SortBean implements javax.ejb.SessionBean {

    public void setSessionContext(javax.ejb.SessionContext sessionContext) {}
    public void ejbCreate() throws java.rmi.RemoteException {}
    public void ejbRemove() throws java.rmi.RemoteException {}
    public void ejbActivate() throws java.rmi.RemoteException {}
    public void ejbPassivate() throws java.rmi.RemoteException {}

    public Vector sort(Vector v, Compare c) throws java.rmi.RemoteException {
        try {
            ... return result;
        }
        catch(java.ejb.CreateException e) {
            throw new java.rmi.ServerException("Could not create sort instance", e);
        }
        catch(java.ejb.RemoveException e) {
            throw new java.rmi.ServerException("Could not remove sort instance", e);
        }
    }

    public Vector merge(Vector a, Vector b, Compare c) throws java.rmi.RemoteException {
        ...
    }
}
```

3-12  Enterprise JavaBeans Programmer's Guide
Stateless session bean example

Write the client code

The client application, SortClient, must first locate the SortHome home interface for the enterprise bean SortBean. It accomplishes this task using the JNDI API. In the code in Code sample 3.4, SortClient creates a JNDI naming context and then uses the JNDI’s lookup method to locate the home interface for “sort”.

Code sample 3.4  Excerpt from SortClient client code

```java
// SortClient.java

public class SortClient {

    public static void main(String[] args) throws Exception {
        javax.naming.Context context;
        { // get a JNDI context using the Naming service
            context = new javax.naming.InitialContext();
        }
        Object objref = (SortHome) context.lookup("sort");
        SortHome home = (SortHome) javax.rmi.PortableRemoteObject.narrow(objref, SortHome.class);
        Sort sort = home.create();
        ... //do the sort and merge work
        sort.remove();
    }
}
```

After the client gets the reference to SortHome, it can invoke the home interface’s create() method to obtain a reference to the remote interface Sort. Once the client has the reference to the SortBean’s remote interface, it can call the remote interface’s sort() and merge() methods just as it would call any other method. Effectively, the remote interface receives the method calls from the client and passes them to the enterprise bean itself.

Build the sort session bean and client

We compile and build the sort session bean and its client using a Makefile. The Makefile does all of the tasks that the java2iiop, jar, and verify utilities do.

Be sure that your environment variables—CLASSPATH and PATH—are properly set. Please see “Building the examples” on page 3-2 for information on how to set these variables.

After you have set the environment variables, compile and build the example as follows:

On Solaris:
```shell
prompt% make
```

Note that Makefile requires a Unix-compatible make utility.

On Windows:
```shell
prompt% make_all
```
Stateless session bean example

The make files are simple enough that they can be modified by hand if need be.

The Makefile for the Sort example is shown in Code sample 3.5:

Code sample 3.5  Makefile for Sort example

```makefile
# Makefile
default: all

include ../Makefile.rules

SRCS = \
  SortClient.java \
  Sort.java \
  SortHome.java \
  SortBean.java

CLASSES = $(SRCS:.java=.class)

beans.jar: $(CLASSES)
  $(JAVA2IIOP) SortHome
  $(JAR) cMf beans.jar META-INF *.class
  $(VERIFY) beans.jar

all: $(CLASSES) beans.jar
```

Create the deployment descriptor

The deployment descriptor is an XML file that defines attributes of the enterprise
bean. You use standard XML editor tools to create this descriptor at this time; later,
there will be a deployment descriptor editor available. For your convenience, correct
XML deployment descriptor files ship with all examples. You can find these XML
files in the META-INF subdirectories within the respective example directories.

Caution  If you use a standard XML editor, you must make sure to comply with the
requirements stated in the EJB 1.1 specification and, so that the enterprise bean can
work with the EJB container, you must use Inprise’s Document Type Definition (DTD).

Note  Currently, you must use a standard XML editor to create a deployment descriptor.
Once the deployment descriptor is part of a JAR file, you can use the deployment
descriptor editor to add to it or modify it.

The deployment descriptor provides information about the deployment of the
enterprise bean code; the Deployer can use the descriptor to customize the bean
itself.

Enterprise beans are meant to be distributed to different applications and used by
these applications without modification of the source code. However, because each
application may use the bean somewhat differently, the bean can be customized for
these differences by modifying the deployment descriptor. Note that the SortBean
descriptor defines the bean’s type (stateless session), specifies the names of its home
Stateless session bean example

interface (SortHome) and its remote interface (Sort), and specifies its transaction type (container managed).

For more information about the deployment descriptor, see Chapter 9, “Deploying Enterprise JavaBeans.”

Run the Sort example

To run the example, you need to have a running VisiBroker Smart Agent to handle the initial bootstrap issues, such as how the client locates the naming service, and so on. (Note that the Smart Agent is installed with the Application Server.)

Start the EJB Container with the following command:

```bash
prompt% vbj com.inprise.ejb.Container ejbcontainer sort_beans.jar -jts -jns
```

Please see “Basics of the EJB container and EJB tools” on page 3-5 for more information about the command line flags.

Run the client program from a terminal window as follows:

```bash
prompt% vbj SortClient
```

The client locates and calls the SortBean to sort a set of varying size vectors. You will see the input—the line marked “in”—and sorted output vectors—the line marked “out”—on the console. Your results should be similar to the following:

```plaintext
in:  [1]
out:  [1]
in:  [1, 2]
out:  [1, 2]
in:  [1, 2, 7, 3, 8, 10, 4, 9, 5, 6]
out:  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
out:  [-98, -94, -87, -56, -51, -43, -35, 13, 15, 22, 26, 27, 29, 50, 55, 57, 57, 66, 78, 81, 84, 85, 89, 91, 91]
in:  [-4, 8.0, 3.0, -6, -4, 4]
out:  [-6, -4, -4, 3.0, 4, 8.0]
in:  [this, is, a, test]
out:  [a, is, test, this]
```
This chapter contains the following major topics:

- “Developing the enterprise bean: Quick steps” provides a quick listing of the key steps for developing an enterprise bean.
- “Using JBuilder” describes how to use the JBuilder development tool.
- “Using an IDE other than JBuilder”
- “Enterprise bean development” provides the details for developing an enterprise bean.
- “Programming restrictions” lists the current known programming restrictions.

### Developing the enterprise bean: Quick steps

In general, the entire process of developing, deploying, and assembling enterprise beans includes the following steps:

- Develop the enterprise bean. To develop an enterprise bean, define three classes—the enterprise bean class itself, and the bean’s home and remote interfaces.

- Deploy the enterprise bean. Deployment involves creating the deployment descriptor, which is an XML file that declares enterprise bean attributes, and bundling the bean’s class files, including the stub files and skeleton files, and deployment descriptor into a JAR file. You need to define the environment properties in the deployment descriptor.

- Assemble the application. This involves the installation of the enterprise bean onto the server and testing connectivity on all tiers. The application assembler combines one or more enterprise beans with some additional components (such as servlets, applets, scripts, and so on). The result of the assembly process can be a complete application or another, more complex enterprise bean consisting of several enterprise beans.

- Manage the enterprise beans and related pieces.
Using JBuilder

JBuilder works seamlessly with the Inprise EJB Container. JBuilder and the Inprise Application Server contain all tools and libraries required to develop and deploy enterprise beans:

- Robust container for running and managing enterprise beans.
- Naming service.
- Transaction service.
- Java database.
- API’s for enterprise bean development.
- An enhanced java-to-IIOP compiler that supports value types and RMI signatures.
- Tools for generating ejb-jar files and for verifying your code.
- Examples that illustrate how to use enterprise beans and the EJB Container.

JBuilder provides wizards and tools for rapid application development (RAD) of enterprise beans. These wizards guide you through the steps to create enterprise beans. The steps are simple and intuitive, have suitable default values, and generate template beans that you can extend to add your business logic.

JBuilder also provides an EJB Interface generation wizard. This wizard generates remote and home interfaces based on your bean’s public methods. There is also a deployment wizard that guides you through the steps to create XML deployment descriptor files and to package the generated stubs into a JAR file.

Using an IDE other than JBuilder

When you use an IDE other than JBuilder, be sure to use the container tools that come with that IDE. You should also verify that the IDE support the correct version of the EJB specification and that is supports the correct version of the EJB API’s.

Be sure that the JDK maps with the supported versions of the Inprise EJB Container. You should check the Inprise installation notes for the versions that the Inprise EJB Container supports with this release.

At deployment time, you should use the tools provided by the Inprise Application Server, because these tools provide editing and modification capabilities for third-party vendor Inprise specific deployment descriptors. The tools verify the deployment descriptor and also verify the bean code.
This section focuses on the tasks that the enterprise bean provider must accomplish to develop an enterprise bean. These tasks are:

- Define and write the enterprise bean class. This is the implementation of the business logic within the enterprise bean.
- Write the enterprise bean’s remote interface class.
- Write the enterprise bean’s home interface class.
- Specify the Primary Key class. The primary key class is needed only for entity beans. You specify the name of the primary key class in the entity bean’s deployment descriptor.

The enterprise bean provider defines the remote and home interfaces and implements the enterprise bean class itself. The remote interface provides the calling interface to the client for the business logic methods implemented by the enterprise bean. The home interface provides methods to locate and create instances of the remote interface.

There is no formal relationship (such as inheritance) among the remote and home interfaces and the enterprise bean implementation. However, methods declared within these three classes follow the rules defined by the EJB specification. For example, you declare application-specific methods (also referred to as business logic) within the enterprise bean class and the bean’s remote interface—these methods must have the same signatures in both classes. The bean implementation must have at least one create method, `ejbCreate()`, but it may have more than one create method with different parameters. There must be matching `create()` methods (with the same parameters) defined in the home interface. The `ejbCreate()` methods in the implementation class return null with container-managed persistence; they return the primary key with bean-managed persistence. However, the corresponding `create()` methods in the home interface return the type of the remote interface.

Note that the `ejbCreate()` method differs somewhat for entity bean implementations. An entity bean implementation can choose not to define an `ejbCreate()` method. Typically, an entity bean omits the `ejbCreate()` method if new entities are added to the database only through legacy applications or database management routines. The `ejbCreate()` method return type is the primary key class. When the `ejbCreate()` method is for an entity bean that implements container-managed persistence, it returns null. When the entity bean implements bean-managed persistence, the `ejbCreate()` method returns the type of the primary key class. These differences are discussed in more detail in Chapter 7, “Writing entity beans.”

The bean provider defines the semantics of the enterprise bean. The container provides the glue between the home and remote interfaces and the enterprise bean implementation class. The container ensures, at compile time or runtime, that the remote interface and the enterprise bean implementation match.
Enterprise bean development

Enterprise bean class inheritance

The enterprise bean implementations and the home and remote interfaces all must inherit from different base EJB classes. The home interface always extends javax.ejb.EJBHome. The remote interface extends javax.ejb.EJBObject. The base classes for the remote and home interfaces inherit from java.rmi.Remote.

A session bean must implement the base class javax.ejb.SessionBean. An entity bean implements the base class javax.ejb.EntityBean. These EJB base classes inherit from javax.ejb.EnterpriseBean, which in turn inherits from java.io.Serializable.

These relationships are shown in Figure 4.1. CartBean, for this illustration, can be either an entity bean or a session bean. Its home interface is CartHome and its remote interface is Cart.

Figure 4.1  EJB inheritance relationship

Remote interface

Every enterprise bean must have a remote interface. The remote interface defines the application-specific operations that a client may invoke. These are the public business methods which can be invoked by clients, and which are actually implemented in the enterprise bean class. Note that clients of an enterprise bean do not access the bean directly; instead, they access its methods through its remote interface.

EJBObject base class

An enterprise bean’s remote interface is a public Java interface extending the interface javax.ejb.EJBObject, which is the base interface of all remote interfaces. Its code is shown in Code sample 4.1.
Developing enterprise beans

Enterprise bean development

Code sample 4.1  Base interface EJBObject for remote interfaces

```java
package javax.ejb;
public interface EJBObject extends java.rmi.Remote {
    public EJBHome getEJBHome() throws java.rmi.RemoteException;
    public Object getPrimaryKey() throws java.rmi.RemoteException;
    public void remove() throws java.rmi.RemoteException, java.rmi.RemoveException;
    public Handle getHandle() throws java.rmi.RemoteException;
    boolean isIdentical (EJBObject p0) throws java.rmi.RemoteException;
}
```

The method `getEJBHome()` allows you to get the associated home interface. For entity beans, you can get the primary key of the entity bean using the method `getPrimaryKey()`. The `remove()` method deletes the enterprise bean; this is explained in the context of the life cycle of the various types of enterprise beans. The method `getHandle()` returns a persistent handle to the enterprise bean instance. The `isIdentical()` method allows you to compare enterprise beans.

Method requirements

All remote interface methods must be declared public and must throw the exception `java.rmi.RemoteException`. In addition, the arguments and return values for all methods defined in the remote interface must be of valid types for RMI-IIOP. There must be a matching method in the enterprise bean’s class for each method in the remote interface. The matching method must have the same name, the same number and types of arguments, the same return type, and it must throw the same exceptions.

Code sample 4.2 shows the code for a remote interface `Atm` for an ATM session bean that declares a business method called transfer. The EJB specification requires that you include the portions of the code highlighted in bold. The remote interface must extend `javax.ejb.EJBObject`. Declare in the remote interface every business method from the enterprise bean that you want accessible to a client. The `transfer()` method throws two exceptions; the `InsufficientFundsException` is an application-specific exception.

Code sample 4.2  Example remote interface

```java
public interface Atm extends javax.ejb.EJBObject {
    public void transfer(String source, String target, float amount)
        throws java.rmi.RemoteException, InsufficientFundsException;
}
```

The same rules apply to the definition of the remote interface for both session and entity beans.

Home interface

An enterprise bean’s home interface controls the life cycle of a bean. It provides operations to create, find, and remove an enterprise bean object, that is, an instance of
an enterprise bean. Session and entity beans have different life cycles. Consequently, their home interfaces must define different methods.

The enterprise bean provider must define the home interface. However, the container implements the home interface.

Similar to the remote interface, the methods in the home interface must define arguments and return values that are valid RMI-IIOP types. All methods must also include the `java.rmi.RemoteException` in their throws clauses.

The home interface must define one or more `create()` methods. Each such `create()` method must be named “create” and its arguments—both type and number of arguments—must match the corresponding `ejbCreate()` method in the bean class. Note that the return type is different between the home interface `create()` method and the `ejbCreate()` method in the bean class.

The home interface for an entity bean also contains find methods. This is discussed in “Entity bean home interface” on page 4-8.

**EJBHome base class**

Every home interface extends the interface `javax.ejb.EJBHome`. Code sample 4.3 shows the definition of the `javax.ejb.EJBHome` interface:

**Code sample 4.3 EJBHome interface definition**

```java
package javax.ejb;

public interface EJBHome extends java.rmi.Remote {
    void remove(Handle handle) throws java.rmi.RemoteException, RemoveException;
    void remove(Object primaryKey) throws java.rmi.RemoteException, RemoveException;
    EJBMetaData getEJBMetaData() throws RemoteException;
    HomeHandle getHomeHandle() throws RemoteException;
}
```

There are two `remove()` methods provided to remove enterprise bean instances. The first `remove()` method removes an instance identified by a handle. The second `remove()` method removes an instance identified by a primary key.

A handle is a unique enterprise bean object identifier that is serializable. A handle has the same lifetime as the enterprise bean object it is referencing. In the case of an entity object, a client can use a serialized handle to re-obtain a reference to the entity object that it identifies. The handle can be valid for multiple instantiations of an enterprise bean object; for example, it is valid even if the server that hosted the enterprise bean object crashes, or when the enterprise bean object moves between different servers and machines. A serialized handle is a concept very similar to a stringified CORBA object reference.

The second `remove()` operation on the `javax.ejb.EJBHome` interface uses a primary key to determine the object to be removed. A primary key may be any Java type that extends the `java.util.EJBObject` and `java.lang.Serializable` interfaces.

Primary keys are the main means of identification for entity beans. A primary key is typically a key in a database table which uniquely defines the data represented by the entity object.
The method `getEJBMetaData()` returns the metadata interface for the enterprise bean object. This interface allows the client to obtain metadata information about the enterprise bean. It is intended to be used by development tools that build applications that use deployed enterprise beans. The `javax.ejb.EJBMetaData` interface provides methods to obtain the `javax.ejb.EJBHome` interface, the type (class) of the home and the remote interfaces, as well as the type of the primary key. It also provides a method `isSession()` to determine if the object hosted by this home interface is a session or entity bean. The `isStatelessSession()` method indicates whether or not the session bean is stateful or stateless. Code sample 4.4 shows the definition of the `javax.ejb.EJBMetaData` interface.

**Code sample 4.4  EJBMetaData interface definition**

```java
package javax.ejb;

public interface EJBMetaData {
    EJBHome getEJBHome();
    Class getHomeInterfaceClass();
    Class getRemoteInterfaceClass();
    Class getPrimaryKeyClass();
    boolean isSession();
    boolean isStatelessSession();
}
```

**Session bean home interface**

Recall from “Session beans” on page 2-3 that an instance of a session bean has only one client. That is, when a client creates a session bean, that session bean instance exists only for the client that created it. (In this case, we are referring to a stateful session bean that maintains a “conversational” state for its client. A stateless session bean, because it does not maintain state across method invocations, may be used for multiple clients.)

The home interface functions as a session bean factory because it includes the definition of one or more `create()` methods. The EJB specification defines the following naming convention for each `create()` method:

- It returns the remote interface type of the session bean.
- The method name is always “create.”
- There must be a `create()` method that matches each `ejbCreate()` method in the session bean class. The number and types of arguments for each `create()` method must match its corresponding `ejbCreate()` method in the session bean class.
- It must throw the exception `java.rmi.RemoteException`.
- It must throw the exception `javax.ejb.CreateException`.
- The parameters of the `create()` method are used to initialize the new session bean object.

Code sample 4.5 shows different create methods of a session home interface. Required parts are shown in bold.
Code sample 4.5  Examples of create() methods

```java
public interface AtmHome extends javax.ejb.EJBHome {

    Atm create() throws java.rmi.RemoteException, javax.ejb.CreateException;

    Atm create(Profile preferredProfile) throws java.rmi.RemoteException, javax.ejb.CreateException;
}
```

Note that the session home interface does not define finder methods to locate objects, because a stateful session bean is intended to be used only by the client that created it. No client should have a need to locate a session bean instance that it did not create.

Entity bean home interface

The entity bean home interface provides create methods in the exact same fashion as the session bean home interface. In addition, the entity bean home interface provides finder methods so that clients can locate and use entity objects. Finder operations are necessary because entity beans are long lived and can be used by multiple clients. For most applications, an entity bean instance already exists and a client just has to find the one it wants to invoke.

An entity bean home interface must provide the default finder method, `findByPrimaryKey(primaryKey)`. This method allows a client to locate an entity object using a primary key. The method has a single argument, the primary key, and its return type is the entity bean’s remote interface. The type of the primary key can be any Java type which extends the Java `Object` class. In the deployment description, you tell the container the type of the primary key. Note that the `findByPrimaryKey()` method, by definition, always returns a single entity object, while other finder methods may return collections of entity objects.

Code sample 4.6  `findByPrimaryKey()` method

```java
<entity bean's remote interface> findByPrimaryKey(
    <primary key type> key )
throws java.rmi.RemoteException, FinderException;
```

The home interface can define additional finder methods. Each finder method must have a corresponding implementation in the entity bean class. Each home interface finder method must adhere to the following conventions:

- The return type is the remote interface type or, for finder methods that return more than one entity object, a collection type that has the remote interface type as the content type. Valid Java collection types are the `java.utilEnumeration` interface (for JDK 1.1 implementations) and the `java.util.Collection` interface (for Java 2 implementations).

- The finder method always starts with the prefix “find.” The corresponding finder method in the entity bean class begins with the prefix “ejbFind.”

- It must throw the exception `java.rmi.RemoteException`.

- It must throw the exception `javax.ejb.FinderException`.

Enterprise bean development

Code sample 4.5  Examples of create() methods

```java
public interface AtmHome extends javax.ejb.EJBHome {

    Atm create() throws java.rmi.RemoteException, javax.ejb.CreateException;

    Atm create(Profile preferredProfile) throws java.rmi.RemoteException, javax.ejb.CreateException;
}
```

Note that the session home interface does not define finder methods to locate objects, because a stateful session bean is intended to be used only by the client that created it. No client should have a need to locate a session bean instance that it did not create.

Entity bean home interface

The entity bean home interface provides create methods in the exact same fashion as the session bean home interface. In addition, the entity bean home interface provides finder methods so that clients can locate and use entity objects. Finder operations are necessary because entity beans are long lived and can be used by multiple clients. For most applications, an entity bean instance already exists and a client just has to find the one it wants to invoke.

An entity bean home interface must provide the default finder method, `findByPrimaryKey(primaryKey)`. This method allows a client to locate an entity object using a primary key. The method has a single argument, the primary key, and its return type is the entity bean’s remote interface. The type of the primary key can be any Java type which extends the Java `Object` class. In the deployment description, you tell the container the type of the primary key. Note that the `findByPrimaryKey()` method, by definition, always returns a single entity object, while other finder methods may return collections of entity objects.

Code sample 4.6  `findByPrimaryKey()` method

```java
<entity bean's remote interface> findByPrimaryKey(
    <primary key type> key )
throws java.rmi.RemoteException, FinderException;
```

The home interface can define additional finder methods. Each finder method must have a corresponding implementation in the entity bean class. Each home interface finder method must adhere to the following conventions:

- The return type is the remote interface type or, for finder methods that return more than one entity object, a collection type that has the remote interface type as the content type. Valid Java collection types are the `java.utilEnumeration` interface (for JDK 1.1 implementations) and the `java.util.Collection` interface (for Java 2 implementations).

- The finder method always starts with the prefix “find.” The corresponding finder method in the entity bean class begins with the prefix “ejbFind.”

- It must throw the exception `java.rmi.RemoteException`.

- It must throw the exception `javax.ejb.FinderException`. 
The throws clause of the find() method in the home interface must match the throws clause of the corresponding ejbFind() method in the entity bean class. Additionally, an entity bean's home interface can provide one or more create() methods. These methods return an object reference of the bean's remote interface type. Their arguments are application specific.

The create methods in the home interface must comply with the following rules:

- They must throw the exception java.rmi.RemoteException.
- They must throw the exception javax.ejb.CreateException.
- They return the remote interface type of the entity bean.
- The method name is always "create."
- There must be a create() method that matches each ejbCreate() method in the entity bean class. The number and types of arguments for each create() method must match its corresponding ejbCreate() method in the entity bean class.
- The exceptions in the throws clause of the home interface create() method must include all the exceptions thrown by the corresponding ejbCreate() and ejbPostCreate() methods in the entity bean class. That is, the set of exceptions for the create() method must be a superset of the union of exceptions for both the ejbCreate() and ejbPostCreate() methods. The return type of the ejbCreate() method is the primary key class.
- The parameters of the create() method are used to initialize the new entity bean object.

Code sample 4.7 shows different types of finder and create methods. Required parts are shown in bold.

**Code sample 4.7  Create and find methods**

```java
public interface AccountHome extends javax.ejb.EJBHome {

    Account create( String accountId )
    throws java.rmi.RemoteException, javax.ejb.CreateException;

    Account create( String accountId, float initialBalance )
    throws java.rmi.RemoteException, javax.ejb.CreateException;

    Account findByPrimaryKey( String key )
    throws java.rmi.RemoteException, javax.ejb.FinderException;

    Enumeration findBySocialSecurityNumber( String socialSecurityNumber )
    throws java.rmi.RemoteException, javax.ejb.FinderException;
}
```
Enterprise bean implementation

The enterprise bean implementation contains all of the application-specific semantics. It is the principal work of a bean provider. A session bean class implements the `javax.ejb.SessionBean` interface, while an entity bean class implements the `javax.ejb.EntityBean` interface. Both of these interfaces extend the enterprise bean base interface, `javax.ejb.EnterpriseBean`.

The implementation of a session bean differs from that of an entity bean. Refer to Chapter 6, “Writing session beans,” for details on how to implement a session bean. Refer to Chapter 7, “Writing entity beans,” for details on the implementation of an entity bean.

Enterprise bean interface

The enterprise bean interface `javax.ejb.EnterpriseBean` defines a common base interface and extends the `java.io.Serializable` interface. It defines no methods. Its definition is shown in Code sample 4.8.

Code sample 4.8 EnterpriseBean interface definition

```java
package javax.ejb;

public interface EnterpriseBean extends java.io.Serializable {}
```

Handles

A handle is an abstract way for a remote object to reference an enterprise bean object or the enterprise bean object’s home interface. The `javax.ejb.Handle` interface, which must be implemented by all enterprise bean object handles, provides a persistent reference to the enterprise bean object. Its definition is shown in Code sample 4.9:

Code sample 4.9 Handle interface

```java
public interface javax.ejb.Handle extends java.io.Serializable {
    public EJBObject getEJBObject() throws java.rmi.RemoteException;
}
```

A handle can be used to persistently store a reference to an enterprise bean object by serializing an instance of the class that implements the handle interface. This is similar to a stringified CORBA IOR. The `javax.ejb.Handle` interface extends the interface `java.io.Serializable`.

There is also a `javax.ejb.HomeHandle` interface that provides a persistent reference to an enterprise bean’s home interface. The `javax.ejb.HomeHandle` interface is implemented by all home object handles. Its definition is shown in Code sample 4.10:

Code sample 4.10 HomeHandle interface

```java
public interface HomeHandle extends java.io.Serializable {
    public EJBHome getEJBHome() throws RemoteException;
}
```

Handle implementations are typically provided by a container.
Programming restrictions

Below is a list of programming restrictions defined in the EJB 1.1 specification.

- An enterprise bean is not allowed to manage threads and thread groups. It should not start new threads or resume suspended threads, nor should it terminate or suspend the running thread. In addition, an enterprise bean should not change a thread’s priority or its name.

- An enterprise bean is not allowed to use read/write static fields. Using read-only static fields is allowed. Therefore, all static fields must be declared as final.

- An enterprise bean is not allowed to use thread synchronization primitives to synchronize the execution of multiple instances.

- An enterprise bean must not use the Java AWT functionality to output information to a display or to accept information from a keyboard.

- An enterprise bean should not use the `java.io` package to access files and directories in the file system.

- An enterprise bean should refrain from using sockets; in particularly, the bean should not listen on a socket, accept connections on a socket, or use a socket for multicast. In addition, it should not set the socket factory used by `ServerSocket`, `Socket`, or the stream handler factory used by URL.

- An enterprise bean must not access classes or packages, or attempt to obtain information about classes, in a manner normally disallowed by the Java programming language or if the classes are normally unavailable to the enterprise bean.

- An enterprise bean must not access the runtime environment functions normally handled by the container—create a class loader or access or change its context, set or create a security manager, stop the JVM, change input, output, or error streams.

- An enterprise bean must not obtain security policy information for a particular code source as this would compromise security.

- An enterprise bean must not load a native library.

- An enterprise bean must not define a class in a package, as this is a function reserved for the container for security reasons.

- An enterprise bean must not use the subclass and object substitution features of the Java Serialization Protocol.

- An enterprise bean should be careful if passing `this` as an argument or method result. It is safer for the bean to pass the result of `SessionContext.getEJBObject()` or `EntityContext.getEJBObject()`.

- An enterprise bean is not allowed to access or modify security configuration objects. For example, it is not allowed to change its `java.security.Identity`. Any such attempt will result in the `java.security.SecurityException` being thrown.
Writing enterprise bean clients

This chapter includes information about the following major topics:

- Client view of an enterprise bean
- Managing transactions
- Getting information about an enterprise bean
- Support for JNDI
- EJB to CORBA mapping

Client view of an enterprise bean

A client of an enterprise bean is an application—a stand-alone application, servlet, or applet—or another enterprise bean. In all cases, the client must do the following things to use an enterprise bean:

- Locate the bean’s home interface. The EJB specification states that the client should use the JNDI (Java Naming and Directory Interface) API to locate home interfaces.
- Obtain a reference to an enterprise bean object’s remote interface. This involves using methods defined on the bean’s home interface. You can either create a session bean, or you can create or find an entity bean.
- Invoke one or more methods defined by the enterprise bean. A client does not directly invoke the methods defined by the enterprise bean. Instead, the client invokes the methods on the enterprise bean object’s remote interface. The methods defined in the remote interface are the methods that the enterprise bean has exposed to clients.
Client view of an enterprise bean

Initializing the client

The SortClient application imports the necessary JNDI classes and the SortBean home and remote interfaces. (See Code sample 5.1 on page 5-2.) The client uses the JNDI API to locate an enterprise bean’s home interface.

Locating the home interface

A client locates an enterprise bean’s home interface using JNDI, as shown in Code sample 5.1. The client first needs to obtain a JNDI initial naming context. The code instantiates a new javax.naming.Context object, which in our example it calls initialContext. Then, the client uses the context lookup() method to resolve the name to a home interface. Note that the initialization of the initial naming context factory is EJB container/server specific.

The context’s lookup() method returns an object of type java.lang.Object. Your code must cast this returned object to the expected type. Code sample 5.1 shows a portion of the client code for the sort example. The main() routine begins by using the JNDI naming service and its context lookup method to locate the home interface. You pass the name of the remote interface, which in this case is sort, to the context.lookup() method. Notice that the program eventually casts the results of the context.lookup() method to SortHome, the type of the home interface.

Code sample 5.1 Using JNDI to locate a bean’s home interface

// SortClient.java

import javax.naming.InitialContext;
import SortHome; // import the bean’s home interface
import Sort; // import the bean’s remote interface

public class SortClient {

    public static void main(String[] args) throws Exception {
        javax.naming.Context context;
        { // get a JNDI context using the Naming service
            context = new javax.naming.InitialContext();
        }
        Object objref = context.lookup("sort");
        SortHome home = (SortHome) javax.rmi.PortableRemoteObject.narrow(objref,
            SortHome.class);
        Sort sort = home.create();
        ... //do the sort and merge work
        sort.remove();
    }
}

The main() routine of the client program throws the generic exception Exception. When coded this way, the SortClient program does not have to catch any exceptions that might occur, though if an exception occurs it will terminate the program.
**Obtaining the remote interface**

Now that we have obtained the home interface of an enterprise bean we can get a reference to the enterprise bean’s remote interface. To do this, we use the home interface’s create or finder methods. The exact method to invoke depends on the type of the enterprise bean and the methods the enterprise bean provider has defined in the home interface.

For example, Code sample 5.1 shows how SortClient obtains a reference to the Sort remote interface. Once SortClient obtains the reference to the home interface and casts it to its proper type (SortHome), then the code can create an instance of the bean and call its methods. It calls the home interface’s `create()` method, which returns a reference to the bean’s remote interface, Sort. (Because SortBean is a stateless session bean, its home interface has only one `create()` method and that method by definition takes no parameters.) SortClient can then call the methods defined on the remote interface—`sort()` and `merge()`—to do its sorting work. When the work finishes, the client calls the remote interface’s `remove()` method to remove the instance of the enterprise bean.

**Session beans**

A client obtains a reference to a session bean’s remote interface by calling one of the create methods on the home interface.

All session beans must have at least one `create()` method. A stateless session bean must have only one `create()` method, and that method must have no arguments. A stateful session bean can have one `create()` method, and may have additional `create()` methods whose parameters vary. If a `create()` method does have parameters, the values of these parameters are used to initialize the session bean.

The default `create()` method has no parameters. For example, the sort example uses a stateless session bean. It has, by definition, one `create()` method that takes no parameters:

```java
Sort sort = home.create();
```

The cart example, on the other hand, uses a stateful session bean, and its home interface, CartHome, implements more than one `create()` method. One of its `create()` methods takes three parameters, which together identify the purchaser of the cart contents, and returns a reference to the Cart remote interface. The CartClient sets values for the three parameters—`cardHolderName`, `creditCardNumber`, and `expirationDate`—then calls the `create()` method. This is shown in Code sample 5.2:

**Code sample 5.2 Calling a create method**

```java
Cart cart;
{
    String cardHolderName = "Jack B. Quick";
    String creditCardNumber = "1234-5678-9012-3456";
    Date expirationDate = new GregorianCalendar(2001, Calendar.JULY, 1).getTime();
    cart = home.create(cardHolderName, creditCardNumber, expirationDate);
}
```

Session beans do not have finder methods.
Client view of an enterprise bean

Entity beans
A client obtains a reference to an entity object either through a find operation or a create operation. Recall that an entity object represents some underlying data stored in a database. Because the entity bean represents persistent data, entity beans typically exist for quite a long time; certainly for much longer than the client applications that call them. Thus, a client most often needs to find the entity bean that represents the piece of persistent data of interest, rather than creating a new entity object, which would create and store new data in the underlying database.

A client uses a find operation to locate an existing entity object, such as a specific row within a relational database table. That is, find operations locate data entities that have previously been inserted into data storage. The data may have been added to the data store by an entity bean or it may have been added outside of the EJB context, such as directly from within the database management system (DBMS). Or, in the case of legacy systems, the data may have existed prior to the installation of the EJB container.

A client uses an entity bean object’s create() method to create a new data entity that will be stored in the underlying database. An entity bean’s create() method inserts the entity state into the database, initializing the entity’s variables according to the values in the create() method’s parameters. A create() method for an entity bean always returns the remote interface, but the corresponding ejbCreate() method returns primary key of the entity instance.

Every entity bean instance must have a primary key that uniquely identifies it. An entity bean instance may also have secondary keys that can be used to locate a particular entity object.

Find methods and primary key class
The default find method for an entity bean is the findByPrimaryKey() method, which locates the entity object using its primary key value. Its signature is as follows:

```java
<remote interface> findByPrimaryKey(<key type> primaryKey)
```

Every entity bean must implement a findByPrimaryKey() method. The primaryKey parameter is a separate primary key class that is defined in the deployment descriptor. The key type is the type for the primary key, and it must be a legal value type in RMI-IIOP. The primary key class can be any class—a Java class or a class you’ve written yourself.

For example, we might have an Account entity bean for which we have defined the primary key class AccountPK. AccountPK is a String type, and it holds the identifier for the Account bean. We can obtain a reference to a specific Account entity bean instance by setting the AccountPK to the account identifier and invoking the findByPrimaryKey() method, as shown in Code sample 5.3.

**Code sample 5.3** Finding an entity bean instance using the primary key

```java
AccountPK accountPK = new AccountPK("1234-56-789");
Account source = accountHome.findByPrimaryKey(accountPK);
```

The bean provider can define additional finder methods that a client can use.
Create and remove methods
A client can also create entity beans using create methods defined in the home interface. When a client invokes a `create()` method for an entity bean, the new instance of the entity object is saved in the data store. The new entity object always has a primary key value that is its identifier. Its state may be initialized to values passed as parameters to the `create()` method.

Keep in mind that an entity bean exists for as long as data is present in the database. The life of the entity bean is not bound by the client’s session. The entity bean can be removed by invoking one of the bean’s `remove()` methods—these methods remove the bean and the underlying representation of the entity data from the database. It is also possible to directly delete an entity object, such as by deleting a database record using the DBMS or with a legacy application.

Invoking methods
Once the client has obtained a reference to the bean’s remote interface, the client can invoke the methods defined in the remote interface for this enterprise bean. The methods pertaining to the bean’s business logic are of most interest to the client. There are also methods for getting information about the bean and its interfaces, getting the bean object’s handle, testing if one bean is identical to another bean, and methods for removing the bean instance.

Code sample 5.4 illustrates how a client calls methods of an enterprise bean, in this case, a cart session bean. We pick up the client code from the point where it has created a new session bean instance for a card holder and retrieved a Cart reference to the remote interface. At this point, the client is ready to invoke the bean methods.

First, the client creates a new book object, setting its title and price parameters. Then, it invokes the enterprise bean business method `addItem()` to add the book object to a shopping cart. The `addItem()` method is defined on the CartBean session bean, and is made public through the Cart remote interface. The client adds other items (not shown here), then calls its own `summarize()` method to list the items in the shopping cart. This is followed by the `remove()` method to remove the bean instance. Notice that a client calls the enterprise bean methods in the same way that it invokes any method, such as its own method `summarize()`.

**Code sample 5.4 Invoking bean methods**

```java
Cart cart;
{
    ...
    // obtain a reference to the bean’s remote interface
    cart = home.create(cardHolderName, creditCardNumber, expirationDate);
}
// create a new book object
Book knuthBook = new Book("The Art of Computer Programming", 49.95f);
// add the new book item to the cart
    cart.addItem(knuthBook);
    ...
    // list the items currently in the cart
    summarize(cart);
    cart.removeItem(knuthBook);
    ...
```
Client view of an enterprise bean

Removing bean instances

The `remove()` method operates differently for session beans than for entity beans. Because a session object exists for one client and is not persistent, a client of a session bean should call the `remove()` method when finished with a session object. There are two `remove()` methods available to the client: the client can remove the session object with the `javax.ejb.EJBObject.remove()` method, or the client can remove the session handle with the `javax.ejb.EJBHome.remove(Handle handle)` method. (See “Using a bean’s handle” on page 5-6 for more information on handles.)

While it is not required that a client remove a session object, it is considered to be good programming practice. If a client does not remove a stateful session bean object, the container will eventually remove the object after a certain time, specified by a timeout value. The timeout value is a deployment property. However, a client can also keep a handle to the session for future reference.

Clients of entity beans do not have to deal with this problem as entity beans are only associated with a client for the duration of a transaction and the container is in charge of their life cycles, including their activation and passivation. A client of an entity bean calls the bean’s `remove()` method only when the entity object is to be deleted from the underlying database.

Using a bean’s handle

A handle is another way to reference an enterprise bean. A handle is a serializable reference to a bean. You can obtain a handle from the bean’s remote interface. Once you have the handle, you can write it to a file (or other persistent storage). Later, you can retrieve the handle from storage and use it to reestablish a reference to the enterprise bean.

However, you can only use the remote interface handle to recreate the reference to the bean; you cannot use it to recreate the bean itself. If another process has removed the bean, or the system crashed or shutdown and removed the bean instance, then an exception is thrown when the client application tries to use the handle to reestablish its reference to the bean.

When you are not sure that the bean instance will still be in existence, rather than using a handle to the remote interface, you can store the bean’s home handle and recreate the bean object later by invoking the bean’s create or find methods.

After the client creates a bean instance, it can use the `getHandle()` method to obtain a handle to this instance. Once it has the handle, it can write it to a serialized file. Later, the client program can read the serialized file, casting the object that it reads in to a `Handle` type. Then, it calls the `getEJBObject()` method on the handle to obtain the bean reference, casting the results of `getEJBObject()` to the correct type for the bean.

To illustrate, the CartClient program might do the following to utilize a handle to the CartBean session bean:
Managing transactions

A client program can manage its own transactions rather than letting the enterprise bean (or container) manage the transaction. A client that manages its own transaction does so in exactly the same manner as a session bean than manages its own transaction.

When a client manages its own transactions, it is responsible for delimiting the transaction boundaries. That is, it must explicitly start the transaction and end (commit or roll back) the transaction.

A client uses the `javax.transaction.UserTransaction` interface to manage its own transactions. It must first obtain a reference to the `UserTransaction` interface, using JNDI to do so. Once it has the `UserTransaction` context, the client uses the `UserTransaction.begin()` method to start the transaction, followed later by the `UserTransaction.commit()` method to commit and end the transaction (or...
Getting information about an enterprise bean

UserTransaction.rollback() to rollback and end the transaction). In between, the client
does its queries and updates.

Code sample 5.6 shows the code that a client would implement to manage its own
transactions. The parts that pertain specifically to client-managed transactions are
highlighted in bold.

**Code sample 5.6**  
Client managed transaction

```java
import javax.naming.InitialContext;
import javax.transaction.UserTransaction;
...
public class clientTransaction {

    public static void main (String[] argv) {
        UserTransaction ut = null;
        InitialContext initContext = new InitialContext();
        ...
        ut = (UserTransaction)initContext.lookup("java:comp/UserTransaction");
        // start a transaction
        ut.beginTransaction();
        // do some transaction work
        ...  
        // commit or rollback the transaction
        ut.commit(); // or ut.rollback();
        ...
    }
}
```

---

**Getting information about an enterprise bean**

Information about an enterprise bean is referred to as metadata. A client can obtain
metadata about a bean using the enterprise bean’s home interface `getMetaData()`
method.

The `getMetaData()` method is most often used by development environments and tool
builders that need to discover information about an enterprise bean, such as for
linking together beans that have already been installed. Scripting clients might also
want to obtain metadata on the bean.

Once the client retrieves the home interface reference, it can call the `getEJBMetaData()`
method on the home interface. Then, the client can call the `EJBMetaData` interface
methods to extract such information as:

- The bean’s `EJBHome` home interface, using `EJBMetaData.getEJBHome()`.
- The bean’s home interface class object, including its interfaces, classes, fields, and
  methods, using `EJBMetaData.getHomeInterfaceClass()`.
- The bean’s remote interface class object, including all class information, using
  `EJBMetaData.getRemoteInterfaceClass()`.
- The bean’s primary key class object, using `EJBMetaData.getPrimaryKeyListener()`.
Support for JNDI

- Whether the bean is a session bean or an entity bean, using `EJBMetaData.isSession()`. The method returns true if this is a session bean.
- Whether a session bean is stateless or stateful, using `EJBMetaData.isStatelessSession()`. The method returns true if the session bean is stateless.

Refer to “EJBHome base class” on page 4-6 to see the definition of the `EJBMetaData` interface.

Support for JNDI

The EJB specification defines the JNDI API for locating home interfaces. JNDI is implemented on top of other services, including CORBA's Naming Service, LDAP/X.500, flat files, and proprietary directory services. Figure 5.1 illustrates the different implementation choices. Typically, the EJB server provider selects a particular implementation of JNDI.

![Figure 5.1: JNDI and sample implementation](image)

The technology implemented beneath JNDI is of no concern to the client. The client needs to use only the JNDI API.

EJB to CORBA mapping

There are a number of aspects to the relationship between CORBA and Enterprise JavaBeans. Three important ones are the implementation of an EJB container/server with an ORB, the integration of legacy systems into an EJB middle tier, and the access of enterprise beans from non-Java components, specifically clients. The EJB specification is currently only concerned with the third aspect.

CORBA is a very suitable and natural platform on which to implement an EJB infrastructure. CORBA addresses all of the concerns of the EJB specification with the CORBA Core specification or the CORBA Services:

- **Support for distribution.** CORBA Core and CORBA Naming Service
- **Support for transactions.** CORBA Object Transaction Service


- **Support for security.** CORBA Security Specification, including IIOP-over-SSL

  Additionally, CORBA allows the integration of non-Java components into an application. These components can be legacy systems and applications, plus different kinds of clients. Back-end systems can be easily integrated using OTS and any programming language for which an IDL mapping exists. This requires an EJB container to provide OTS and IIOP APIs.

  The EJB specification is concerned with the accessibility of enterprise beans from non-Java clients and provides an EJB to CORBA mapping. The goals of the EJB/CORBA mapping are:

  - Supporting interoperability between clients written in any CORBA-supported programming language and enterprise beans running on a CORBA-based EJB server.
  - Enabling client programs to mix and match calls to CORBA objects and enterprise beans within the same transaction.
  - Supporting distributed transactions involving multiple enterprise beans running on CORBA-based EJB servers provided by different vendors.

  The mapping is based on the Java-to-IDL mapping. The specification includes the following parts: mapping of distribution-related aspects, the mapping of naming conventions, the mapping of transactions, and the mapping of security. We explain each of these aspects in the following sections. Since the mapping uses new IDL features introduced by the OMG’s Object-by-Value specification, interoperability with other programming languages requires CORBA 2.3-compliant ORBs.

### Mapping for distribution

An enterprise bean has two interfaces that are remotely accessible: the remote interface and the home interface. Applying the Java/IDL mapping to these interfaces results in corresponding IDL specifications. The base classes defined in the EJB specification are mapped to IDL in the same manner.

For example, let’s examine the IDL interface for an ATM enterprise session bean that has methods to transfer funds between accounts and throws an insufficient funds exception. By applying the Java/IDL mapping to the home and the remote interface, we get the following IDL interface, shown in Code sample 5.7.

**Code sample 5.7**  IDL interface

```idl
module transaction {
  module ejb {

    valuetype InsufficientFundsException : ::java::lang::Exception {};

    exception InsufficientFundsEx {
      ::transaction::ejb::InsufficientFundsException value;
    };

    interface Atm : ::javax::ejb::EJBObject {
```

---

5-10  *Enterprise JavaBeans Programmer’s Guide*
void transfer (in string arg0, in string arg1, in float arg2)
    raises (::transaction::ejb::InsufficientFundsEx);
}

interface AtmHome : ::javax::ejb::EJBHome {
    ::transaction::ejb::Atm create ()
    raises (::javax::ejb::CreateEx);
}

Mapping for naming

A CORBA-based EJB runtime environment that wants to enable any CORBA clients to access enterprise beans must use the CORBA Naming Service for publishing and resolving the home interfaces of the enterprise beans. The runtime can use the CORBA Naming Service directly or indirectly via JNDI and its standard mapping to the CORBA Naming Service.

JNDI names have a string representation of the following form “directory1/directory2/…/directoryN/objectName”. The CORBA Naming Service defines names as a sequence of name components.

typedef string Istring;
struct NameComponent {
    Istring id;
    Istring kind;
};
typedef sequence<NameComponent> Name;

Each “/” separated name of a JNDI string name is mapped to a name component; the leftmost component is the first entry in the CORBA Naming Service name.

A JNDI string name is relative to some naming context, which we call the JNDI root context. The JNDI root context corresponds to a CORBA Naming Service initial context. CORBA Naming Service names are relative to the CORBA initial context.

A CORBA program obtains an initial CORBA Naming Service naming context by calling resolve_initial_references("NameService") on the ORB (pseudo) object. The CORBA Naming Service does not prescribe a rooted graph for organizing naming context and, hence, the notion of a root context does not apply. The initialization of the ORB determines the context returned by resolve_initial_references().

For example, a C++ Client could locate the home interface to our ATMSession bean, which has been registered with a JNDI string name “transaction/corbaEjb/atm”. First we obtain the initial naming context.

    Object_ptr obj = orb->resolve_initial_references("NameService");
    NamingContext initialNamingContext= NamingContext.narrow( obj );
    if( initialNamingContext == NULL ) {
        cerr << "Couldn't initial naming context* << endl;
        exit( 1 );
    }
Then we create a CORBA Naming Service name and initialize it according to the mapping explained previously.

```java
Name name = new Name( 1 );
name[0].id = "atm";
name[0].kind = "";
```

Now we resolve the name on the initial naming context. We assume that we have successfully performed the initialization and that we have the context of the naming domain of the enterprise bean. We narrow the resulting CORBA object to the expected type and make sure that the narrow was successful.

```java
Object_ptr obj = initialNamingContext->resolve( name );
ATMSessionHome_ptr atmSessionHome = ATMSessionHome.narrow( obj );
if( atmSessionHome == NULL ) {
    cerr << "Couldn't narrow to ATMSessionHome" << endl;
    exit( 1 );
}
```

### Mapping for transaction

A CORBA-based enterprise bean runtime environment that wants to enable a CORBA client to participate in a transaction involving enterprise beans must use the CORBA Object Transaction Service for transaction control.

When an enterprise bean is deployed it can be installed with different transaction policies. The policy is defined in the enterprise bean’s deployment descriptor.

The following rules have been defined for transactional enterprise beans: A CORBA client invokes an enterprise through stubs generated from the IDL interfaces for the enterprise bean’s remote and home interface. If the client is involved in a transaction, it uses the interfaces provided by CORBA Object Transaction Service. For example, a C++ client could invoke the ATMSession bean from the previous example as follows:

```cpp
try {
    // obtain transaction current
    Object_ptr obj = orb->resolve_initial_references("Current");
    Current current = Current.narrow( obj );
    if( current == NULL ) {
        cerr << "Couldn't resolve current" << endl;
        exit( 1 );
    }
    // execute transaction
    try {
        current->begin();
        atmSession->transfer("checking", "saving", 100.00 );
        current->commit( 0 );
    } catch( _ ) {
        current->rollback();
    }
}
```

---

5-12 Enterprise JavaBeans Programmer’s Guide
Security aspects of the EJB specification focuses on controlling access to enterprise beans. CORBA defines a number of ways to define the identities, including the following cases:

Plain IIOP. CORBA’s principal interface was deprecated in early 1998. The principal interface was intended for determining the identity of a client. However, the authors of the CORBA security services implemented a different approach, GIOP.

The GIOP specification contains a component called service context, which is an array of value pairs. The identifier is a CORBA long and the value is a sequence of octet. Among other purposes, entries in the service context can be used to identify a caller.

Secure IIOP. The CORBA security specification defines an opaque data type for the identity. The real type of the identity is determined by the chosen security mechanism; for example, GSS Kerberos, SPKM, or CSI-ECMA.

IIOP over SSL. SSL uses X.509 certificates to identify servers and, optionally, clients. When a server requests a client certificate, the server can use the certificate as a client identity.
Writing session beans

This chapter contains the following major topics:

- “Overview of a session bean” provides an introduction to stateful and stateless session beans.
- “Life cycle of a stateful session bean” describes the lifecycle of a stateful session bean, from its inception to the time it is removed.
- “Life cycle of a stateless session bean” describes the lifecycle of a stateless session bean.
- “Developing a session bean” provides the implementation details for all types of session beans.
- “The cart example” illustrates the code that must be written to fully implement a stateful session bean.

Overview of a session bean

A session bean class must implement the session bean (SessionBean) interface. It must also comply with the rules and naming conventions that define the relationship of this class to its remote and home interfaces.

A session bean implementation must also implement methods corresponding to those in the home interface and the remote interface. For example, the home interface’s create() methods have corresponding ejbCreate() methods in the session bean implementation class. The business methods exposed by the remote interface have corresponding implementations in the session bean class.

Recall that there can be both stateless and stateful session beans.

- **Stateless session bean.** A stateless session bean does not maintain conversational state between method invocations. As a consequence, a stateless session bean instance can be used for multiple clients.
Life cycle of a stateful session bean

- **Stateful session bean.** A stateful session bean maintains conversational state across method invocations and transactions. Once a client obtains a specific session bean, it uses this bean instance for the lifetime of the session. A client can, however, have multiple, distinguished sessions and hence multiple session bean instances.

Life cycle of a stateful session bean

The life cycle of a stateful session bean typically consists of the following events:

- The container instantiates a new session bean object in response to a client request.
- The session bean instance is ready for the client to invoke its business methods. This is known as the method ready state. These business methods may be executed in a transaction context or with an unspecified transaction context, depending on the bean’s deployment descriptor attributes and the client’s transaction context.
- The session bean instance executes non-transactional methods while it is in the method ready state.
- The session bean instance is associated with a transaction when the client invokes a transactional method. The container maintains the proper transactional context. Eventually, the transaction service either commits or rolls back the transaction.
- At certain times, the container may decide to evict the bean instance from memory. If so, it passivates the bean and saves the bean’s state to secondary storage. A session bean cannot be passivated when it is within a transaction.
- The container activates a session bean instance that had been passivated when a client invokes another method on the bean. When the container activates the bean, it restores the bean instance’s state from secondary storage and the bean is ready to act on client method calls.
- The container removes the bean instance in response to the client’s invocation of the remove method. The container may also remove the bean instance if the session bean reaches its timeout limit.

Figure 6.1 illustrates the life cycle of a session bean.
Typically, the life of a session bean starts when a client invokes a `create()` method on the session bean’s home interface. The implementation of the home interface is provided by the container, and is based on the session bean implementation class. The container creates a new instance of the session bean, initializes it, and returns an object reference to the client. During this process, the container invokes the `SessionBean` method `setSessionContext()` and then the method `ejbCreate()` on the session bean implementation. The bean provider can use these methods to initialize the session bean. The state of the session bean is now method ready, which means it can perform nontransactional operations or be included in a transaction for transactional operations. (You can find more information on the `SessionBean` methods in “SessionBean interface” on page 6-5. Refer to “Session bean implementation” on page 6-7 for more information on the `ejbCreate()` methods.)

When a client invokes a `remove()` method on the remote or home interface, the corresponding `ejbRemove()` method is invoked by the container on the session bean object. This allows the bean provider to add application-specific cleanup code to the `ejbRemove()` method. Once the invocation is completed, the object is in a nonexistent state again. When a client tries to invoke a method on a nonexistent session object, the container throws the exception `java.rmi.NoSuchObjectException`.

The container can deactivate the session bean instance. This is usually done for resource management reasons, such as when a session object is idle for a certain time or if the container requires memory resources. When a bean instance is deactivated—referred to as passivation—the container stores reference information and the state of the session object on disk and frees the memory allocated by the bean. The container must also explicitly retain references to files or other operating system-controlled resources that may have been opened by the session bean. The container deactivates the bean by triggering the `ejbPassivate()` method; it reactivates the bean instance with the `ejbActivate()` method. The instance is typically reactivated when the client makes the next call on the session bean’s reference that it holds. At that time, the container recreates the session object in memory.
Note that passivation depends on the setting of the EJB passivation time-out flag for the session bean. If this flag is set to a value of zero, the container never passivates the session bean instance. If it is set to a value other than zero, the container passivates the session bean instance after the time specified by this flag. The value reflects the time in seconds.

When a client invokes a method on a session bean object in a transactional context, the container starts a new transaction, or includes the bean instance in an existing transaction. There are points in the transaction's life cycle—called transaction synchronization points—where the session bean instance can receive notification of upcoming transaction events and can take some prior action, if desired. The SessionSynchronization interface defines these transaction synchronization points. (See “Session synchronization interface” on page 6-6.) The session bean that wants to receive notification of transaction events can implement the SessionSynchronization interface. Use of this interface is optional.

The Inprise container extends the EJB architecture to provide long-lived behavior. Normally, a session bean's lifetime is bound to the lifetime of the container in which it is installed. For most EJB containers, this means that once the process, or JVM, in which the container is executed terminates, the session bean's reference usually becomes invalid. However, the Inprise container permits a session bean to be passivated and its state persisted. By doing this, the Inprise container can re-activate the session bean in a different container. This is a preferred behavior that is not supported by all EJB containers.

Life cycle of a stateless session bean

In contrast to a stateful session bean, the EJB Container determines and controls the life cycle of a stateless session bean.

The life cycle of a stateless session bean is fairly simple, as shown in Figure 6.2, “Life cycle of a stateless session bean.” When the container creates a new instance of a stateless session bean, it invokes the methods setSessionContext() and ejbCreate() on the session object. The new instance is placed in a pool of stateless bean instances that are ready to be invoked by clients. Because stateless session objects do not maintain client state, the container can assign any instance to handle an incoming method invocation. When the container removes an instance from that pool, it invokes the method ejbRemove() on the session object.
Developing a session bean

The following section describes the session bean implementation class and the interfaces and methods that a session bean class might implement.

SessionBean interface

The session bean `SessionBean` interface defines the methods all session beans must implement. This interface extends the interface `EnterpriseBean`. The methods of the session bean interface are closely associated with the life cycle of a session bean.

Code sample 6.1  SessionBean interface

```java
package javax.ejb;
public interface SessionBean extends EnterpriseBean {
    void setSessionContext(SessionContext sessionContext)
        throws EJBException, RemoteException;
    void ejbRemove() throws EJBException, RemoteException;
    void ejbActivate() throws EJBException, RemoteException;
    void ejbPassivate() throws EJBException, RemoteException;
}
```

The methods have the following semantics.

- `setSessionContext()` sets a session context. The bean’s container calls this method to associate a session bean instance with its context. The session context interface provides methods to access runtime properties of the context in which a session runs. Typically, a session bean retains its context as part of its conversational state.

- `ejbRemove()` notifies a session object that it is about to be removed. Whenever the container removes a stateful session bean (for example, because a client invoked a remove operation on the remote or home interface), it calls this operation on the bean implementation.

Note  The actual creation and removal of stateless session bean instances from the Container’s pool of stateless bean instances are not related to the invocation of the `create()` and `remove()` methods on the home/remote interfaces. Container policies govern the life cycle of stateless session beans.

Figure 6.2  Life cycle of a stateless session bean

Bean does not exist

```
<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>ejbCreate()</td>
</tr>
<tr>
<td>setSessionContext()</td>
</tr>
<tr>
<td>ejbRemove()</td>
</tr>
<tr>
<td>Bean exists; placed in method ready pool</td>
</tr>
</tbody>
</table>
```

Bean does not exist

**Code sample 6.1**  SessionBean interface

```java
package javax.ejb;
public interface SessionBean extends EnterpriseBean {
    void setSessionContext(SessionContext sessionContext)
        throws EJBException, RemoteException;
    void ejbRemove() throws EJBException, RemoteException;
    void ejbActivate() throws EJBException, RemoteException;
    void ejbPassivate() throws EJBException, RemoteException;
}
```
Developing a session bean

- `ejbActivate()` notifies a stateful session object that it has been activated.
- `ejbPassivate()` notifies a stateful session object that it is about to be deactivated by the container.

The activation and deactivation notifications allow a sophisticated bean implementation to manage resources.

Session synchronization interface

The optional `SessionSynchronization` interface provides methods for session beans to receive transaction synchronization notification—that is, the bean instance receives notification of transition points within the transaction either just after they have completed or just before they are to begin. The bean instance can use this notification to manage database data that it may have cached within a transaction. For example, implementing this interface allows a bean to synchronize cached data with the database within an ongoing transaction.

Only a stateful session bean using container-managed transactions may implement the `SessionSynchronization` interface. A stateless session bean should never implement this interface, nor should a stateful session bean that uses bean-managed transactions.

**Code sample 6.2 SessionSynchronization interface**

```java
package javax.ejb;

public interface SessionSynchronization {
    void afterBegin() throws RemoteException;
    void beforeCompletion() throws RemoteException;
    void afterCompletion(boolean completionStatus) throws RemoteException;
}
```

The Container uses these three methods to notify the bean instance of the start or completion of specific steps in a transaction.

`afterBegin()` notifies the session bean that a new transaction has begun. The container invokes this method prior to invoking the first business method within the bean, though this is not necessarily at the beginning of the transaction. The session bean is already in a transaction and any work that is executed is within the scope of this transaction.

`beforeCompletion()` notifies the session bean that a client has completed work on the current transaction, but the resources involved have not yet been committed. The bean should now update the database with its cached values if there are any. If necessary, the session bean can force a rollback of the current transaction by invoking the method `setRollbackOnly()` on its session context.

`afterCompletion()` notifies the session bean that the current transaction has been completed. The parameter `completionStatus` indicates the outcome of the transaction. A status value of true means the transaction committed; a status of false means the transaction rolled back.

For example, the container invokes the method `afterBegin()` after the client invokes a transactional business method on the remote interface. The invocation of the business
method puts the enterprise bean object in the transaction ready state. Later, the transaction commit triggers the invocation of two methods—first the beforeCompletion() method and then, if the commit is successful, the afterCompletion(true) method on the enterprise bean object. In the case of a rollback, the container invokes the method afterCompletion(false) on the session object. (However, when a rollback situation occurs, the container does not invoke the beforeCompletion() method.) As a result of the commit or rollback, the session object will be back in the method ready state.

Session bean implementation

A session bean class (or implementation) must implement:

- The SessionBean interface
- Methods that correspond to methods defined in the home interface
- Methods that correspond to methods defined in the remote interface

A stateful session bean that uses container-managed transactions may optionally implement the SessionSynchronization interface.

We have already explained the typical implementations of the methods of the SessionBean interface. See Figure 6.1, “Life cycle of a stateful session bean,” on page 6-3 and Figure 6.2, “Life cycle of a stateless session bean,” on page 6-5.

The session bean home interface defines one or more create methods. For each of these methods, the session bean must declare and implement a corresponding create method. The naming convention for the session bean’s create methods corresponds to the names used in the home interface, with the addition of the prefix “ejb.” Thus, the home interface defines one or more create() methods, and the session bean defines the equivalent create methods which it calls ejbCreate(). The number and types of parameters must match those defined in the home interface’s corresponding create() methods. The ejbCreate() methods in the session bean always return void, as opposed to the home interface’s create() methods which return the remote interface.

The signature for all session bean ejbCreate() methods is:

```java
class {  
  public void ejbCreate( <zero or more parameters> ) {  // implementation
    ...
  }
}
```

The ejbCreate() method is not required to throw an exception. However, it may throw application-specific exceptions and other exceptions, such as the javax.ejb.CreateException or javax.ejb.EJBException.

The session bean must also implement the business methods corresponding to those defined in the remote interface. The signatures for these business methods must exactly match those of the methods defined in the remote interface. Type safety is ensured by the container. There is no explicit “implements” relationship between the remote interface and the enterprise bean implementation class. The container usually checks type safety at install time. Development tools for EJB will probably ensure type safety at development time.
Developing a session bean

There is no need for a constructor in the session bean class, and you can omit it. All of the initialization of the bean is done in the ejbCreate() methods.

To illustrate, examine the session bean class for the CartBean session bean shown in Code sample 6.3. This code snippet shows only parts of the CartBean implementation; it is meant to highlight the code that a session bean must implement.

Code sample 6.3  Example of a session bean implementation

```java
import java.util.*;

public class CartBean implements javax.ejb.SessionBean {

    // variables
    private Vector _items = new Vector();
    private String _cardHolderName, _creditCardNumber;
    ...
    // required methods
    public void setSessionContext(javax.ejb.SessionContext sessionContext) {}
    public void ejbCreate(String cardHolderName, String creditCardNumber,
                           Date expirationDate) {
        _cardHolderName = cardHolderName;
        _creditCardNumber = creditCardNumber;
        ...
    }
    public void ejbRemove() {}
    public void ejbActivate() {}  
    public void ejbPassivate() {}

    // business methods
    public void addItem(Item item) {
        System.out.println("addItem(" + item.getTitle() + ": " + this);
        _items.addElement(item);
    }
    public void removeItem(Item item) throws ItemNotFoundException {
        ...
    }
    public float getTotalPrice() {
        ...
    }
    public java.util.Enumeration getContents() {
        ...
    }
    public void purchase()
        ...
    }
    ...
}
```

Notice that the CartBean class has no constructor. It has one ejbCreate() method, and this method is used to initialize the cart instance. It includes declarations of the other required methods—getSessionContext(), ejbRemove(), ejbActivate(), and ejbPassivate(). However, CartBean provides a trivial implementation of these methods.
The cart example

The cart example demonstrates the use of a shopping cart for an online store. You select items and put them in your cart. You might leave the site briefly, return, and add more things to your cart. At any time you can view the items you’ve placed in the cart and their total cost. Eventually, you checkout and buy the items.

The cart example implements a stateful session enterprise bean.

Stateful session beans and caching

The EJB Container supports stateful session enterprise beans using a high-performance caching architecture. There are two basic pools of objects: the ready pool and the passive pool. Enterprise beans transition from the ready pool to the passive pool after a configurable timeout. Transitioning an enterprise bean to the passive pool stores the enterprise bean’s state in the EJB Container’s Java database.

The cart example’s files

The cart example consists of a number of different files. This section focuses on those files that you might write yourself, or that illustrate interesting things about session beans. Some of the files in the cart directory are generated files (stubs, skeletons, and other CORBA code) and we do not discuss them here.

Briefly, we focus on the following files:

- The cart home interface, CartHome.java. This file defines the home interface for the CartBean session bean. See “Cart home interface” on page 6-10.
- The cart remote interface, Cart.java. This file defines the remote interface for the CartBean session bean. See “Cart remote interface” on page 6-11.
- The cart session bean implementation, CartBean.java. This is the session bean class for CartBean. See “CartBean session bean” on page 6-12.
- The item class, Item.java. This file is used by CartBean. It provides methods for the getting the price and title of items to be placed in a cart. See “Item class” on page 6-16.
- The deployment descriptor file, Cart.xml. For EJB 1.1, the deployment descriptor is contained in an XML file. See “XML deployment descriptor file” on page 6-18. (For EJB 1.0, there is a file that generates a serialized deployment descriptor for CartBean, GenerateDescriptors.java. See “Generate deployment descriptor” on page B-1.)
- Exception files. These files define the application-specific exceptions thrown by CartBean. There are three such exceptions, and each is defined in its own file: ItemNotFoundException, PurchaseProblemException, and CardExpiredException. See “Exceptions” on page 6-17.
- The cart client program, CartClient.java. This is the client application.
- The makefile for compiling and building the bean components and the client application. See “Makefile” on page 3-3.
The cart example

Cart home interface

Enterprise beans are always defined by two interfaces: the remote and home interfaces. Thus, a session bean called CartBean (in our example) has a public EJB remote interface called Cart and a home interface called CartHome.

CartHome is the home interface for the CartBean stateful session bean. Like all other home interfaces, it extends the EJBHome interface. While the home interface can potentially perform two actions, create bean instances and find bean instances, session bean home interfaces only create new bean instances. Find functionality is not needed because a session bean lacks persistence—by definition, the bean instance terminates when the client’s session ends. Only home interfaces for entity beans include find operations, because entity beans are used by multiple clients and persist for as long as the data entity exists.

Code sample 6.4 shows the code for the CartHome interface.

Code sample 6.4 CartHome interface

```java
// CartHome.java
public interface CartHome extends javax.ejb.EJBHome {
    Cart create(String cardHolderName, String creditCardNumber,
                 java.util.Date expirationDate)
    throws java.rmi.RemoteException, javax.ejb.CreateException;
}
```

The shopping cart bean’s CartHome interface is a very simple interface—it just defines one create() method. Because this is a stateful session bean, there can be more than one create() method and a create() method may have parameters. In our example, the CartHome create() method takes three parameters: cardHolderName, creditCardNumber, and expirationDate. The client invokes the create() method to request a shopping cart and the container creates one specifically for that user. The client may use the shopping cart intermittently (and the server might crash and resume operation), but the session bean remains active for that one client until the user exits and removes the session bean. The Inprise Application Server (IAS) stores session beans in a database. By default, this is an all-Java database (JDataStore), though it could alternatively be any other database. Because they are stored in a database, they can be persistent.

Note

With the Inprise Application Server, unlike other EJB products, stateful session beans survive (are persisted across) server failures.

A stateful session bean maintains state across method invocations, regardless of whether those methods are within the context of a transaction or not. The state is the data carried by an object, and it remains associated with the object for the lifetime of the object. The container flushes the state of a session bean from memory when the session is completely over.

The create() method follows the rules defined in the EJB specification: it throws an RMI remote exception, java.rmi.RemoteException, and it also throws an EJB create exception, javax.ejb.CreateException. The signature of the create() method matches that of the ejbCreate() method in the session bean class, in terms of the number and types of arguments. The return value of create() is a Cart remote interface; this is
because the CartHome interface essentially functions as a factory for Cart. (The return value for the matching `ejbCreate()` method is void.)

**Cart remote interface**

The CartBean session bean has a remote interface Cart that extends the EJBObject interface. The EJBObject interface is a base interface for all remote interfaces. It defines methods that enable you to:

- Get information about the session bean. You can test if the bean object is identical to another enterprise bean object. You can also get the primary key for an entity bean, but this does not apply to session beans.

- Obtain a reference and/or a handle to the session bean. You can obtain a reference to the bean’s home interface or a serializable handle to the bean instance. You can store the handle and retrieve it at a later time and then use it to regain your reference to the bean instance.

- Remove the bean instance. The EJBObject interface defines the `remove()` method for removing the bean instance.

The Cart remote interface defines five business methods, in addition to the methods it inherits from EJBObject. These business methods are implemented in the CartBean session bean class; the Cart remote interface merely exposes these methods to clients. A client may only invoke the methods of an enterprise bean that the remote interface exposes. The exposed business methods are:

- `addItem()`—Adds an item to the shopping cart.
- `removeItem()`—Removes an item from the shopping cart.
- `getTotalPrice()`—Adds the prices on all the items and returns the total price.
- `getContents()`—Gathers all the items in the shopping cart and returns them in a list that can be viewed or printed.
- `purchaseItems()`—Attempts to purchase the items.

The Cart remote interface code is shown in Code sample 6.5.

**Code sample 6.5  Cart remote interface**

```java
// Cart.java
public interface Cart extends javax.ejb.EJBObject {
void addItem(Item item) throws java.rmi.RemoteException;
void removeItem(Item item)
    throws ItemNotFoundException, java.rmi.RemoteException;
float getTotalPrice() throws java.rmi.RemoteException;
java.util.Enumeration getContents() throws java.rmi.RemoteException;
void purchase()
    throws PurchaseProblemException, java.rmi.RemoteException;
}
```
The cart example

The Cart remote interface follows the rules for defining an enterprise bean remote interface:

1. The remote interface must extend the `javax.ejb.EJBObject` interface.
2. Each method in the remote interface matches a method implemented in the session bean class. The signature of the method in the remote interface matches the signature of the method in the session bean class—that is:
   - They have the same name.
   - They have the same number and types of arguments, and the same return type.
   - The exceptions defined in the throws clause of the method in the session bean class must be a subset of the exceptions defined in the throws clause of the method in the remote interface.
3. All the methods must throw the `java.rmi.RemoteException`. They can, optionally, throw other application-specific exceptions as well. Note that the `removeItem()` method throws `ItemNotFoundException` in addition to `java.rmi.RemoteException`.
4. All the methods have arguments and return values that are valid Java RMI-IIOP values.

CartBean session bean

This section provides the details for implementing the CartBean session bean.

Code sample 6.3, “Example of a session bean implementation,” on page 6-8, shows the overall structure and principal parts of the CartBean class implementation. Here, we’ll provide the details of the session bean class.

An enterprise bean must implement either the `javax.ejb.SessionBean` interface, if it is a session bean, or the `javax.ejb.EntityBean` interface, if it is an entity bean. Because the CartBean object is a session enterprise bean, it implements the `javax.ejb.SessionBean` interface, as follows:

```java
public class CartBean implements javax.ejb.SessionBean {

    // Details of the session bean class implementation

    // A session bean is a normal Java object which can have instance variables. CartBean
    // has four instance variables, and it declares all four as private variables. See Code
    // sample 6.6. They cannot be accessed directly by a client program. Rather, these
    // variables may only be accessed or set through CartBean's business methods and its
    // create method.

    Code sample 6.6  CartBean instance variables
    private Vector _items = new Vector();
    private String _cardHolderName;
    private String _creditCardNumber;
    private Date _expirationDate;

    The _items variable corresponds to the items owned by the cart object. It is a vector
    type—that is, a collection of items. The three instance variables—_cardHolderName,
Writing session beans

The cart example

_creditCardNumber, and _expirationDate—store the parameters in the home interface’s create() method, and CartBean’s ejbCreate() method.

Required methods

A session bean is required to implement the four methods that are defined by the javax.ejb.SessionBean interface. The EJB container invokes these methods on the bean instance at specific points in a session bean’s life cycle. At a minimum, the bean provider must provide a trivial implementation of these methods in the session bean implementation. The bean provider can add additional code to these methods, if desired. The CartBean session bean adds no code of its own to these methods. The SessionBean methods are shown in Code sample 6.7.

Code sample 6.7  Methods inherited from SessionBean

```
public void setSessionContext(javax.ejb.SessionContext sessionContext) {}
public void ejbActivate() {}
public void ejbPassivate() {}
public void ejbRemove() {}
```

The container calls the setSessionContext() method to associate the bean instance with its session context. The bean can choose to retain this session context reference as part of its conversational state, but it is not required to do so. Our example bean does not save the session context. The session bean can use the session context to get information about itself, such as environment variables or its home interface.

The container calls the ejbPassivate() method on the bean instance when it needs to place the bean instance into a passive state. The container writes the bean’s current state to secondary storage when it passivates the bean; it restores this state when it later activates the bean. Because the container calls the ejbPassivate() method just before it actually passivates the bean instance, the bean provider can add code to this method to do any special variable caching that it desires. Similarly, the container calls the ejbActivate() method on the bean instance just prior to returning the bean instance to an active state from a passive state. When it activates the bean, it restores all persisted state values. The bean provider can choose to add code to the ejbActivate() method, if desired. The CartBean provides a trivial implementation of these two methods.

A session bean is not required to implement a constructor. Instead, the bean implements at least one ejbCreate() method, which serves as a constructor to create a new bean instance. A stateful session bean may implement more than one ejbCreate() method; they would differ only by their parameters. According to the EJB specification, the parameters for each ejbCreate() method in the session bean class must match the parameters for a create() method declared in the bean’s home interface. That is, there must be a matching create() method in the home interface with the same number and types of arguments as in the bean’s ejbCreate() method. In addition, the session bean’s ejbCreate() method always returns a void, while the home interface create() method returns the remote interface type.
The cart example

The CartBean declares one `ejbCreate()` method, which takes three parameters, as shown in Code sample 6.8:

**Code sample 6.8**  CartBean `ejbCreate()` method

```java
public void ejbCreate(String cardHolderName, String creditCardNumber,
                      Date expirationDate) throws CreateException {
  _cardHolderName = cardHolderName;
  _creditCardNumber = creditCardNumber;
  _expirationDate = expirationDate;
}
```

As with the other SessionBean methods, the `ejbRemove()` method is required. The container calls this method just prior to removing the bean instance. A bean provider might want to add some application-specific code that would execute before the bean is removed, but it is not required. CartBean provides a trivial implementation of this method.

**Business methods**

The session bean class must implement the business methods that its remote interface declared. There are a few rules that the session bean’s business method implementations must follow:

- The method name must not start with “ejb” to avoid conflict with names reserved by the EJB architecture.
- The method must be declared as public.
- The method must not be declared as `final` or `static`.
- The arguments and return types must be legal RMI-IIOP types.
- The throws clause may include the `javax.ejb.EJBException` exception and it may define arbitrary application-specific exceptions.

In the case of the cart example, the Cart remote interface declared five methods. The CartBean class provides the implementation for these five business methods. The signatures (method name, number of arguments, argument types, and return type) of the session bean class methods must match those of the remote interface.

The CartBean class implements the methods `addItem()`, `removeItem()`, `getTotalPrice()`, `getContents()`, and `purchase()`. (Note that the `java.rmi.RMIException` is deprecated for EJB 1.1.)

To help you follow the flow of the program, each business method includes a line of code that displays the method name and what it is doing.

The `addItem()` method adds an item to the vector which holds the list of items in the cart, as follows:

**Code sample 6.9**  CartBean `addItem()` method

```java
public void addItem(Item item) {
  System.out.println("addItem(" + item.getTitle() + "): " + this);
  _items.addElement(item);
}
```
The `removeItem()` method is a bit more complicated. The program loops through the elements in the item list and checks if the class and title of the item to be removed matches the class and title of one in the list. This method verifies that you are removing the item you really want removed. If no matching item is found, the method throws an `ItemNotFoundException`.

**Code sample 6.10** CartBean `removeItem()` method

```java
public void removeItem(Item item) throws ItemNotFoundException {
    System.out.println("removeItem(" + item.getTitle() + ")": " + this);
    Enumeration elements = _items.elements();
    while(elements.hasMoreElements()) {
        Item current = (Item) elements.nextElement();
        // items are equal if they have the same class and title
        if(item.getClass().equals(current.getClass()) &&
            item.getTitle().equals(current.getTitle())) {
            _items.removeElement(current);
            return;
        }
    }
    throw new ItemNotFoundException
        ("The item " + item.getTitle() + " is not in your cart");
}
```

The `getTotalPrice()` initializes the total price to zero, then loops through the item list, adding the price of each element to the total price. It returns the total price rounded to the nearest penny.

**Code sample 6.11** CartBean `getTotalPrice()` method

```java
public float getTotalPrice() {
    System.out.println("getTotalPrice(): " + this);
    float totalPrice = 0f;
    Enumeration elements = _items.elements();
    while(elements.hasMoreElements()) {
        Item current = (Item) elements.nextElement();
        totalPrice += current.getPrice();
    }
    // round to the nearest lower penny...
    return (long) (totalPrice * 100) / 100f;
}
```

All data types passed between a client and a server must be *serializable*. That is, they must implement `java.io.Serializable`. In the cart example, the program needs to return a list of items to the client. If there were no serializable restriction, you could use `_items.elements()` to return the contents of the item vector. However, `_items.elements()` returns a Java `Enumeration` object which is not serializable. To avoid this problem, the program implements a class called `com.inprise.ejb.util.VectorEnumeration(_items)`. This class takes a vector and returns an actual enumeration, which is serializable, for the contents of that vector. The CartBean object passes this serializable vector to the client, and can receive a serializable vector passed from the client side. The `getContents()` method, shown in
The cart example

Code sample 6.12, does the conversion between a Java Enumeration and a serializable VectorEnumeration.

**Code sample 6.12 CartBean getContents() method**

```java
code sample 6.12
CartBean getContents() method

public java.util.Enumeration getContents() {
    System.out.println("getContents(): " + this);
    return new com.inprise.ejb.util.VectorEnumeration(_items);
}
```

The purchase() method does the following:

1. Gets the current time.
2. Compares the expiration date of the credit card with the current time. If the expiration date is prior to the current time, it throws the application exception CardExpiredException.
3. The method normally completes the purchasing process, including updating inventory, posting the charge to the credit card company, and initiating shipment of the item. (None of this has actually been implemented.) If an error occurs at any point, the purchase process does not complete and the method throws a PurchaseProblemException.

**Code sample 6.13 CartBean purchase() method**

```java
code sample 6.13
CartBean purchase() method

public void purchase() throws PurchaseProblemException {
    System.out.println("purchase(): " + this);
    // make sure the credit card has not expired...
    Date today = Calendar.getInstance().getTime();
    if(_expirationDate.before(today)) {
        throw new CardExpiredException("Expiration date: " + _expirationDate);
    }
    // complete purchasing process
    // throw PurchaseProblemException if error occurs
}
```

CartBean includes a toString() method to print out CartBean and the name of the card holder.

```java
code sample 6.13
CartBean toString() method

public String toString() {
    return "CartBean[name=" + _cardHolderName + "]";
}
```

**Item class**

The Item interface is a public class. It extends java.io.Serializable. Serializable data can be passed on the wire.
The cart example

The Item interface has two fields (title and price), two get methods—getTitle() and getPrice()—and a constructor. Code sample 6.14 shows the code for Item:

Code sample 6.14 Item interface

```java
// Item.java
public class Item implements java.io.Serializable {
    private String _title;
    private float _price;
    public Item(String title, float price) {
        _title = title;
        _price = price;
    }
    public String getTitle() {
        return _title;
    }
    public float getPrice() {
        return _price;
    }
}
```

Exceptions

There are three exceptions in the cart example. All of them are extensions of the Java class Exception. They are shown in Code sample 6.15

Code sample 6.15 Cart exceptions

```java
// ItemNotFoundException.java
public class ItemNotFoundException extends Exception {
    public ItemNotFoundException(String message) {
        super(message);
    }
}
// PurchaseProblemException.java
public class PurchaseProblemException extends Exception {
    public PurchaseProblemException(String message) {
        super(message);
    }
}
// CardExpiredException.java
public class CardExpiredException extends PurchaseProblemException {
    public CardExpiredException(String message) {
        super(message);
    }
}
```

In the Cart interface, the removeItem() method throws an ItemNotFoundException. The purchase() method throws a PurchaseProblemException. PurchaseProblemException has a subclass called CardExpiredException.

Note Java exceptions and exceptions with subclassing were not supported before the implementation of the OMG objects-by-value specification.
According to the EJB 1.1 specification, the deployment descriptor must be an XML file. The XML file follows the Document Type Definition (DTD) approved by Sun Microsystems. A deployment descriptor contains a set of properties that describe how the container will deploy the enterprise bean or application.

The deployment descriptor includes a set of tags and attributes whose values indicate the properties of the bean. For example, some of the tags for the cart application are as follows:

- The `<session>` tag indicates that the enterprise bean is a session bean. (An `<entity>` tag indicates it is an entity bean.)
- Within the `<session>` tag, other tags indicate:
  - `<ejb-class>`—The name of the implementation class.
  - `<home>`—The home interface name.
  - `<remote>`—The remote interface name.
  - `<session-type>`—If the session bean is stateful or stateless.
  - `<transaction-type>`—Whether persistence is container managed or bean managed.
  - `<trans attribute>`—The transaction attribute for each method.
  - `<timeout>`—The timeout value for the session bean.

**Code sample 6.16  XML deployment descriptor file**

```xml
<ejb-jar>
  <enterprise-beans>
    <session>
      <description>
        XML deployment descriptor created from file:
        D:\Kodiak\kodiak04\ejb_ea_0_4\examples\cart\cart.ser
      </description>
      <ejb-name>cart</ejb-name>
      <home>CartHome</home>
      <remote>Cart</remote>
      <ejb-class>CartBean</ejb-class>
      <session-type>Stateful</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>
  <assembly-descriptor>
    <container-transaction>
      <method>
        <ejb-name>cart</ejb-name>
        <method-name>*</method-name>
        <trans-attribute>NotSupported</trans-attribute>
      </method>
    </container-transaction>
  </assembly-descriptor>
</ejb-jar>
```
The cart example

CartClient.java

The CartClient program is the client application that utilizes the sort enterprise bean. Let’s examine the CartClient main() routine first. The main() routine includes elements which all enterprise bean client applications must implement. It shows you

- How to use JNDI to locate the cart bean’s home interface.
- How to use the home interface’s create() method to create a new remote Cart object.
- How to invoke methods declared in the Cart object.

Figure 6.3 lists the more interesting and relevant parts of the CartClient main() routine. It shows the key portions of code. The numbers to the left of the code correspond to the list of steps that follows the figure. (For the complete listing of the CartClient, refer to the source code in the cart examples directory that came with your installation.)
The cart example

Figure 6.3  CartClient main() routine

```java
public static void main(String[] args) throws Exception {
    // get a JNDI context using the Naming service
    javax.naming.Context context = new javax.naming.InitialContext();
    Object objref = context.lookup("cart");

    CartHome home = (CartHome) javax.rmi.PortableRemoteObject.narrow(objref, SortHome.class);

    Cart cart;
    {
        String cardHolderName = "Jack B. Quick";
        String creditCardNumber = "1234-5678-9012-3456";
        Date expirationDate = new GregorianCalendar(2001, Calendar.JULY, 1).getTime();
        cart = home.create(cardHolderName, creditCardNumber, expirationDate);
        Book knuthBook = new Book("The Art of Computer Programming", 49.95f);
        cart.addItem(knuthBook);
        // create compact disk and add it to cart, then list cart contents
        summarize(cart);
        cart.removeItem(knuthBook);
        // add a different book and summarize cart contents
        try {
            cart.purchase();
        } catch(PurchaseProblemException e) {
            System.out.println("Could not purchase the items:
	" + e);
        }
        cart.remove();
    }
```

1. The `main()` routine begins by using JNDI context to look up objects. Construct an initial context—a Java naming context. This is standard JNDI code.

2. Look up the CartHome object called `cart`. Looking up a name with JNDI involves invoking a call from the client to the CosNaming service to look up the name in CosNaming. The CosNaming service returns an object reference to the client. In this example, it returns a CORBA object reference. The program must do a `PortableRemoteObject.narrow()` operation on the object reference and cast the returned object to the type CartHome, and at the same time assigns it to the variable `home`. This call is typical for distributed applications. The call uses CORBA and IIOP to do the following:
   - Talk to a server.
   - Do a CosNaming lookup.
   - Obtain a CORBA object reference.
   - Return the object reference to the client.

3. The program declares a reference to the remote cart object, initializes the three create parameter variables—user name, credit card number, and card expiration date—and creates a new cart remote object.
4 The program creates two shopping cart items—a book and a compact disk—and adds these items to the shopping cart using the cart's addItem() method.

5 The program then lists the items currently in the cart. To do this, it calls the summarize() function. The summarize() function retrieves the elements or items in the cart using the cart's getContents() method, which returns a Java Enumeration. It then uses the Java Enumeration interface methods to read every element in the Enumeration, extracting the title and price for each one. shows the details of the summarize() function:

**Code sample 6.17 CartClient summarize() function**

```java
static void summarize(Cart cart) throws Exception {
    System.out.println("======= Cart Summary ========");
    Enumeration elements = cart.getContents();
    while(elements.hasMoreElements()) {
        Item current = (Item) elements.nextElement();
        System.out.println("Price: $" + current.getPrice() + ", " +
                           current.getClass().getName() + ", title: " +
                           current.getTitle());
    }
    System.out.println("Total: $" + cart.getTotalPrice());
    System.out.println("============================");
}
```

6 The program then calls the cart's removeItem() method to remove an item from the cart. It adds a different item and summarizes the cart contents again.

7 The program then attempts to purchase the items. The purchase operation fails—it is not implemented on the server—and the program throws a PurchaseProblemException.

8 At this point, the user is finished with the shopping session and the program removes the cart.

**Note** It is not necessary to remove the cart. It is done here for simplicity and to illustrate good programming practice. A session bean exists for the client that created it; when the client ends its session, the container automatically removes the session bean object. The container also removes the session bean object when it eventually times out, though this does not happen immediately.

CartClient also includes code that extends the generic Item class (see “Item class” on page 6-16) with two types of items: a book and a compact disc. Book and CompactDisc are the concrete classes used in the example.

**Code sample 6.18 CartClient extends item class**

```java
// CartClient.java
import java.util.*;
class Book extends Item {
    Book(String title, float price) {
        super(title, price);
    }
}

class CompactDisc extends Item {
    CompactDisc(String title, float price) {
        super(title, price);
    }
}```
Chapter 7

Writing entity beans

This chapter includes the following major topics:

- “Overview of entity beans” provides an introduction to entity beans.
- “Handling persistence” explains the two different ways an entity bean can handle persistence.
- “Understanding the lifecycle of an entity bean” depicts the lifecycle of an entity bean from the time it is created to when it is removed and deleted.
- “Implementing an entity bean” describes the details for implementing an entity bean

Overview of entity beans

An entity bean is a component that directly represents data stored in persistent storage, such as a database, or a data entity implemented by an existing enterprise application. It is intended to be used concurrently by multiple clients, as opposed to session beans which are dedicated to a particular client.

An entity bean maps to a row or rows within a table in a relational database, or to an entity object in an object-oriented database. For example, in the case of a relational database, each column of the row in the table maps to an instance variable in the entity bean. Each row within the table is identified by a primary key—one or more columns which uniquely identify that row. Similarly, each entity bean includes a primary key to identify a particular bean instance.

While multiple clients can interact concurrently with the same entity bean, the EJB Container handles this interaction in a way that preserves the integrity of the underlying database data. The container itself may handle the concurrent entity bean access, such as by queuing client requests so that only one request executes at a time. Or, the container may defer the concurrency handling to the database management system (DBMS)—that is, the container may create an instance of the entity bean for
Handling persistence

Each client and then let the DBMS handle synchronizing the database access. The Inprise EJB Container uses this latter approach.

Because an entity bean typically represents data stored in a database, its lifetime matches the lifetime of the data—an entity bean is a long-lived component that normally outlives the client applications that use it. Entity beans not only persist after a client application or session ends, they also persist despite server crashes. Just like data in a database, an entity bean is guaranteed to survive a server crash and an orderly server restart.

Unlike session beans which have a timeout value associated with them (and, when the timeout expires, the container removes the session bean instance from persistent storage), entity beans do not have a timeout period associated with them. Regardless of how long they remain inactive, the container does not remove them from the persistent storage. The container removes only the entity bean instance from the container itself.

An entity bean can only be removed by explicitly deleting it. This can be done by invoking its \texttt{remove()} method, which removes both the bean and the underlying data from the database, or by using the DBMS (or another existing enterprise application outside of the EJB realm) to remove the data entry from the database.

Handling persistence

Persistence refers to the data access protocol for transferring the state of an entity between the entity bean instance and the underlying database. With entity beans, the bean provider has a choice about how persistence is implemented.

The bean provider can implement the bean’s persistence directly in the entity bean class (or in other classes provided with the entity bean). This is referred to as bean-managed persistence.

Or, the bean provider can delegate the handling of the entity bean’s persistence to the EJB Container. At deployment, container tools specify how to handle the bean’s persistence. This is referred to as container-managed persistence.

Bean-managed persistence

An entity bean with bean-managed persistence contains code to do its own database access and updates. That is, the bean provider writes database access calls directly into the entity bean component, or into associated classes. These calls are usually written using JDBC.

The database access calls may appear in the entity bean’s business methods, or in one of the following entity bean interface methods—\texttt{ejbCreate()}, \texttt{ejbRemove()}, \texttt{ejbLoad()}, and \texttt{ejbStore()}—or in the finder \texttt{ejbFind<methodname>()} methods declared in the bean’s home interface. (The entity bean interface methods are discussed in “Entity bean interface” on page 7-6.)
In general, a bean with bean-managed persistence is more difficult to implement because the bean provider must write the additional data access code. And, because the database access code may be embedded within the bean methods, it may also be more difficult to adapt the entity bean to different databases or to a different schema.

Though there are many reasons to not use bean-managed persistence, there are some situations where you might use this approach. Entity beans using bean-managed persistence do not require any extra support from the EJB Container. Plus, they can handle situations that might be beyond the capabilities of the EJB Container.

**Container-managed persistence**

In the case of container-managed persistence, the entity bean does not include code to do database access. The bean provider writes no database access code into the bean. Instead, the bean relies on the EJB Container to handle database access and updates.

The container’s tools generate the database access calls at the time that the entity bean is deployed; that is, when the entity bean is installed into the container. The tools use the deployment descriptor to determine the instance fields for which they must generate database access calls. Instead of coding the database access directly in the bean (as with bean-managed persistence), the bean provider of a container-managed entity bean must specify in the deployment descriptor those instance fields for which the container tools must generate access calls. The container has sophisticated deployment tools capable of mapping the fields of an entity bean to its data source.

Container-managed persistence has many advantages over bean-managed persistence. It is simpler to code—the bean provider does not have to code the database access calls. Handling of persistence can also be changed without having to modify and recompile the entity bean code. The Deployer or Application Assembler can do this by modifying the deployment descriptor when deploying the entity bean. Shifting the database access and persistence handling to the container not only reduces the complexity of code in the bean, it also reduces the scope of possible errors. The bean provider can focus on debugging the business logic of the bean and not be concerned with the underlying system issues.

However, there are some limitations to container-managed persistence. While admittedly obscure, there is a potential problem if container-managed fields of different entity beans map to the same data item in the underlying database. In a case such as this, the different entity beans may see an inconsistent view of the data item if they are invoked within the same transaction. In addition, with container-managed persistence, the container may load the entire state of the entity object into the bean instance’s container-managed fields before invoking the `ejbLoad()` method. This may lead to performance problems if the entity object has a large amount of state.
Understanding the lifecycle of an entity bean

There are three distinct stages or states in the lifecycle of an entity bean, as follows:

- **Nonexistent.** The entity bean instance does not exist.
- **Pooled.** The entity bean instance exists but it is not associated with a particular entity object identity; that is, there is no specific data associated with the instance.
- **Ready.** The entity bean instance is associated with a particular entity object; that is, there is specific data associated with the instance.

Figure 7.1 depicts the lifecycle of an entity bean instance. Every entity bean must extend the `javax.ejb.EntityBean` interface. An EJB Container uses the methods of the `EntityBean` interface to notify an entity bean instance of lifecycle events. See “Entity bean interface” on page 7-6 for more information on this interface. The methods that appear in the lifecycle diagram are implemented in the `EntityBean` interface.

![Lifecycle of an entity bean instance](image)

A container creates an instance of an entity bean by constructing it. It then invokes the `setEntityContext()` method on the entity bean to pass the instance a reference to its context; that is, a reference to the `EntityContext` interface. The `EntityContext` interface gives the instance access to container-provided services and allows it to obtain information about its clients.

The entity bean instance is now in the pooled state. Each type of entity bean has its own pool. None of the instances in the pool are associated with data; that is, they have no identity—none of their instance variables have been set—and, thus, they are...
Understanding the lifecycle of an entity bean

all equivalent. The container is free to assign any instance to a client that requests such an entity bean. The container implementation determines when to create entity bean instances and how many such instances should be in the pool.

When a client application invokes one of the finder methods for an entity bean, the container executes the corresponding `ejbFind()` method on an arbitrary instance in the pooled state. The instance does not move to the ready state during the execution of a finder method.

When the container selects an instance to service a client’s request to an entity object, that instance moves from the pooled to the ready state. There are two ways that an entity instance transitions from the pooled state to the ready state:

- Through the `ejbCreate()` and `ejbPostCreate()` methods.
- Through the `ejbActivate()` method.

The container may select the instance to handle a client’s `create()` request on the bean’s home interface. In response to the `create()` call, the container creates an entity object and invokes the `ejbCreate()` and `ejbPostCreate()` methods when the instance is assigned to the entity object. The container uses the `ejbActivate()` method to activate an instance so that it can respond to an invocation on an existing entity object. Typically, the container uses `ejbActivate()` when there is no suitable instance in the ready state to handle the client’s call.

When the instance is in the ready state, it is associated with a specific entity object identity. Clients can invoke the application-specific methods on the entity bean. The container uses the `ejbLoad()` and `ejbStore()` methods to instruct the bean to load and store its data. All these methods—the business methods, `ejbLoad()`, and `ejbStore()`—may be invoked multiple times, or not at all, according to the needs of the client application. The `ejbLoad()` and `ejbStore()` methods enable the bean instance to synchronize its state with that of the underlying data entity.

When an entity instance moves back to the pooled state, the instance is decoupled from the data represented by the entity. The container can now assign the instance to any entity object within the same entity bean home. There are two means by which the container moves an entity instance from the ready state to the pooled state:

- The container passivates the entity bean instance by invoking the `ejbPassivate()` method. It uses the `ejbPassivate()` method to disassociate the instance from its entity identity without removing the underlying entity object. The container typically passivates a bean when the transaction completes, However, it may choose to passivate the bean instance within a transaction; if so, it first invokes the `ejbStore()` method so that the instance can synchronize its state with the database state. When the instance is later activated, the container automatically invokes the `ejbLoad()` method to again synchronize the state between the instance and the database.

- The container removes the entity object by invoking the `ejbRemove()` method. It uses the `ejbRemove()` method in response to the client application invoking the `remove()` method on the entity object’s remote or home interface. The container removes the bean instance and the underlying entity object (the data from the database).
Implementing an entity bean

Implementing an entity bean is very much like implementing a session bean. You must implement a home interface, a remote interface, and the entity bean implementation class. The entity bean class must implement the methods that correspond to those declared in the remote and home interfaces.

An entity bean implementation class extends the EntityBean interface rather than the SessionBean interface.

Refer to “Remote interface” on page 4-4 for information on developing an entity bean’s remote interface. Refer to “Entity bean home interface” on page 4-8 for information on developing an entity bean’s home interface. You should have read the material on session beans in Chapter 6, “Writing session beans.”

An entity bean class must implement:

- The EntityBean interface.
- Methods that correspond to methods defined in the home interface—in particular, the create methods and, for bean-managed persistence, the finder methods.
- Methods that correspond to methods defined in the remote interface—the business methods and the EntityBean interface methods.

The entity bean class also contains the implementation of the business methods that pertain to this entity object. In addition, if the bean does bean-managed persistence, it must include methods to access and update the database.

Entity bean interface

An entity bean implementation must extend the javax.ejb.EntityBean interface and implement these interface methods. The container uses these EntityBean interface methods to notify the entity bean instance of lifecycle events. The EntityBean interface is parallel in functionality to the SessionBean interface, which applies to session beans.

When an entity bean uses container-managed persistence, the methods from the EntityBean interface (except for the two context-related methods) and the ejbPostCreate() method serve as notification callbacks. This means that the bean implementation has to provide only a skeletal implementation for them. While not required, the bean developer is free to add application-specific code to these methods.

Entity beans that use bean-managed persistence must provide a more complete implementation of these same methods. Because a bean-managed entity bean is solely responsible for its database management, it must correctly handle reading data from the database, updating database entries, and removing an entity object from the database. It accomplishes these operations by the appropriate implementation of the
Implementing an entity bean

EntityBean interface methods—ejbLoad(), ejbStore(), ejbRemove()—and the ejbCreate() method.

The entity bean interface definition appears in Code sample 7.1.

**Code sample 7.1  EntityBean interface definition**

```java
package javax.ejb;
import java.rmi.RemoteException;

public interface EntityBean extends EnterpriseBean {
    public void setEntityContext(EntityContext ctx) throws EJBException, RemoteException;
    public void unsetEntityContext() throws EJBException, RemoteException;
    public void ejbRemove() throws RemoveException, EJBException, RemoteException;
    public void ejbActivate() throws EJBException, RemoteException;
    public void ejbPassivate() throws EJBException, RemoteException;
    public void ejbLoad() throws EJBException, RemoteException;
    public void ejbStore() throws EJBException, RemoteException;
}
```

The EntityBean interface methods have the following semantics.

- **setEntityContext()** sets an entity context. The container uses this method to pass a reference to the EntityContext interface to the bean instance. The entity context interface provides methods to access properties of the runtime context for the entity bean. An entity bean instance that uses this context in its lifetime needs to store it in an instance variable.

- **unsetEntityContext()** is called by the container before it terminates the life of the current instance of the entity bean. The bean can free resources that have been allocated during the setEntityContext() call.

- **ejbRemove()** removes the database entry or entries associated with this particular entity bean. The container calls this method when a client invokes a remove() method.

- **ejbActivate()** notifies an entity bean that it has been activated. The container invokes this method on the instance selected from the pool of available instances and assigned to a specific entity object identity. When activated, the entity bean instance has the opportunity to acquire additional resources that it might need.

- **ejbPassivate()** notifies an entity bean that it is about to be deactivated—that is, the instance is about to be disassociated from an entity object identity and returned to the pool of available instances. The instance has the chance to release any resources allocated during the ejbActive() method which it might not want to hold while in the pool.

- **ejbLoad()** refreshes the data the entity object represents from the database. The container invokes this method on the entity bean instance so that the instance synchronizes the entity state cached in its instance variables to the entity state in the database.

- **ejbStore()** stores the data the entity object represents in the database. The container invokes this method on the entity bean instance so that the instance synchronizes the database state to the entity state cached in its instance variables.
Implementing an entity bean

Entity bean methods

An entity bean typically declares and implements create and finder methods.

Create methods
An entity bean must declare and implement \texttt{ejbCreate()} methods and \texttt{ejbPostCreate()} that correspond to the \texttt{create()} methods defined in its home interface. Keep in mind that entity beans are not required to implement create methods. A create method on an entity bean results in the insertion of a new entity object to the database. You may have entity beans without create methods if new instances of entity objects should only be added to the database via DBMS updates or through a legacy application.

The names for the create methods in the bean implementation match those used in the home interface, with the addition of the prefix "ejb." The parameters, both in number and type, must match those defined in the home interface’s corresponding \texttt{create()} methods.

The \texttt{ejbCreate()} methods in the entity bean insert a new entity object into the underlying database. While the home interface’s \texttt{create()} methods return a reference to the remote interface, the corresponding \texttt{ejbCreate()} methods in the implementation class return the primary key type. For container-managed entity beans, the \texttt{ejbCreate()} methods return a value of \texttt{null}. This is because the container has complete responsibility for creating the entity bean. For bean-managed entity beans, the \texttt{ejbCreate()} methods return an instance of the primary key class for the newly created entity object. The container uses this primary key to create the actual entity bean instance.

The signature for the \texttt{ejbCreate()} methods for entity beans is the same, regardless if the bean uses bean-managed persistence or container-managed persistence. The signature is:

\begin{verbatim}
public <PrimaryKeyClass> ejbCreate( <zero or more parameters> )
// implementation
}
\end{verbatim}

Note that the signature for the \texttt{ejbCreate()} methods for entity beans must be defined to return the type of the primary key class. Entity beans with container-managed persistence do not have a primary key class to return, because they let the container create the actual entity instance. Instead, the Bean Developer codes the \texttt{ejbCreate()} method to return a \texttt{null} value.

The container executes the \texttt{ejbCreate()} method in response to the client’s invocation of the \texttt{create()} method and inserts a record representing the entity object into the database. The container then invokes a matching \texttt{ejbPostCreate()} method to allow the instance to complete its initialization. The \texttt{ejbPostCreate()} method matches the \texttt{ejbCreate()} method in its arguments, but it returns \texttt{void}. Its signature is:

\begin{verbatim}
public void ejbPostCreate( <zero or more parameters> ) throws RemoteException {
    // implementation
}
\end{verbatim}

The \texttt{ejbCreate()} methods typically initialize some entity state. Thus, they often have one or more parameters and their implementation includes code that sets the entity
state to the parameter values. For example, the bank example has a checking account
entity bean whose ejbCreate() method takes two parameters, a string and a float
amount. The method initializes the name of the account to the string value and the
account balance to the float amount, as shown in Code sample 7.2:

Code sample 7.2 Bank example ejbCreate() method

```java
public AccountPK ejbCreate(String name, float balance) {
    this.name = name;
    this.balance = balance;
    return null;
}
```

Finder methods

An entity bean that uses bean-managed persistence must also implement finder
methods that correspond to the home interface finder methods. The finder method
names in the bean implementation match those used in the home interface, with the
addition of the prefix “ejb.” The parameters—number and type—must match those
defined in the home interface’s corresponding finder methods.

An entity bean that uses container-managed persistence does not implement finder
methods to match those in the home interface. For these beans, the container
provides the implementations of the finder methods. However, you have to provide
information in the deployment descriptor so that the container knows how to
implement the respective finder methods. The Inprise EJB Container uses special
fields in the deployment descriptor.

The home interface must provide a default finder method,
findByPrimaryKey(primaryKey), to allow a client to locate a single entity object using a
primary key. The method has a single argument, the primary key, and its return type
is the entity bean’s remote interface, as follows:

```java
public <entity bean's remote interface> findByPrimaryKey(<primary key type> key )
```

The home interface may define additional finder methods. Each finder method,
including the findByPrimaryKey() method, must have a corresponding implementation
in the entity bean class (if the bean uses bean-managed persistence). Similar to the
ejbCreate() methods, the entity bean class has an ejbFindByPrimaryKey() method and
ejbFind<methodname>() methods that match the home interface’s finder methods.

A finder method returns a single entity instance or multiple instances. In either case,
if the entity bean uses container-managed persistence, a finder method returns a
java.util.Collection type, if using Java 2, or a java.util Enumeration type if using an
earlier JDK. When you extract the individual entity instances from the Collection or
Enumeration, you cast each instance to the bean’s remote interface type.

Likewise, if the entity bean uses bean-managed persistence and the finder method
returns more than one entity instance, it should return a collection—either a
Collection or an Enumeration—of primary keys. The container transforms these
primary keys into a collection of remote interfaces.
Implementing an entity bean

Business logic

The entity bean must also implement the business methods defined in the remote interface. The signatures for the bean’s business methods must exactly match those of the corresponding methods defined in the remote interface.

There is no need for a constructor in the entity bean class, and you can omit it. The ejbCreate() methods do all bean initialization.

The entity bean implementation, when using container-managed persistent, must also declare as public those variables that it wants the container to manage. An entity bean using bean-managed persistence may declare its variables public or private.

Method synchronization

Method synchronization pertains to concurrent access to entity beans and reentrant entity beans.

Concurrent access to entity beans

The container manages the synchronization between concurrent invocations and the bean provider need not be concerned with this. The container typically uses a strategy of database synchronization or container synchronization.

With database synchronization, the container provides multiple instances of the same entity bean, then leaves the synchronization to the database access calls in the ejbCreate(), ejbRemove(), ejbLoad(), and ejbStore() methods.

For container synchronization, the container serializes the invocations of the entity bean instances. The container creates only one instance of the entity bean at a time, and that entity bean instance acquires an exclusive lock on the database. The container keeps subsequent instances in a queue, and invokes the next entity bean instance only when the currently executing one completes its work.

Reentrant entity beans

By default, entity beans are not reentrant. When a call within the same transaction context arrives at the entity bean, it causes the exception java.rmi.RemoteException to be thrown.

You can declare an entity bean reentrant in the deployment descriptor; however, take special care in this case. The critical issue is that a container can generally not distinguish between a (loopback) call within the same transaction and a concurrent invocation (in the same transaction context) on that same entity bean.

When the entity bean is marked reentrant, it is illegal to allow a concurrent invocation within the same transaction context on the bean instance. It is the programmer’s responsibility to ensure this rule.
Entity bean primary keys

Every entity bean instance must have a primary key. A primary key is a value (or combination of values) that uniquely identifies the entity instance. For example, a database table that contains employee records might use the employee’s social security number for its primary key. Likewise, the entity bean modeling this employee table would also use the social security number for its primary key.

With enterprise beans, the primary key is represented by a Java class containing the unique data. This primary key class can be any class as long as that class is a legal value type in RMI-IIOP—which means it extends the java.io.Serializable interface. It must also provide an implementation of the Object.equals(Object other) and Object.hashCode() methods, two methods which all Java classes inherit by definition.

The primary class may be specific to an particular entity bean class. That is, each entity bean may define its own primary key class. Or, multiple entity beans may share the same primary key class.

The bank application uses two different entity beans to represent savings and checking accounts. Both types of accounts use the same field to uniquely identify a particular account record. In this case, they both use the same primary key class, AccountPK, to represent the unique identifier for either type of account. Code sample 7.3 shows the definition of the account primary key class:

Code sample 7.3  Account primary key class

```java
public class AccountPK implements java.io.Serializable {
  public String name;
  public AccountPK() {}  
  public AccountPK(String name) {
    this.name = name;
  }
}
```

Handling transactions with optimistic concurrency

The Inprise EJB container does not lock entity beans with respect to transactions. This means that multiple client transactions can access the same “conceptual” entity bean in parallel, but they all run against different “physical” copies of the data. This can be problematic in practice for most DBMSs.

For example, suppose three different transactions were each accessing the SavingsAccount entity bean to update a field in the Account record. If they try to update the same field, one transaction could overwrite the data just committed by another transaction with no knowledge of what that other transaction had done, which is not a good situation. Or, if they are accessing different fields in the same record, only one transaction executes successfully and the other two rollback.

The Inprise container uses a form of optimistic concurrency to handle this situation with the least amount of disruption to executing transactions and also to maintain data integrity. Essentially, the container checks the value in the database row that the transaction is about to modify. If the value in the database row at the time of the
write operation is the same as the value read in at the start of the transaction, then the container does the update. If not—that is, the value in the database now differs from the value read at the start of the transaction, then the container disallows the second transaction and rolls it back.

The container takes a snapshot of the entity bean at the time it is first read in. Then, over the course of the transaction, other transactions modify the bean. At the end of the original transaction, the container determines which fields have been modified, and does a tuned update only of the modified fields. The container allows transactions to complete successfully if they are modifying different fields in the same database row. However, if two transactions attempt to modify the same field simultaneously—both transactions have read in the same data from the row, and one transaction has already committed its update—the container rolls back the second transaction when it attempts its commit because the data in the field has changed and this second transaction (or “last to commit” transaction) had no knowledge of that change.

**Bank entity bean example**

The bank example illustrates the use of entity beans. It includes two implementations of the same Account remote interface: one implementation uses bean-managed persistence and the other implementation uses container-managed persistence.

Savings accounts are modeled by the Account entity bean called SavingsAccount. They illustrate bean-managed persistence. When you examine the entity bean code you’ll see that it includes direct JDBC calls.

Checking accounts are modeled by the Account entity bean called CheckingAccount. They rely on the container to implement persistence; that is, they illustrate container-managed persistence.

A third Teller enterprise bean transfers funds from one account to the other. This stateless session bean shows how calls to multiple entity beans can be grouped within a single container-managed transaction. Thus, even if the credit occurs before the debit in the transfer operation, the container rolls back transaction if the debit fails, and neither the debit nor the credit occurs.

**Entity bean home interface**

Multiple entity beans can share the same home and remote interfaces, even if one entity bean uses container-managed persistence and the other uses bean-managed persistence. Both the SavingsAccount and CheckingAccount entity beans use the same home interface, called AccountHome. Likewise, they use the same remote interface.

The home interface for entity beans is very much like the home interface for session beans. They extend the same `javax.ejb.EJBHome` interface. The home interface for entity beans must include at least one finder method and a create method is optional.

Code sample 7.4 shows the code for the AccountHome interface.
Bank entity bean example

Code sample 7.4  AccountHome home interface

```java
public interface AccountHome extends javax.ejb.EJBHome {
    Account create(String name, float balance)
        throws java.rmi.RemoteException, javax.ejb.CreateException;
    Account findByPrimaryKey(AccountPK primaryKey)
        throws java.rmi.RemoteException, javax.ejb.FinderException;
    java.util.Enumeration findAccountsLargerThan(float balance)
        throws java.rmi.RemoteException, javax.ejb.FinderException;
}
```

The AccountHome home interface implements three methods. While the create() method is not required for entity beans, the bank example does implement one. The create() method inserts a new entity object into the underlying database. The entity bean can choose to defer the creation of new entity objects to the DBMS or to another application, in which case it would not define a create() method.

The create() method takes two parameters, an account name string and a balance amount. The implementation of this method in the entity bean class uses these two parameter values to initialize the entity object state—the account name and the starting balance—when it creates a new object. The throws clause of a create() method must include the java.rmi.RemoteException and the java.ejb.CreateException; it may include additional application-specific exceptions.

Entity beans must have the findByPrimaryKey() method; the AccountHome interface declares this method. It takes one parameter, the AccountPK primary key class, and returns a reference to the Account remote interface. This method finds one particular account entity and returns a reference to it.

While not required, the home interface also declares a second finder method, findAccountsLargerThan(). This method returns a Java Enumeration containing all the accounts whose balance is greater than some amount.

All finder methods must include the java.rmi.RemoteException and java.ejb.FinderException exceptions in their throws clause.

Entity bean remote interface

More than one entity bean can use the same remote interface, even when the beans use different persistence management strategies. The bank example’s two entity beans both use the same Account remote interface.

The remote interface for entity beans is virtually identical to the remote interface for session beans—it extends the javax.ejb.EJBObject interface and exposes the business methods available to clients.

Code sample 7.5 shows its implementation:

Code sample 7.5  Account remote interface

```java
public interface Account extends javax.ejb.EJBObject {
    public float getBalance()
        throws java.rmi.RemoteException;
    public void credit(float amount) throws java.rmi.RemoteException;
    public void debit(float amount) throws java.rmi.RemoteException;
}
```
Bank entity bean example

The Account remote interface declares three business methods: `getBalance()`, `credit()`, and `debit()`.

Entity bean with container-managed persistence

The bank example implements a CheckingAccount entity bean that illustrates the basics for using container-managed persistence. In many ways, this implementation is like a session bean implementation. However, there are some key things to note in the implementation of an entity bean that uses container-managed persistence:

- The entity bean has no implementations for finder methods. The EJB Container provides the finder method implementations for entity beans with container-managed persistence. Rather than providing the implementation for finder methods in the bean's class, the deployment descriptor contains information that enables the container to implement these finder methods.

- The entity bean declares all fields `public` that are managed by the container for the bean. The CheckingAccount bean declares `name` and `balance` to be public fields.

- The entity bean class implements the seven methods declared in the `EntityBean` interface—`ejbActivate()`, `ejbPassivate()`, `ejbLoad()`, `ejbStore()`, `ejbRemove()`, `setEntityContext()`, and `unsetEntityContext()`. However, the entity bean is required to provide only skeletal implementations of these methods, though it is free to add application-specific code where appropriate. The CheckingAccount bean saves the context returned by `setEntityContext()` and releases the reference in `unsetEntityContext()`. Otherwise, it adds no additional code to the `EntityBean` interface methods.

- There is an implementation of the `ejbCreate()` method (because this entity bean allows callers of the bean to create new checking accounts), and the implementation initializes the instance's two variables—account name and balance amount—to the argument values. The `ejbCreate()` method returns a `null` value because, with container-managed persistence, the container creates the appropriate reference to return to the client.

- The entity bean provides the minimal implementation for the `ejbPostCreate()` method, though this method could have performed further initialization work if needed. For beans with container-managed persistence, it is sufficient to provide just the minimal implementation for this method because `ejbPostCreate()` serves as a notification callback. Note that the same rule applies to the methods inherited from the `EntityBean` interface as well.

Code sample 7.6 Implementation of an entity bean using container-managed persistence

```java
import javax.ejb.*;
import java.rmi.RemoteException;

public class CheckingAccount implements EntityBean {
    private javax.ejb.EntityContext _context;
    public String name;
    public float balance;
```
Bank entity bean example

```java
public float getBalance() {
    return balance;
}

public void debit(float amount) {
    if(amount > balance) {
        // mark the current transaction for rollback...
        _context.setRollbackOnly();
    } else {
        balance = balance - amount;
    }
}

public void credit(float amount) {
    balance = balance + amount;
}

public AccountPK ejbCreate(String name, float balance) {
    this.name = name;
    this.balance = balance;
    return null;
}

public void ejbPostCreate(String name, float balance) {}
public void ejbRemove() {}
public void setEntityContext(EntityContext context) {
    _context = context;
}

public void unsetEntityContext() {
    context = null;
}

public void ejbActivate() {}
public void ejbPassivate() {}
public void ejbLoad() {}
public void ejbStore() {
    return "CheckingAccount\[name=\" + name + ",balance=\" + balance +\"]";
}
}
```

Entity bean with bean-managed persistence

The bank example also implements a SavingsAccount entity bean. The SavingsAccount entity bean is a good illustration of bean-managed persistence.

The SavingsAccount bean accesses a different account table than the CheckingAccount bean. Though these two entity beans use different persistence-management approaches, they may both use the same home and remote interfaces. However, the implementation class for the SavingsAccount bean is different in some key ways from the CheckingAccount implementation.
Bank entity bean example

An entity bean implementation with bean-managed persistence does the following:

- It can declare its instance variables to be private rather than public. The bean includes its own code to access these variables—to load the database values into these instance variables and to store their changes to the database. As such, it can limit access to these variables as it sees fit. This differs from a bean using container-managed persistence, which must declare all container-managed variables to be public so that the container can access them.

- The ejbCreate() method returns the primary key class AccountPK. The container takes the returned primary key class and uses it to construct a remote reference to the entity bean instance.

- Just like beans with container-managed persistence, a bean with bean-managed persistence may optionally provide more than an empty implementation of the ejbPostCreate() method. The SavingsAccount bean does not need to include additional initialization code in this method.

- There are implementations for the ejbLoad() and ejbStore() methods. A bean using container-managed persistence typically provides only an empty implementation of these methods because the container handles persistence. An entity bean with bean-managed persistence must provide its own code to read the database values into its instance variables—the ejbLoad() method—and to write the database with changed values—the ejbStore() method.

- There must be implementations for all finder methods. The SavingsAccount entity bean implements two finder methods: the required ejbFindByPrimaryKey() method and the optional ejbFindAccountsLargerThan() method.

- An entity bean with bean-managed persistence must implement the ejbRemove() method, declared in the EntityBean interface. Because the bean is managing the underlying database entity object, it must implement this method so that it can remove the entity object from the database. A bean with container-managed persistence will omit the implementation of this method because the container is responsible for the database management.

- Each method that accesses the underlying database object—ejbCreate(), ejbRemove(), ejbLoad(), ejbStore(), ejbFindByPrimaryKey(), and other finder methods and business methods—must include the correct database access code. Each such method includes code to connect to the database, followed by code to build and then execute SQL statements that accomplish the functionality encompassed by the method. When the SQL statements complete, the method closes the statements and the database connection before returning.

Code sample 7.7 shows the interesting code portions from the SavingsAccount implementation class. The example removes the code that is merely the empty implementations of the EntityBean interface methods—ejbActivate(), ejbPassivate(), and so forth.

Let’s examine the ejbLoad() method, which accesses the database entity object, to see how a bean with bean-managed persistence implements database access. Note that all of the methods implemented in the SavingsAccount class follow the same approach as ejbLoad(). The ejbLoad() method begins by establishing a connection to the database. It calls the internal getConnection() method, which uses a DataSource to...
obtain a JDBC connection to the database from a JDBC connection pool. Once the connection is established, \texttt{ejbLoad()} creates a \texttt{PreparedStatement} object and builds its SQL database access statement. Because \texttt{ejbLoad()} reads the entity object values into the entity bean’s instance variables, it builds an SQL \texttt{SELECT} statement for a query that selects the balance value for the savings account whose name matches a pattern. The method then executes the query. If the query returns a result, it extracts the balance amount. The \texttt{ejbLoad()} method finishes by closing the \texttt{PreparedStatement} objects and then closing the database connection. Note that the \texttt{ejbLoad()} method does not actually close the connection; instead, it simply returns the connection to the connection pool.

\textbf{Code sample 7.7} Implementation of an entity bean using bean-managed persistence

```java
import java.sql.*;
import javax.ejb.*;
import java.util.*;
import java.rmi.RemoteException;

public class SavingsAccount implements EntityBean {
    private EntityContext _context;
    private String _name;
    private float _balance;

    public float getBalance() {
        return _balance;
    }

    public void debit(float amount) {
        if(amount > _balance) {
            // mark the current transaction for rollback...
            _context.setRollbackOnly();
        } else {
            _balance = _balance - amount;
        }
    }

    public void credit(float amount) {
        _balance = _balance + amount;
    }

    // setEntityContext(), unsetEntityContext(), ejbActivate(), ejbPassivate(),
    // ejbPostCreate() skeleton implementations not shown
    ...

    public AccountPK ejbCreate(String name, float balance) throws RemoteException, CreateException {
        _name = name;
        _balance = balance;
        try {
            Connection connection = getConnection();
            PreparedStatement statement = connection.prepareStatement("INSERT INTO Savings_Accounts (name, balance) VALUES (?, ?)");
            statement.setString(1, _name);
            statement.setFloat(2, _balance);
            if(statement.executeUpdate() != 1) {
                throw new CreateException("Could not create: " + name);
            }
            statement.close();
        }
```
Bank entity bean example

connection.close();
return new AccountPK(name);
} catch(SQLException e) {
    throw new RemoteException("Could not create: " + name, e);
}
}

public void ejbRemove() throws RemoteException, RemoveException {
    try {
        Connection connection = getConnection();
        PreparedStatement statement = connection.prepareStatement
                ("DELETE FROM Savings_Accounts WHERE name = ?");
        statement.setString(1, _name);
        if(statement.executeUpdate() != 1) {
            throw new RemoveException("Could not remove: " + _name);
        }
        statement.close();
        connection.close();
    } catch(SQLException e) {
        throw new RemoteException("Could not remove: " + _name, e);
    }
}

public AccountPK ejbFindByPrimaryKey(AccountPK key) throws RemoteException,
        FinderException {
    try {
        Connection connection = getConnection();
        PreparedStatement statement = connection.prepareStatement
                ("SELECT name FROM Savings_Accounts WHERE name = ?");
        statement.setString(1, key.name);
        ResultSet resultSet = statement.executeQuery();
        if(!resultSet.next()) {
            throw new FinderException("Could not find: " + key);
        }
        statement.close();
        connection.close();
        return key;
    } catch(SQLException e) {
        throw new RemoteException("Could not find: " + key, e);
    }
}

public java.util.Enumeration ejbFindAccountsLargerThan(float balance)
    throws RemoteException, FinderException {
    try {
        Connection connection = getConnection();
        PreparedStatement statement = connection.prepareStatement
                ("SELECT name FROM Savings_Accounts WHERE balance > ?");
        statement.setFloat(1, balance);
        ResultSet resultSet = statement.executeQuery();
        Vector keys = new Vector();
        while(resultSet.next()) {
            String name = resultSet.getString(1);
            keys.addElement(new AccountPK(name));
        }
    } catch(SQLException e) {
        throw new RemoteException("Could not find: " + name, e);
    }
}
Bank entity bean example

```java
statement.close();
connection.close();
return keys.elements();
} catch(SQLException e) {
    throw new RemoteException("Could not findAccountsLargerThan: " + balance, e);
}
}

public void ejbLoad() throws RemoteException {
    // get the name from the primary-key
    _name = ((_AccountPK) _context.getPrimaryKey()).name;
    try {
        Connection connection = getConnection();
        PreparedStatement statement = connection.prepareStatement
            ("SELECT balance FROM Savings_Accounts WHERE name = ?");
        statement.setString(1, _name);
        ResultSet resultSet = statement.executeQuery();
        if(!resultSet.next()) {
            throw new RemoteException("Account not found: " + _name);
        }
        _balance = resultSet.getFloat(1);
        statement.close();
        connection.close();
    } catch(SQLException e) {
        throw new RemoteException("Could not load: " + _name, e);
    }
}

public void ejbStore() throws RemoteException {
    try {
        Connection connection = getConnection();
        PreparedStatement statement = connection.prepareStatement
            ("UPDATE Savings_Accounts SET balance = ? WHERE name = ?");
        statement.setFloat(1, _balance);
        statement.setString(2, _name);
        statement.executeUpdate();
        statement.close();
        connection.close();
    } catch(SQLException e) {
        throw new RemoteException("Could not store: " + _name, e);
    }
}

private Connection getConnection() throws SQLException {
    Properties properties = _context.getEnvironment();
    String url = properties.getProperty("db.url");
    String username = properties.getProperty("db.username");
    String password = properties.getProperty("db.password");
    if(username != null) {
        return DriverManager.getConnection(url, username, password);
    } else {
        return DriverManager.getConnection(url);
    }
}
```
Bank entity bean example

```java
public String toString() {
    return "SavingsAccount\[name=\" + _name + ",balance=\" + _balance +\"];"
}
}
```

Entity bean deployment descriptor

The deployment descriptor for the bank example deploys three kinds of beans—the Teller session bean, the CheckingAccount entity bean with container-managed persistence, and the SavingsAccount entity bean with bean-managed persistence. You should refer to Chapter 9, “Deploying Enterprise JavaBeans,” for complete information on deploying an enterprise bean. This section focuses on the deployment descriptor for entity beans.

You use properties in the deployment descriptor to specify information about the entity bean’s interfaces, transaction attributes, and so forth, just like you do for session beans, plus information that is unique to an entity bean. Code sample 7.8 on page 7-21 shows typical deployment descriptor property tags for an entity bean with bean-managed persistence. Code sample 7.9 on page 7-22 illustrates the typical deployment descriptor tags for an entity bean that uses container-managed persistence. When you compare the descriptor tags for the two types of entity beans, you will notice that the deployment descriptor for an entity bean with container-managed persistence is more complex.

An entity bean deployment descriptor specifies such information as:

- For an entity bean, the bean’s deployment descriptor type is set to `<entity>`. Notice that the first tags within the `<enterprise-beans>` section in both Code sample 7.8 and Code sample 7.9 specify that the bean is an entity bean.
- The names of the related interfaces (home and remote) and the bean implementation class. Every enterprise bean specifies its home interface using the `<home>` tag, its remote interface using the `<remote>` tag, and its implementation class name using the `<ejb-class>` tag.
- The JNDI names under which the entity bean is registered and by which clients locate the bean.
- The bean’s transaction attributes and its transaction isolation level. This typically appears in the `<assembly-descriptor>` section of the deployment descriptor.
- The name of the entity bean’s primary key class. In our example, the primary key class is `AccountPK` and it appears within the `<prim-key-class>` tag.
- The persistence used by the bean. An entity bean uses either container-managed persistence or bean-managed persistence. The CheckingAccount bean uses container-managed persistence, so the deployment descriptor sets the `<persistence-type>` tag to `Container`.
- The reentrancy of the bean. If the bean class is reentrant or not. Neither the SavingsAccount nor the CheckingAccount bean is reentrant, so the `<reentrant>` tag is set to `False` for both.
The fields that the container manages, if the bean uses container-managed persistence. A bean that uses bean-managed persistence does not specify any container-managed fields. Thus, the deployment descriptor for the SavingsAccount bean does not specify any container-managed fields. An entity bean using container-managed persistence must specify the names of its fields or instance variables that the container must manage. Use a combination of the `<cmp field>` and `<field name>` tags for this. The first tag, `<cmp field>`, indicates that the field is container-managed. Within this tag, the `<field name>` tag specifies the name of the field itself. For example, the CheckingAccount bean deployment descriptor indicates that the balance field is container-managed as follows:

```
<cmp field><field name>balance</field name></cmp field>
```

Information related to the database entity objects. In particular, this information identifies the database (including user information) and the table within the database that holds the entity object. You can provide this information using the Inprise-specific `<datasource>` section. This is discussed in more detail in Chapter 9, “Deploying Enterprise JavaBeans,” in the section “Datasource” on page 9-8.

For container-managed beans only, you must provide information in the deployment descriptor about the container-managed fields so that the container can generate the finder methods for these fields. Use the `<datasource>` tag for the Inprise-specific datasource section. See “Datasource” on page 9-8 in Chapter 9, “Deploying Enterprise JavaBeans.”

Code sample 7.8 below shows the key parts of the deployment descriptor for an entity bean using bean-managed persistence. Because the bean handles its own fetches from the database entity values and updates to these values, rather than the container, the descriptor does not specify fields for the container to manage. Nor does it tell the container how to implement its finder methods, because the bean has already provided their implementation.

**Code sample 7.8 Deployment descriptor for a bean-managed entity bean**

```
...<enterprise-beans>
  <entity>
    <description>This Entity bean is an example of Bean Managed Persistence</description>
    <ejb-name>savings</ejb-name>
    <home>AccountHome</home>
    <remote>Account</remote>
    <ejb-class>SavingsAccount</ejb-class>
    <persistence-type>Bean</persistence-type>
    <prim-key-class>AccountPK</prim-key-class>
    <reentrant>False</reentrant>
  </entity>
...<assembly-descriptor>
  <container-transaction>
    <method>
      <ejb-name>savings</ejb-name>
      <method-name>*</method-name>
    </method>
  </container-transaction>
</enterprise-beans>
```
Bank entity bean example

<trans-attribute>Required</trans-attribute>
</container-transaction>
</assembly-descriptor>

Code sample 7.9 shows the key parts of the deployment descriptor for an entity bean using container-managed persistence. Because the bean lets the container handle loading database entity values and updating these values, the descriptor specifies the fields that the container will manage.

Code sample 7.9  Deployment descriptor for a container-managed entity bean

<enterprise-beans>
<entity>
<description>
This Entity bean is an example of Container Managed Persistence </description>
<ejb-name>checking</ejb-name>
<home>AccountHome</home>
<remote>Account</remote>
<ejb-class>CheckingAccount</ejb-class>
<persistence-type>Container</persistence-type>
<primkey-class>AccountPK</primkey-class>
<reentrant>False</reentrant>
<cmp-field>
<field-name>name</field-name>
</cmp-field>
<cmp-field>
<field-name>balance</field-name>
</cmp-field>
</entity>
<assembly-descriptor>
<container-transaction>
<method>
<ejb-name>checking</ejb-name>
<method-name>*</method-name>
</method>
<trans-attribute>Required</trans-attribute>
</container-transaction>
</assembly-descriptor>

Using the datasource section

An entity bean that uses container-managed persistence configures itself in the deployment descriptor using the Inprise-specific datasource section. The datasource section specifies an entry point into the database and tells the program how to instantiate the data source.

You can specify all the information about the database—such as the database URL, password, username, and so forth—using the datasource section and the <datasource> element.

Refer to “Datasource” on page 9-8 for more information on the datasource section of the deployment descriptor. Also refer to “Datasource” on page 9-8 in Chapter 9, “Deploying Enterprise JavaBeans.”
Deploying the bank enterprise beans

Because this example relies on an external database for persistence, deploying the enterprise beans is a little more involved than the other examples. For more details about creating tables and modifying the deployment descriptor, see “An example using Oracle” on page 7-23.

The deployment steps are as follows:

1. Obtain a database which can be accessed via JDBC.
   1. Set up a user account and give it the ability to create new tables.
   2. Create two tables in the database and give each the following layout: a variable length string of at least 5 characters in length and a floating point field with at least 3 digits of precision. Call one table “Savings_Accounts” and the other “Checking_Accounts”.
   3. Modify the deployment descriptor in the EJB-JAR.XML file in the bank example meta-inf directory. Specify the URL for the JDBC driver and database, user name, and password.
   4. Add the JDBC driver to your classpath.
   5. Start the Inprise EJB container. The command line to start the container is:
      ```
prompt% vbj com.inprise.ejb.Container test beans.jar -jts -jns
      ```
   6. Start the client using this command:
      ```
prompt% vbj BankClient
      ```

You should see the following output:

Peter's balance: 200
Paul's balance: 100
Taking from Peter and giving to Paul
Peter's balance: 200
Paul's balance: 100

Using debug mode

If you run into trouble, use the standard EJB debugging mode (EJBDebug). To run the server with debugging mode enabled:

```
prompt% vbj -DEJBDebug com.inprise.ejb.Container test beans.jar -jts -jns
``` 

An example using Oracle

In this example we deploy with an Oracle 7.3.3 server, running on the machine “gemini”, port 1525, with the alias of “GEORA733”. The account or user name is “scott” and the password is “tiger”. 

An example using Oracle
Advanced container-managed persistence issues

To create tables:

```sql
prompt% sqlplus scott/tiger@GEORA73
create table Savings_Accounts (name varchar(10), balance float(10));
create table Checking_Accounts (name varchar(10), balance float(10));
exit;
prompt%
```

You must type everything between the prompts. This creates two tables, each with a variable string field of length 10 or less, and a floating point field with 10 digits of precision.

For this example, you might have a `<datasource>` entry in the deployment descriptor for both the CheckingAccount and SavingsAccount beans, as follows:

```xml
<deployment-descriptor>
  <datasource>
    <res-ref-name>jdbc/SavingsDataSource</res-ref-name>
    <url>jdbc:oracle:thin:@avicenna:1521:avi73a</url>
    <username>scott</username>
    <password>tiger</password>
    <driver-class-name>oracle.jdbc.driver.OracleDriver</driver-class-name>
  </datasource>
</deployment-descriptor>
```

You will need to modify the datasource entries to reflect the host and port on which you are running Oracle, or to reflect the JDBC URL for whatever database you are using.

Advanced container-managed persistence issues

The previous sections covered the basics of container-managed persistence. It described how you must specify the container-managed fields as public fields in the entity bean implementation, and then you must list them as container-managed fields in the deployment descriptor. The home interface declares the finder methods, but provides no implementation for these methods. The deployment descriptor also includes a specification of the logic for the finder methods. The EJB Container uses these deployment descriptor properties to determine the fields it must manage and to know how to implement the various finder methods.

This section addresses some of the more advanced issues with container-managed persistence, especially as implemented in the Inprise EJB Container environment. In particular, this section focuses on the object relational mapping between tables in a relational database and object-oriented entity beans. It also discusses the various mapping strategies that you can implement using the deployment descriptor.

Object relational mapping

You have already seen how a single entity bean maps to a single table in a database; that is, how an instance of an entity bean represents one row in a table and each field
Advanced container-managed persistence issues

in the bean represents a column in the table. The entity bean instance is identified by
a primary key that matches the primary key identifier of the table row.

However, many tables in a database have relationships to other tables. Often, a table
has a one-to-one or a one-to-many (and, conversely, a many-to-one) relationship to
another table. The foreign key reference maintains the relationship between the
tables.

Let’s see how this might look in a relational database that models a simple customer
order system. A customer places an order for some product or products, and this
action generates one order with one or more separate line items for each product. In
the database, there might be an Order table that contains information identifying the
customer order. Each entry or row in the Order table has a primary key column,
order_id, that uniquely identifies this particular order. Each row also has columns
with such information as order date, customer name, total order price, and so forth.
Each row in a second table, called LineItem, represents a line item entry for an order.
A particular order may have one or many line items. A LineItem row has a foreign
key column which is the order_id primary key value identifying the Order row to
which this line item belongs.

Figure 7.2 shows two orders within the Order table; their primary key values are
order1 and order2. The foreign key column for the LineItem table indicates that order1
has three line items and order2 has one line item entry. Each LineItem entry, however,
is connected to one and only one Order entry.

**Figure 7.2** Relationships between database tables and rows

While an entity bean may use bean-managed persistence to represent these
intra-bean relationships, it can also do so when using container-managed persistence.
In fact, by modeling these relationships with entity beans that uses
container-managed persistence, the developer’s coding task is much simpler.

**Implementing a one-to-one relationship**

Entity beans with container-managed persistence implement one-to-one
relationships, such as the one from LineItem to Order, in a straight-forward manner.
To see how an instance of a LineItem bean can find its associated instance of the Order bean, let’s look at the entity beans.

The OrderBean entity bean class, along with its Order remote interface and OrderHome home interface, models the Order database table. Similarly, the LineItemBean entity bean class models the LineItem table, and it has a LineItem remote interface and LineItemHome interface.

The OrderBean entity bean implements public fields that correspond to the columns in the Order table. It declares such public fields as order_id, customer, price, and so on. (Note that an entity bean can instruct the EJB Container to use different names from the SQL column names—there is a property in the deployment descriptor that specifies the mapping between the SQL column name and the Java field name.) The order_id field is the primary key field for OrderBean. The code for OrderBean includes the following field declarations (see Code sample 7.10):

**Code sample 7.10** OrderBean container-managed field declarations

```java
public class OrderBean implements EntityBean{
    public String order_id;
    public String customer;
    public float price;
    ...
```

The LineItemBean entity bean also implements public fields corresponding to the table columns. While its order field represents the foreign key reference to the OrderBean object, the field is implemented not as the primary key class but as the type of the remote interface Order. That is, it is an object reference to the Order objects. In the Inprise EJB environment, any entity bean field using container-managed persistence can correspond to a foreign key column in a database table. The EJB Container treats these foreign key container-managed fields as EJBObject references. The LineItemBean also includes a getOrder() method that returns the associated order object. This is shown in Code sample 7.11.

**Code sample 7.11** LineItemBean container-managed field declarations

```java
public class LineItemBean implements EntityBean {
    public itemno; 
    public Order order;
    ...
    public Order getOrder() { 
        return order;
    }
    ...
```

At the same time, the deployment descriptor for the LineItemBean includes a property that specifies the JNDI name for the OrderHome interface. The container uses this property to locate the correct home interface. Figure 7.3 illustrates this.
Advanced container-managed persistence issues

Figure 7.3 Implementing a one-to-one relationship with a container-managed entity bean

Normally, a container would select the primary key value from the LineItem instance, then it would call the `findByPrimaryKey()` method on the OrderHome home interface to obtain the reference to the correct OrderBean instance.

The Inprise EJB Container uses the foreign key—the object reference to the Order remote interface—in the LineItemBean class to locate the correct OrderBean instance for that line item. It also uses the LineItem’s deployment descriptor entry that specifies the location of the home interface for the Order bean. By using the deployment descriptor OrderHome reference and the foreign key reference from the LineItem object to its Order object, the Inprise EJB Container avoids calling the `findByPrimaryKey()` method on the OrderHome interface. Eliminating the `findByPrimaryKey()` call represents a significant performance saving, because that method typically involves going out to the database to return some data.

The container constructs the object reference to the Order remote interface, and thus to the particular OrderBean instance, directly. However, the container does not load the state from the referenced object—the state from the OrderBean instance—until that object is used. That is, when the container loads the LineItemBean instance state, it does not load the OrderBean state. This is referred to as a lazy reference, and it provides a significant performance saving.

Implementing a many-to-one relationship

The Inprise EJB Container can also handle a many-to-one (or one-to-many) relationship within a container-managed entity bean. Using the previous Order and LineItem example, the container can find all of the line items associated to a particular order.

To do this, the EJB Container matches the `EJBObject` reference that models the foreign key value in the entity bean on the “many” side to the primary key field in the entity
Advanced container-managed persistence issues

bean on the “one” side. In the above example, the Order bean has a primary key that identifies each individual order, with values such as order1 and order2. Each LineItemBean instance—each line item for the order—has a foreign key whose value identifies the order to which this line item is associated.

The process is illustrated in Figure 7.4 and the explanation follows.

**Figure 7.4** Implementation of a one-to-many relationship with container-managed persistence

For this to work, the entity bean implementation on the “one” side of the one-to-many relationship—the OrderBean—must include a method to return all of its associated “many” instances—all LineItemBean entity objects. This method, which we call `getLineItems()`, relies on a finder method in the “many” side entity bean home interface—LineItemHome—that finds its entity objects whose foreign key value matches the primary key value of the “one,” or OrderBean, side. Because this is container-managed, the home interface finder method is just a declaration of the method. The logic for it is specified in the deployment descriptor. A deployment descriptor property tells the EJB Container how to construct the SQL `WHERE` clause to locate the LineItemBean objects whose foreign key values equal the primary key value of the OrderBean side of the relationship.

In our example, the OrderBean includes a method to return a `java.util.Enumeration` or `java.util.Collection` of all line item entity instances. (Use the `Collection` interface if you are using Java 2. Use the `Enumeration` interface if you are using an earlier version of the JDK. You can continue to use the `Enumeration` interface even if you use Java 2; however, the `Collection` interface is not available in earlier JDK versions.) For example, you could have the following method declaration:

```java
java.util.Enumeration getLineItems() throws java.rmi.RemoteException;
```
Advanced container-managed persistence issues

or:

```java
java.util.Collection getLineItems() throws java.rmi.RemoteException;
```

This method uses the entity context of the bean, which it knows from its `setEntityContext()` method, to find the LineItemHome home interface. Code sample 7.12 shows the significant portions of the OrderBean code.

**Code sample 7.12  OrderBean getLineItems() method**

```java
public class OrderBean implements EntityBean {
  private LineItemHome _lineItemHome;
  ...
  public void setEntityContext(EntityContext context) throws java.rmi.RemoteException {
    _context = context;
    ...
  }
  ...
  public java.util.Collection getLineItems() throws java.rmi.RemoteException {
    try {
      Order self = (Order) _context.getEJBObject();
      return _lineItemHome.findByOrder(self);
    } catch (javax.ejb.FinderException e) {...}
  }
  ...
}
```

The OrderBean class declares a `private` field `_lineItemHome` to hold the reference to the LineItemHome interface. In the `setEntityContext()` method, it saves the bean’s context in `_context`. In the `getLineItems()` method, it uses the `_context.getEJBObject()` to get a reference to its remote interface Order. Once it has that reference, the method can call the `findByOrder()` method, defined on the LineItemHome interface, and pass it the reference to its remote interface.

For the `getLineItems()` method to work, the LineItemHome interface must declare the `findByOrder()` method, as follows:

```java
java.util.Collection findByOrder(Order order) throws java.rmi.RemoteException, java.ejb.FinderException;
```

The EJB Container implements the `findByOrder()` method according to instructions in a property in the deployment descriptor. This deployment descriptor property contains the SQL statement to select all LineItem entities where the value of the LineItem `order_id` column is equal to the value of the parameter `o`, the key of the OrderBean entity. This statement might look as follows:

```sql
WHERE LineItem = :o
```

The deployment descriptor also contains a property that specifies the JNDI name for the OrderHome interface so that the EJB Container can locate this interface.

**Client view of object relational mapping**

It is straight-forward for the client code to use the object relational mapping implementation that was just described.

The OrderClient code shows how a client might use the one-to-one and one-to-many implementations. (The interesting portions of the code appear in Code sample 7.13).
Given a particular line item, the client retrieves the order associated with that line item. Then, given a particular order, the client retrieves all the line items on that order. The code sample illustrates the code when using the Collection interface. Note that only the significant portions of code are shown.

**Code sample 7.13  OrderClient code snippet**

```java
import java.util.*;
public class OrderClient {
    public static void main(String[] args) throws Exception {
        ...
        Order myOrder;
        LineItem myItem;
        // get order for a line item
        System.out.println(" Line item: "+myItem.getPrimaryKey()+" belongs to Order "+myItem.getOrder().getPrimaryKey());
        ...
        // get all line items for an order using a Collection
        System.out.println("The order "+myOrder.getPrimaryKey()+" contains these items:");
        Iterator iterator = myOrder.getLineItems().iterator();
        while (iterator.hasNext()) {
            LineItem item = (LineItem) javax.rmi.PortableRemoteObject.narrow
            (iterator.next(), LineItem.class);
            System.out.print(" "+item.getPrimaryKey());
        }
        ...
    }
}
```

### Mapping java types to SQL types

When you develop an enterprise bean for an existing database, you must map the SQL data types specified in the database schema to Java programming language data types.

The Inprise EJB Container follows the JDBC rules for mapping Java programming language types to SQL types. JDBC defines a set of generic SQL type identifiers that represent the most commonly used SQL types. You should use these default JDBC mapping rules when you develop an enterprise bean to model an existing database table. (These types are defined in the class `java.sql.Types`.)

Table 7.1 shows the default SQL to Java type mapping as defined by the JDBC specification.

<table>
<thead>
<tr>
<th>Java type</th>
<th>JDBC SQL type</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>BIT</td>
</tr>
<tr>
<td>byte</td>
<td>TINYINT</td>
</tr>
<tr>
<td>char</td>
<td>CHAR(1)</td>
</tr>
<tr>
<td>double</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>float</td>
<td>REAL</td>
</tr>
<tr>
<td>int</td>
<td>INTEGER</td>
</tr>
</tbody>
</table>
While all relational databases implement SQL types, there may be significant variations in how they implement these types. Even when they support SQL types with the same semantics, they may use different names to identify these types. For example, Oracle implements a Java boolean as a NUMBER(1,0), while Sybase implements it as a BIT and DB2 implements it as a SMALLINT.

When the Inprise EJB Container creates the database tables for your enterprise beans, it automatically maps entity bean fields and database table columns. The container must know how to properly specify the SQL types so that it can correctly create the tables in each supported database. As a result, the EJB Container maps some Java types differently, depending on the database in use. Table 7.2 shows the mapping for Oracle, Sybase/MSSQL, and DB2:

### Table 7.2  Mapping of Java types to SQL types for Oracle, Sybase/MSSQL, and DB2

<table>
<thead>
<tr>
<th>Java type</th>
<th>Oracle</th>
<th>Sybase/MSSQL</th>
<th>DB2</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>NUMBER(1,0)</td>
<td>BIT</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>byte</td>
<td>NUMBER(3,0)</td>
<td>TINYINT</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>char</td>
<td>CHAR(1)</td>
<td>CHAR(1)</td>
<td>CHAR(1)</td>
</tr>
<tr>
<td>double</td>
<td>NUMBER(10,0)</td>
<td>REAL</td>
<td>REAL</td>
</tr>
<tr>
<td>float</td>
<td>NUMBER(10,0)</td>
<td>INT</td>
<td>INTEGER</td>
</tr>
<tr>
<td>long</td>
<td>NUMBER(19,0)</td>
<td>NUMERIC(19,0)</td>
<td>BIGINT</td>
</tr>
<tr>
<td>short</td>
<td>NUMBER(5,0)</td>
<td>SMALLINT</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>java.math.BigDecimal</td>
<td>NUMBER(38)</td>
<td>DECIMAL(28,28)</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>byte[]</td>
<td>RAW(2000)</td>
<td>IMAGE</td>
<td>BLOB</td>
</tr>
<tr>
<td>java.sql.Date</td>
<td>DATE</td>
<td>DATETIME</td>
<td>DATE</td>
</tr>
<tr>
<td>java.sql.Time</td>
<td>DATE</td>
<td>DATETIME</td>
<td>TIME</td>
</tr>
<tr>
<td>java.sql.Timestamp</td>
<td>DATE</td>
<td>DATETIME</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>java.util.Date</td>
<td>DATE</td>
<td>DATETIME</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>java.io.Serializable</td>
<td>RAW(2000)</td>
<td>IMAGE</td>
<td>BLOB</td>
</tr>
</tbody>
</table>
Advanced container-managed persistence issues

Table 7.3 shows the Java to SQL type mapping for JDatastore, Informix, and Interbase:

Table 7.3  Mapping of Java types to SQL types for JDatastore, Informix, and Interbase

<table>
<thead>
<tr>
<th>Java types</th>
<th>JDatastore</th>
<th>Informix</th>
<th>Interbase</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>BOOLEAN</td>
<td>SMALLINT</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>byte</td>
<td>SMALLINT</td>
<td>SMALLINT</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>char</td>
<td>CHAR(1)</td>
<td>CHAR(1)</td>
<td>CHAR(1)</td>
</tr>
<tr>
<td>double</td>
<td>DOUBLE</td>
<td>FLOAT</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>float</td>
<td>FLOAT</td>
<td>SMALLFLOAT</td>
<td>FLOAT</td>
</tr>
<tr>
<td>int</td>
<td>INTEGER</td>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>long</td>
<td>LONG</td>
<td>DECIMAL(19,0)</td>
<td>NUMBER(15,0)</td>
</tr>
<tr>
<td>short</td>
<td>SMALLINT</td>
<td>SMALLINT</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>java.math.BigDecimal</td>
<td>NUMERIC</td>
<td>DECIMAL(32)</td>
<td>NUMBER(15,15)</td>
</tr>
<tr>
<td>byte[]</td>
<td>OBJECT</td>
<td>BYTE</td>
<td>BLOB</td>
</tr>
<tr>
<td>java.sql.Date</td>
<td>DATE</td>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>java.sql.Time</td>
<td>TIME</td>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>java.sql.Timestamp</td>
<td>TIMESTAMP</td>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>java.util.Date</td>
<td>TIMESTAMP</td>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>java.io.Serializable</td>
<td>OBJECT</td>
<td>BYTE</td>
<td>BLOB</td>
</tr>
</tbody>
</table>
This chapter describes how to handle transactions. It covers the following topics:

- Understanding transactions
- Transaction manager services
- Enterprise beans and transactions
- Using the transaction API
- Handling transaction exceptions
- JDBC support
- Distributed transactions

**Understanding transactions**

Application programmers benefit from developing their applications on platforms such as Java 2 Enterprise Edition (J2EE) that support transactions. A transaction-based system simplifies application development because it frees the developer from the complex issues of failure recovery and multi-user programming. Transactions are not limited to single databases or single sites. Distributed transactions can simultaneously update multiple databases across multiple sites.

A programmer typically divides the total work of an application into a series of units. Each unit of work is a separate transaction. As the application progresses, the underlying system ensures that each unit of work—each transaction—fully completes without interference from other processes. If not, it rolls back the transaction and completely undoes whatever work the transaction had performed.
Characteristics of transactions

Typically, transactions refer to operations that access a database. All access to a database is performed in the context of a transaction. All transactions share the following characteristics:

- Atomicity
- Consistency
- Isolation
- Durability

These characteristics are denoted by the acronym ACID.

A transaction often consists of more than a single operation. Atomicity requires that all of the operations of a transaction are performed successfully for the transaction to be considered complete. If all of a transaction’s operations cannot be performed, then none of them may be performed.

Consistency refers to database consistency. A transaction must transition the database from one consistent state to another. The transaction must preserve the database’s semantic and physical integrity.

Isolation requires that each transaction appear to be the only transaction currently manipulating the database. Other transactions may run concurrently. However, a transaction should not see the intermediate data manipulations of other transactions until and unless they successfully complete and commit their work. Because of interdependencies among updates, a transaction might get an inconsistent view of the database were it to see just a subset of another transaction’s updates. Isolation protects a transaction from this sort of data inconsistency.

Isolation is related to transaction concurrency. There are levels of isolation and higher degrees of isolation limit the extent of concurrency. The highest level of isolation occurs when all transactions can be serialized. That is, the database contents look as if each transaction ran by itself to completion before the next transaction started. However, some applications may be able to tolerate a reduced level of isolation for a higher degree of concurrency. Typically, these applications run a greater number of concurrent transactions even if transactions are reading data that may be partially updated and perhaps inconsistent.

Lastly, durability means that updates made by committed transactions persist in the database regardless of failure conditions. Durability guarantees that committed updates remain in the database despite failures that occur after the commit operation and that databases can be recovered after a system or media failure.

Transaction support

The container supports flat transactions, but not nested transactions. The container also propagates transactions implicitly. This means that the user does not have to explicitly pass the transaction context as a parameter, because the container transparently handles this for the client.
Developers should keep in mind that JSPs and servlets, while they can act as clients, are not designed to be transactional components. Transactional work should be done in the appropriate enterprise bean. When you invoke an enterprise bean to perform the transactional work, the enterprise bean and container take care of properly setting up the transaction.

**Transaction manager services**

There are two transaction services, or engines, that ship with the Inprise EJB Container—Inprise’s implementation of Java Transaction Service (JTS) and Integrated Transaction Service (ITS), both products from Inprise. JTS is the default transaction engine that ships with the container. JTS offers many features, but it does have a significant shortcoming—it lacks support for transaction timeouts. A transaction may never complete for some reason, for example, because the transaction hangs. Without a transaction timeout, a bad transaction could hang forever. If a session bean is associated with a bad transaction, then the session bean never times out nor can the container ever remove the session bean—because a container cannot remove a session bean that is involved in a transaction. It is easy to see that this places a limit on the container’s ability to manage resources effectively.

The Inprise ITS transaction service supports transaction timeouts. In fact, it lets you set a system-wide default timeout value and a global maximum timeout values. (See the ITS documentation for more information on setting these values.)

When ITS is used as the transaction service, the container can passivate a session bean even if it is involved in a bad transaction that hangs and never completes. If the transaction times out, which it will eventually do if it hangs, the container can then passivate the session bean. Then, if the session bean remains idle past its own timeout value, the container can remove the session bean. This prevents the situation of a session bean starting a transaction, but then the transaction hangs, resulting in memory and transactional resources being tied up. With ITS, the container can remove a session bean regardless of whether it is involved in a transaction, as long as both the transaction and the session bean have both exceeded their respective timeout values.

**Enterprise beans and transactions**

Enterprise beans and the EJB container handle transaction management. Enterprise beans make it possible for applications to update data in multiple databases within a single transaction. The data can be stored not only in multiple databases but also at multiple sites.

Enterprise beans utilize a declarative style of transaction management that differs from the traditional transaction management style. With declarative management, the enterprise bean declares its transaction attributes at deployment time. The transaction attributes indicate whether the container will manage the bean’s
transactions or whether the bean itself will manage its own transactions, and, if so, to what extent it will do its own transaction management.

Traditionally, the application was responsible for managing all aspects of a transaction. This entailed such operations as:

- Creating the transaction object.
- Explicitly starting the transaction.
- Keeping track of the transaction context.
- Committing the transaction when all updates completed.

It requires a developer with more expertise to write an application that is responsible for managing a transaction from start to finish—the code for such an application is more complex and difficult to write and it is easy for “pilot error” to occur.

With declarative transaction management, the container manages most if not all aspects of the transaction for you. The container handles starting and ending the transaction, plus maintains its context throughout the life of the transaction object. This greatly simplifies an application developer’s responsibilities and tasks, especially for transactions in distributed environments.

**Understanding bean- and container-managed transactions**

When an enterprise bean programmatically performs its own transaction demarcation as part of its business methods, then that bean is considered to be using bean-managed transaction. On the other hand, when the bean defers all transaction demarcation to its EJB container, and the container performs the transaction demarcation based on the Application Assembler’s deployment instructions, then the bean is referred to as using container-managed transaction.

Session beans, both stateful and stateless varieties, can use either container- or bean-managed transaction. However, a bean cannot use both types of transaction management at the same time. Entity beans can only use container-managed transaction. It is the bean provider who decides the type of transaction which a session bean will use.

A bean might want to manage its own transaction if it wishes to start a transaction as part of one operation and then finish the transaction as part of another operation. However, such a design might be problematic if one operation calls the transaction starting method but no operation calls the transaction ending method.

Whenever possible, enterprise beans should use container-managed transactions as opposed to bean-managed transactions. Container-managed transactions require less programming work and are less prone to programming error. In addition, a container-managed transaction bean is easier to customize and compose with other beans.
Transaction attributes

Session beans that use bean-managed transaction have transaction attributes associated with each method of the bean. The attribute value tells the container how it must manage the transactions that involve this bean. There are six different transaction attributes that can be associated with each method of a bean. This association is done at deployment time by the Application Assembler or Deployer.

These attributes are:

- **Required**—This attribute guarantees that the work performed by the associated method is within a global transaction context. If the caller already has a transaction context, then the container uses the same context. If not, the container begins a new transaction automatically. This attribute permits easy composition of multiple beans and co-ordination of the work of all the beans using the same global transaction.

- **RequiresNew**—This attribute is used when the method does not want to be associated with an existing transaction. It ensures that the container begins a new transaction.

- **Supports**—This attribute permits the method to avoid using a global transaction. This should only be used when a bean’s method only accesses one transaction resource—or no transaction resources—and does not invoke another enterprise bean. It is used solely for optimization, because it avoids the cost associated with global transactions. When this attribute is set and there is already a global transaction, the Inprise Container will invoke the method and have it join the existing global transaction. However, if this attribute is set but there is no existing global transaction, the Container will start a local transaction for the method, and that local transaction completes at the end of the method.

- **NotSupported**—This attribute also permits the bean to avoid using a global transaction. When this attribute is set, the method should not be in a global transaction. Instead, the Inprise Container suspends any existing global transaction and starts a local transaction for the method, and the local transaction completes at the conclusion of the method.

- **Mandatory**—It is recommended that this attribute not be used. Its behavior is similar to Requires but the caller must already have an associated transaction. If not, the container throws a `javax.transaction.TransactionRequiredException`. This attribute makes the bean less flexible for composition because it makes assumptions about the caller’s transaction.

- **Never**—It is recommended that this attribute not be used. However, if used, the Inprise Container starts a local transaction for the method. The local transaction completes at the conclusion of the method.

Under normal circumstances only two attributes—Required and RequiresNew—should be used. The attributes Supports and NotSupported are strictly for optimization. The use of Never and Mandatory are not recommended because they affect the compositibility of the bean. In addition, if a bean is concerned about transaction synchronization and implements the `javax.ejb.SessionSynchronization` interface, then the Assembler/Deployer can specify only the attributes Required, RequiresNew, or
Mandatory. These attributes ensure that the container invokes the bean only within a global transaction, because transaction synchronization can only occur within a global transaction.

Local and global transactions

A local transaction is a transaction that is managed by the resource manager. A global transaction, on the other hand, is a transaction managed by the ITS transaction service’s global transaction manager.

A bean that uses bean-managed transaction demarcation uses the javax.transaction.UserTransaction interface to explicitly demarcate global transaction boundaries. When transaction demarcation is container managed, the container intercepts each client call to control the transaction demarcation. It then controls this demarcation declaratively, according to the transaction attribute set by the Application Assembler in the deployment descriptor. The transaction attribute also determines whether the transaction is local or global.

The container follows certain rules to determine when it is to do a local versus a global transaction for container-managed transactions. In general, a container calls the method within a local transaction after verifying that no global transaction already exists. It also verifies that it is not expected to start a new global transaction and that the transaction attributes are set for container-managed transactions. The container automatically wraps a method invocation within a local transaction if one of the following is true:

- If the transaction attribute is set to NotSupported and the container detects that database resources were accessed.
- If the transaction attribute is set to Supports and the container detects that the method was not invoked from within a global transaction.
- If the transaction attribute is set to Never and the container detects that database resources were accessed.

The Inprise Container supports the following characteristics for local transactions:

- Local transactions support the javax.ejb.EJBContext methods setRollbackOnly() and getRollbackOnly().
- Local transactions support time-outs for database connections and transactions.
- Local transactions are lightweight from a performance standpoint.

Using the transaction API

All transactions use the Java Transaction API (JTA). When transactions are container managed, the platform handles the demarcation of transaction boundaries and the container uses the JTA API; you do not need to use this API in your bean code.

A bean that manages its own transactions (bean-managed transaction), however, must use the JTA javax.transaction.UserTransaction interface. This interface allows a
Handling transaction exceptions

client or component to demarcate transaction boundaries. Enterprise beans that use bean-managed transactions use the method `EJBContext.getUserTransaction()`.

In addition, all transactional clients use JNDI to look up the `UserTransaction` interface. This simply involves constructing a JNDI `InitialContext` using the JNDI naming service, as shown in the following line of code:

```java
javax.naming.Context context = new javax.naming.InitialContext();
```

Once the bean has obtained the `InitialContext` object, it can then use the JNDI `lookup()` operation to obtain the `UserTransaction` interface, as shown in.

**Code sample 8.1  Obtaining the UserTransaction interface**

```java
```

Note that an enterprise bean can obtain a reference to the `UserTransaction` interface from the `EJBContext` object. This is because an enterprise bean by default inherits a reference to the `EJBContext` object. Thus, the bean can simply use the `EJBContext.getUserTransaction()` method rather than having to obtain an `InitialContext` object and then using the JNDI `lookup()` method. However, a transactional client that is not an enterprise bean must use the JNDI lookup approach.

When the bean or client has the reference to the `UserTransaction` interface, it can then initiate its own transactions and manage these transactions. That is, you can use the `UserTransaction` interface methods to begin and commit (or rollback) transactions. You use the `begin()` method to start the transaction, then the `commit()` method to commit the changes to the database. Or, you use the `rollback()` method to abort all changes made within the transaction and restore the database to the state it was in prior to the start of the transaction. Between the `begin()` and `commit()` methods, you include code to carry out the transaction’s business.

**Handling transaction exceptions**

Enterprise beans may throw application and/or system level exceptions if they encounter errors while handling transactions. Application-level exceptions pertain to errors in the business logic and are intended to be handled by the calling application. System-level exceptions, such as runtime errors, transcend the application itself and may be handled by the application, the bean, or the bean container.

The enterprise bean declares application-level exceptions and system-level exceptions in the `throws` clauses of its `Home` and `Remote` interfaces. You must check for checked exceptions in your program’s `try/catch` block when calling bean methods.

**System-level exceptions**

An enterprise bean throws a system-level exception, which is a `java.ejb.EJBException` (but may also be a `java.rmi.RemoteException`), to indicate an unexpected system-level failure. For example, it throws this exception if it cannot open a database connection.
Handling transaction exceptions

The `java.ejb.EJBException` is a runtime exception and does not have to be listed in the throws clause of the bean’s business methods.

System-level exceptions usually require the transaction to be rolled back. Often, the container managing the bean does the rollback. Other times, especially with bean-managed transactions, the client must rollback the transaction.

**Application-level exceptions**

The bean throws an application-level exception to indicate application-specific error conditions—that is, business logic errors and not system problems. These application-level exceptions are exceptions other than `java.ejb.EJBException`. Application-level exceptions are checked exceptions, which means you must check for them when you call a method that potentially can throw this exception.

The enterprise bean’s business methods use application exceptions to report abnormal application conditions, such as unacceptable input values or amounts beyond acceptable limits. For example, a bean method that debits an account balance may throw an application exception to report that the account balance is not sufficient to permit a particular debit operation. A client can often recover from these application-level errors without having to rollback the entire transaction.

The application or calling program gets back the same exception that was thrown and this allows the calling program to know the precise nature of the problem. When an application-level exception occurs, the enterprise bean instance does not automatically rollback the client’s transaction. The client now has the knowledge and the opportunity to evaluate the error message, take the necessary steps to correct the situation, and recover the transaction. Or, the client can abort the transaction.

**Handling application exceptions**

Because application-level exceptions report business logic errors, you as the client are expected to handle these exceptions. While these exceptions may require transaction rollback, they do not automatically mark the transaction for rollback. You often have the option to retry the transaction, though there are times when you must abort and rollback the transaction.

The bean Provider is responsible for ensuring that the state of the bean is such that if the client continues with the transaction there is no loss of data integrity. If the Provider cannot ensure this degree of integrity, then the bean marks the transaction for rollback.

**Transaction rollback**

When your client program gets an application exception, you should first check if the current transaction has been marked for “rollback only”. For example, a client may receive a `javax.transaction.TransactionRolledbackException`. This exception indicates that the helper enterprise bean failed and the transaction has been aborted or marked “rollback only”. In general, the client does not know the transaction context within which the called enterprise bean operated. The called bean may have operated in its
Handing transaction exceptions

own transaction context separate from the calling program’s transaction context, or it may have operated in the calling program’s context.

If the enterprise bean operated in the same transaction context as the calling program, then the bean itself (or its container) may have already marked the transaction for rollback. When an EJB container has marked a transaction for rollback, the client should stop all work on the transaction. Normally, a client using declarative transactions will get an appropriate exception, such as `javax.transaction.TransactionRolledbackException`. Note that declarative transactions are those transactions where the container manages the transaction details.

A client that is itself an enterprise bean should call the `javax.ejb.EJBContext.getRollbackOnly` method to determine if its own transaction has been marked for rollback or not.

For bean-managed transactions—those transactions managed explicitly by the client—the client should rollback the transaction by calling the `rollback` method from the `java.transaction.UserTransaction` interface.

Options for continuing a transaction

When a transaction is not marked for rollback, then the client has three options:

- Rollback the transaction.
- Pass the responsibility by throwing a checked exception or re-throwing the original exception.
- Retry and continue the transaction. This might entail retrying portions of the transaction.

When a client receives a checked exception for a transaction not marked for rollback, its safest course is to rollback the transaction. The client does this by either marking the transaction as “rollback only” or, if the client has actually started the transaction, calling the `rollback` method to actually rollback the transaction.

The client can also throw its own checked exception or re-throw the original exception. By throwing an exception, the client lets other programs further up the transaction chain decide whether or not to abort the transaction. However, in general it is preferable for the code or program closest to the occurrence of the problem to make the decision about saving the transaction.

Lastly, the client can continue with the transaction. The client can evaluate the exception message and decide if invoking the method again with different parameters is likely to succeed. However, you need to keep in mind that retrying a transaction is potentially dangerous. You have no knowledge of nor guarantee that the enterprise bean properly cleaned up its state.

Clients that are calling stateless session beans, on the other hand, may retry the transaction with more confidence if they can determine the problem from the thrown exception. Because the called bean is stateless, the client does not have the problem of not knowing the state in which the bean left the transaction.
Transactions involve accessing data from one or more datasources and, usually, updating or changing the data in these datasources. For this to happen, the enterprise bean involved in the transaction must establish a connection to the underlying database or databases. The container uses the JDBC for establishing these connections.

The Inprise EJB Container implements the JDBC 2.0 DataSource interface. This means that an enterprise bean can use the javax.sql.DataSource interface rather than the java.sql.DriverManager interface, though the DriverManager interface is still supported for backward compatibility.

It is recommended that applications use the DataSource interface because of its performance optimizations. Using the DataSource interface implementation does not require the JDBC 2.x driver; applications using the JDBC 1.x driver can use this implementation. In addition, applications will not need to change when migrating from the JDBC 1.x driver to the 2.x driver.

For your bean to use the DataSource interface (or to use it with container-managed persistence), you need to specify the datasource in the deployment descriptor and the bean must access it by using the JNDI look up. The Container deploys the datasources to JNDI and configures them based on the deployment descriptor information. Code sample 8.2 shows how you might use the datasource in a program.

**Code sample 8.2 Using DataSource in a program**

```java
javax.sql.DataSource ds;
try {
    javax.naming.Context ctx = (javax.naming.Context) new
    javax.naming.InitialContext();
    ds = (javax.sql.DataSource)ctx.lookup( "java:comp/jdbc/SavingsDataSource" );
} catch (javax.naming.NamingException exp){
    exp.printStackTrace();
}
```

**Specifying DataSources**

You specify information about datasources in the deployment descriptor. There are three required elements to specify a datasource. These elements are:

- The datasource URL. When using the DataSource interface, the URL might look as follows:
  ```java
db:oracle:thin:@avicenna:1521:avi73a
  ```
- The JNDI name of the datasource.
- The JDBC driver class name.

For a complete description of these elements, please see “Datasource” on page 9-8 in Chapter 9, “Deploying Enterprise JavaBeans.”
Code sample 8.3 shows an example of a datasource specification:

**Code sample 8.3 Deployment descriptor DataSource specification**

```xml
<deployment-descriptor>
  <datasource>
    <res-ref-name>jdbc/SavingsDataSource</res-ref-name>
    <url>jdbc:oracle:thin:@avicenna:1521:avi73a</url>
    <username>scott</username>
    <password>tiger</password>
    <driver-class-name>oracle.jdbc.driver.OracleDriver</driver-class-name>
  </datasource>
</deployment-descriptor>
```

If you are using the `DriverManager` interface, rather than the `DataSource` interface, to obtain a database connection, then you must include the `inprise:its_direct` prefix in the URL, as follows:

```
jdbc:inprise:its_direct:oracle:thin:@avicenna:1521:avi73a
```

Also, if using the `DriverManager` interface, you need to specify the `jdbc.drivers` system property on the command line when you start the container. For example, you might use the following command line for the Oracle driver:

```bash
vbj -Djdbc.drivers=oracle.jdbc.driver.OracleDriver com.inprise.ejb.Container test
  bank_beans.jar -jts -jns
```

When you use the `DataSource` interface, it is not necessary to specify the driver system property if you specify the `driver-class-name` in the deployment descriptor.

Two optional datasource components that you may specify in the deployment descriptor are the user name and the password. They are optional because they may be embedded in the URL. There is a third optional component, called dialect. The dialect component is the name of the database that hosts the application’s persistent data, and the Inprise Container uses this field when it manages persistence. For example, to specify that the dialect component is Sybase, you would include the following line in your deployment descriptor:

```xml
<dialect>sybase</dialect>
```

When dialect is specified, the container automatically creates database tables for you for the specified database.

Keep in mind that the container sets properties specified in the deployment descriptor during startup; the bean cannot change these properties.

### Handling database connections and pooling

An enterprise bean must first establish a database connection to access data from a datasource or database. Establishing new database connections is a time-consuming activity and should be kept to a minimum.

Inprise’s EJB Container provides a database connection pooling mechanism that allows enterprise beans to reuse previously obtained connections. Pooling and reusing connections reduces the number of times that new connections need to be established, and make for more efficient performance.
How does this database connection pooling mechanism work? After a bean object obtains a database connection, the container pools the connection for a configurable amount of time. It keeps the connection in the pool for this configured amount of time and only releases it after the time period elapses. When a bean uses container-managed persistence, the container automatically uses pooled connections.

To use connection pooling, you specify some information for the datasource. You also specify options to control timeouts of idle connections. However, you currently cannot configure other options, such as the size of the pool. There are two system properties for specifying timeouts:

- The **ITSJDBCIdle_timeout** system property lets you specify the timeout value for the pooled JDBC connections. The system default value is ten minutes. You can specify a value for this property at the command line, and the value is in seconds. For example, the following line sets the idle timeout to twenty minutes:

  ```
  Vbj -DITSJDBCIdle_timeout=1200 com.inprise.ejb.Container myContainer beans.jar -jts -jns
  ```

- The **ITS_timeout** system property controls the frequency that the transaction service checks the idle timeout value for pooled connections. Its default value is five minutes, or 300 seconds, and you can set a different value at the command line, as follows:

  ```
  Vbj -DITS_timeout=600 -DITSJDBCIdle_timeout=1200 com.inprise.ejb.Container myContainer beans.jar -jts -jns
  ```

Here is how the container handles connection pooling. In general, the container only establishes connections as required, and not just because certain operations occur. For example, it does not establish a connection just because it starts a transaction, but instead it waits until the first call to the `javax.sql.DataSource` method `getConnection()`. This call from within a transaction assigns the transaction a connection. The container can use a connection from the pool or create a new one if necessary. When there are additional calls to the `getConnection()` method within the same transaction, the container provides the original connection. When the bean invokes the `commit()` method, the container ensures that the commit occurs in the same connection. Once the transaction completes, the container keeps the connection in its internal cache of connections. If another transaction needs a new connection to the same datasource, the container returns the cached connection. The container does this pooling of connections automatically. The code might look as shown in Code sample 8.4:

**Code sample 8.4 Using connection pooling**

```java
Context ctx = (Context) new InitialContext();
ds = (DataSource)ctx.lookup("java:comp/jdbc/SomeDataSource");
ds.getConnection();
// some database work...
ds.getConnection(); // you get the same connection
```

The EJB Container relies on the underlying JDBC driver to handle all RDBMS-specific activities, such as preparing and executing queries. The container passes through most of the calls, except for `java.sql.Connection.setAutoCommit()`. It intercepts this call because it is disallowed when the application is in a global transaction. In addition, the `AUTOCOMMIT` mode for JDBC connections returned by the Inprise DataSource is set to OFF, and the bean cannot set this mode to ON. By setting `AUTOCOMMIT` to OFF, the
Inprise Container commits all changes at the end of the transaction, rather than committing each change as it occurs.

**Transaction isolation level**

When a transaction isolation level is not explicitly set, the container uses the default isolation level of the JDBC driver and the RDBMS. For example, the Oracle RDBMS supports only two isolation levels: \texttt{TRANSACTION\_READ\_COMMITTED} and \texttt{TRANSACTION\_SERIALIZABLE}. The default is \texttt{TRANSACTION\_READ\_COMMITTED} and the container sets this isolation level unless the deployment descriptor specifies a different level.

There are five valid transaction isolation levels that can be specified in the deployment descriptor. These descriptors are case-sensitive, and their support depends on the RDBMS in use. The valid entries are:

- \texttt{TRANSACTION\_NONE}
- \texttt{TRANSACTION\_READ\_COMMITTED}
- \texttt{TRANSACTION\_READ\_UNCOMMITTED}
- \texttt{TRANSACTION\_REPEATABLE\_READ}
- \texttt{TRANSACTION\_SERIALIZABLE}

When the deployment descriptor specifies a particular isolation level for a datasource, all connections using that datasource have the same isolation level. Please check your RDBMS documentation for performance tuning of your data access. If you are not sure about the isolation level, it is recommended that you use the default isolation level.

Refer to Chapter 9, “Deploying Enterprise JavaBeans,” for more information.

**Distributed transactions**

The Inprise EJB Container supports distributed transactions. Distributed transactions are those transactions that cross systems/platforms and cross Java Virtual Machines (JVMs).

**Two-phase commit**

Transactions that manipulate data across multiple databases use a two-phase commit process. This process ensures that the transaction correctly updates all databases involved in the transaction. If it cannot update all databases, then it updates none of the databases.

There are two steps to a two-phase commit. The first step is the preparation phase. The transaction readies its updates to each database and the database must signal that it is ready to complete and commit the updates. The second step initiates the actual database updates only when all databases have signaled that they can complete the update process.
Distributed transactions

The ITS and JTS transaction services support both heterogeneous distributed (two-phase commit) transactions and two-phase commit for homogeneous databases using the RDBMS’s built-in transaction coordinator.

By default, JTS does not allow multiple resources to participate in a global transaction, though it does support a nonrecoverable two-phase commit process. You must set the `EJBAllowUnrecoverableCompletion` flag to true to allow JTS to support a nonrecoverable two-phase commit. When set to true, the container makes a one-phase commit call on each participating database during the transaction commit process. You have to be careful when using this flag because there is no recovery if a problem occurs.

To support heterogeneous two-phase commit transactions, the ITS service must integrate with XA support in the underlying DBMS. (Please see the ITS documentation for details.) Currently, two-phase commit requires using native access—that is, C or C++ code. In the future, when an XA-enabled JDBC driver does become available, ITS and the EJB container will allow multiple databases to participate in a single transaction.

Figure 8.1 shows how the bean and the container control the two-phase commit process with heterogeneous databases. Note that this is a non-recoverable operation when the `EJBAllowOnRecoverableCompletion` flag is set.

**Figure 8.1** Two-phase commit with heterogeneous databases

Two-phase commit for homogeneous databases require some configuration of the RDBMS server. While the container controls the commit to the first database, the RDBMS server controls the commits to the subsequent databases. See Figure 8.2.
You need to consult the manual for the RDBMS server.
Deploying Enterprise JavaBeans

This chapter includes the following major topics:

- Deploying Enterprise JavaBeans: Quick steps
- Creating a deployment descriptor file
- Specifying the EJB’s runtime environment properties
- Creating the EJB jar file
- Deploying your EJB jar file to the container

Deploying Enterprise JavaBeans: Quick steps

Deploying an enterprise bean generally involves the following five steps:

1. Create a deployment descriptor XML-based file compliant with Sun’s EJB 1.1 specification.
2. If necessary, edit the environment properties that an EJB requires at runtime.
3. Create an EJB jar file containing the deployment descriptor and all of the classes required to operate the EJB (bean class, remote interface, home interface, stubs and skeletons, primary key class if the EJB is an entity bean, and any other associated classes). You can do this using Java’s jar utility.
4. Deploy your EJB to an EJB container using either the Inprise Application Server’s Console or command line utilities.

Creating a deployment descriptor file

To create or change a deployment descriptor, use the Inprise Application Server’s Deployment Descriptor editor or any standard XML editor. If you use a standard
Creating a deployment descriptor file

XML editor, you must make sure to comply with the requirements stated in the EJB 1.1 specification and, so that the enterprise bean can work with the EJB container, you must create the Inprise-specific file called ejb-inprise.xml.

The Inprise Application Server’s Deployment Descriptor editor creates a deployment descriptor that conforms to the following requirements stated in the EJB 1.1 specification:

- Is XML based and conforms to the rules of XML.
- Is valid with respect to the DTD in the EJB 1.1 specification.
- Conforms to the semantics rules specified in the DTD. You do not have to learn the rules, the Deployment Descriptor editor imposes these rules on the data you enter.
- Refers to the DTD using the following statement:
  ```xml
  <!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems Inc.//DTD Enterprise JavaBeans 1.2//EN" "http://java.sun.com/j2ee/dtds/ejb-jar_1_2.dtd">
  ```

We strongly recommend that you use the Inprise Application Server’s Deployment Descriptor editor for the following reasons:

- It automatically creates a deployment descriptor that is XML based so you do not have to learn XML.
- It conforms to the semantics rules specified in Sun’s DTD. You do not have to learn the rules, the Deployment Descriptor editor imposes these rules on the data you enter.
- It automatically sets up the Inprise-specific extensions in a separate file.
- As you fill in the information, it lets you know what data are required.

For more information about the Deployment Descriptor editor, see the Inprise Application Server User’s Guide. This section shows snippets from a sample deployment descriptor. To see an example of an entire deployment descriptor, see the examples directory located under your installation directory. It will be in a subdirectory called META-INF:

```
[install_directory]/examples/bank/META-INF/ejb-jar.xml
```

To see an example of the Inprise-specific extensions:

```
[install_directory]/examples/bank/META-INF/ejb-inprise.xml
```

The role of the deployment descriptor

The role of the deployment descriptor is to provide information about each EJB that is to be bundled and deployed in a particular EJB jar file. It is intended for use by the consumer of the EJB jar file. It is the EJB developer’s responsibility to create the deployment descriptor. The deployer may modify some of the attributes at deployment time. You can also modify the deployment descriptor once the EJB is deployed.

The information in the deployment descriptor is used in setting EJB attributes. These attributes define how the EJB operates within a particular environment. For example,
Creating a deployment descriptor file

when you set the EJB’s transactional attributes, they define how the EJB operates when involved with a transaction—or if it participates in a transaction at all. The deployment descriptor keeps the following information:

- Type information, which defines the types—that is, the name of the classes—for the Home and Remote interfaces and the implementation class.
- JNDI names, which set the name under which the Home interface of the enterprise bean is registered.
- Fields to enable container-managed persistence.
- Transactional policies that govern the transactional behavior of a bean.
- Security attributes that govern access to an enterprise bean.
- Inprise-specific information such as datasource information used for database connectivity.

The types of information in the deployment descriptor

The information in the deployment descriptor can be divided into two basic kinds of information:

- Enterprise beans’ structural information. Structural information describes the structure of an enterprise bean and declares an enterprise bean’s external dependencies. This information is required. The structural information cannot, in general, be changed because doing so could break the enterprise bean’s function.
- Application assembly information. Application assembly information describes how the enterprise bean(s) included in the ejb-jar file are composed into a larger application deployment unit. This information is optional. Assembly level information can be changed without breaking the enterprise bean’s function, although doing so may alter the behavior of an assembled application.

Structural information

The EJB developer must provide the following EJB structural information for each EJB in the EJB jar file. Some information is required for all enterprise beans, some for enterprise session beans, some for enterprise entity beans, some for enterprise entity beans that have container-managed persistence.

All enterprise beans:
- Enterprise bean’s name
- Enterprise bean’s class
- Enterprise bean’s home interface
- Enterprise bean’s remote interfaces
- Enterprise bean’s type
- Environment entries
- Resource factory references, if a datasource is used
Creating a deployment descriptor file

- EJB references, if an enterprise bean references another enterprise bean
- Security role references

Enterprise session beans:
- Session EJB’s state management type
- Session EJB’s transaction demarcation type

Enterprise entity beans:
- Entity bean’s persistence management type
- Entity bean’s primary key class

Enterprise entity beans with container-managed persistence:
- Container-managed fields

**Application assembly information**
You can specify any of the following application assembly information. During application assembly it is optional. It is not optional for the Deployer.

- Binding of enterprise bean references
- Security roles
- Method permissions
- Linking of security role references
- Transaction attributes

During the process of application assembly or deployment, you can modify the following structural information:

- Values of environment entries. The Application Assembler may change existing and/or define new values of environment properties.
- Description fields. The Application Assembler may change existing or create new description elements.

You cannot modify any other types of structural information; however, you may modify any application assembly information at deployment time.

For complete information and details about the deployment descriptor’s content and semantics rules, see the Sun EJB 1.1 specification.

For information about what to specify using the Deployment Descriptor editor, see the Inprise Application Server's User’s Guide.

**Security**
A developer—usually the application assembler—specifies the following information in the deployment descriptor:

- Security roles
- Method permissions
- Links between security role references and security roles
**Security roles**
Using the security role elements in the deployment descriptor, the developer may define one or more security roles. These define the recommended security roles for the clients of the enterprise beans.

**Method permissions**
Using the method-permission elements in the deployment descriptor, the developer may define method permissions. This is a paired relation between the security roles and the methods of the EJBs' remote and home interfaces.

**Linking of security role references**
If security roles are defined, the developer must link them with security role references using the role-link element in the deployment descriptor.

**Inprise-specific information needed to deploy enterprise beans**
Inprise-specific elements are stored in the file called ejb-inprise.xml. This file resides in the same location as the ejb-jar.xml file. You will be able to specify the following information in the ejb-inprise.xml file:
- bean-home-name
- a stateful session bean’s timeout
- datasource information
- container-managed persistence information
- mapping of the JNDI names used in the enterprise beans to the actual names in the namespace

The two main sections are:
- `<enterprise-beans>`
- `<datasource-definitions>`

The DTD of the Inprise-specific file is:
```
<!ELEMENT inprise-specific (enterprise-beans, datasource-definitions?)>
<!ELEMENT enterprise-beans (session|entity)+>
<!ELEMENT session (ejb-name, bean-home-name, timeout?, ejb-ref*, resource-ref*, property*)>
<!ELEMENT entity (ejb-name, bean-home-name, ejb-ref*, resource-ref*, cmp-info?, property*)>
<!ELEMENT ejb-ref (ejb-ref-name, jndi-name)>
<!ELEMENT resource-ref (res-ref-name, jndi-name, cmp-resource?)>
<!ELEMENT datasource-definitions (datasource*)>
<!ELEMENT datasource (jndi-name, url, username?, password?, isolation-level?, driver-class-name?, jdbc-property*, property*)>
<!ELEMENT jdbc-property (prop-name, prop-value)>
Creating a deployment descriptor file

```xml
<!ELEMENT cmp-info (description?, database-map?, finder*)>
<!ELEMENT database-map (table, column-map*)>
<!ELEMENT finder (method-signature, where-clause, load-state?)>
<!ELEMENT column-map (field-name, column-name, ejb-ref-name?)>
<!ELEMENT cmp-resource (#PCDATA)>
<!ELEMENT method-signature (#PCDATA)>
<!ELEMENT where-clause (#PCDATA)>
<!ELEMENT load (#PCDATA)>
<!ELEMENT prop-name (#PCDATA)>
<!ELEMENT prop-type (#PCDATA)>
<!ELEMENT prop-value (#PCDATA)>
<!ELEMENT field-name (#PCDATA)>
<!ELEMENT column-name (#PCDATA)>
<!ELEMENT table (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT ejb-name (#PCDATA)>
<!ELEMENT bean-home-name (#PCDATA)>
<!ELEMENT timeout (#PCDATA)>
<!ELEMENT ejb-ref-name (#PCDATA)>
<!ELEMENT jndi-name (#PCDATA)>
<!ELEMENT res-ref-name (#PCDATA)>
<!ELEMENT url (#PCDATA)>
<!ELEMENT username (#PCDATA)>
<!ELEMENT password (#PCDATA)>
<!ELEMENT isolation-level (#PCDATA)>
<!ELEMENT driver-class-name (#PCDATA)>
```

Some of these elements are optional and some are required.

You must have a corresponding session or entity section for each EJB you are planning to deploy in an EJB jar file. For example, if you have five session beans, you must have five session elements in the Inprise-specific file.

**Bean information**

In the enterprise-beans section you provide additional information about the EJB. The Inprise-specific elements are described in Table 9.1.

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ejb-name</td>
<td>The ejb-name is the element that references the bean and is used to correlate all of the EJB’s data. It provides a link between the bean information in the ejb-jar.xml file and the information in the ejb-inprise.xml file. This name must be unique per XML file. You can deploy the same bean but under different names. This is mandatory information.</td>
</tr>
<tr>
<td>bean-home-name</td>
<td>The bean-home-name is the name used by the client to look up the EJB via JNDI. This is mandatory information.</td>
</tr>
<tr>
<td>timeout</td>
<td>Timeout is the stateful session EJB timeout. It is optional and you only use it for a stateful session EJB.</td>
</tr>
<tr>
<td>ejb-ref</td>
<td>This links the JNDI names used by the EJB to actual names in the naming system. It is optional information.</td>
</tr>
</tbody>
</table>
Creating a deployment descriptor file

The entity beans section may have an additional element that you can specify. If the bean has container-managed persistence, you can specify cmp-info.

### Table 9.1 Inprise-specific bean information in deployment descriptor (continued)

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>resource-ref</td>
<td>This links JNDI names used by datasources to actual names in the naming system. It is optional.</td>
</tr>
<tr>
<td>property</td>
<td>Specify a name=value pair. These are properties the container uses for the enterprise bean.</td>
</tr>
</tbody>
</table>

### Table 9.2 Elements in cmp-info

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>description</td>
<td>The description describes the container-managed persistence. This is optional information.</td>
</tr>
<tr>
<td>database-map</td>
<td>This element maps the cmp-bean to a particular table in the database. It maps the bean’s fields to actual columns in the table. Default maps are: bean name = table name; field name = column name. This is optional information.</td>
</tr>
<tr>
<td>finders</td>
<td>This element defines the operation done by the finder methods. It specifies the SQL WHERE clause used by the cmp-bean to execute the finder methods and retrieve records from the database. The load state is true by default and the Container pre-fetches all the container-managed fields when a find is done. It is optional.</td>
</tr>
<tr>
<td>tuned-writes</td>
<td>This element ensures that minimal database updates are performed. In particular, if an entity bean was not modified in a given transaction, the bean is not written to the database. It is optional information.</td>
</tr>
</tbody>
</table>

### Example of session bean element

```xml
<inprise-specific>
    <enterprise-beans>
        <session>
            <ejb-name>sort</ejb-name>
            <bean-home-name>ejb/sort</bean-home-name>
            <ejb-ref>
                <ejb-ref-name>ejb/sort</ejb-ref-name>
                <jndi-name>cosnm/mySortHome</jndi-name>
            </ejb-ref>
        </session>
    </enterprise-beans>
</inprise-specific>
```

### Example of entity bean with container-managed persistence

```xml
<inprise-specific>
    <enterprise-beans>
        <entity>
            <ejb-name>checking</ejb-name>
        </entity>
    </enterprise-beans>
</inprise-specific>
```
Creating a deployment descriptor file

```xml
<bean-home-name>accounts/savings</bean-home-name>
<cmp-info>
  <database-map>
    <table>Checking_Accounts</table>
  </database-map>
  <finders>
    <method-signature>findAccountsLargerThan(float balance)</method-signature>
    <where-clause>balance > :balance</where-clause>
  </finders>
</cmp-info>
<resource-ref>
  <res-ref-name>jdbc/CheckingDatasource</res-ref-name>
  <jndi-name>oracleDataSource/CheckingDS</jndi-name>
  <cmp-resource>True</cmp-resource>
</resource-ref>
</entity>
</enterprise-beans>
<datasource-definitions>
  <datasource>
    <jndi-name>oracleDataSource/CheckingDS</jndi-name>
    <url>jdbc:oracle:thin:@avicenna:1521:avi73a</url>
    <username>scott</username>
    <password>tiger</password>
  </datasource>
</datasource-definitions>
</inprise-specific>
```

**Datasource**

If you are using a database, you must specify elements in the datasource section. The datasource section is a combination of elements that gives you an entry point to the database.

The datasource section tells the Container how to instantiate the datasource.

It specifies the correlation between a datasource, a particular EJB, and the elements in the resource-ref node specified in Sun's EJB 1.1. specification. The Inprise-specific extensions in the datasource section tell you how to correlate the res-ref-name to an actual datasource to perform a connection. One of the elements in the resource-ref node is res-ref-name. This is a lookup into the JNDI for a particular datasource. The Inprise-specific extensions help the Container to bind a JNDI name with a particular datasource.
Creating a deployment descriptor file

Table 9.3 describes the elements present in `<datasource>`.

Table 9.3  Inprise-specific datasource information in deployment descriptor

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jndi-name</td>
<td>This is the actual JNDI name under which the datasource is registered in the naming system. This is mandatory information.</td>
</tr>
</tbody>
</table>
| url          | The URL—uniform resource locator—to access the datasource over a network. It is comprised of:  
  `<JDBC driver><server><port><host>`  
  This is mandatory information if you are using a database. |
| username     | User name for access to the database. This is optional. This can also be specified as part of the URL. This must be specified if the `<res-auth>` element is set to “Container.” |
| password     | Password for access to the database. This is optional. This can also be specified as part of the URL. This must be specified if the `<res-auth>` element is set to “Container.” |
| isolation level | See the section which follows this table. This is optional. If you do not specify it, it defaults to whatever is the default for the JDBC driver. |
| driver-class-name | This is the name of the driver class to be loaded. This is the same name as the one specified in the “jdbc.drivers” system property. This is optional. |
| jdbc-property | This is a list of arbitrary string name/value pairs passed to DriverManager.getConnection. There is no type associated with it. This is optional. |
| property     | This is a name=value pair. These are the properties used by the container for the datasource. This is optional. |

**Note**  For EJBs with container-managed persistence (CMP), a connection is obtained by specifying a user name and password. Therefore, you must specify username and password either in these elements or in the URL element. For enterprise beans with `<res-auth>` set to “Bean”, information about username and password is not required.

**Transaction isolation levels**

The transaction isolation levels are defined based on the permitted violations. Also, isolation levels are dependent on what is allowed by the JDBC driver you are using. Make sure you are familiar with the isolation levels that your JDBC driver supports.

**Permitted violations**—Transaction isolation levels are originally defined in the ANSI SQL specification. The concept has been adopted by the ODBC and JDBC standards and found its way in the EJB specification. Isolation level refers to the degree to which multiple interleaved transactions are prevented from interfering with each other in a multi-user database system. Ideally, one would like to have serializable transactions. That means that the interleaved execution of any set of concurrent transactions produces the same effect as a serial execution of the same transactions. There are three specific ways in which the serializability of a transaction may be violated.

- **Dirty Read.** Transaction t1 modifies a row. Transaction t2 then reads the row. Now t1 performs a rollback and t2 has seen a row that never really existed.
Specifying the EJB’s runtime environment properties

- **Non-repeatable Read.** Transaction t1 retrieves a row. Then transaction t2 updates this row and t1 retrieves the same row again. Transaction t1 has now retrieved the same row twice and has seen two different values for it.

- **Phantoms.** Transaction t1 reads a set of rows that satisfy certain search conditions. Then transaction t2 inserts one or more rows that satisfy the same search condition. If transaction t1 repeats the read, it will see rows that did not exist previously. These rows are called phantoms.

Transaction isolation levels are as follows:

- **TRANSACTION_READ_COMMITTED.** Does not allow dirty reads but allows the other two violations.
- **TRANSACTION_READ_UNCOMMITTED.** Allows all three violations.
- **TRANSACTION_REPEATABLE_READ.** Allows phantoms but not the other two violations.
- **TRANSACTION_SERIALIZABLE.** Does not allow any of the three violations.
- **TRANSACTION_NONE.** Does not support transactions.

Specifying the EJB’s runtime environment properties

Environment properties can be edited right before you deploy an EJB jar file to a container. For example, if a particular EJB must be associated with a particular user but you want to avoid hard-coding this information, you can edit the environment properties at deployment time.

Creating the EJB jar file

To deploy an EJB, you must build an EJB jar file containing all of the classes which enable the EJB to operate. You can include more than one EJB in the EJB jar file.

The contents of the EJB jar file

Using the Java jar utility, create an EJB jar file that contains all the files needed so that the EJB will operate. This includes the following files:

- The XML-based deployment descriptor files—the file required by the EJB specification 1.1 and the Inprise-specific file. These must be in a particular directory and in a particular format. Name the files ejb-jar.xml and ejb-inprise.xml put them in a subdirectory called META-INF.

- Class files which must be compiled and in the form of byte code. These files are as follows:
  - Bean implementation
  - Remote
  - Home
Deploying your EJB jar file to the container

To deploy an EJB, you must deploy its jar file to the EJB container. You can do this in two ways:

• using the Deployment wizard in the Inprise Application Server console
• using a command line utility

You can deploy multiple EJB jar files to an EJB container. And, as a result, you can start the container with multiple jar files. Currently, an EJB built with Inprise products may only be deployed to an Inprise EJB container.

Syntax
The syntax for the command utility is as follows:

```
prompt% vbj com.inprise.ejb.Container <server_name> <jar_file> <jar_file> <jar_file> <options>
```

Example
An example of using the command line utility to deploy an EJB jar file is as follows:

```
prompt% vbj com.inprise.ejb.Container test beans.jar -jts -jns -jdb
```
There are various tools used during the development and deployment of EJBs:

- `java2iiop` compiler
- `verify`
- `dd2xml`

The following sections describe their use.

**java2iiop**

Use this tool during assembly to generate IIOP-compliant stubs and skeletons from the home interface. You must generate stubs and skeletons so that your client can communicate with the server.

The stubs and skeletons that are generated are compliant with the CORBA 2.3 specification. This means that RMI-over-IIOP is implemented in terms of objects-by-value. Complex Java data types (such as dictionaries, vectors, etc.) are written into IIOP using the new IDL “value” types, as specified by CORBA 2.3. This ensures true interoperability.

**When do you use it?**

Here’s a summary of steps that tells you when to use the `java2iiop` compiler:

1. Write all of the classes associated with the EJB.
2. Build all the Java classes that you have developed using the Java compiler (javac).
3. Run the `java2iiop` compiler against the home class. It generates the IIOP-compliant stubs and skeletons. Like other Java files, these stubs and skeletons must be in byte
code format before you package them in the jar file. To automatically generate them in byte code format, use the java2iiop \-compile option.

4 Package all of the EJB classes—including the stubs and skeletons—into a jar file. You can enter one or more home classes. If you enter more than one file name, make sure you include spaces in between the file names. For more information about using this command, see “Defining CORBA interfaces with Java” in the VisiBroker for Java Programmer’s Guide.

**Note** The java2iiop compiler does not support overloaded methods on Caffeine interfaces.

**Syntax**

```
java2iiop [options] {filename filename filename}
```

**Example**

```
java2iiop -verbose -no_examples -compile ContainerHome ContainedHome
```

**Options**

The following table lists and describes some of the options available when using the java2iiop compiler. To view the usage syntax and a list of all options for java2iiop, enter the command followed by a dash and the word “help.” For example,

```
prompt>java2iiop -help
```

The usage syntax and a list of options appear.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-verbose</td>
<td>Turns on verbose mode.</td>
</tr>
<tr>
<td>-no_comments</td>
<td>Suppresses comments in generated code.</td>
</tr>
<tr>
<td>-no_examples</td>
<td>Suppresses the generation of example code.</td>
</tr>
<tr>
<td>-compile</td>
<td>Builds and generates all the class files for the stubs and skeletons.</td>
</tr>
</tbody>
</table>

**Note** In addition, you can use any of the command line options for the vbj command as command line options for the java2iiop command. For a complete list of the vbj options, see the VisiBroker for Java Reference Guide.

**Verify**

Use this tool to verify that the contents of an EJB jar file follow rules specified in the EJB 1.1 specification such as inheritance of the EJB classes and that the jar uses XML-based deployment descriptors—not obsolete serialized ones per the EJB 1.0 specification. See the EJB 1.1 specification for all the rules.

You must run verify on the EJB jar file before you deploy it to the container. You might want to use verify during development to make sure there are no errors.

Input to the verify utility can be multiple jar files.
When do you use it?

Here’s a summary of steps that tells you when to use the verify utility:

1. Develop EJB classes.
2. Run java2iiop.
3. Create the deployment descriptor in XML.
5. Run verify on the EJB jar file(s).

Syntax

```
vbj com.inprise.ejb.util.Verify [-verbose] <ejb-jar-file>.jar ...
```

Example

```
vbj com.inprise.ejb.util.Verify bank753.jar storexyz.jar
```

Options

The following table lists and describes the options available when using Verify.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[-verbose] &lt;ejb-jar-file&gt;.jar ...</td>
<td>Turns on verbose messages when verifying the ejb jar file.</td>
</tr>
</tbody>
</table>

**dd2xml**

Use this tool to convert existing serialized deployment descriptors (developed according to the EJB 1.0 specification) into XML-based deployment descriptors compliant with the EJB 1.1 specification. This utility helps you migrate from serialized deployment descriptors to XML-based ones.

Enter one or more .ser file(s). When you run the dd2xml tool, it generates an .xml file that is compliant in content and format with the EJB 1.1 specification. The dd2xml utility incorporates all of the deployment descriptors specified on the command line into a single output which can be saved to a file.

You can combine more than one serialized deployment descriptor. The output is one XML file. So, if you have a jar that has, for example, three beans in it, there would be three serialized deployment descriptors in the jar. Now, you have only one XML-based deployment descriptor that describes all three beans. The beans can have different attributes. Plus, you can put session beans in with entity beans. There are no restrictions about mixing types of enterprise beans in a jar file.
Verify

**When do you use it?**

Here’s a summary of steps that tells you when to use the dd2xml utility if you are migrating from EJB 1.0 compliant jars to EJB 1.1 compliant jars:

1. Extract the .ser files from the EJB jar file.
2. Run dd2xml specifying all the .ser files on the command line. Two XML files (ejb-jar.xml and ejb-inprise.xml) get generated.
3. Move the XML files that were generated to the directory META-INF/.
4. Rebuild the jar file using the Java jar utility.
5. Run verify.

**Syntax**

```
  vbj com.inprise.ejb.util.dd2xml <deployment_descriptor_file>.ser ...
```

**Example**

```
vbj com.inprise.ejb.util.dd2xml bank753.ser
```

**Note**

There are no options for this utility.
Analyzing the cart example’s output

The following section provides a detailed explanation of the cart example’s output. This output is the result of turning on EJBDebug mode for both the container and the client. For information about how to turn on debug mode, see “Using debug mode” on page 7-23.

This chapter contains the following major topics:

- Summary information
- Deployment descriptor
- List of the EJB’s methods
- The container’s statistics
- The client and container interacting
- Client output
- Purchasing an item

Summary information

When you start the container, a summary of information about Kodiak appears:

**Code sample 10.1 Container output 1**

Inprise EJB Container

=================================
server version : 0.2.0
server build date : December 21, 1998
java version : JDK1.1.6_Borland
java vendor : Sun Microsystems Inc.
java class path : C:\kodiak\ejb_ea_0_2\lib\ejb.jar
: C:\Inprise\APPLIC-1\bin\vbj.exe\..\..\lib\vjorb.jar
Deployment descriptor

: C:\Inprise\APPLICATION\bin\vbj.exe\..\..\lib\vbjapp.jar
: C:\Inprise\APPLICATION\bin\vbj.exe\..\..\lib\vbj30ssl.jar
: .
: c:\Inprise\ApplicationServer\java\bin\..\classes
: c:\Inprise\ApplicationServer\java\bin\..\lib\classes.zip
: c:\Inprise\ApplicationServer\java\bin\..\lib\classes.jar
: c:\Inprise\ApplicationServer\java\bin\..\lib\rt.jar
: c:\Inprise\ApplicationServer\java\bin\..\lib\i18n.jar

=====================  
Creating POA: EJB[test]cart

You can use this information to troubleshoot problems.  
The container output shows the name of the EJB, called *cart*, that will be instantiated 
when create() is called.

Deployment descriptor

Next is information from the deployment descriptor file:

**Code sample 10.2  Container output 2**

Deployment Descriptor

---Generic---
getEnterpriseBeanClassName: CartBean
getHomeInterfaceClassName: CartHome
getRemoteInterfaceClassName: Cart
getBeanHomeName: cart
getEnvironmentProperties: {}
getReentrant: false

---Session---
getSessionTimeout: 0
getStateManagementType: STATEFUL_SESSION

---Access---
getAccess.getMethod: *default*
getAccess.getAllowed: null

---Control---
getControl.getMethod: *default*
getControl.getTxAttribute: TX_NOT_SUPPORTED
getControl.getRunAsMode: SPECIFIED_IDENTITY
getControl.getRunAsId: null
getControl.getMethod: purchase
getControl.getTxAttribute: TX_REQUIRED
getControl.getRunAsMode: SPECIFIED_IDENTITY
getControl.getRunAsId: null

A deployment descriptor includes the declarative attributes associated with an EJB.  
The attributes instruct the container how to manage the EJBs. The output shows the 
following:

- The name of the implementation class is set to CartBean.
• The name of the EJBHome interface class name is set to CartHome.

• The remote interface class name is set to Cart.

• The JNDI name is set to cart. JNDI does not actually use strings as names. This example uses a composite name “cart.”

There are no environment properties so this is an empty list.

The EJB is not reentrant. Because getReentrant() is set to false in this example, the container ensures that only one thread can be executing a particular EJB at any time. If a client request arrives for an EJB while the EJB is executing another request, the container throws the java.rmi.RemoteException to the second request. Note that a session EJB is intended to support only a single client. One implication of this rule is that it is not possible for an application to make loopback calls to a session EJB.

Session-specific information includes timeout information. In this example, the session timeout is set to 0 which means that there is no timeout set, therefore the objects are not removed by the container. If it were set to 10 or 100, the timeout would expire after 10 or 100 seconds and the container would remove the objects. And, concluding session information, its state management type is “stateful session bean,” which means that it stores its state and the EJB instance can be passivated and then reactivated. So, the shopping cart is a stateful session EJB.

The access information shows that it is set to the default: no access control is being done.

The control descriptor is set so that the default is used: for all methods not listed here, transactions are not supported. Further down you can see that the purchase() method requires transactional support. So, you can set control one way for most methods and then specify differently for individual methods. More control information is shown about user identity.

List of the EJB’s methods

All of this is followed by a printout of all the EJB methods on the object. Five methods and an ejbCreate() and ejbRemove().

Code sample 10.3  Container output 3

```java
--Methods (bean)--
  0: addItem
  1: removeItem
  2: getTotalPrice
  3: getContents
  4: purchase

--Methods (create)--
  0: ejbCreate

--Methods (miscellaneous)--
  0: ejbRemove

*sc* user sending request: CartHome._is_a
*sc* user received request: CORBA::Object._is_a
```
The container is ready to interact with the client and prints out a message declaring this.

**Code sample 10.4** Container output 4

Container [test] is ready

Often you will also see output about the container’s statistics:

**Container output 5**

```
EJB Container Statistics
========================
Time                    Fri Mar 19 10:27:11 PST 1999
Memory (used)           514 Kb (max 514 Kb)
Memory (total)          1023 Kb (max 1023 Kb)
Memory (free)           49.0%
========================
Home                    cart
Total created           0
Total active            0
========================
```

The frequency of displaying the output about statistics varies. It happens approximately every five seconds. This is controllable—you can specify how often you want this to occur. In this example, the container and EJB are using about 500 K. That may seem like a lot, but there is a significant amount of infrastructure.

When evaluating the container’s statistics, the ideal situation is where a client comes in, does its job, goes away, and the memory usage stays constant. The container should not grow in its memory usage over time.

In terms of the cart, it has not been created yet and the output shows this. The information shown in this section of the output is related to the EJB’s state and tells you what the EJB is actually doing. If it were active, it could be in other states. It could be in the passive state if it had been written to secondary storage. Or it could be in the transactional ready state. The output is giving you a snapshot of what’s actually going on.

Once the container tells you its ready, you can start the client.

**The client and container interacting**

Once the client and container interact, the container’s output shows more information.

**Code sample 10.5** Container output 6

```
*sc* user received request: CORBA:Void.root_context
*sc* user received request: CORBA:Void.resolve
```
Analyzing the cart example's output

The client and container interacting

*sc* user received request: CORBA::Object._is_a
*sc* user received request: CORBA::Object._is_a
*sc* user received request: CORBA::Object._is_a
*sc* user received request: CORBA::Object._is_a
*sc* user received request: CORBA::Object._get_EJBMetaData
*sc* user received request: CORBA::Object._is_a

*sc* user received request: CORBA::Object.create

*st* prepare context NOT_EXIST --{setContext:setSessionContext}--> SET_CONTEXT
Invoking method
CartBean.setSessionContext(java.ejb.SessionContext=SessionContext{id=1,status=NoTransaction ,caller=null})
caller identity null
run as identity null
completed CartBean.setSessionContext()

*st* commit context NOT_EXIST --{setContext:setSessionContext}--> SET_CONTEXT
*st* prepare context SET_CONTEXT --{create:ejbCreate}--> READY
Invoking method
void CartBean.ejbCreate(java.lang.String=Jack B. Quick,
java.lang.String=1234-5678-9012-3456, java.util.Date
=Sun Jul 01 00:00:00 PDT 2001)
caller identity null
run as identity null
transaction attribute TX_NOT_SUPPORTED
transaction status StatusNoTransaction
completed CartBean.ejbCreate()

*st* commit context SET_CONTEXT --{create:ejbCreate}--> READY
*sc* user received request: CORBA::Object.addItem

*st* prepare context READY --{method:addItem}--> READY
Invoking method
void CartBean.addItem(Item=Book@314860)
caller identity null
run as identity null
transaction attribute TX_NOT_SUPPORTED
transaction status StatusNoTransaction
completed CartBean.addItem()

*st* commit context READY --{method:addItem}--> READY
*sc* user received request: CORBA::Object.addItem

*st* prepare context READY --{method:addItem}--> READY
Invoking method
void CartBean.addItem(Item=CompactDisc@317725)
caller identity null
run as identity null
transaction attribute TX_NOT_SUPPORTED
transaction status StatusNoTransaction
completed CartBean.addItem()

*st* commit context READY --{method:addItem}--> READY
*sc* user received request: CORBA::Object._get_contents

*st* prepare context READY --{method:getContents}--> READY
Invoking method
java.util.Enumeration CartBean.getContents()
caller identity null
run as identity null
transaction attribute TX_NOT_SUPPORTED
transaction status StatusNoTransaction
completed CartBean.getContents()

getContents(): CartBean[name=Jack B. Quick]
result java.util.Enumeration=com.inprise.ejb.util.VectorEnumeration@317933
Debug mode turns on interceptors and, if you look at the container’s output, it shows what RPC messages are coming in. Debug mode also turns on diagnostics on the state machine. The output shows when transitions occur on objects within the state machine and it tells you what methods are being called on objects.

Client output

Now take a look at the client’s output.

Code sample 10.6  Client output 1

A few calls are being made on CosNaming. One is to look up CosNaming and then another to look up the name “cart.” The lookup actually turns into a call to resolve on the transaction service. Then there’s a call to create() which moves the EJB from DOES NOT EXIST to METHOD READY.

Meanwhile, the container’s output is showing that the program calls ejbCreate() on the EJB with a couple of arguments “Jack B. Quick”, a string for the credit card number, and a date. The output shows the transition to the READY state.

Take a look at the client’s output.

Code sample 10.7  Client output 2

A few calls are being made on CosNaming. One is to look up CosNaming and then another to look up the name “cart.” The lookup actually turns into a call to resolve on the transaction service. Then there’s a call to create() which moves the EJB from DOES NOT EXIST to METHOD READY.

Meanwhile, the container’s output is showing that the program calls ejbCreate() on the EJB with a couple of arguments “Jack B. Quick”, a string for the credit card number, and a date. The output shows the transition to the READY state.

Take a look at the client’s output.
Purchasing an item

Code sample 10.8  Client output 3

*sc* user  sending request: Cart.removeItem
*sc* user  sending request: Cart.addItem
====== Cart Summary ======
*sc* user  sending request: Cart._get_contents
Price: $11.97  CompactDisc title: Kind of Blue
*sc* user  sending request: Cart._get_totalPrice
Total: $61.92

Note

The create() method does not print any output on the container's side—which is how the create() was written. There are printouts for the addItem() call. The container's output shows the addition of the book—The Art of Computer Programming—and shows a printed statement whenever addItem() is called. These statements are from the EJB—the container did not provide a call that printed the statements. The EJB has printout statements in it to show what is going on. When writing your own EJB, if you want to see these types of messages in the container's output, you have to include code for the printout statements.

Code sample 10.9  Container output 7

*sc* user  received request: CORBA::Object.removeItem
*st* prepare context READY --{method:removeItem}--> READY
Invoking method void CartBean.removeItem(Item=Book@317ed1)
caller identity null
run as identity null
transaction attribute TX_NOT_SUPPORTED
transaction status StatusNoTransaction
removeItem(The Art of Computer Programming): CartBean[name=Jack B. Quick]
Purchasing an item

completed CartBean.removeItem()
*st* commit context READY --[method:removeItem]--> READY
*sc* user received request: CORBA::Object.addItem
*st* prepare context READY --[method:addItem]--> READY
Invoking method void CartBean.addItem(Item=Book@318040)
caller identity null
run as identity null
transaction attribute TX_NOT_SUPPORTED
transaction status StatusNoTransaction
addItem(Programming with VisiBroker): CartBean[name=Jack B. Quick]
completed CartBean.addItem()
The container output is showing that the purchase() call came in from the client. Remember that the deployment descriptor showed that the purchase operation actually required a transaction. But the client did not process a transaction. The container actually begins a transaction and registers the synchronization after it calls afterBegin(). Then it calls purchase() on the CartBean. According to Sun’s EJB specification, if an EJB throws an exception within a transactional method, the transaction is rolled back. The transaction is rolled back and afterCompletion() is called. The purchase method is done.

Looking now at the client, the output shows that the cart is removed.

**Code sample 10.10 Client output 4**

```java
*sc* user sending request: Cart.remove
```
Generate deployment descriptor

The GenerateDescriptors class generates a serialized deployment descriptor. Deployment descriptors are Java objects that have properties describing how to deploy a particular Java program. For example, a property may describe how to run the program or where to put its name in the naming service. For EJB 1.0, a deployment descriptor is stored as a serialized object in a file. The build process includes incorporating the file containing the deployment descriptor into the jar file.

This class creates a serialized output file, sets the deployment descriptor properties, and then writes the deployment descriptor to the file. Let’s look first at the class’s `main()` routine in Code sample 10.1. The `main()` creates a file called cart.ser, which it then wraps within an object output stream. The `ObjectOutputStream` class enables a Java object to be streamed. Then, it calls `createDescriptor()`, which constructs the deployment descriptor and writes it to the cart.ser file. See Code sample 10.2.

**Code sample 10.1** main() routine for generatedescriptors

```java
public static void main(String[] args) throws Exception {
    ObjectOutputStream objectOutput =
        new ObjectOutputStream(new FileOutputStream("cart.ser"));
    objectOutput.writeObject(createDescriptor());
}
```

To create a deployment descriptor, the `createDescriptor()` routine initializes a session descriptor (because CartBean is a session bean; for an entity bean, it would initialize an entity descriptor) and sets the following properties:

- The enterprise bean implementation class name to CartBean.
- The home interface class name to CartHome.
- The remote interface class name to Cart.
The JNDI name to cart. JNDI does not actually use strings as names. It uses Java naming names. The simplest way to convert a string to a Java naming name is to create a `javax.naming.CompositeName`, as is done in this example.

The CartBean’s state type to STATEFUL_SESSION by calling the function `setStateManagementType()`. (A stateless session bean’s state type would be STATELESS_SESSION.) This sets the transactional mode for the session bean.

Transaction attributes. The CartBean is given a transaction attribute of TX_NOT_SUPPORTED. The `purchase()` method is given the transaction attribute TX_REQUIRES.

A session timeout value of zero. The session timeout is a configurable value that pertains to the bean’s lifecycle. There are two factors which control the lifecycle of a session bean:

- The user’s need for the session bean. Normally, a user creates a session bean, uses the bean, and then removes it.
- The session bean times out, if a timeout is set. If the timeout value has been set, the container removes the session bean if the session bean has not been used within a certain period of time. For example, this GenerateDescriptors file sets the session timeout to zero. A value of zero means that no timeout is set and the session bean objects are not removed by the container even if left unused by the client. However, if the session timeout is set to 100, then the bean’s timeout expires after 100 seconds of non-use and the container removes the objects.

**Code sample 10.2 Setting the descriptor properties**

```java
public class GenerateDescriptors { 
  static DeploymentDescriptor createDescriptor() throws Exception { 
    SessionDescriptor d = new SessionDescriptor(); 
    d.setEnterpriseBeanClassName("CartBean"); 
    d.setHomeInterfaceClassName("CartHome"); 
    d.setRemoteInterfaceClassName("Cart"); 
    d.setBeanHomeName(new javax.naming.CompositeName("cart")); 
    d.setSessionTimeout(0); 
    d.setStateManagementType(d.STATEFUL_SESSION); 
    { // set the default control descriptor 
      ControlDescriptor defaultControl = new ControlDescriptor(null); 
      defaultControl.setTransactionAttribute(ControlDescriptor.TX_NOT_SUPPORTED); 
      // set the transaction mode on purchase to TX_REQUIRES 
      ControlDescriptor purchaseControl = 
        new ControlDescriptor(CartBean.class.getMethod("purchase", null)); 
      purchaseControl.setTransactionAttribute(ControlDescriptor.TX_REQUIRED); 
      ControlDescriptor[] controls = { defaultControl, purchaseControl }; 
      d.setControlDescriptors(controls); 
    } 
    return d; 
  }
}
```
Index

Symbols

... ellipsis 1-2
[ ] brackets 1-2
| vertical bar 1-2

A

Application Assembler 2-4
audiences, description of 1-2

B

business methods
  entity bean 7-10
  implementation of 6-14

C

caching
  passive pool 6-9
  ready pool 6-9
  stateful session bean 6-9

client
  definition of 5-1
  enterprise bean
    client 3-13
  get bean information 5-8
  initialization of 5-2
  invoke enterprise bean
    methods 5-5
  locate home interface 5-2
  manage transaction 5-7
  obtain remote interface 5-3
  use bean handle 5-6
  concurrent access 7-10
  connection pooling 8-11
  timeout 8-12

contacting Borland 1-4

Container
  concepts 3-6
  diagnostic flags 3-7
  starting 3-5

conventions
  platform 1-3
  platform icons 1-3
  typographic 1-2

CORBA 2-9
  distribution mapping 5-10
  mapping to EJB 5-9

D

database connection 8-11
data source 9-8
data source element 7-22
DataSource interface 8-10
dd2xml
  description 10-3
  example of usage 10-4
  syntax 10-4
  when to use 10-4
dd2xml utility 3-8
debug mode 3-4
definition of 2-3
Deployer 2-5
deploying
  deployment descriptor 9-1
  Quick steps 9-1
deployment
  creating jar file 9-10
  deployment descriptor
    cmp-info elements 9-7
    converting to XML 10-3
    creating 3-14, 9-1
data source 9-8
  data source element 7-22
  description 6-18
  entity bean 7-20
  example 9-2, 9-7
  Inprise-specific 9-5
  requirements 9-2
  role of 9-2
  transaction isolation 9-9
deployment descriptor
  editor 9-2
developer support 1-4
diagnostic flags 3-7
distributed transaction
  two-phase commit 8-13
DTD 9-5

E

EJB
  application assembly
    information 9-4

   architecture 2-1
   attributes 9-2
   container’s services 2-1
   container-managed
     persistence 9-2
   Inprise-specific
     information 9-2
   JNDI names 9-2
   mapping to CORBA 5-9
   roles during
     development 2-4
     runtime environment
       properties 9-10
       security 9-2, 9-4
       structural information 9-3
       timeout 9-6
       transactional policies 9-2
       type information 9-2
   EJB Container 2-7, 3-6
   diagnostic flags 3-7
   Inprise 2-8
   Provider role 2-4
   starting 3-5
   EJB infrastructure 2-6
   EJB roles
     Application Assembler 2-4
     Container Provider 2-4
     Deployer 2-5
     Enterprise bean provider 2-4
     Server Provider 2-4
     System Administrator 2-5
     EJB Server 3-6
     Provider role 2-4
     EJBException 8-7
     EJBHome interface 4-6
     EJBMetaData interface 4-7
     EJBOBJECT interface 4-4

enterprise bean
  bean-managed
    transaction 8-4
    characteristics of 2-2
    compile 3-13
    components 2-2
    container-managed
      transaction 8-4
    design approaches 2-5
    development steps 3-9, 4-1
    development tasks 4-3
    get information about 5-8
    home interface 3-9, 3-10, 4-6
    locate 5-2
platform designation with icons 1-3
primary key 4-3, 7-11

R
reentrancy 7-10
remote interface 3-9, 3-11, 4-4
  EJBObject interface 4-4
  methods 4-5
  obtain reference to 5-3
RMI-IIOP 2-8

S
Serializable interface 6-16
session bean 2-3
  business methods 6-14
  deployment descriptor 6-18
  home interface 4-7, 6-10
  implementation of 6-7
  remote interface 6-11
  reference to 5-3
  remove instances of 5-6
  SessionBean interface 6-5, 6-13
  SessionSynchronization interface 6-6
    stateful
      life cycle 6-2 to 6-4
    stateless
      life cycle 6-4 to 6-5
  transaction attributes 8-5
  session bean class 6-7, 6-12
  session bean implementation class 3-11
  SessionBean interface 6-5
  SessionSynchronization interface 6-6
  skeletons
    java2iiop 10-1
  SQL types
    mapped to Java types 7-30
    stubs
    java2iiop 10-1
  support options 1-4
  System Administrator 2-5

T
technical support 1-4
  timeout
    connection pooling 8-12
tools
  code generator 3-8
  code verifier 3-8
  container shutdown 3-8
  converter 3-8
  dd2xml 10-3
  deployment descriptor
eeditor 3-8
  java2iiop 10-1
  JBuilder 4-2
  listed 10-1
  verify 10-2
transaction
  bean-managed 8-4
    UserTransaction interface 8-6
  characteristics of 8-2
  client management of 5-7
  connection pooling 8-11
  container support for 8-2
  container-managed 8-3, 8-4
  database connection 8-11
  DataSource 8-10
  declarative management of 8-3
  definition of 8-2
  deployment descriptor
    DataSource 8-10
distributed 8-13
two-phase commit 8-13
  EJBEException 8-7
  enterprise bean management of 8-3
  exceptions 8-7
    application-level 8-8
    continuing 8-9
    handling of 8-8
    rollback 8-8
    system-level 8-7
  flat 8-2
  global 8-6
  Integrated Transaction Service 8-3
  isolation level 8-13
  Java Transaction API 8-6
  Java Transaction Service 8-3
  JDBC support 8-10
  local 8-6
  Mandatory attribute 8-5
  nested 8-2
  Never attribute 8-5
  NotSupported attribute 8-5
  optimistic concurrency 7-11
  Required attribute 8-5
  RequiresNew attribute 8-5
  Supports attribute 8-5
  timeout 8-3
  transaction attributes 8-5
  two-phase commit 8-14
  transactions
    isolation levels 9-9
    two-phase commit 8-13
    completion flag 8-14
type mapping 7-30
typographic conventions 1-2

U
UserTransaction interface 5-7, 8-6

V
verify
description 10-2
  example of usage 10-3
  options 10-3
  syntax 10-3
  when to use 10-3
  Verify utility 3-8
  VisiBroker 2-8
  VisiBroker ORB, additional information 1-3

X
XML file 6-18
  Inprise-specific 9-5

Y
Y2K issues 1-4
  year 2000 issues 1-4