



Building Enterprise JavaBeans™ Components

Sun™ ONE Studio 4 Programming Series

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054 U.S.A.
650-960-1300

Part No. 816-7864-10
September 2002, Revision A

Send comments about this document to: docfeedback@sun.com

Copyright © 2002 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, U.S.A. All rights reserved.

Sun Microsystems, Inc. has intellectual property rights relating to technology embodied in the product that is described in this document. In particular, and without limitation, these intellectual property rights may include one or more of the U.S. patents listed at <http://www.sun.com/patents> and one or more additional patents or pending patent applications in the U.S. and in other countries.

This document and the product to which it pertains are distributed under licenses restricting their use, copying, distribution, and decompilation. No part of the product or of this document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any.

Third-party software, including font technology, is copyrighted and licensed from Sun suppliers.

This product includes code licensed from RSA Data Security.

Sun, Sun Microsystems, the Sun logo, Forte, Java, NetBeans, iPlanet, docs.sun.com, and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon architecture developed by Sun Microsystems, Inc.

UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd.

Netscape and Netscape Navigator are trademarks or registered trademarks of Netscape Communications Corporation in the United States and other countries.

Federal Acquisitions: Commercial Software—Government Users Subject to Standard License Terms and Conditions.

DOCUMENTATION IS PROVIDED “AS IS” AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

Copyright © 2002 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, Etats-Unis. Tous droits réservés.

Sun Microsystems, Inc. a les droits de propriété intellectuels relatants à la technologie incorporée dans le produit qui est décrit dans ce document. En particulier, et sans la limitation, ces droits de propriété intellectuels peuvent inclure un ou plus des brevets américains énumérés à <http://www.sun.com/patents> et un ou les brevets plus supplémentaires ou les applications de brevet en attente dans les Etats-Unis et dans les autres pays.

Ce produit ou document est protégé par un copyright et distribué avec des licences qui en restreignent l'utilisation, la copie, la distribution, et la décompilation. Aucune partie de ce produit ou document ne peut être reproduite sous aucune forme, par quelque moyen que ce soit, sans l'autorisation préalable et écrite de Sun et de ses bailleurs de licence, s'il y en a.

Le logiciel détenu par des tiers, et qui comprend la technologie relative aux polices de caractères, est protégé par un copyright et licencié par des fournisseurs de Sun.

Ce produit comprend le logiciel licencié par RSA Data Security.

Sun, Sun Microsystems, le logo Sun, Forte, Java, NetBeans, iPlanet, docs.sun.com, et Solaris sont des marques de fabrique ou des marques déposées de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays.

Toutes les marques SPARC sont utilisées sous licence et sont des marques de fabrique ou des marques déposées de SPARC International, Inc. aux Etats-Unis et dans d'autres pays. Les produits portant les marques SPARC sont basés sur une architecture développée par Sun Microsystems, Inc.

UNIX est une marque enregistrée aux Etats-Unis et dans d'autres pays et licenciée exclusivement par X/Open Company Ltd.

Netscape et Netscape Navigator sont des marques de Netscape Communications Corporation aux Etats-Unis et dans d'autres pays.

LA DOCUMENTATION EST FOURNIE “EN L'ÉTAT” ET TOUTES AUTRES CONDITIONS, DECLARATIONS ET GARANTIES EXPRESSES OU TACITES SONT FORMELLEMENT EXCLUES, DANS LA MESURE AUTORISÉE PAR LA LOI APPLICABLE, Y COMPRIS NOTAMMENT TOUTE GARANTIE IMPLICITE RELATIVE A LA QUALITE MARCHANDE, A L'APTITUDE A UNE UTILISATION PARTICULIERE OU A L'ABSENCE DE CONTREFAÇON.



Contents

Before You Begin xvii

1. Enterprise JavaBeans Concepts 1

The J2EE Architecture 2

The Roles of EJB Components 4

The Roles of Application Builders 5

Inside an EJB Application 6

 The Elements of an Enterprise Bean 8

 Bean Methods 8

 Types of Interfaces 10

 The Bean Class 12

 EJB QL 13

 The Deployment Descriptor 13

 The Work Flow of an EJB Application at Runtime 14

 An Enterprise Bean's Development Life Cycle 15

The IDE's Support for Enterprise Beans 16

 Developing Enterprise Beans in the IDE 17

 Creating Sets of Related CMP Entity Beans 17

 Providing Transactions 18

 Providing Persistence 18

Providing Security	18
Creating Application Clients	19
Further Reading	19
2. Design and Programming Issues	21
Deciding Which Type of Bean You Need	21
Understanding Session Beans	22
Deciding When to Use a Stateless Session Bean	23
Deciding When to Use a Stateful Session Bean	24
Selecting a Transaction Mode	25
Understanding the Life Cycle of a Session Bean	26
Understanding Entity Beans	29
Taking Advantage of the EJB Container's Services	30
Understanding the Life Cycle of an Entity Bean	31
Understanding Sets of Related CMP Entity Beans and Container-managed Relationships	35
Understanding Message-Driven Beans	36
Using Message Sources (Destinations)	37
Deciding When to Use a Message-Driven Bean	37
Deciding When Another Bean Type Is Better	38
Understanding the Life Cycle of a Message-Driven Bean	39
Using Enterprise Beans in Applications	41
Using Exceptions to Handle Problems	41
Working With Deployment Descriptors	42
Enforcing Security Policies	42
Declaring Security in Enterprise Beans	43
Programming Security Into Enterprise Beans	43
Understanding the Application Servers and Databases	44
Further Reading	45

3. Developing Session Beans	47
Using the EJB Builder With Session Beans	48
Selecting a Session Bean Type	49
Stateless or Stateful Session Beans	49
Container-Managed or Bean-Managed Transactions	51
Defining a Session Bean	52
Creating a Package	52
Starting the EJB Builder Wizard	52
Generating the Default Session Bean	53
Looking at a Session Bean's Classes	55
Expanding the Nodes	56
Reviewing the Generated Classes	56
Default Create Method	57
Life-Cycle Methods	57
Completing Your Session Bean	59
Using Recommended Approaches When Working With Enterprise Beans	59
Completing Create Methods	60
Completing a Stateless Bean's Create Method	60
Completing a Stateful Bean's Create Methods	60
Adding a Create Method to a Stateful Bean	61
Completing Life-Cycle Methods	61
Completing the <code>ejbPassivate</code> Method	62
Completing the <code>ejbActivate</code> Method	62
Adding Business Methods	63
Coding Transactions	63
Understanding Transaction Spans	64
Specifying Transaction Boundaries and Rollbacks	64

After Creating Your Session Bean 67

Further Reading 67

4. Developing CMP Entity Beans 69

Using the EJB Builder With CMP Entity Beans 69

Comparing CMP and BMP Entity Beans 71

Creating Sets of Related CMP Entity Beans 72

Defining a CMP Entity Bean 72

 Creating a Package 73

 Having a Data Source Ready 73

 Starting the EJB Builder Wizard 74

 Generating a CMP Entity Bean's Infrastructure 74

 Specifying Persistent Fields From a Database Table 76

 Creating Your Bean's Persistent Fields From Scratch 82

Looking at a CMP Entity Bean's Classes 84

 Expanding the Nodes 85

 Reviewing the Generated Classes 88

 Default Finder Method 88

 Persistent Fields and Accessor Methods 88

 Primary-Key Class and Required Methods 90

 A CMP Entity Bean's Life-Cycle Methods 91

Completing Your CMP Entity Bean 92

 Using Recommended Approaches When Working With Enterprise Beans 93

 Defining Create Methods 93

 Adding or Replacing a Primary Key 95

 Creating a New Primary Key 95

 Handling Foreign Keys 97

 Defining Business Methods 97

Adding Finder Methods	98
Defining Home Methods	99
Defining Select Methods	100
Defining Additional Fields	101
After Creating Your CMP Entity Bean	101
Further Reading	101
5. Developing Sets of Related CMP Entity Beans	103
Using the EJB Builder With Sets of Related CMP Entity Beans	104
Creating All Related CMP Entity Beans at Once	104
Creating a Set of Related CMP Entity Beans One at a Time	105
Defining a Set of Related CMP Entity Beans	105
Creating a Package	106
Preparing to Use a Database or Schema	106
Starting the EJB Builder Wizard	107
Generating the Bean Set's Infrastructure	107
Using a Database Connection	108
Using a Database Schema Object	114
Looking at the Components of a CMP Entity Bean Set	115
Expanding the EJB Module's Node	116
Reviewing the Generated Classes	117
Completing Your Set of Related CMP Entity Beans	117
Using Recommended Approaches When Working With Enterprise Beans	118
Adding a Bean to the Set	119
After Creating Your Set of Related CMP Beans	121
6. Developing BMP Entity Beans	123
Deciding on an Approach	123

Building a BMP Entity Bean	124
Creating a Package	124
Starting the EJB Builder Wizard	124
Generating a BMP Entity Bean's Infrastructure	125
Looking at a BMP Entity Bean's Classes	125
Expanding the Nodes	126
Reviewing the Generated Classes	126
findByPrimaryKey Method	126
A BMP Entity Bean's Life-Cycle Methods	127
Completing Your BMP Entity Bean	128
Using Recommended Approaches When Working With Enterprise Beans	129
Adding Persistence Logic	129
Adding a Primary-Key Class	129
Adding Methods	130
Defining Create Methods	130
Adding Finder Methods	131
Defining Business and Home Methods	131
After Creating Your BMP Entity Bean	132
Further Reading	132
7. Developing Message-Driven Beans	133
Using the EJB Builder With Message-Driven Beans	134
Deciding on Transaction Management	135
Defining a Message-Driven Bean	136
Creating a Package	136
Starting the EJB Builder Wizard	136
Generating the Basic Message-Driven Bean	137
Looking at Your Message-Driven Bean in the Explorer	137

Expanding the Nodes	138
Reviewing the Generated Class	138
Completing Your Message-Driven Bean	139
Using Recommended Approaches When Working With Enterprise Beans	140
Completing the <code>onMessage</code> Method	140
Completing the <code>setMessageDrivenContext</code> Method	141
After Creating Your Message-Driven Bean	141
Specifying a Message-Driven Destination	142
Specifying a Message Selector	143
Specifying Resources for Client Message-Driven Beans	143
Specifying Resource Factories	143
Specifying Resources	144
Specifying Resources for Listener Message-Driven Beans Deployed to the RI	145
Avoiding Pitfalls of Message-Driven Beans	146
Further Reading	146
8. Preparing Enterprise Beans for Deployment	147
Understanding Deployment Information	147
Looking at a Generated Deployment Descriptor	148
Editing an EJB Module's Deployment Descriptor	149
Editing an EJB Module's Deployment Descriptor Directly	149
Reverting to the EJB Module's Last Generated Descriptor	149
Using Properties to Edit a Deployment Descriptor	150
Specifying Bean Properties	150
Using the Properties Tabbed Pane	150
Properties of Entity Beans	151
Properties of Session Beans	152

Properties of Message-Driven Beans	152
Using the References Tabbed Pane	153
Specifying EJB Local References	153
Specifying EJB References	155
Specifying Environment Entries	155
Specifying Resource Environment References	156
Specifying Resource References	156
Specifying Security-Role References	158
Using the J2EE RI Tabbed Pane	159
Setting J2EE RI Properties for Individual Session and Entity Beans	159
Setting J2EE RI Properties for Message-Driven Beans	160
Creating an EJB Module	161
Deciding What Should Go Into an EJB Module	162
Putting Enterprise Beans in an EJB Module	162
Setting Database-related Properties for CMP Entity Beans	163
Understanding the RI's Generated SQL	165
If Your CMP Beans Don't Need to Use an Existing Database Table	165
If Your CMP Beans Need To Use an Existing Database Table	168
If Your EJB Module Contains an EJB 1.1 CMP Entity Bean	168
Understanding the Order of CMP Field Values	170
Adding Transaction Attributes to an EJB Module	170
Changing EJB References Within the EJB Module or Application	172
Overriding a Reference at the Module Level	173
Overriding a Reference at the Application Level	174
Creating an EJB JAR	174
Adding a JAR File to an EJB Module	174

9. Testing Enterprise Beans 177

Fulfilling the Prerequisites for Testing	177
Preparing to Deploy to the J2EE RI	178
Preparing to Test Beans Against the PointBase Database	178
Starting the Database Server and Web Browser	179
Generating Test Objects	180
Deploying the Test Application to a Server	182
Deploying and Executing the Test Application in One Step	183
Executing the Test Application	184
Using the Test Client to Test Your Beans	184
Understanding the Test Client Page	184
Testing the Sample Bean's Home Interface	186
Testing the Sample Bean's Business Method	187
Creating New Testing Classes	188
Making Changes After Deployment	188
Preparing to Test: Some Variations	189
If You Want to Test a CMP or BMP Bean	189
If You Want to Test a Bean With EJB References	190
Adding Remote Interfaces to a Bean	190
A. Working With Enterprise Beans	195
Using Recommended Approaches When Editing Beans	195
Working Through the Logical Node	195
Using the Customizer or Property Sheet	197
Using the Source Editor to Edit Beans	197
Understanding the IDE's Error Information	199
Compiling and Validating Enterprise Beans	199
Saving Your Changes	201
Renaming an Enterprise Bean	201

Modifying a Bean Based on Another Bean	202
Copying and Pasting an Enterprise Bean	202
Exchanging Bean Classes or Interfaces	203
Editing a Bean's Methods	204
Viewing a Method	204
Changing an Entity Bean's Fields	204
Renaming a Field	205
Changing the Type of a Field	205
Deleting an Enterprise Bean	205
B. Migrating and Upgrading EJB 1.1 Enterprise Beans	207
Understanding Updates in the Current Release	207
Making Specific Changes	209
Converting a CMP 1.x Entity Bean	209
Avoiding the Use of New Features in an Old Bean	210
Don't Add Local Interfaces to a CMP 1.x Entity Bean	210
Don't Add Local EJB References, Either	210
Changing the PointBase User Name and Password	211
Avoiding the transient Modifier	211
Shifting a Bean's RI Properties to the EJB Module Level	212
Changing CMP Entity Bean Properties Before Testing the Bean	212
Index	215

Figures

FIGURE 1-1	Model J2EE Application as Supported by the Sun ONE Studio IDE	3
FIGURE 1-2	Typical Basic Configuration for an EJB Application	4
FIGURE 1-3	Example of an Application With All Three Kinds of Enterprise Beans	7
FIGURE 1-4	Work Flow Inside the Application at Runtime	14
FIGURE 1-5	Development, Assembly, and Deployment of an Enterprise Bean	15
FIGURE 1-6	Generated Elements of an Enterprise Bean Shown in the Explorer Window	17
FIGURE 2-1	Basic Choices About Enterprise Beans in the Sun ONE Studio IDE	22
FIGURE 3-1	Possible Wizard Selections for a Stateless (or Stateful BMT) Session Bean	53
FIGURE 3-2	Default Classes of a Typical Session Bean With Remote Interfaces	55
FIGURE 3-3	Explorer's Detailed View of a Typical Session Bean With Remote Interfaces	56
FIGURE 4-1	Selections in the EJB Builder Wizard for CMP Entity Beans	75
FIGURE 4-2	Default Classes of a Typical CMP Entity Bean	84
FIGURE 4-3	Explorer's Detailed View of a Typical CMP Entity Bean With Local Interfaces	86
FIGURE 4-4	Explorer's Detailed View of a Typical CMP Entity Bean With a Composite Primary Key	87
FIGURE 5-1	Selections in the EJB Builder Wizard for a CMP Entity Bean Set	108
FIGURE 5-2	Default Classes of a Typical Set of Related CMP Entity Beans	116
FIGURE 5-3	Expanded Nodes of an EJB Module Containing Related CMP Entity Beans	117
FIGURE 6-1	Explorer's Detailed View of a BMP Entity Bean	126
FIGURE 7-1	Default Class and Methods of a Typical Message-Driven Bean	137
FIGURE 7-2	Explorer's Detailed View of a Typical Message-Driven Bean	138

FIGURE 8-1	References Tabbed Pane of the Properties Dialog Box for a CMP Entity Bean	153
FIGURE 8-2	J2EE RI Tabbed Pane Showing Properties for an EJB Module Containing CMP Beans	164
FIGURE 8-3	Table-related Settings for a CMP Entity Bean in an EJB Module	166
FIGURE 8-4	Example of SQL Code Generated by the RI Plugin for a CMP Bean's <code>createTable</code> Method	167
FIGURE 8-5	Example of SQL Code Generated for a CMP Bean's Finder Method	169
FIGURE 8-6	EJB Local References Property Editor, Showing an Example of Override Selections for an Enterprise Bean's Local References	173
FIGURE 9-1	Example of Test Objects Generated for Enterprise Beans	181
FIGURE 9-2	Client JSP Page Generated to Test Simple Session Bean <code>dollarToYen</code>	185
FIGURE 9-3	Customizer for Adding an Interface Class to a Bean	192

Tables

TABLE 3-1	Deciding Between Stateless and Stateful Session Beans	50
TABLE 3-2	Deciding Between Container-Managed and Bean-Managed Transactions	51
TABLE 3-3	Purpose of Life-Cycle Methods In a Session Bean Class	58
TABLE 3-4	Purpose of Session-Synchronization Methods in a Session Bean Class	59
TABLE 3-5	Relationship Between Transactions and Methods	64
TABLE 4-1	Deciding Between CMP and BMP Entity Beans	71
TABLE 4-2	Purpose of Default Life-Cycle Methods in a CMP Entity Bean Class	91
TABLE 6-1	Purpose of Default Life-Cycle Methods in a BMP Entity Bean Class	127
TABLE 7-1	Deciding Between Container-Managed and Bean-Managed Transactions	135
TABLE 7-2	Purpose of <code>ejbCreate</code> and <code>onMessage</code> Methods in a Message-Driven Bean's Bean Class	139
TABLE 7-3	Purpose of Default Life-Cycle Methods in a Message-Driven Bean's Bean Class	139
TABLE 7-4	Example of a <code>setMessageDrivenContext</code> Method	141

Before You Begin

This book describes how to build Enterprise JavaBeans™ components (enterprise beans) using the Sun™ Open Net Environment (Sun ONE) Studio, Enterprise Edition for Java, integrated development environment (IDE).

Enterprise beans come in several varieties. A session bean can be stateful or stateless, and can manage its own transactions or have them managed by the EJB™ container. An entity bean can manage its own persistence or let the container manage its relationship with the underlying database. You can use the Sun ONE Studio IDE to build those enterprise beans as well as message-driven beans and sets of entity beans whose relationships are managed by the EJB container. Flexible support is available to the developer in building all of these types of enterprise beans. The IDE streamlines the task of coding and helps ensure that the results are consistent with the Java 2 Platform, Enterprise Edition Blueprints (the J2EE™ Blueprints).

Another book in this series, *Building J2EE Applications*, suggests designs for industrial-strength applications that use enterprise beans and other J2EE components. It offers various application scenarios, and explains how to assemble finished enterprise beans and other components into modules, how to deploy them in applications, and how to run those applications. This book, *Building Enterprise JavaBeans Components*, concentrates on the design and creation of enterprise beans, and on basic issues of assembly, deployment, and testing. If you are responsible for providing enterprise beans, assembling them into applications, and deploying them on application servers, you should refer to both books.

You can use the Sun ONE Studio IDE to create the examples in this book on the systems listed in the Release Notes. Find the Release Notes on the following website:

<http://forte.sun.com/ffj/documentation/index.html>

Screen shots vary slightly from one platform to another. You should have no trouble translating the slight differences to your platform. Although almost all procedures use the IDE's user interface, you might occasionally be instructed to enter a command at the command line. Here too, there are slight differences from one

platform to another. For example, a Microsoft Windows command might look like this:

```
c:\>cd MyWorkDir\MyPackage
```

To translate for UNIX® or Linux environments, simply change the prompt and use forward slashes:

```
% cd MyWorkDir/MyPackage
```

Before You Read This Book

If you want to use the Sun ONE Studio IDE to build enterprise beans, you will benefit from reading this document. Before you start, you should be familiar with the following subjects:

- The Java programming language
- The EJB component model
- The JDBC™ API and JDBC-enabled driver syntax
- Relational database concepts (such as tables, columns, and keys)
- How to use the chosen database
- The Java Message Service (JMS) API
- XML syntax

To develop enterprise beans, you need to know J2EE concepts and generally to understand enterprise beans. Whenever further details are needed, refer to the following list of resources:

- *Enterprise JavaBeans Specification, version 2.0*
<http://java.sun.com/products/ejb/docs.html>
- *Java 2 Platform, Enterprise Edition Blueprints*
<http://java.sun.com/j2ee/blueprints>
- *Java 2 Platform, Enterprise Edition Specification*
<http://java.sun.com/j2ee/download.html#platformspec>
- *The J2EE Tutorial (for J2EE SDK version 1.3)*
http://java.sun.com/j2ee/tutorial/1_3-fcs/index.html
- *Java Message Service Tutorial*
<http://java.sun.com/products/jms/tutorial/index.html>
- *Java Transaction API (JTA) Specification*
<http://java.sun.com/products/jta>

At some stages of bean development, you also need to know about specific application servers. Refer to a server's documentation for details.

Note – Sun is not responsible for the availability of third-party web sites mentioned in this document and does not endorse and is not responsible or liable for any content, advertising, products, or other materials on or available from such sites or resources. Sun will not be responsible or liable for any damage or loss caused or alleged to be caused by or in connection with use of or reliance on any such content, goods, or services available on or through any such sites or resources.

How This Book Is Organized

Chapter 1 introduces J2EE and Enterprise JavaBeans concepts, and gives an overview of the Sun ONE Studio IDE's support for creating enterprise beans and assembling them into EJB modules.

Chapter 2 discusses design and programming issues for those who use the IDE to build enterprise beans and assemble EJB modules.

Chapter 3 tells how to use the IDE to create stateless or stateful session beans that manage their own transactions or delegate their transaction management to the EJB container.

Chapter 4 tells how to use the IDE to create single entity beans with container-managed persistence (CMP entity beans).

Chapter 5 tells how to use the IDE to create sets of CMP entity beans with their relationships automatically included.

Chapter 6 tells how to use the IDE to create entity beans with bean-managed persistence (BMP entity beans).

Chapter 7 tells how to use the IDE to create message-driven beans.

Chapter 8 shows how to prepare a bean for deployment by specifying properties on the bean and its EJB module.

Chapter 9 explains how to use the IDE's testing feature to test enterprise beans.

Appendix A contains instructions for working with enterprise beans in the IDE.

Appendix B provides tips on updating and converting EJB 1.1 enterprise beans so that they are maintainable and executable in the current version of the IDE.

Typographic Conventions

Typeface	Meaning	Examples
AaBbCc123	The names of commands, files, and directories; on-screen computer output	Edit your <code>.cvspass</code> file. Use <code>DIR</code> to list all files. Search is complete.
AaBbCc123	What you type, when contrasted with on-screen computer output	> login Password:
<i>AaBbCc123</i>	Book titles, new words or terms, words to be emphasized	Read Chapter 6 in the <i>User's Guide</i> . These are called <i>class</i> options. You <i>must</i> save your changes.
<i>AaBbCc123</i>	Command-line variable; replace with a real name or value	To delete a file, type DEL <i>filename</i> .

Related Documentation

Sun ONE Studio 4 documentation includes books delivered in Acrobat Reader (PDF) format, online help, release notes, readme files for example applications, and Javadoc™ documentation.

Documentation Available Online

The documents described in this section are available from the docs.sun.comSM web site and from the documentation page of the Sun ONE Studio Developer Resources portal (<http://forte.sun.com/ffj/documentation>).

The docs.sun.com web site (<http://docs.sun.com>) enables you to read, print, and buy Sun Microsystems manuals through the Internet. If you cannot find a manual, see the documentation index installed with the product on your local system or network.

- Release Notes (HTML format)

Available for each Sun ONE Studio 4 edition. Describe last-minute release changes and provide technical notes.

- Getting Started guides (PDF format)

Describe how to install the Sun ONE Studio 4 integrated development environment (IDE) on each supported platform and include other pertinent information, such as system requirements, upgrade instructions, application server configuration instructions, command-line switches, installed subdirectories, database integration, and information on how to use the Update Center.

- *Sun ONE Studio 4, Community Edition Getting Started Guide* - part no. 816-7871-10
- *Sun ONE Studio 4, Enterprise Edition for Java Getting Started Guide* - part no. 816-7859-10
- *Sun ONE Studio 4, Mobile Edition Getting Started Guide* - part no. 816-7872-10

- The Sun ONE Studio 4 Programming series (PDF format)

This series provides in-depth information on how to use various Sun ONE Studio 4 features to develop well-formed J2EE applications.

- *Building Web Components* - part no. 816-7869-10

Describes how to build a web application as a J2EE web module using JSP pages, servlets, tag libraries, and supporting classes and files.

- *Building J2EE Applications* - part no. 816-7863-10

Describes how to assemble EJB modules and web modules into a J2EE application, and how to deploy and run a J2EE application.

- *Building Enterprise JavaBeans Components* - part no. 816-7864-10 (this book)
- *Building Web Services* - part no. 816-7862-10

Describes how to use the Sun ONE Studio 4 IDE to build web services, to make web services available to others through a UDDI registry, and to generate web service clients from a local web service or a UDDI registry.

- *Using Java DataBase Connectivity* - part no. 816-7870-10

Describes how to use the JDBC productivity enhancement tools of the Sun ONE Studio 4 IDE, including how to use them to create a JDBC application.

- Sun ONE Studio 4 tutorials (PDF format)

These tutorials demonstrate how to use the major features of each Sun ONE Studio 4 edition.

- *Sun ONE Studio 4, Community Edition Tutorial* - part no. 816-7868-10

Provides step-by-step instructions for building a simple J2EE web application.

- *Sun ONE Studio 4, Enterprise Edition for Java Tutorial* - part no. 816-7860-10

Provides step-by-step instructions for building an application using EJB components and Web Services technology.

- *Sun ONE Studio 4, Mobile Edition Tutorial* - part no. 816-7873-10

Provides step-by-step instructions for building a simple application for a wireless device, such as a cellular phone or personal digital assistant (PDA). The application will be compliant with the Java 2 Platform, Micro Edition (J2ME™ platform) and conform to the Mobile Information Device Profile (MIDP) and Connected, Limited Device Configuration (CLDC).

You can also find the completed tutorial applications at
<http://forte.sun.com/ffj/documentation/tutorialsandexamples.html>

Online Help

Online help is available inside the Sun ONE Studio IDE. You can open help by pressing the help key (F1 in Microsoft Windows and Linux environments, Help key in the Solaris environment), or by choosing Help → Contents. Either action displays a list of help topics and a search facility.

Examples

You can download several examples that illustrate a particular Sun ONE Studio 4 feature, as well as completed tutorial applications, from the Sun ONE Studio Developer Resources portal at:

<http://forte.sun.com/ffj/documentation/tutorialsandexamples.html>

Javadoc Documentation

Javadoc documentation is available within the IDE for many Sun ONE Studio 4 modules. Refer to the release notes for instructions on installing this documentation. When you start the IDE, you can access this Javadoc documentation within the Javadoc pane of the Explorer.

Sun Welcomes Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions. You can email your comments to Sun at:

docfeedback@sun.com

Please include the part number (816-7864-10) of this document in the subject line of your email.

Enterprise JavaBeans Concepts

Enterprise JavaBeans™ components (enterprise beans) are key building blocks in the the Java™ 2 Platform, Enterprise Edition (J2EE™) architecture. This chapter introduces:

- The main ideas behind the J2EE architecture
- The roles of enterprise beans and other elements of the J2EE model's EJB™ tier
- The components and work flow of an EJB application
- The EJB Builder, which is a collection of wizards and other GUI support in the Sun™ Open Net Environment (Sun ONE) Studio, Enterprise Edition for Java software (Sun ONE Studio 4 IDE, the integrated development environment)

If you are already conversant with J2EE and developing enterprise beans, and you just want to know specifically how to use the IDE to create and work with your beans, see Chapters 3 through 9 and the appendixes.

An enterprise bean is usable only when it has been placed with any related beans in an EJB module, assembled into an application, and deployed on a server. The tasks of development, assembly, and deployment can be distributed among developers or working units according to their expertise. This document focuses on the work that the EJB developer does before handing over a finished enterprise bean or related group of beans to be assembled into an application and deployed on a server. Another document in this series, *Building J2EE Applications*, discusses J2EE application design, assembly, and deployment.

Note – The Sun ONE Studio 4 IDE supports *Enterprise JavaBeans Specification*, version 2.0.

The J2EE Architecture

The Java 2 Platform, Enterprise Edition (J2EE) documents describe a services-based application architecture within which transactional, scalable, secure, portable Java components can be deployed and redeployed. Combining tiers of databases, servers, and client-access mechanisms on the J2EE model, your organization can develop applications that support your entire enterprise.

A J2EE application architecture typically has the following major features:

- **A client tier.** According to the J2EE specification, this tier can contain HTML or Java applets running in a browser, Extensible Markup Language (XML) documents transmitted through HTTP, and Java clients running in a client container.

The Sun ONE Studio 4 IDE supports the execution and deployment of applications that use Java clients, JSP pages, servlets, and other enterprise beans as clients.

- **One or more EJB or server tiers.** These tiers can contain:
 - **Presentation logic.** Servlets or JavaServer Pages™ (JSP™ pages) running in web servers.
 - **Application logic.** Enterprise JavaBeans components (enterprise beans) running in application servers.
- **A database tier.**

The server tiers of a typical J2EE application can contain any or all of the elements shown in FIGURE 1-1.

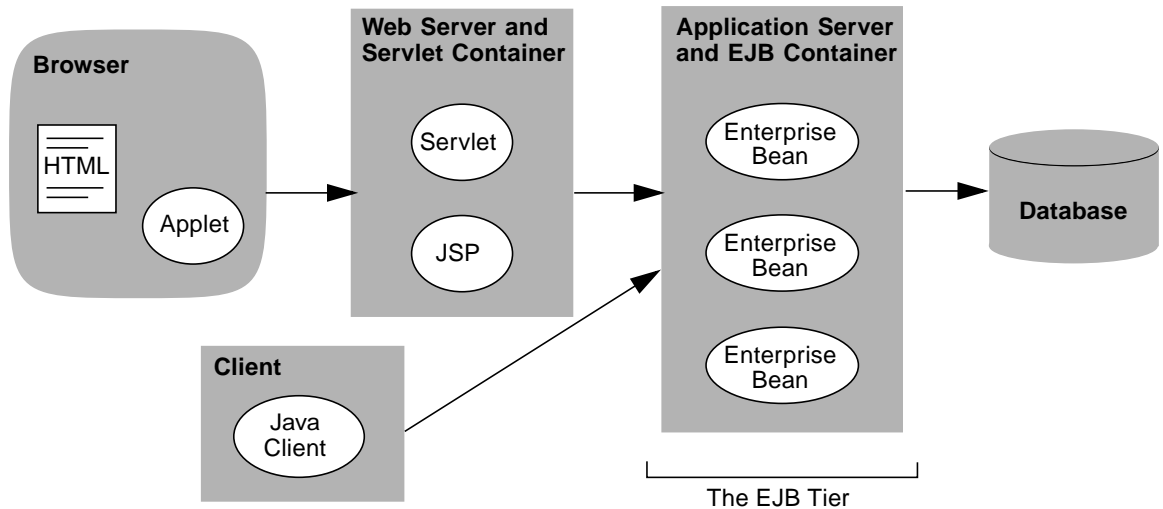


FIGURE 1-1 Model J2EE Application as Supported by the Sun ONE Studio IDE

An enterprise bean is a Java component, a set of Java interfaces and classes that make up a business entity. These interfaces and classes contain methods that implement business logic on an application server. One type of enterprise bean also contains fields that can be mapped to database columns. Another type of enterprise bean can manage interactions between other enterprise beans in the same application. Enterprise beans can be combined with any of the different types of components shown in FIGURE 1-1 to create applications.

Although both enterprise beans and JavaBeans™ components are written in the Java programming language, they are not the same. You can use JavaBeans components with design tools to customize instances of Java classes, and you can link the customized objects through events. Enterprise beans, on the other hand, implement distributed, container-managed transactional services for multiple users.

The design of the EJB tier carries the modularity and portability of Java components several steps farther. For that reason, your job as an EJB developer is more modular: You can focus more on the business data of an application than on distributed computing. When you build an application using JavaBeans components, you must also build the server framework. However, when you build an application on the J2EE model using enterprise beans, the server-side infrastructure is already built into the application server. You don't need to provide generic services such as support for transactions, security, or remote access.

The Roles of EJB Components

The most basic configuration for a typical EJB application is shown in FIGURE 1-2: an application client, an application server, the EJB container, at least one enterprise bean, and a data store of some kind. In this figure, a database is used.

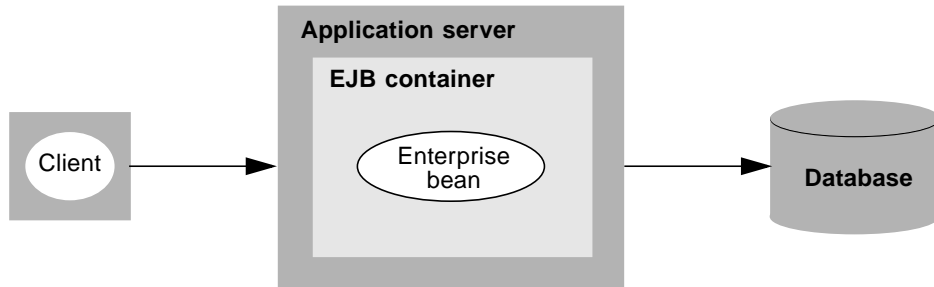


FIGURE 1-2 Typical Basic Configuration for an EJB Application

The contracts (that is, the interaction and implied agreements) between an enterprise bean, its EJB container, and the application server lend flexibility and power to the J2EE application, while simplifying the task of creating enterprise beans.

The EJB container is more a concept than an object. It's the environment that surrounds the enterprise beans on the EJB server to provide life-cycle management, security, distributed transaction support, and other services.

One or more enterprise beans can be deployed in a single container, which uses the standard Java Naming and Directory Interface™ (JNDI) API to locate an individual bean and make it available to a client.

The container intercedes between its beans and their clients. When a client needs work done by an enterprise bean, the container intercepts the method call. Working on behalf of many enterprise beans and their clients, the container can manage services (such as security and transactions) across calls, components, and even other containers running on other servers. This design feature allows the container to apply its services flexibly and transparently.

For individual enterprise beans, the container is designed to manage database persistence and transactions. This ensures a standard approach to state-management events. It also means that your beans can perform database-access operations without you, the EJB developer, having to write the SQL code or use the JDBC™ API directly (unless you need to override the container's default behaviors).

The container's services ensure that persistent data in an enterprise bean is saved if a client terminates or if the server shuts down.

The application server provides lower-level support such as naming, directory, and email services.

Enterprise beans are of three main types: session beans, entity beans, and message-driven beans. These types are discussed in more detail starting in Chapter 2, but to briefly describe the role of each type:

- A session bean manages the conversation between a client and the EJB server, and can direct complex interactions with entity beans. For example, a session bean can pass in requests for data to entity beans, package the resulting data, and pass it back out to a client.
- An entity bean usually represents an entity, or a table of data, in a database. Many entity beans can work cooperatively inside a Java virtual machine (JVM™) on an application server.
- A message-driven bean forms a functional layer between client and server. This type of bean receives client message notifications and starts asynchronous interactions among other enterprise beans that are deployed to the server.

Finished enterprise beans are packaged into an EJB module (which is a logical construct for an EJB JAR file) for assembly into an application and deployment on a server. An application server can house one or more J2EE applications, a J2EE application can house one or more EJB modules, and an EJB module can house one or more enterprise beans.

The Roles of Application Builders

The J2EE architecture implies a methodology and supports the division of responsibilities in the application-building process into different roles. In a typical development organization, some team members have more knowledge of the business, while others are more adept at systems-level development. If the J2EE model is applied to an organization where not everyone is responsible for the same kind of work, the business or domain experts might take the role of bean provider or EJB developer, while the more systems-minded developers might do the assembly and deployment work.

At least three roles are common in development environments:

- **EJB development.** The EJB developers (who are application developers and, often, domain experts) build enterprise beans without having to focus on the surrounding framework. For testing purposes and for convenience, these people might also package finished beans into EJB modules.

- **EJB module assembly.** The assembler or assemblers can make final groupings of finished enterprise beans into EJB modules, and combine those and other J2EE building blocks into applications. The modularity of J2EE design enables container-neutral decisions at this stage.
- **Deployment.** One or more deployers can deploy J2EE applications in a specific environment, making container-specific and server-specific decisions.

The IDE, with its built-in support for enterprise beans, is designed to serve the J2EE approach to building applications. When you use this IDE to build your enterprise beans, you can concentrate on writing the business logic your application needs. In the EJB development role, you need to make only minimal gestures toward the assembly and deployment steps.

However, when necessary, one person can assume all three roles. The IDE provides seamless support for all stages of enterprise bean development, assembly, and deployment.

Inside an EJB Application

In the typical EJB application shown in FIGURE 1-3, many client programs can get access simultaneously to the heart of the application, which resides on the EJB server and is managed by the EJB container. Within the EJB tier, instances of two different session beans (one prompted by a message-driven bean) manage interactions with instances of four different entity beans to let clients look up calendar appointments and schedule meeting spaces. Data from the database is read into instances of the entity beans, and clients' updates to the entity-bean instances are posted to the database.

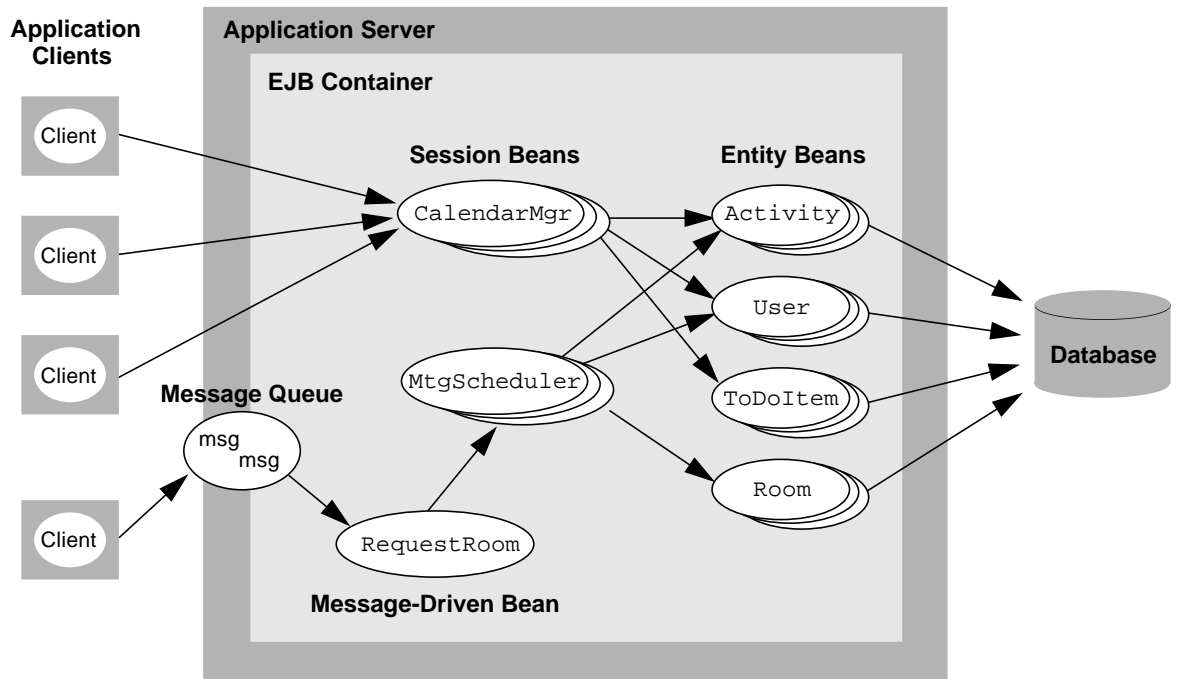


FIGURE 1-3 Example of an Application With All Three Kinds of Enterprise Beans

The Elements of an Enterprise Bean

Every enterprise bean has at least one class.

- A message-driven bean needs only its bean class, with no interfaces.
- A session bean typically is composed of three elements: its bean class and two remote-type interfaces (the home interface and the remote interface). However, a session bean can have local-type interfaces as well.
- An entity bean typically has three elements: its bean class and two local-type interfaces (the local home interface and the local interface). An entity bean can also have remote-type interfaces, and it might have a primary-key class.

For more information about enterprise bean interfaces, see “Types of Interfaces” on page 10.

Responsibilities in regard to the elements of an EJB application are as follows:

- As the EJB developer, you use the IDE to generate the bean class and the interfaces for each enterprise bean. For an entity bean, you also define a primary-key class, if needed. You complete the code that the IDE generates, and you declare deployment information.
- The container in which the bean is deployed implements the bean’s interfaces and manages interactions between components and the data storage.
- The client using the bean creates its own stubs to call the bean’s interfaces, which interact with the bean class to get the application’s work done. Or, the client sends messages to a destination, and a message-driven bean listens for those messages and interacts as requested with a session bean.

Bean Methods

A J2EE application gets its work done through methods that the client calls on the bean. The kinds of methods that an enterprise bean includes are briefly discussed next (and are discussed in detail in Chapters 3 through 7). All method declarations are added either automatically by the IDE or explicitly by the developer. To add all necessary parts of a method declaration, you follow a single, short sequence of actions in dialog boxes. The IDE generates the corresponding parts of the method and places them in the appropriate classes.

- **Finder Methods.** The client goes through the home interface to find an entity bean instance by its primary key. The developer can also add other finder methods.

The IDE automatically generates a `findByPrimaryKey` method declaration in the local home interface of every entity bean (and in the bean’s home interface, if it has one). The IDE also places a corresponding `ejbFindByPrimaryKey` method declaration in the bean class of every entity bean that manages its own persistence (that is, a bean-managed persistent entity bean, or BMP entity bean).

If the developer adds another finder method, the IDE automatically places the corresponding method declarations in the local home (and home) interface and, for BMP entity beans, in the bean class.

Added finder methods include EJB Query Language (EJB QL) statements, which the bean's application server plugin converts automatically to the kind of SQL code the server needs.

- **Create Methods.** The container initializes the enterprise bean instance, using the create method's arguments.

The IDE automatically generates a create method declaration in the home interface of every session bean (and in the bean's local home interface, if it has one). The IDE places a corresponding `ejbCreate` method declaration in the bean class.

The IDE also generates an `ejbCreate` method declaration in the bean class of every message-driven bean.

Since an entity bean doesn't have to contain a create method, the IDE doesn't automatically generate a declaration for an entity bean. However, if the developer adds a create method to an entity bean, the IDE generates the corresponding `create`, `ejbCreate`, and `ejbPostCreate` method declarations in the appropriate classes. An entity bean can have more than one create method.

- **Business Methods.** A client calls business methods on a bean through the bean's remote interface (or local interface, as applicable).

The developer explicitly adds business methods to the bean; the IDE doesn't generate any default business method declarations. However, when the developer does specify a business method, the IDE places matching method declarations in the bean class and in the remote, local, or remote and local interfaces.

- **Home Methods.** An entity bean can use a home method for a lightweight operation that doesn't require access to any particular instance of the bean. (By contrast, a business method does require access to a particular instance.) The developer explicitly adds a home method, and the IDE generates the corresponding method declaration in the bean class and the bean's local home or home interface. An entity bean can have any number of home methods.
- **Select Methods.** An entity bean that delegates its persistence to the container (that is, a container-managed persistent entity bean, or CMP entity bean) can use a select method. Like a finder method, a select method can query the database and return a local or remote interface or a collection. In addition, a select method can query a related entity bean within the same EJB module and return values from its persistent fields. Select methods aren't exposed in remote-type interfaces and can't be invoked by a client.

The developer explicitly adds one or more select methods to the bean class. Select methods include EJB Query Language (EJB QL) statements, which the bean's application server plugin converts automatically to the kind of SQL code the server needs.

- **OnMessage Methods.** A client sends a message through a Java Message Service (JMS) destination to call an `onMessage` method on a message-driven bean.

The IDE automatically generates the `onMessage` method declaration in the bean's class. The developer completes the method body.

- **Life-cycle Methods.** The container calls several methods to manage the life cycle of an enterprise bean. Depending on the type of bean, the container works through the methods in slightly different ways. The developer has the option of specifying parameters for some of these methods.

The IDE automatically generates the appropriate life-cycle method declarations for each type of bean and places them in the bean class.

Types of Interfaces

Since session beans are often called by application clients, which live outside the enterprise beans' application server, the IDE offers remote-type interfaces (that is, a remote interface and a home interface) as a default for each session bean. However, depending on how a particular session bean will be called, the developer can choose either or both types of interfaces when creating the bean.

Entity beans are normally called by session beans and by other entity beans inside the same application server. Such entity beans need only local-type interfaces (that is, a local interface and a local home interface). Local-type interfaces save processing time because they pass parameters by reference instead of serializing the parameter values. However, again, depending on the situation, the developer can apply either or both kinds of interfaces to an entity bean.

Note – Any bean that will be tested using the IDE's testing feature must have remote interfaces.

All four of these interface types are discussed next.


The Remote Interface

A client views and gets access to the enterprise bean through the bean's remote interface. Signatures for the business methods that the client can call on the bean are in the remote interface, but the complete code for the business methods is in the bean class. The container creates a class that implements the remote interface.

The remote interface extends `javax.ejb.EJBObject`. A client uses this interface to locate the home interface through a JNDI lookup call, calls a method on the home interface to retrieve a specific instance of the bean (with the remote interface as the return type), and then calls business methods on that instance.

When you use the Sun ONE Studio 4 IDE to create an enterprise bean with the remote-type interfaces (through which the bean can be called from outside the server), the EJB Builder's GUI support and validation help ensure that the remote interface's methods follow the rules defined in the J2EE documents. Those rules include the following:

- The method signatures in the remote interface have corresponding methods in the bean class.
- The arguments and return values are valid RMI types.
- The methods' throws clauses include the appropriate exception classes.


The interface node looks like this in the IDE's Explorer window:  Order
The default label is the name of the enterprise bean.

The Home Interface

The enterprise bean's home interface extends `javax.ejb.EJBHome` and defines the create and finder methods that the client can call on the enterprise bean. A client uses JNDI to locate the home interface, and the container provides a class that implements the home interface.

When you use the IDE to create an enterprise bean with remote-type interfaces, the EJB Builder's GUI support and validation help ensure that the home interface's methods follow the basic rules for enterprise beans. Those rules include the following:

- The method signatures in the home interface have corresponding methods in the bean class (except in the case of finder methods in an enterprise bean that relies on the container to manage its persistence).
- The arguments and return values are valid RMI types.
- The methods' throws clauses include the appropriate exception classes.

The interface node looks like this in the IDE's Explorer window:  OrderHome
The default label is `bean_nameHome`.


The Local Interface

The local interface is similar to the remote interface in some respects. This type of interface contains signatures for the business methods that can be called on the bean. The methods' complete code is in the bean class. The container creates a class that implements the local interface. However, a call to a bean's local interface must come from another bean or a web component inside the same server.

The local interface extends `javax.ejb.EJBLocalObject`. A client uses this interface to locate the local home interface through a JNDI lookup call, calls a method on the local home interface to retrieve a specific instance of the bean (with the remote interface as the return type), and then calls business methods on that instance.

When you use the Sun ONE Studio IDE to create an enterprise bean with local-type interfaces, the EJB Builder's GUI support and validation help ensure that the local interface's methods follow the rules defined in the J2EE documents. Those rules include the following:

- The method signatures in the local interface have corresponding methods in the bean class.
- The methods' throws clauses include the appropriate exception classes.


The interface node looks like this in the IDE's Explorer window:  LocalReps
The default label is `Localbean_name`.

The Local Home Interface

Similar in some ways to the home interface, the enterprise bean's local home interface extends `javax.ejb.EJBLocalHome` and defines the create and finder methods that can be called on the enterprise bean by another bean within the same server. The container provides a class that implements the local home interface.

When you use the IDE to create an enterprise bean with local-type interfaces, the EJB Builder's GUI support and validation help ensure that the local home interface's methods follow the basic rules for enterprise beans. Those rules include the following:

- The method signatures in the local home interface have corresponding methods in the bean class (except in the case of finder methods in an enterprise bean that relies on the container to manage its persistence).
- The methods' throws clauses include the appropriate exception classes.

The interface node looks like this in the Explorer window:  LocalRepsHome
The default label is `Localbean_nameHome`.


The Bean Class

The bean class is the heart of the enterprise bean, containing the implementation defined in the other two classes. The bean class of an entity bean extends the `javax.ejb.EntityBean` interface, the bean class of a session bean extends `javax.ejb.SessionBean`, and the bean class of a message-driven bean extends

`javax.ejb.MessageDrivenBean`. The bean class implements the enterprise bean's finder, create, business, home, and select methods. The class also implements life-cycle methods that the container calls.

When you use the Sun ONE Studio IDE to create an enterprise bean, the EJB Builder's GUI support and validation help ensure that the bean class follows these and other basic rules for enterprise beans:

- The class is defined as public and abstract.
- The class contains a public constructor with no parameters.
- The class implements an `ejbCreate` method to match each create method defined in the home or local home interface.
- The class, if it is an entity bean that manages its own persistence, contains an `ejbFind` method to match each finder method in the home or local home interface.

A bean class node looks like this in the IDE's Explorer window:  **RepsBean**
The default label is the name of the enterprise bean plus Bean.

EJB QL

When you add a finder method or a select method to a CMP entity bean, you embed a statement in the EJB Query Language to define the method's query. In a query written in EJB QL, your bean can navigate over the relationships defined in its abstract schema, that is, the part of your bean's deployment descriptor that defines the bean's persistent fields and relationships. An EJB QL query can span the abstract schemas of all related entity beans that are packaged in the same EJB JAR file.

When your bean is deployed to an application server, your EJB QL queries are translated into the target language of the underlying data store. Thus, an entity bean that uses EJB QL is portable across different data stores.

The Deployment Descriptor

The enterprise bean's deployment descriptor states how the bean is to be deployed in the server. The deployment descriptor, which is an XML file, lists and describes the classes that compose the enterprise bean, the bean's references to other beans, settings for the environment in which the bean will operate, and how the bean should be managed at runtime. This file also lists the persistent fields of an entity bean that delegates its persistence management to the container.

When you use the Sun ONE Studio IDE to create an enterprise bean, the EJB Builder automatically creates a deployment descriptor and ensures that it follows the J2EE standard. (Because you normally work through the enterprise bean's property sheets

rather than manipulating the deployment descriptor directly, the descriptor file does not appear in the IDE's Explorer window. However, you can open the descriptor file through the Explorer.)

The Work Flow of an EJB Application at Runtime

At runtime, the application client communicates first with the enterprise bean's home interface and then with the remote interface, but never directly with the enterprise bean object. All work is done for the client through the EJB container.

At runtime, these application elements interact as shown in FIGURE 1-4. The figure's numbered steps are explained next. (Notice that this is a generic view of the work flow. The example uses only one enterprise bean with remote-type interfaces. Some steps, for example, instance pooling, do not apply to certain types of enterprise beans.)

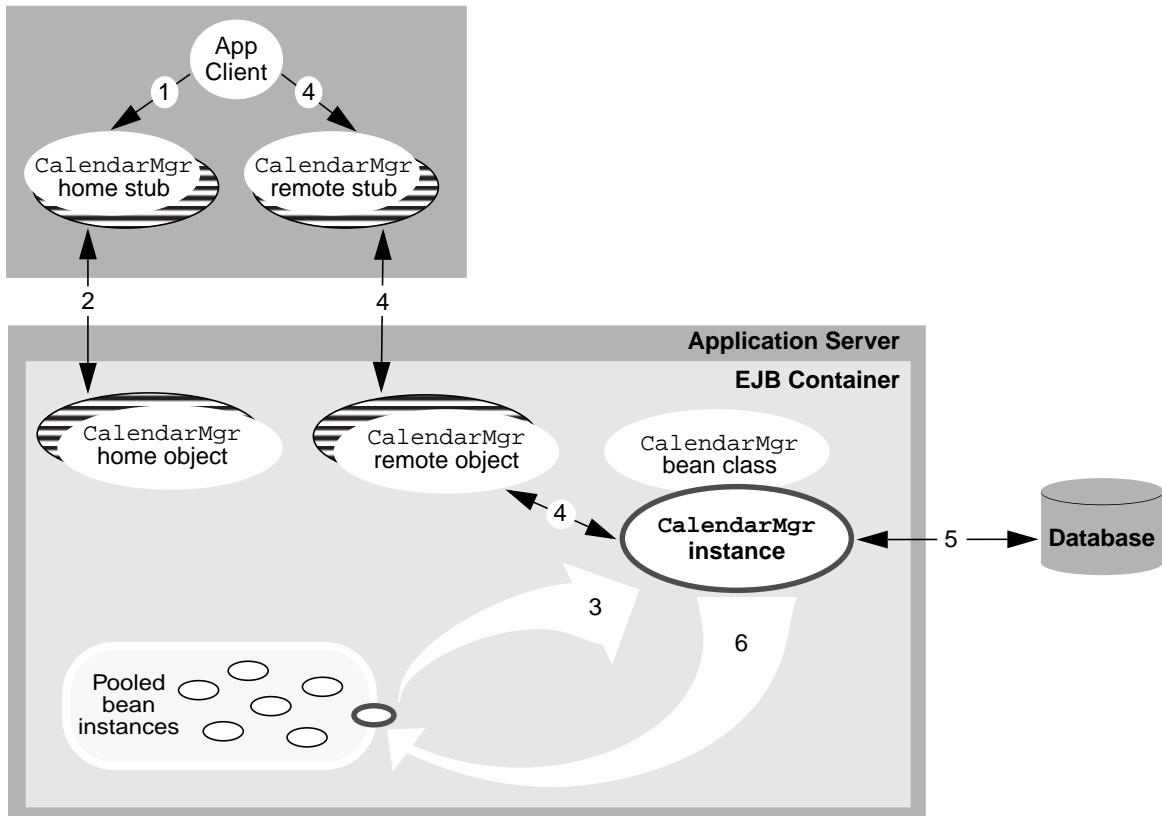


FIGURE 1-4 Work Flow Inside the Application at Runtime

1. The client finds the enterprise bean in the application server and container. (That is, the client uses a JNDI lookup method to get a remote reference to the enterprise bean's home interface.) A corresponding home stub is created in the client.
2. A home object is created on the server side to implement the bean's home interface. The home stub asks the bean's home object (which acts as a factory) to have an instance of the enterprise bean created for the use of this client in this session.
3. The container takes a bean instance from the pool.
4. A remote object is created on the server side to implement the bean's remote interface. The client works through its remote stub and the remote object to call business methods on the bean instance.
5. Data is read from the database into the bean instance and transmitted to the client. Any updates are written to the database in transactions.
6. The client has received the results it asked for, and the container returns the instance to the pool.

Notice how this architecture supports multiple concurrent users without multithreaded programming. Because enterprise bean users get their own instances of the bean from the pool, the developer can write simple, single-threaded code.

An Enterprise Bean's Development Life Cycle

As depicted in FIGURE 1-5, an enterprise bean goes through several steps after you create it, but before it is ready for use.

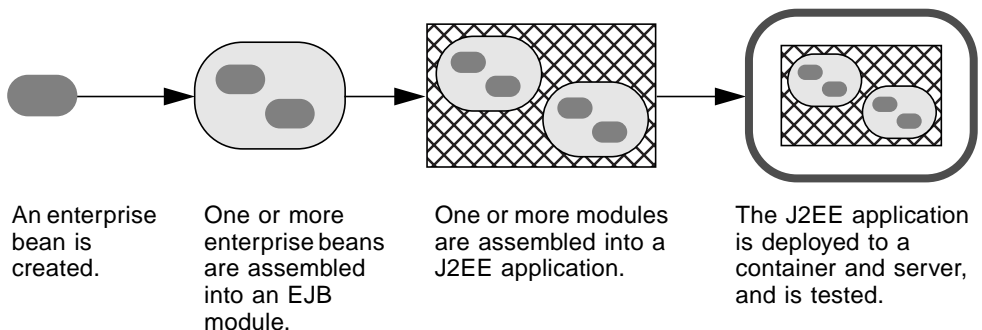


FIGURE 1-5 Development, Assembly, and Deployment of an Enterprise Bean

As an EJB developer using the Sun ONE Studio IDE, you follow these basic steps to create an enterprise bean and prepare it for assembly and deployment:

1. Use the EJB Builder wizard and other GUI support (as described in Chapters 3 through 7) to generate the enterprise bean's classes.
 2. Use the IDE's Source Editor and GUI support to code the enterprise bean. If you can let the EJB container manage your session bean's transactions and your entity bean's persistence, you have much less code to write.
 3. Use the IDE to package the enterprise bean, along with any other related enterprise beans, in an EJB module. Use the EJB's property sheets to add your beans' external dependencies to the deployment descriptor.
 4. Use the IDE's EJB test application as described in Chapter 9 to create an EJB web test client for the bean and to run tests. To prepare for this step, you perform some assembly and deployment steps. Notice that the bean can be deployed later in a production environment to a different server.
-

The IDE's Support for Enterprise Beans

The Sun ONE Studio IDE invisibly takes care of, or automates, many tasks that you would be obliged to do if you developed your enterprise beans by hand. Here are a few of the tasks you *do not have to do* when you use the IDE:

- Write method declarations for the basic classes. The IDE generates the necessary classes for each bean and the method declarations within those classes.
- Provide code to manage persistence. The application server takes care of those tasks for you when you create CMP beans.
- Provide code to manage transactions. The application server takes care of those tasks for you when you create CMP beans or when you choose to let the container manage transactions for your session beans and message-driven beans.
- Keep your bean classes, interfaces, and methods in synch. The IDE maintains consistency for you.
- Write XML code for the deployment descriptor. The IDE generates this file.
- Manually create a test client to test the enterprise bean. The IDE provides comprehensive, GUI-based support for testing session and entity beans.
- Search the J2EE documentation. The enterprise bean source code generated by the IDE conforms to J2EE standards. The code automatically includes comments and references to related documentation. The IDE also provides the following:
 - Code completion for the application programming interface (API) encompassed by the J2EE standard. (Press Ctrl-Space when you're editing code.)
 - Convenient access to the pertinent Javadoc™ documentation. (Press Shift-F1 on the selected class or interface name.)

Developing Enterprise Beans in the IDE

You use the EJB Builder Wizard to generate the infrastructure of your enterprise bean. The wizard is tailored to the type of bean you've chosen: a session, entity, or message-driven bean, with options for the source of the bean's persistent data and for management of the bean's transactions and persistence. The wizard leads you through the steps of creating all the basic components.

FIGURE 1-6 shows the elements of a typical enterprise bean that the IDE generates for you, and how the elements appear in the IDE's Explorer window. This figure uses the example of a session bean with remote-type interfaces.

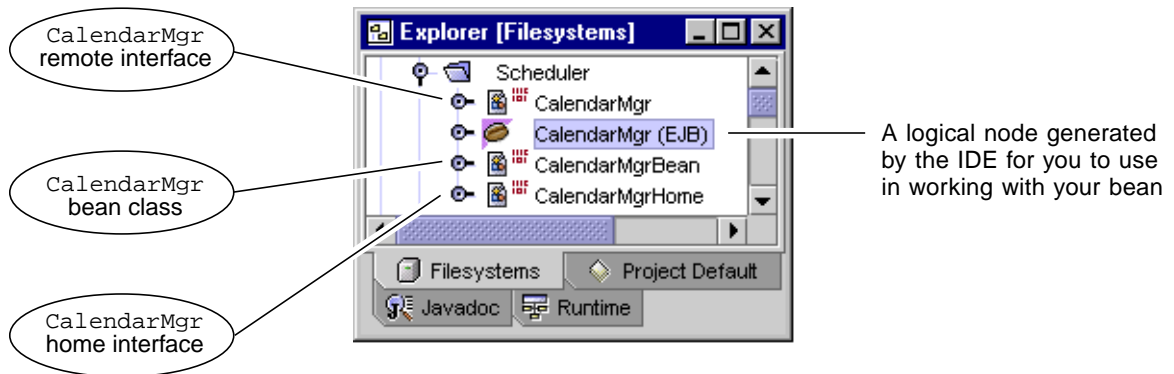


FIGURE 1-6 Generated Elements of an Enterprise Bean Shown in the Explorer Window

After using the wizard to generate these basic elements, you use the EJB Builder's other GUI features to add methods to your bean, and you use the Source Editor to finish coding the bean.

Creating Sets of Related CMP Entity Beans

The EJB Builder Wizard allows you, in one process, to generate the infrastructure of an entire set of CMP entity beans along with an EJB module to house them. This feature is particularly useful if the database tables that your beans represent are related by foreign keys. In the entity bean set, these foreign keys are preserved as container-managed relationships.

Providing Transactions

In the enterprise bean model, transactional behavior is designed to be handled both implicitly and declaratively. When a method is invoked on a bean instance, the EJB container intervenes and manages the transaction for you. You don't have to be expert in writing transactions; you don't have to write or debug code that controls transaction boundaries. By making a simple selection in the EJB Builder wizard, you can declare your bean's transactional attributes, and you can fine-tune those attributes later using the property sheets of the bean's EJB module.

Sometimes, however, you might need to program transactions explicitly in a session bean. The IDE lets you declaratively override the container and supports the use of the JDBC API and the Java Transaction API (JTA) to manage your beans' transactional behavior.

Providing Persistence

As with transactions, the IDE allows you to delegate your beans' persistence handling entirely to the EJB container, or to code persistence yourself. If you handle it yourself, you can write JDBC code. If you want container-managed persistence, you start by making a few selections in the EJB Builder wizard, and finish by making a few declarations in a property sheet to enable the container to find your underlying data store.

Providing Security

If you want only users in certain roles to call certain methods in your enterprise bean, you can add programmatic security to your bean. However, you don't have to write full security routines in your bean's source code. A security reference in your bean code matches a security role that you declare for a method. To make that match, you simply change a field in the bean's property sheet, and the security information is added to the bean's deployment descriptor.

When a client tries to call that a secured method on your bean, the EJB container compares the user's role with your access control list (the user roles that you have authorized to execute the bean's methods), and permits or refuses execution.

Creating Application Clients

In addition to developing the enterprise beans that make up the EJB tier of your application, you can use the IDE to create an application client. In this context, an application client is a stand-alone Java program that starts with its own main method, runs in a J2EE client container, and interacts with other J2EE application components including EJB modules. For details on the design and development of clients, refer to *Building J2EE Applications*.

Further Reading

For details on the design of enterprise beans and the EJB tier, refer to *Enterprise JavaBeans Specification*, version 2.0 at:

<http://java.sun.com/products/ejb/docs.html>

Other information sources are listed in “Before You Read This Book” on page xviii.

Design and Programming Issues

If you're not already familiar with the design and programming issues associated with enterprise beans, you need to consider the differences between various kinds of beans and what they are meant to do. You should be aware of the life cycle of each kind of bean, how methods and exceptions are applied, and how beans are set up for reuse in different application environments. You need to understand how persistence, transactions, and security are handled. This chapter discusses those topics, and ends with a list of recommended readings for further details.

Deciding Which Type of Bean You Need

The *Enterprise JavaBeans Specification*, version 2.0, defines three types of enterprise beans: session beans, entity beans, and message-driven beans. There are also several types of session and entity beans, each with built-in functionality for different purposes. The EJB Builder in the Sun ONE Studio IDE guides you and streamlines the process of creating all these types of enterprise beans.

To help you make design decisions before you start, this chapter describes the enterprise bean types.

FIGURE 2-1 shows the basic choices before you when you use the IDE's template to create enterprise beans.

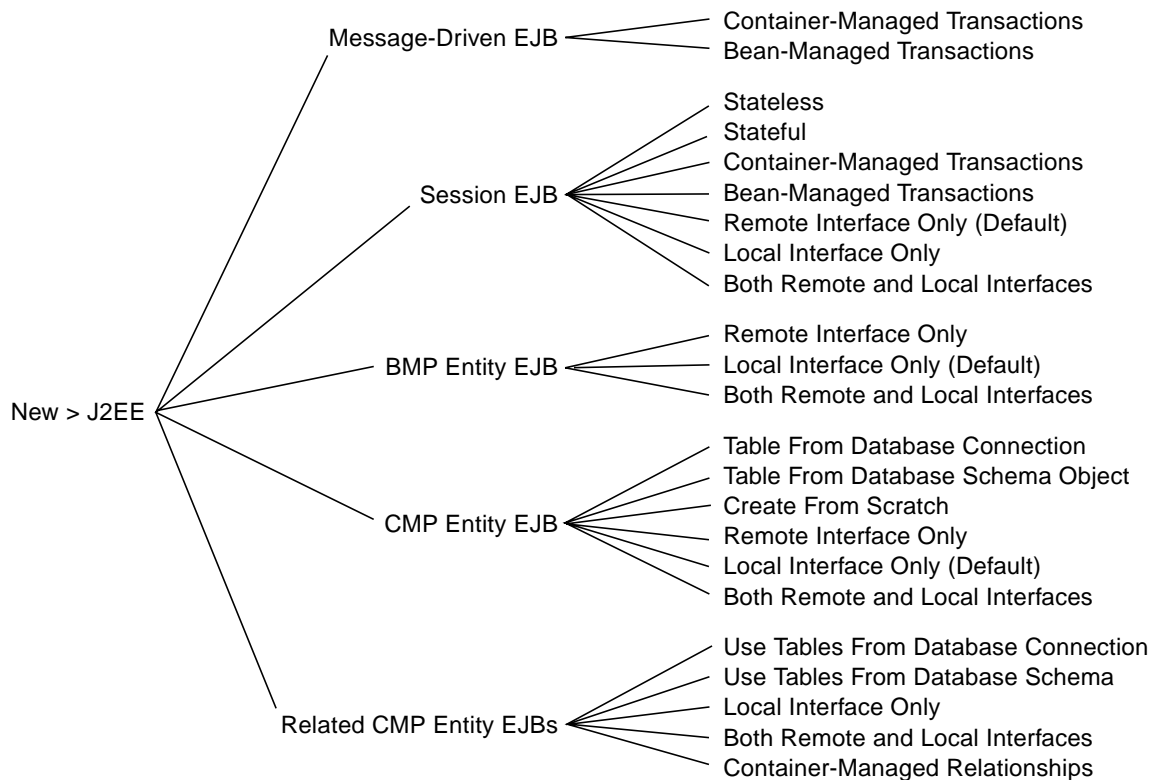


FIGURE 2-1 Basic Choices About Enterprise Beans in the Sun ONE Studio IDE

Understanding Session Beans

A session bean acts as the traffic director for an application, controlling the work flow of the application and encapsulating its business processes. If you think of a model-view-controller architecture, a session bean is like the controller tier, but in an EJB application. On behalf of a client, a session bean can do work such as accessing a database or calculating balances. A session bean doesn't represent database data directly, but it can access the database or manipulate entity beans that access the database.

In the context of an application that uses enterprise beans, a session bean manages the conversation between a single client and the parts of the application that reside on the EJB server and are managed by the EJB container. These other application parts often include entity beans and (in its own separate tier) the database with which any persistence-capable beans interact.

A session bean can manipulate one or more entity beans, control interactions between them, and bridge gaps between the data represented by the entity beans and the business logic that acts on the data. A single session bean can direct transactional work by several entity beans in the same application.

The conversation (or session) that the session bean manages is transient and so is any data in the session bean. When the client-server session is over, or when the client or the server shuts down, the instance of the session bean that the client created for that session is destroyed. However, the client can store a handle to the session, shut down, and then resume the session later.

A session bean has no primary key. Unlike an entity bean, a session bean is intended for use by only one client at a time. Therefore the session bean can appear anonymous to the client; the bean doesn't need the unique identity that a primary key provides.

Occasionally, a session bean represents an entity, as does the `ShoppingCart` object in an application for ordering merchandise online. However, most session beans are not intended to save entity state to a database. For example, while a user is shopping, the `ShoppingCart` bean instance temporarily holds items that the user intends to buy. If the server goes down before the user has actually committed to buying the items, it would be inappropriate to save those items to the database in a transaction. A common design approach is to let that data go and have the user start a new shopping cart with the next session.

Deciding When to Use a Stateless Session Bean

The conversation carried on between the client and the session bean can be short and simple, involving only work that can be accomplished by loading the parameters of one method. Or a session bean can manage a long, complicated conversation that involves many methods and database transactions. Such a conversation requires the session bean to retain information between method invocations.

In the first situation, a session that consists of a request and a response, a stateless session bean is best. A stateless bean retains no state between method calls. Such a lightweight bean costs the application few resources, is easy for the container to manage, and promotes faster processing. A stateless bean can provide better scalability to an application that has many clients.

Of course, the tradeoff is that this bean can do less with data. A stateless bean operates only on the arguments that the client passes to it. Every call to a method of a stateless bean is independent of previous calls.

For example, a stateless bean can get the ZIP code for an address. Each retrieval can be completed in one method call, `getZip`, because all the information needed to process the retrieval is in the method parameters. Any transactions are completed within the method call and within the container. (Transactions are discussed a little later in this chapter and in Chapter 3.)

The instance variables of a stateless bean can contain state only while the method executes. All instances of a stateless bean are the same when they are pooled. As a result, the EJB container can pool and assign bean instances very flexibly, swapping instances in and out between the client's method calls. In effect, clients can share stateless beans. These beans seem anonymous to their clients.

A session bean can be stateless if it is designed to be used sequentially by many different clients and needs no tailoring to suit a specific client. A stateless bean holds no state information for a specific client. However, the bean can have state that isn't specific to a client, for example, an open database connection.

Deciding When to Use a Stateful Session Bean

The conversation between the client and the session bean can be complicated. The bean might need more than one method to encapsulate business logic and the application might need the session bean to remember state between and across method calls. By definition, such a bean is stateful. If your client is an interactive application, or if the session bean's state must be initialized when it is created, use a stateful bean.

The bean's state can be written to a database if necessary. The state is specific to the client and is held in memory during the session, but is not persistent. If a stateful session bean must be removed from memory, the EJB container manages the state. The state of the bean instance can survive a session, but not the client's termination or a server crash.

Notice that the `ejbRemove` method is not called after a container crash, a timeout while the instance is passivated, or a system exception thrown by a method. You might need to provide a clean-up program for such an event.

A stateful bean is not shared by more than one client. By servicing only one client, the bean can maintain conversational state throughout the session. Stateful bean instances are not pooled.

The online shopping cart mentioned earlier in this chapter is an example of how stateful session beans can be used. Just as the logical business transaction of shopping includes multiple individual decisions by the user, so the stateful bean in this application includes multiple method calls. The `ShoppingCart` bean must accumulate items that the user has chosen until the user is ready to review the list of items, approve or reject each one, and place the order.

Selecting a Transaction Mode

Whether you're programming a stateless or a stateful session bean, you must make one of these selections in the EJB Builder wizard:

- **Container-Managed Transactions.** The bean's transactions are managed by the EJB container, and you don't intend to provide code to manage transactions. The result is referred to as a CMT session bean.
- **Bean-Managed Transactions.** The bean manages its own transactions, and you intend to explicitly demarcate each transaction as you code the bean's methods. The result is referred to as a BMT session bean.

For a CMT session bean, you have less coding to do and all the transactions are handled in a predictable, consistent way. Also, the transaction policy that you select for your bean can be changed declaratively. The tradeoff is that each method can be associated with no more than one transaction. The container typically has a transaction begin just before a method starts, and commits the transaction just before the method exits. Nested and multiple transactions are not allowed in a single method.

Assigning Transaction Attributes

If you decide to let the EJB container manage your bean's transactions, the container looks for transaction attributes on your bean or on specific methods within your bean. A transaction attribute specifies the scope of a transaction: which methods it includes and how the results of those methods are treated in relation to the transaction. These attributes are assigned as follows.

- **CMT session beans.** The IDE automatically assigns the Required transaction attribute to every CMT session bean, and that transaction attribute applies to every business method in your bean. However, you can manually assign a transaction attribute to a particular method or set an overriding transaction attribute for the bean. (You set transaction attributes for a CMT bean at the EJB module level.)
- **Entity beans.** The same is true for entity beans, all of which have container-managed transactions.

You don't set transaction attributes for a BMT session bean. All of its transaction boundaries must be explicitly demarcated in the bean class.

Using JTA or JDBC

To code bean-managed transactions explicitly, you can use the Java Transaction API (the `javax.transaction.UserTransaction` interface, or JTA) or the JDBC API.

- **JTA.** If you're using the Sun ONE Studio IDE to create new BMT session beans, consider using JTA. It can be more powerful and flexible than the JDBC API.
- **The JDBC API.** On the other hand, if you're wrapping legacy code inside a session bean, and that code uses JDBC technology or encapsulates SQL code, it's a good idea to use the JDBC API.

JTA can include transactions for other resources such as the JDBC API. When you use JTA to code transactions in an enterprise bean, you're using the JDBC API for database connections and JTA for transactions.

In handling transactions, your bean's method calls the JTA methods, which then call the lower-level routines of the Java Transaction Service (JTS), the transaction manager used by the Java 2 Platform, Enterprise Edition (J2EE). Because of that level of indirection, JTA lets you demarcate transactions independently of the transaction-manager implementation. A JTA transaction can also span updates to multiple databases from different vendors.

A JDBC transaction is controlled by the transaction manager of the database you're using.

A disadvantage of using JTA is that it doesn't support nested transactions. One transaction must end before another starts.

For more information on transactions, refer to *Building J2EE Applications*.

Understanding the Life Cycle of a Session Bean

At runtime, the application server creates bean instances as requested by EJB clients. A bean instance passes through several stages of activity managed by the EJB container. When the instance is no longer needed, it is destroyed.

The individual stages in a session bean's life, the methods that cause the bean to transition between stages, and the programmer's responsibilities are described next.

Creating and Initializing a Bean Instance

A session bean's runtime life cycle starts when an EJB client requests some work from the bean. This stage of the life cycle goes as follows:

The client calls a create method on the bean's home (or local home) interface. In response, the container calls these three methods in sequence:

1. `newInstance` to create a new instance of the session bean
2. `setSessionContext` to associate the instance with a session-context object
3. `ejbCreate` to initialize the instance

Note – The IDE generates method signatures for the `setSessionContext` and `ejbCreate` methods. It's up to the programmer to complete the body of the methods.

The client receives a reference to the bean instance's remote object.

Executing Business Logic

Now that a bean instance has been created and initialized, the EJB client asks the instance to do some work. This stage of the life cycle goes as follows:

The client calls a business method on the bean's remote object. In response, the container does the following:

- Checks security permissions to make sure that the requestor is entitled to execute that business method
- Applies the transaction control specified by the method's transaction attribute
- Calls a business method on the instance

The client receives the result of the business method.

Note – The programmer can specify security control either programmatically, within the bean code, or declaratively, using the property inspector of the EJB module. The programmer uses the EJB module's property sheet to set transaction attributes for a bean's methods.

Removing the Bean Instance

The client is finished with the session and can let go of the bean instance. This stage of the life cycle goes as follows:

The client calls the `remove` method on either the home (or local home) interface or the remote (or local) interface. In response, the container calls the `ejbRemove` method to close any open resources that the instance has used. The container removes the instance from memory.

Note – The IDE generates the method signature for the `ejbRemove` method. It's up to the programmer to complete the method's code.

Pooling Stateless Instances

Ordinarily, in a production environment, many clients concurrently request work from an enterprise bean. To support this need, the container can concurrently create many instances of a *stateless* session bean and can pool them for use. The container can populate the instance pool at its own discretion.

An instance of a stateless session bean maintains no client-related state information between method calls. Therefore, stateless session bean instances in a pool are interchangeable. The container can call different session beans from the pool to handle requests from a single client.

So that the container always has an adequate supply of stateless session bean instances to serve the volume and frequency of client requests, it keeps adjusting the volume of pooled instances. For example, the container creates new instances of a stateless session bean when the number of client requests increases, and removes instances when memory becomes scarce. To maintain its pool, the container calls the stateless session bean's `ejbCreate` and `ejbRemove` methods at its own discretion.

Passivating Stateful Instances

A *stateful* session bean must maintain its conversational state with the client throughout the client's session. Therefore, the EJB container does not pool instances of stateful session beans. Instead, the container only creates and removes stateful bean instances upon explicit instructions from the client.

However, to control the use of resources, the container might still need to control the number of active stateful session bean instances at a given time. When memory becomes scarce, the container can passivate an instance, writing the instance's conversational state to secondary storage so that it can be used to handle another client's session. On passivation, the container first calls the instance's

`ejbPassivate` method, which the programmer codes to release resources and put all fields in a serializable state. The container then writes the instance's non-transient fields to secondary storage.

When a client calls a business method on a stateful bean instance that has been passivated, the container restores the instance's state from secondary storage and calls the `ejbActivate` method on the instance. The programmer codes this method to acquire resources that were released by the `ejbPassivate` method, and to restore the values of fields that were not serializable.

Note – The IDE generates the method signatures for the `ejbPassivate` and `ejbActivate` methods in both stateless and stateful session beans. It's up to the programmer to complete the code for these methods.

Synchronizing State in a Session

A programmer can choose to implement the session-synchronization interface in a stateful CMT session bean. During the stateful bean's life cycle and at certain points in a transaction, the container uses the interface to notify the instance that it is about to enter or complete a transaction. The programmer can program the methods on this interface to synchronize the bean's instance variables with the data store's most current data, or to abort the transaction. The interface includes three methods: `afterBegin`, `beforeCompletion`, and `afterCompletion`.

Note – The IDE generates the method signatures for the session-synchronization methods. It's up to the programmer to complete the code for these methods.

Understanding Entity Beans

An entity bean represents persistent data in an underlying data store. This type of bean provides an object view of a set of data such as rows in a database table. Each entity bean instance contains one entity of that data and can also contain business logic that is intrinsic to the entity. A client, or a session bean working on behalf of a client, can use an entity bean to find or insert data in a database.

An entity bean's state isn't dependent on its environment. With its primary key and its remote reference, the bean can survive a crash of the server, the EJB container, or the client. The entity's state automatically reverts to the way it was after the last committed transaction.

Because each client gets its own instance of the entity bean, many different users can share access to one set of data. If two clients execute the same finder method on an entity bean, they both reference the same remote object. Each find is independent, which eliminates contention problems. Multithreaded code is not needed in an enterprise bean. (However, there might be a situation in which you need to run concurrent processes. With message-driven beans, you can approximate multithreading in a J2EE application. That discussion can be found in “Understanding Message-Driven Beans” on page 36.)

A client finds a particular entity bean by its unique object identifier, which is the bean’s primary key.

Taking Advantage of the EJB Container’s Services

All transactions that are part of an entity bean are automatically managed by the EJB container. When you have finished coding a bean and have created an EJB module for it, you use the module’s property sheets to declare the bean’s transaction attributes. The container demarcates the bean’s transaction boundaries accordingly. The IDE automatically assigns the default transaction attribute to all business, create, remove, finder, select, and home methods in your entity bean.

As an EJB programmer, you can choose to have your entity bean’s persistence managed by the container, or you can code the bean yourself to manage its relationship with the data store.

When you use the IDE to create an entity bean with container-managed persistence, that is, a CMP entity bean, you complete the bean class without writing JDBC calls to the data store. The container provides the code to synchronize your bean’s instance variables with the data store. You provide information to the container on how to map the instance variables to columns in database tables. You also use the EJB query language (EJB QL) to define how servers will implement your bean’s query methods.

An EJB QL query in a finder method can be used by a client to select an existing entity object. Or, without exposing the result to the client, an EJB QL query in a select method can select objects or values related to the state of an entity bean. To find this kind of information, the EJB QL query can use the bean’s abstract persistence schema, which defines the bean’s persistent fields and relationships and is part of the bean’s deployment descriptor.

Suppose you are deploying your application to the J2EE reference-implementation server. The server looks at your bean’s methods and the EJB QL queries you have supplied, and generates its own server-specific SQL statements to do this mapping.

In some cases, you might want or need to adjust the SQL that a given server plugin generates for its own use, assuming that the server allows access to its generated SQL. For example, if you are using the J2EE reference implementation (RI) server

and if your application includes a CMP entity bean that was created in the EJB 1.1 environment, you must make a few small changes in the server's generated SQL for certain methods. You can also change any CMP entity bean's mapping rules by adjusting the generated SQL.

Relationships between entity beans can be managed by the EJB container. If you generate a set of related CMP entity beans from a database in which tables use foreign keys, the IDE automatically preserves these relationships.

Put simply, the advantages of using container-managed persistence are that you have less coding to do and that the resulting entity bean is not dependent on any particular data store.

At some point, you might need to create an EJB application in which you wrap legacy code that isn't supported by mapping tools. Or, you might need to implement complex joins between tables, or even between different databases (for example, non-relational databases). In these situations, depending on the capabilities of the application server you use to deploy your EJB application, you might need to choose bean-managed persistence and code all database calls yourself in the entity bean class. If the server supports the persistence style you need, then container-managed persistence is the best approach. However, as a general rule, bean-managed persistence affords more flexibility in how an entity's state is managed.

Understanding the Life Cycle of an Entity Bean

The application server creates a pool of entity bean instances to be used by EJB clients. At runtime, a bean instance passes through several stages of activity as requested by the bean's client and as managed by the EJB container. When the instance is no longer needed, it is destroyed.

The individual stages in an entity bean's life, the methods that cause the bean to transition between stages, and the programmer's responsibilities are described next.

Creating and Managing a Pool of Bean Instances

An entity bean's runtime life cycle starts when the container creates and pools instances of the bean.

Many EJB clients might concurrently need many entity beans to do work for them. At its own discretion, the container creates and pools multiple, anonymous instances of a bean before they are needed. These instances can be used to run queries with finder methods, or they can be assigned identities. When a particular instance is needed to hold data from the data store, the container transitions a pooled instance into the ready state. (A ready instance has a primary key that uniquely identifies it.) Finally, the container can adjust the size of the pool by constructing new instances or deleting unneeded ones.

To create a new instance for the pool, the container calls:

1. The `newInstance` method to create a new instance of the entity bean
2. The `setEntityContext` method to associate the instance with an entity-context object

The instance is now in the pooled state.

The container cycles instances between the ready state and the pooled state. When the client requests an entity using its identity, but the corresponding instance is not in the ready pool, the container transitions an instance from the pooled state to the ready state. As part of this process, the container calls the `ejbActivate` method on the instance. The programmer can code this method to acquire resources that are needed by instances with identity, but not by instances in the pooled state. The container then loads the values of the entity's instance variables and associates the instance with its remote object.

The instance is now ready.

Notice that the `ejbActivate` method does not load the values of the entity's instance variables. For BMP entity beans, this is handled by the `ejbLoad` method; for CMP entity beans, it's handled by the container.

When the container has too many instances in the ready state, it can passivate one or more instances, moving them into the pooled state. As part of this process, the container calls the `ejbPassivate` method on the instance. The programmer can code this method to release resources that are not needed by instances in the pooled state. The container also dissociates the instance from its remote object, and stores the current values of the entity's instance variables in the database.

Again, the `ejbPassivate` method does not store the values of the entity's instance variables in the database. For BMP entity beans, this is handled by the `ejbStore` method; for CMP entity beans, it's handled by the container.

To remove an inactivated instance from the pool, the container calls the `unsetEntityContext` method on the instance and dissociates the instance from its entity-context object. The container then destroys the instance.

Note – The IDE generates method signatures for the `setEntityContext`, `unsetEntityContext`, `ejbActivate`, and `ejbPassivate` methods. It's up to the programmer to complete the methods when specific entity contexts or resources are needed.

Using a Bean Instance to Create a New Entity

Whenever an EJB client wants to create a new entity (to insert data into the data store), the client calls a create method on the bean's home interface. In response, the container:

1. Does the appropriate security checking and applies the transaction control specified by the method's transaction attribute.
2. Calls the `ejbCreate` method on an instance in the pool. In a CMP entity bean, this method initializes the persistent field values to prepare for the container to populate the data store. In a BMP entity bean, this method initializes field values and inserts the record into the database.
3. Creates a remote object for the bean, and associates it with the new bean instance.
4. Calls the `ejbPostCreate` method on the instance to complete initialization. Because the container has already assigned an identity to the bean instance, the `ejbPostCreate` method can forward identity information, such as the associated remote (or local) interface or primary key, to another enterprise bean.

The client receives a reference to the instance's remote object. The instance is now in the ready state and can run business methods for the client. See "Executing Business Logic" on page 34.

Note – The IDE generates method signatures for `ejbCreate` and `ejbPostCreate`. It's up to the programmer to complete those methods. The programmer must also specify the security control and transaction attributes to be applied by the container.

Locating an Existing Bean Instance

An EJB client can locate one or more existing entities by calling a finder method on the bean instance's home object. A finder method returns one or more entities that meet specific search criteria. Besides the `findByPrimaryKey` method, an entity bean can have any number of other finder methods.

When a client calls a finder method on an instance's home object, the following steps happen:

1. The container does the appropriate security checking and applies the transaction control specified by the method's transaction attribute.
2. The container calls a finder method on an anonymous instance in the pool.
3. The finder method returns the primary key of the instance (or multiple keys of multiple instances, if appropriate). Notice that only the primary key is returned.

4. The container locates or creates a remote object with each primary key and returns a reference to the object to the client.

Note – The IDE generates a method signature for the `findByPrimaryKey` method. It's up to the programmer to furnish any other finder methods that a particular bean might need.

The client can go on to call business methods on the located instance, using the methods named in the remote object. See “Executing Business Logic” on page 34.

Executing Business Logic

When an EJB client needs an entity-bean instance to do some work, the client calls a business method on the instance's remote object. In response, the container:

1. Does the appropriate security checking and applies the transaction control specified by the method's transaction attribute.
2. Calls a business method on the instance.

The business method finishes and the client receives the result. When appropriate, the container passivates the instance as discussed in “Creating and Managing a Pool of Bean Instances” on page 31.

Note – The IDE provides support for creating business-method signatures on both the remote (or local) interface and the bean class. It's up to the programmer to finish coding the business method in the bean class.

Using a Bean Instance to Remove an Existing Entity

Whenever an EJB client wants to remove an existing entity (to delete data from the data store), the client calls a remove method on the instance's home or remote object. In response, the container:

1. Does the appropriate security checking and applies the transaction control specified by the method's transaction attribute.
2. Calls the `ejbRemove` method on the instance. A CMP entity bean instance responds by readying the data for the container to delete. A BMP entity bean instance responds by deleting the data.
3. Commits the transaction as appropriate.

Note – The IDE generates a method signature for the `ejbRemove` method. It's up to the programmer to complete the method.

Synchronizing an Instance With the Data Store

At certain points during a transaction, the container must make sure that the data in the bean instance is synchronized with the data in the data store. To do this, the container:

- Calls the `ejbLoad` method on the instance when the entity enters an active transaction.
 - In a CMP entity bean, this method is called after the container has read the entity object's state from the data store into the bean's container-managed fields. The programmer can use this method to perform some computation on the values of the fields that were read by the container.
 - In a BMP entity bean, this method usually reads the data from the underlying data store and assigns the values to the bean's instance variables.
- Calls the `ejbStore` method on the instance when the transaction is committed or when the instance is passivated.
 - In a CMP entity bean, the container calls this method first, before writing the container-managed fields to the data store. The programmer can use this method to prepare the container-managed fields before they are written to the data store.
 - In a BMP entity bean, this method writes the values in its instance variables to the underlying data store.

Note – The IDE generates method signatures for the `ejbLoad` and `ejbStore` methods. In a BMP entity bean, it's up to the programmer to complete those methods. In a CMP entity bean, those methods typically require no further coding, because the container manages synchronization with the data store.

Understanding Sets of Related CMP Entity Beans and Container-managed Relationships

If you wish, you can use the EJB Builder wizard that generates the infrastructure of one CMP entity bean at a time. However, if you want to base several CMP entity beans on database tables that have foreign keys or table-to-table joins, it's easier and more reliable to generate the infrastructure for the whole group of beans at one time. A special EJB Builder wizard displays the tables in a database or schema, and, from

the tables you select, generates a corresponding set of CMP entity beans. Along with the beans, the wizard creates logical entities to model foreign keys and joins between database tables, and generates an EJB module to store and track the set of beans and relationships.

A CMP entity bean that you created as part of a set is no different from a CMP entity bean that you created individually. Its function, capacities, properties, and life cycle are the same. However, if you use the wizard to generate a set of related CMP entity beans, you don't have to hand-code information about the enterprise-bean equivalent of joins and foreign keys. The IDE presents these links as logical fields called container-managed relationship (CMR) fields. A CMR field is like a foreign key. In an EJB QL query, you can perform the equivalent of a table-to-table join using a CMR field instead of a CMP field.

In accordance with the *Enterprise JavaBeans Specification*, the EJB container manages CMRs to ensure referential integrity between associated CMP entity beans. The IDE uses the Collections API to let you manipulate your beans' CMRs. Information about CMRs is stored at the level of the EJB module in which a set of related beans resides.

In the bean class are abstract accessor methods that specify a CMR's directionality and cardinality. For example, in a relationship between the beans `Order` and `LineItems`:

- The `Order` bean has the methods `getLineItems` and `setLineItems`. These methods give the `Order` bean access to the collection that represents an order's line items.
- The `LineItems` bean has the methods `getOrder` and `setOrder`. These methods give the `Order` bean access to the order to which the line items belong.

A CMR allows for cascade-delete functionality, which is specified declaratively and stored in the deployment descriptor.

A CMR field provides access to local instances of a CMP entity bean; thus, only a bean with local-type interfaces can have CMR fields.

Understanding Message-Driven Beans

A special kind of enterprise bean acts as a go-between for application components, taking messages from the client and acting on the messages to start processes asynchronously. This is the message-driven bean, which combines many features of enterprise beans with the ability to be a listener for the Java Message Service (JMS) message-oriented middleware (MOM). With message-driven beans, you can approximate threading or parallel processing in an EJB environment.

Whereas in another J2EE application an enterprise bean might respond to RMI calls, a message-driven bean listens to certain resources for messages arriving from other application components, usually the client. When such a message arrives, regardless

of what processes or servers are running at the time, the message-driven bean is notified of the message receipt by the invocation of the `onMessage` method. The message-driven bean then acts on the message, calling a stateless session bean to start a process.

Using Message Sources (Destinations)

A destination is a resource to which a client sends messages and to which a message-driven bean listens. A destination can be a queue or a topic.

- **Queue.** A message queue uses the point-to-point or “pull” model (analogous to email from a sender to a receiver). The client sends messages to a queue object. A message-driven bean polls the queue periodically and consumes messages meant for that bean. One message is sent to one consumer.
- **Topic.** A message topic uses the publish-and-subscribe or “push” model (analogous to an online news subscription). The client sends a message to a topic object. All consumers who subscribe to that topic receive a copy of the message. One message can be broadcast to many consumers.

A topic subscription can be durable or non-durable.

- **Durable.** Messages are saved for the consumer, which can retrieve them the next time it connects to the system.
- **Non-durable.** Messages are available to the consumer only when it is connected, and old messages aren’t saved.

Deciding When to Use a Message-Driven Bean

An application using message-driven beans has minimal dependence on the state of other application components. A message-driven bean is designed for one-way operation.

As long as a destination is available, an application client can reliably send a message to it, whether or not the message-driven bean’s server or the target application are currently deployed. The container doesn’t have to wait for a client-invoked process to complete. The client can even be decoupled from the server while the message-driven bean and its called bean do their work. One or many clients can send messages to one or many servers, invoking multiple processes.

If an application needs to start a process that might go on a long time, if a server might go down, or if, for any other reason, resources might become unavailable before the message arrives, you can use a middle layer of message-driven beans to keep processing going. A message-driven bean is ideal if your client needs to start a process and then continue to be available to the user. For example, in a shopping application, you could use a message-driven bean to check the customer’s credit

card number for validity while the customer continues to browse the product list. The client component of the application sends a message to the message-driven bean and then continues processing.

The use of message-driven beans can help your application with load balancing and scheduling. For example, you can start processes at your database's off-peak times. Asynchronous processing is particularly advantageous for communications and processing over different time zones and geographically dispersed systems.

If your application needs to interface with another application it doesn't know much about, you can use message-driven beans to loosely couple the applications. Many legacy systems use messaging and can be interfaced in this way with J2EE applications.

A message-driven bean interacts with the JMS environment only when the bean's `onMessage` method is called. The message-driven beans that you generate using IDE's EJB Builder wizard incorporate JMS transparently, so that you don't have to write JMS code. Because you specify the JMS connections and the message channels (destinations) as properties on the bean, you can easily change a single message-driven bean to point toward a different destination if needed.

Deciding When Another Bean Type Is Better

In certain situations, message-driven beans are not appropriate. For example:

- When you need to return data. A message-driven bean must be hand-coded to return anything other than `VOID`, and a specific client must be targeted. If you need to return a result, a session bean is more efficient.
- When you need confirmation that an operation has succeeded. A message-driven bean can't throw exceptions as other enterprise beans can.
- When the bean's operation is part of a transaction that must complete within a given time.
- When the server needs to know the client's security identity. Messaging doesn't propagate this identity to the message-driven bean. With this kind of bean, all instances are the same.
- When performance is an issue. Messaging becomes a middle layer between the client and the server. Even though message-driven beans are relatively lightweight, an extra layer can add time to your system response.
- When you want your application to stay small and uncomplicated. An application that doesn't need asynchronous processing can be easier to code and debug.

Understanding the Life Cycle of a Message-Driven Bean

At runtime, an application client sends messages to a destination to which a message bean is listening. When these messages arrive, the application server creates instances of the bean to service the client's requests.

This type of bean has a very simple life cycle. As with a stateless session bean, its instances pass through several stages of activity managed by the EJB container, and when an instance is no longer needed, it is destroyed.

The individual stages in a message-driven bean's life, the methods that cause the bean to transition between stages, and the EJB programmer's responsibilities are described next.

Creating and Initializing a Bean Instance

A message-driven bean's runtime life cycle starts when a client sends a message to a queue or a topic, to be consumed (read and processed) by a message-driven bean. In response, the EJB container calls these three methods in sequence:

1. `newInstance` to create new instances of the message-driven bean
2. `setMessageDrivenContext` to associate each instance with a message-driven-context object
3. `ejbCreate` to initialize the instances

Note – The IDE generates method signatures for the `setMessageDrivenContext` and `ejbCreate` methods. It's up to the programmer to complete the body of the methods as needed.

After sending the message, the client doesn't need to be involved again unless results are returned.

Invoking Another Bean to Execute Business Logic

Now that instances of the message-driven bean have been created and initialized, the instance and the container collaborate as follows:

- A bean instance consumes the message and finds out what work the client has requested
- The bean instance asks the container to create instances of the appropriate stateless session bean

- The container creates the session-bean instances and applies the transaction control specified by the transaction attribute of the message-driven bean's `onMessage` method
- The message-driven bean instance calls a business method on an instance of the stateless session bean

Eventually, if appropriate, the client can make a separate call to the session bean or another enterprise bean in the server and receive the result of the business method.

Note – The IDE generates the method signature for the `onMessage` method. It's up to the programmer to complete the body of the method. Also, the programmer uses the EJB module's property sheet to set transaction attributes for a bean's methods.

Removing the Bean Instance

When the message-driven bean has handed off its assigned task to another bean in the application, its job is done. The container calls the `ejbRemove` method on the message-driven bean instance to close any open resources that the instance has used. The container removes the instance from memory.

Note – The IDE generates the method signature for the `ejbRemove` method. It's up to the programmer to complete the method's code, if needed.

Pooling Message-Driven Bean Instances

As is true for stateless session beans, the container can concurrently create and pool many instances of a message-driven bean. The container populates the instance pool at its own discretion, creating new instances when the number of arriving messages increases and removing instances when memory becomes scarce.

An instance of a message-driven bean maintains no state information, and so message-driven bean instances in a pool are identical and interchangeable.

The J2EE specification does not guarantee that messages to multiple instances of a message-driven bean will be delivered in any particular order; therefore, the application must be able to handle out-of-order messages.

Using Enterprise Beans in Applications

The needs of your application dictate whether and how to combine message-driven, session, and entity beans. In some cases, you might get the best results by using only one kind of bean. In the case of a very simple application (for example, an application that performs only one CRUD operation), you might place a single session bean or entity bean in the EJB module. In other cases, you will want to exercise the full power and capability of several types of enterprise beans.

You can continue to increase the scope and power of an application by adding enterprise beans to an EJB module. EJB applications are highly extensible.

Here are a few possible combinations of enterprise beans and other components in applications:

- An EJB module containing one stateful session bean and several CMP entity beans. The session bean models user sessions. In each session, an instance of the session bean directs instances of the entity beans to retrieve and write data from and to a database. The EJB container handles persistence and transactions for the entity beans.
- An EJB module containing several CMP entity beans. This module interacts with a web module within the same application. The web module acts as the application client, and one or more of its components call methods on individual entity beans within in the EJB module. The entity beans interact with the database and return results through the web module components to the end user.
- An EJB module containing a message-driven bean, a session bean, and one or more entity beans. This module interacts with a web module in which a client component sends messages to a queue. The message-driven bean listens to the queue, consumes messages, and starts asynchronous processes in the session bean, which causes database work to be done by the entity beans.

These scenarios and more are discussed in detail in *Building J2EE Applications*.

Using Exceptions to Handle Problems

In the bean class, you define how your bean is to handle problems it encounters at runtime. A system-level problem (such as an unavailable database connection, a database so full that an SQL insert fails, or an object that can't be found) is expressed in a system exception that uses the `javax.ejb.EJBException` interface. The container sees an exception of this kind, wraps it in a remote exception, and passes it back to the client to be handled by a system administrator.

An application-level problem (such as an error in the business logic of an enterprise bean, or an input error), can be addressed with a predefined exception such as the `javax.ejb` package offers, or with a customized exception that the programmer writes. The container sees an exception of this kind and passes it back to the client for handling.

When you use the Sun ONE Studio IDE wizards to create an enterprise bean or its methods, the IDE includes the required exceptions in the method signatures. For example, `java.rmi.RemoteException` is included in the signature of all methods on the home and remote interfaces. As another example, `javax.ejb.CreateException` is included in the signature of all create methods.

When you create a method using the GUI support available from the IDE's Explorer window, you are also given the option of specifying application-level exceptions that should be thrown by the method. These application exceptions are automatically added to both the remote (or local) interface and the bean class.

Working With Deployment Descriptors

The basic design of enterprise beans makes them reusable in different applications and deployable in different servers. Toward that end, all the information that a particular server needs to know at runtime is captured in an XML meta-file called the deployment descriptor. This descriptor file includes information about the bean's structure, its relationships to other beans, where its data store is, what is needed for the user to gain access to the data store, and all other external dependencies.

Whenever you create an enterprise bean, the IDE generates a starter deployment descriptor for the bean. You use the bean's property sheets to declare whatever you know of the bean's external dependencies. When you assemble beans into an EJB module, the IDE gives you the opportunity to override the default values of bean properties, and to set properties for the EJB module as a whole. These properties can also be set through the EJB module's property sheets. At deployment time, the IDE generates the EJB module's deployment descriptor, incorporating all specified properties.

Enforcing Security Policies

The EJB container offers you mechanisms for securing your application, that is, for restricting the set of users who can call methods on an enterprise bean. You can specify the security policies for your application either declaratively or

programmatically. Declarative security is specified within the deployment descriptor, and therefore it can be changed at any point up through deployment. Programmatic security is defined within the code of the enterprise bean, and therefore it is supplied by the programmer.

In most cases, declarative security is preferable. It's easier to provide, and it's configurable throughout the development, assembly, and deployment process.

Programmatic security is more complicated. However, it provides a more granular control of security, and therefore it's sometimes the only option to meet the security requirements of an application. For example, if you want to perform different logic within the body of a method depending on the identity of the caller, you must use programmatic security.

To specify security policies for your enterprise beans, you define a set of security roles for your application. A security role is a set of users who share common permissions for executing the methods of your enterprise beans.

With declarative security, each security role is assigned a set of bean methods that callers in that role are permitted to execute. At runtime, the container checks the security role of each caller, and decides whether the caller is permitted to execute the requested method.

In providing programmatic security, you can use methods supplied by the container (`getCallerPrincipal` and `isCallerInRole`) to determine the identity or role of the caller, and then you can use conditional logic as appropriate.

Declaring Security in Enterprise Beans

You declare security roles and method permissions after you have assembled the enterprise beans into an EJB module. On the module's property sheet, you define security roles for the EJB module. On the property sheet of the assembled EJB component, you define, for each security role, the list of methods that a caller in that role is permitted to execute.

When you take the declarative approach, you can modify the security permissions at any time during development and testing. Also, you can use different security roles and method permissions for each different EJB module that includes your bean.

Programming Security Into Enterprise Beans

Programmatic security allows you to determine:

- The individual identity of the caller
- Whether the caller has a particular security role

With this information, you can branch your logic conditionally, depending on the identity or role of the caller.

To programmatically determine the identity of the caller, you use the `getCallerPrincipal` method on the `javax.ejb.EJBContext` object. This returns a `java.security.Principal` object, which allows you to get the caller's name. You might use this to query a database for more information about the caller.

To programmatically determine whether the caller has a particular logical role, use the `isCallerInRole(String roleName)` method on the `javax.ejb.EJBContext` object. This returns a Boolean value indicating whether the caller has the specified logical role. If you use the `isCallerInRole` method, you must also declare the `roleName` used in your code as a security-role reference on the bean's property sheet.

At assembly time, when the bean is included in an EJB module, the assembler can map the bean's security-role reference to one of the security roles defined in the EJB module. Therefore, the programmer does not need to know the actual security role names before they are determined at assembly time.

For more information on implementing security features in enterprise beans and J2EE applications, refer to *Building J2EE Applications*.

Understanding the Application Servers and Databases

The enterprise beans that you create using the Sun ONE Studio IDE are typically tested using the application server that comes with the IDE. This is the J2EE Reference Implementation (the RI), a non-commercial, operational server made freely available for demonstrations, prototyping, and educational use. You can test your enterprise beans on the RI to see how they will behave under different application conditions. All examples in this manual use the RI as the application server.

Refer to the Sun ONE Studio 4 Release Notes for information on any other application servers and server plugins that are available for the IDE.

The entity beans you create using the IDE can be tested using the database that is included: PointBase Server 4.2 Restricted Edition. All examples in this manual use PointBase as the database.

Further Reading

In addition to the specifications and blueprints mentioned earlier in this book, there are many information resources for EJB programmers. For example, the following documents suggest ways that you can improve the design and programming of your enterprise beans:

- Sun ONE Studio 4 tutorials and example applications
<http://forte.sun.com/ffj/documentation/tutorialsandexamples.html>
- Java 2 Platform, Enterprise Edition Blueprints
<http://java.sun.com/j2ee/blueprints>
- *Designing Enterprise Applications with the J2EE Platform*, Second Edition, at:
http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/index.html
- “Seven Rules for Optimizing Entity Beans” by Akara Sucharitakul, at:
<http://developer.java.sun.com/developer/technicalArticles/ebeans/sevenrules/>
- “Working with J2EE Application Clients” by Monica Pawlan, at:
<http://developer.java.sun.com/developer/technicalArticles/J2EE/appclient/>
- “Designing Entity Beans for Improved Performance” by Beth Stearns, at:
<http://developer.java.sun.com/developer/technicalArticles/ebeans/ejbperformance/>

Developing Session Beans

You can use the EJB Builder in the Sun ONE Studio 4 IDE to program the session beans that perform server-side business logic on behalf of clients in your Enterprise JavaBeans application. This chapter discusses the process of creating and working with stateless and stateful session beans.

A session bean of either type can use the EJB container to manage transactions, or you can code a session bean to manage its own transactions. Session beans access persistent data using the JDBC API and the Java Transaction API (JTA). A session bean can manage one or more entity beans.

The IDE provides wizards that help you create the parts of an enterprise bean: a bean class and its two or four interfaces. The default is a remote interface and a home interface, but you can substitute or add a local interface and a local home interface. Much of the task of creating the session bean is automated for you.

When you're programming session beans, you have many options in addition to those described in this chapter. Although the Sun ONE Studio IDE is designed to take care of much of your coding work, the IDE also supports those options flexibly and leaves many decisions up to you. For detailed instructions in coding session beans, refer to the resources listed in "Before You Read This Book" on page xviii, or to one of the many excellent texts on programming enterprise beans.

Using the EJB Builder With Session Beans


The EJB Builder is a collection of wizards, property sheets, and editors with which you can build enterprise beans consistently and easily. To see if the EJB Builder is installed, go to the main window and choose Tools → Options → IDE Configuration → System → Modules → J2EE Support. If you see EJB 2.0 Builder in the list of modules, and the Enabled field in the property sheet is set to True, the EJB Builder is ready for use.

You can take several approaches to creating your session beans in the Sun ONE Studio IDE. However, you get the most comprehensive support and, in general, the fastest path to bean completion, if you use the approach recommended in this chapter. The methodology described here takes full advantage of the IDE's ability to ensure consistency and its adherence to the J2EE standard.

For best results, use the EJB Builder to program session beans by:

- Creating a session bean and its required classes.

When you finish the EJB Builder wizard's short sequence, you have the framework of your session bean. The bean's three (or five) classes and a logical node are shown in the Explorer's Filesystems tabbed pane. The wizard generates declarations for all classes, and you supply the methods' implementations.

The logical node is the best place from which to work with a session bean. All logical nodes appear in the Explorer with this icon: 

- Adding methods, parameters, and exceptions.

Use the IDE's GUI support as described later in this chapter. You can add a method to a bean by using a dialog box available from the contextual menu or by directly editing the set of required classes.

- Setting values in a bean's deployment descriptor.

Use the session bean's property sheet, available at the logical node, to edit properties.

From a session bean's logical node, you can validate the bean. You can also use the IDE's testing feature against your bean.

Selecting a Session Bean Type

A session bean handles interaction between a client and an application service; the duration of this interaction is the session. Should your session bean be stateful or stateless? Should it manage its own transactions or have its container manage them? All of these choices are discussed next.

The EJB Builder is designed to support all of these choices, and you use the same wizard to generate the session bean's infrastructure. Later, you finish specifying each type of session bean.

Details follow in these sections:

- "Stateless or Stateful Session Beans" on page 49
- "Container-Managed or Bean-Managed Transactions" on page 51

Stateless or Stateful Session Beans

The main purpose of a session bean is to perform work on behalf of a client application, that is, to help a client carry on a conversation with one or more entity beans on the server side. When such a conversation consists of more than a simple question and a simple answer, the conversation's manager (that is, the session bean) must remember certain information until the conversation is finished. In that case, the session bean must have state. A stateless session bean might manage a less complex conversation.

A more detailed discussion of this choice is in Chapter 2. TABLE 3-1 highlights some of the design considerations.

TABLE 3-1 Deciding Between Stateless and Stateful Session Beans

Issue	Stateless	Stateful
Scope	A stateless session bean manages a simple interaction between a client and an entity, and calls only one method per session.	A stateful session bean manages a more complex interaction between a client and an entity, and calls more than one method per session.
Initialization	A stateless bean carries no data that must be initialized.	A stateful bean's state must be initialized. For example, if the bean is designed to set up access to remote resources, it acquires a reference to a resource factory.
Information saved	During its session, a stateless bean saves no state information between method invocations.	During its session, a stateful bean maintains a conversational state between the client and server. It saves state information between method invocations but discards the information when the session ends.
Relationship with clients	A stateless bean instance performs one operation on behalf of one client at a time. Once it has completed a method call, the instance can be pooled and reassigned to a different client, even during the same session.	A stateful bean instance performs a series of operations on behalf of one client at a time. Once that client's session is complete, the bean instance is destroyed, not pooled.
Application examples	A stateless bean could represent a catalog viewer. The bean's one method lets the end user look up an item in an online catalog.	A stateful bean could represent an online shopping cart that invokes several methods to accumulate items until the end user is ready to start processing the entire order.

Container-Managed or Bean-Managed Transactions

As is discussed in more detail in Chapter 2, you must specify whether your bean's container or the bean itself will manage the bean's transactions. TABLE 3-2 summarizes the differences between these choices.

TABLE 3-2 Deciding Between Container-Managed and Bean-Managed Transactions

Issue	Container-Managed Transactions	Bean-Managed Transactions
How transaction boundaries are set	The EJB container decides when to begin and commit a transaction according to the <i>Java 2 Platform, Enterprise Edition Specification</i> .	The programmer explicitly codes the transaction's boundaries to obtain finer-grained control over transactions.
Transaction manager	The container itself is the transaction manager.	To manage transactions, use JTA, which can include transactions for other resources such as JDBC.
Transactions and methods	One transaction is allowed per method. However, a method does not have to be associated with a transaction.	This case is more complex, but you can code more than one transaction per method.

In a typical enterprise bean using container-managed transactions (CMT), the container begins the transaction just before a method starts and commits the transaction just before the method exits. With CMT, you can let the client control the transaction. For example, a client might string together a logical business transaction by using different methods called by a stateful CMT session bean.

In a session bean with bean-managed transactions (BMT), you must specify in the code where a transaction begins and ends.

Defining a Session Bean

The EJB Builder wizard automates much of the task of creating the three default classes that your session bean requires: a bean class, and the interfaces you choose (remote, local, or both remote and local). To define a session bean, you take the following steps:

1. Select or create a package to contain the session bean.
2. Use the EJB Builder wizard to generate the infrastructure of your session bean.
3. Add create and business methods to the bean's code.
4. Complete the bodies of the methods you added.

These basic steps are explained in detail next.

Creating a Package

If you need to create a package to house your session bean, select a filesystem, right-click, and choose New Package.

Starting the EJB Builder Wizard

When you're ready to create a session bean, do as follows:

1. **In the IDE's main window, choose View → Explorer to open the Explorer window.**
2. **In the Filesystems pane of the Explorer, select the package where you want your session bean to reside.**
3. **Right-click and choose New → J2EE → Session EJB.**

The EJB Builder wizard appears. Notice that the panel on the left shows the current step and the steps you still must complete before your entity bean is created.

Generating the Default Session Bean

In the EJB Builder's Session Bean Name and Properties pane, you must make choices about state, transaction type, and type of interfaces. Do as follows:

1. **Type a name for your session bean, and select the type of session bean you need.**

Click the appropriate buttons to specify your bean's state, transaction mode, and which type of interfaces to implement. FIGURE 3-1 shows your choices. Notice the defaults: Stateless, Container-Managed, and Remote Interface Only.

Session Bean Name and Properties

EJB Name:

Package:

State

☒ Stateless

☐ Stateful

Transaction Type

☒ Container-Managed

☐ Bean-Managed

Component Interfaces

☒ Remote Interface Only (Default)

☐ Local Interface Only

☐ Both Remote and Local Interfaces

FIGURE 3-1 Possible Wizard Selections for a Stateless (or Stateful BMT) Session Bean

Note – The selections you make in this first pane of the wizard determine the code that the wizard generates. If you later want to change any of these most basic selections, you can use the bean's property sheets, as described in Chapter 8.

2. Click Next.

The Session Bean Class Files pane appears as shown next for a stateless session bean.

The dialog titled "Session Bean Class Files" contains two sections. The "Bean Class" section has a text field labeled "Bean Class:" with the value "session.ProcessOrderEJB" and a "Modify Class..." button. The "Remote Client Interfaces" section has two rows: "Home Interface:" with "session.ProcessOrderHome" and a "Modify Interface..." button, and "Remote Interface:" with "session.ProcessOrder" and a "Modify Interface..." button.

For a stateful session bean, you have an additional selection to make: Session Synchronization.

The dialog titled "Session Bean Class Files" is similar to the one above but includes an additional checkbox labeled "implement SessionSynchronization Interface" which is checked. The "Bean Class:" text field contains "session.runWeeklyTotals.monthlyTotalsBean".

This selection is explained in TABLE 3-4 and in "Using Session Synchronization" on page 65.

3. Check the bean class and interfaces, and change them if necessary.

The classes that make up your session bean are shown with their paths in this pane.

- You can change the package location of the bean.
- You can use a Modify button to change any of the class names, specifying either a class that already exists or creating a new one. For example, you might be implementing a bean whose home and remote interfaces have already been specified, and now you want to generate a new bean class.

If you specify any classes outside the named package, the resulting bean classes appear differently than shown in FIGURE 3-2.

- Don't change the superclass of your bean's interfaces. (The IDE's code generator delegates to the superclass implementation, if one exists. However, as a general principle, you should inspect the code.)

Before you change these fields, also consider the following points:

- **Server requirements.** The EJB Builder wizard lets you move parts of the session bean to other locations. For example, you can change the package name on one or more of the related objects so that the bean class is in one directory and the home and remote interfaces are in another. First, however, you should find out whether the application server you plan to use supports this distribution of files.
- **Reuse of classes.** At this point you can, if you want, substitute a bean class or home and remote interfaces from another session bean. The wizard prompts you if the class you substitute is missing any required methods or exceptions.
- **Package and directory names.** Use valid Java identifiers.

4. Click Finish when you're done.



The wizard automatically generates the parts of your session bean's infrastructure. These parts are discussed next.

Looking at a Session Bean's Classes

The EJB Builder wizard generates the default session bean classes for you and sets up the relationships between all the classes. FIGURE 3-2 shows how a typical session bean (all of whose classes are in the same package) appears in the Explorer's Filesystems pane.



FIGURE 3-2 Default Classes of a Typical Session Bean With Remote Interfaces

The nodes marked with the class icon  represent classes of the session bean. The node marked with the bean icon  is a logical node for the session bean. Do all your editing in the logical node. The example bean's primary nodes are described next.

- The remote interface extends the `javax.ejb.EJBObject` interface and provides signatures for the session bean's business methods that are called from outside the bean's EJB module.
- The bean class implements the `javax.ejb.SessionBean` interface and implements the session bean's methods.
- The home interface extends the `javax.ejb.EJBHome` interface and provides signatures for the session bean's create methods that are called from outside the bean's EJB module.

- If you chose a local interface, you see a node labeled `Localbean_name`. This interface extends the `javax.ejb.EJBLocalObject` interface and provides signatures for the session bean's business methods that are called from within the bean's EJB module.
- If you chose a local home interface, you see a node labeled `Localbean_nameHome`. This interface extends the `javax.ejb.EJBLocalHome` interface and provides signatures for the session bean's create methods that are called from within the bean's EJB module.
- The logical node is created in the Explorer to group all the elements of your enterprise bean and let you work with them more conveniently.

Expanding the Nodes

When you expand the four nodes under your session bean's package node, you see something like the tree view in FIGURE 3-3.

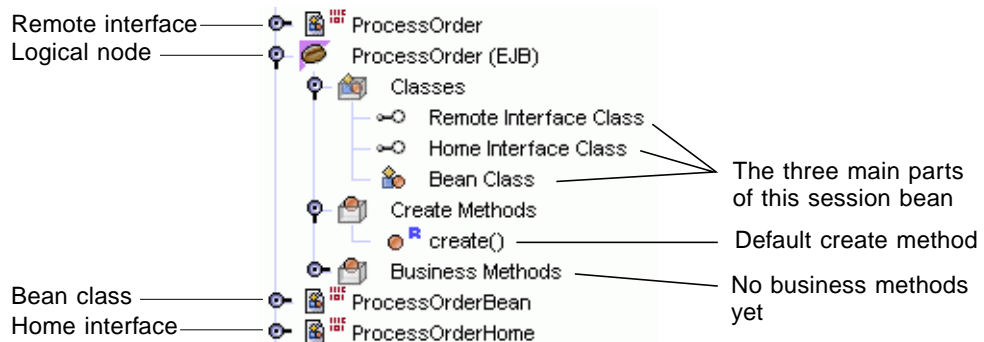


FIGURE 3-3 Explorer's Detailed View of a Typical Session Bean With Remote Interfaces

Reviewing the Generated Classes

The EJB Builder has automatically placed a create method and several life-cycle methods in your session bean. These methods are discussed next.

Default Create Method

The wizard places the following `ejbCreate` method signature in every session bean class:

```
public void ejbCreate() {  
}
```

The corresponding `create` method is placed in the session bean's home interface:

```
public interface AcctBalHome extends EJBHome {  
    public AcctBal create()  
        throws RemoteException, CreateException;  
}
```

See “Completing Create Methods” on page 60 for more information.

Life-Cycle Methods

The wizard adds the life-cycle methods shown next to the bean class of any session bean.

```
public void setSessionContext(SessionContext context) {  
    this.context = context; }  
public void ejbActivate() {  
}  
public void ejbPassivate() {  
}  
public void ejbRemove() {  
}
```

TABLE 3-3 shows the purposes of these methods in the session bean class.

TABLE 3-3 Purpose of Life-Cycle Methods In a Session Bean Class

Method	Purpose
<code>setSessionContext</code>	This method lets you store the <code>SessionContext</code> reference in a field and populate instance variables. You can use it to allocate resources that last for the session bean's lifetime, resources such as a database-connection factory. By default, the EJB Builder wizard generates code that assigns the <code>SessionContext</code> to a field named <code>context</code> .
<code>ejbActivate</code>	This method initializes the bean, prepares it for use, and acquires the resources needed by the instance.
<code>ejbPassivate</code>	Before the bean instance is passivated, this method releases the resources the bean was using.
<code>ejbRemove</code>	This method releases resources that were acquired within the <code>ejbCreate</code> and business methods.

If you have opted to have your session bean use the session-synchronization interface, the wizard generates three more methods in the bean class:

```
public void afterBegin() {  
}  
public void beforeCompletion() {  
}  
public void afterCompletion(boolean committed) {  
}
```

The session-synchronization methods are explained in TABLE 3-4.

TABLE 3-4 Purpose of Session-Synchronization Methods in a Session Bean Class

Method	Purpose and Use
<code>afterBegin</code>	This method tells the instance that a new transaction has begun. The EJB container calls the method right before it calls the business method. In <code>afterBegin</code> , you can load instance variables from the database.
<code>beforeCompletion</code>	This method tells the instance that a business method has completed, but the transaction has not been committed yet. This is the session bean's last chance to roll back the transaction. If the database hasn't yet been updated with the instance variables, you can code that update in the method body.
<code>afterCompletion</code>	This method tells the instance that the transaction has completed. In its one parameter, the Boolean value <code>true</code> means the transaction was committed, and <code>false</code> means the transaction was rolled back. If the transaction failed and was rolled back, this method can make the session bean refresh its instance variables from the database.

Completing Your Session Bean

The steps to completion of your session bean vary by the type of bean you have chosen. Guidelines follow for:

- Completing Create Methods
- Completing Life-Cycle Methods
- Adding Business Methods
- Coding Transactions

Using Recommended Approaches When Working With Enterprise Beans

Appendix A discusses the best ways to make changes in your enterprise beans, and the errors and anomalies that you might see if you use other approaches. As a general rule, you should work through the logical node rather than the individual class nodes, use the bean's property sheets or the Customizer dialog box to edit methods, and use the IDE's Source Editor to complete or edit any bean code that isn't available to you through one of the dialog boxes.

Completing Create Methods

If your bean is stateless, it takes only one create method, which can have no parameters. A stateless session bean can contain no user-specific or client-specific data.

If your bean is stateful, it can have one or more create methods, each of which can have parameters.

In any case, work under the logical node. Open the create method in the Source Editor by selecting the node labeled `create()`, right-clicking, and choosing Open. Complete the generated create method in the Source Editor.

Completing a Stateless Bean's Create Method

In a stateless session bean, the create method is often used to connect to resources. For example, this method can look up a resource-factory reference and store it as a field, so that JDBC connections can be acquired in later method calls.

Completing a Stateful Bean's Create Methods

In a stateful session bean, you can use a create method's parameters to look up a resource-factory reference or to send client-specific information (such as a user name and password), as shown in CODE EXAMPLE 3-1. The method can store the information for later use. Notice that this create method uses a helper class, `IdVerifier`.

CODE EXAMPLE 3-1 Create Method in a Stateful Session Bean

```
public void ejbCreate(String userid, String pwd)
    throws CreateException {

    if (userid == null) {
        throw new CreateException("Please enter a user ID.");
    }
    else {
        this.userid = userid;
    }

    IdVerifier idChecker = new IdVerifier();
    if (idChecker.validate(pwd)) {
        this.pwd = pwd;
    }
}
```

CODE EXAMPLE 3-1 Create Method in a Stateful Session Bean (*Continued*)

```
        else {  
            throw new CreateException("Invalid password: " + pwd);  
        }  
  
        contents = new Vector();  
    }  
}
```

Adding a Create Method to a Stateful Bean

To add one or more create methods to your stateful session bean, do as follows:

1. **Select the bean's logical node, right-click, and choose Add Create Method.**

The Add Create Method dialog box appears.

2. **Type a method name starting with `create`, add parameters and exceptions as necessary, and click OK.**

The create method signature is generated in your bean's home interface and the corresponding `ejbCreate` method is generated in the bean class.

3. **Finish any coding needed in the Source Editor.**

Under the bean's logical node, expand the `Classes` node, select `Bean Class`, right-click, and choose `Open`.

Completing Life-Cycle Methods

The EJB Builder has generated four life-cycle methods for you. For a stateless session bean, the generated methods are sufficient. In a stateful session bean, you might need to add code to two of these methods: `ejbPassivate` and `ejbActivate`.

For example, your stateful bean might contain nonserializable fields that became serializable by replacing references. Or, your bean's conversational state might contain open resources, which the container can't retain when the bean's instance is passivated. In each case, you must complete the `ejbPassivate` method to release the nonserializable fields. Then complete the corresponding `ejbActivate` method to restore those fields.

Completing the ejbPassivate Method

This method must leave the instance fields ready to be serialized by the container. For example, as shown in CODE EXAMPLE 3-2, you must close all JDBC connections in this method and assign the instance's fields that store the connections to null.

CODE EXAMPLE 3-2 ejbPassivate Method

```
public void ejbPassivate() {  
  
    try {  
        con.close();  
    } catch (Exception ex) {  
        throw new EJBException("ejbPassivate Exception: " +  
            ex.getMessage());  
    } finally {  
        con = null;  
    }  
}
```

Completing the ejbActivate Method

This method must make the instance fields available again, as demonstrated in CODE EXAMPLE 3-3.

CODE EXAMPLE 3-3 ejbActivate Method

```
public void ejbActivate() {  
  
    try {  
        InitialContext ic = new InitialContext();  
        DataSource ds = (DataSource) ic.lookup(dbName);  
        con = ds.getConnection();  
    } catch (Exception ex) {  
        throw new EJBException("ejbActivate Exception: " +  
            ex.getMessage());  
    }  
}
```

Adding Business Methods

In a session bean, you add business methods to run business tasks for the client. Such a method in a session bean might access a database, or it might manage one or more entity beans that use their persistent fields to manipulate database entities.

To add a business method to your stateful session bean, do as follows:

1. **Select the bean's logical node, right-click, and choose Add Business Method.**

The Add Business Method dialog box appears.

2. **Name the method, check to be sure the return type is appropriate, add parameters and exceptions as necessary, and click OK.**

The business method signature is generated in your bean's remote interface and the corresponding method in the bean class.

3. **Finish any coding needed in the Source Editor.**

Under the bean's logical node, expand the `Classes` node, select `Bean Class`, right-click, and choose `Open`.

If your session bean needs access to the database, you might be able to reduce JDBC calls in the bean (and save system resources and network bandwidth) by encapsulating database access in a data access object (a DAO). A DAO can do the actual work of fetching data for the session bean. Using a DAO might make your session bean's code more simple and straightforward, and it might free your bean from dependence on a particular vendor tool or database.

Coding Transactions

The way you code transactions differs depending on whether your session bean is stateful or stateless, and whether it uses BMT or CMT. Guidelines follow for specifying transaction boundaries, dealing with rollbacks, and using the session-synchronization interface.

Understanding Transaction Spans

The allowable span of a transaction differs according to the type of session bean. TABLE 3-5 summarizes those differences. Notice that CMT and statefulness give a bean more flexibility.

TABLE 3-5 Relationship Between Transactions and Methods

In a stateless BMT bean (that manages its own transactions), a transaction can span only one method.	In a stateful BMT bean, a transaction can span one or more methods on the same session bean.
In a stateless CMT bean (whose transactions are managed by the container), a transaction can span more than one method, but each method must be on a different session bean.	In a stateful CMT bean, a transaction can span one or more methods on the same session bean.

Specifying Transaction Boundaries and Rollbacks

This section discusses guidelines for coding the starting and ending points of transactions in both CMT and BMT beans. Two start with, keep in mind these two general rules:

- Nested transactions are not allowed in session beans or in JTA code.
- Code is easier to maintain when JDBC and JTA transactions are not mixed. JTA is generally preferable because it can include transactions for other resources, including JDBC.

In CMT Beans

In a CMT bean, all transactions' boundaries are set by the EJB container, which means that you don't specify where a transaction begins or ends. Usually, the EJB container begins a transaction just before a method starts and commits the transaction just before the method ends.

Don't call any method that could interfere with the container's transaction boundaries. Problematic methods are:

- `commit`, `setAutoCommit`, and `rollback` methods of `java.sql.Connection`
- `getUserTransaction` method of `javax.ejb.EJBContext`
- Any method of `javax.transaction.UserTransaction`

A session bean can roll back a container-managed transaction in two ways:

- If a system exception is thrown, the container automatically rolls back the transaction.
- Calling the `setRollbackOnly` method of `javax.ejb.EJBContext` tells the container to roll back the transaction even if an application exception is thrown.

In BMT Beans

In a BMT bean, you must explicitly code the beginning and ending of each transaction. Demarcate transaction boundaries explicitly using the interface `javax.transaction.UserTransaction`. In the following code sample, the JTA interface is used:

```
UserTransaction ut = ejbContext.getUserTransaction();
ut.begin();
// perform transactional work here
ut.commit();
```

When the updates specified by a transaction are saved, the transaction ends with a commit. When the transaction fails, it is rolled back, which means that the effects of all statements in the transaction are undone. When you provide for rollbacks in a session bean with BMT, don't use the methods `getRollbackOnly` or `setRollbackOnly`. Those two methods are for use only with an EJB container.

Using Session Synchronization

A stateful CMT session bean can use the session-synchronization interface, which gives the bean more control over database data cached within transactions.

This interface provides callback methods that the EJB container calls before starting, committing, or rolling back a transaction. Using this interface, a session bean's instance variables are automatically synchronized with their corresponding values in the database at specific stages in the transaction. If the transaction doesn't complete, the session bean can roll back the values of the bean's instance variables.

- `afterBegin`. The container calls this method on the session bean before the first business method within a transaction. You can code this method to do any database work required by the instance within the scope of the transaction.
- `beforeCompletion`. The container calls this method when the session bean's client has completed work on its current transaction but before committing the resource managers used by the instance. You can code this method to write out

any database updates the bean has cached. In this method, you can also cause the transaction to roll back by invoking the `setReadbackOnly` method on its session context.

- `afterCompletion`. The container calls this method to signal that the current transaction has completed. The status `True` is sent if the transaction committed, and `False` if the transaction was rolled back. You can code this method to manually reset the instance's state if the transaction was rolled back.

To add the session-synchronization interface to your session bean, make the following choices in the wizard:

1. In the first pane of the session bean wizard, in the **State** section, choose **Stateful**.
2. In the second pane of the wizard, choose **Implement Session Synchronization Interface**.

If you make these selections, code like that shown in CODE EXAMPLE 3-4 is inserted into your session bean class. In the example, `checkingBalance` and `savingBalance` variables have been loaded into the `afterBegin` method.

CODE EXAMPLE 3-4 Example of an `afterBegin` Method

```
public void afterBegin() {
    System.out.println("afterBegin()");
    try {
        checkingBalance = selectChecking();
        savingBalance = selectSaving();
    } catch (SQLException ex) {
        throw new EJBException("afterBegin Exception: " +
            ex.getMessage());
    }
}
```

The sample `afterCompletion` method shown in CODE EXAMPLE 3-5 allows the account-balance fields in the session bean to be refreshed from the database if the transaction fails and is rolled back.

CODE EXAMPLE 3-5 Example of an afterCompletion Method

```
public void afterCompletion(boolean committed) {
    System.out.println("afterCompletion: " + committed);
    if (committed == false) {
        try {
            checkingBalance = selectChecking();
            savingBalance = selectSaving();
        } catch (SQLException ex) {
            throw new EJBException("afterCompletion SQLException: " +
                ex.getMessage());
        }
    }
}
```

After Creating Your Session Bean

Your session bean still needs to be prepared to work in its eventual environment. For information on the deployment descriptor, how to use property sheets, and other considerations of module assembly and application deployment, see Chapter 8.

Recommendations for working with finished enterprise beans are given in Appendix A.

Further Reading

Enterprise beans can be a very powerful and flexible part of your application. Creating the basic parts of an enterprise bean can be very simple, especially with a tool like the Sun ONE Studio IDE. However, completing the bean so that it fulfills the needs of your application can be very complex. For details, refer to *Enterprise JavaBeans Specification*, version 2.0 at:

<http://java.sun.com/products/ejb/docs.html>

Developing CMP Entity Beans

The EJB Builder in the Sun ONE Studio IDE enables you to program the entity beans you need to represent data in your J2EE application. This chapter focuses on how you develop individual entity beans with container-managed persistence (CMP entity beans).

The IDE provides wizards that let you create the classes required for an Enterprise JavaBeans component (enterprise bean): a bean class, interfaces (local, remote, or both), and sometimes a primary-key class. Much of the task is automated for you.

When programming entity beans, you have many options in addition to those described in this chapter. Although the Sun ONE Studio IDE is designed to take care of much of your coding work, the IDE also supports those options flexibly and leaves many decisions up to you. For more information, refer to the resources listed in “Before You Read This Book” on page xviii, or to one of the many excellent texts on programming enterprise beans.


Using the EJB Builder With CMP Entity Beans

The EJB Builder is a collection of wizards, property sheets, and editors with which you can build enterprise beans consistently and easily. To see if the EJB Builder is installed, go to the main window and choose Tools → Options → IDE Configuration → System → Modules → J2EE Support. If you see EJB 2.0 Builder in the list of modules, and the Enabled field in the property sheet is set to True, the EJB Builder is ready for use.

You can take several approaches to creating entity beans in the Sun ONE Studio IDE. However, you get the most comprehensive support and, in general, the fastest path to bean completion, if you use the approach recommended in this chapter. The methodology described here takes full advantage of the IDE's ability to ensure consistency and its adherence to the J2EE standard.

For best results, use the EJB Builder to program entity beans by:

- **Creating an entity bean and its required classes.** After using the EJB Builder wizard, you have the framework of your entity bean. The bean's three or four necessary classes and a logical node are shown in the Explorer's Filesystems tabbed pane. The wizard generates declarations for two of these classes: the home and remote interfaces. The generated bean class contains declarations of required methods, as well as any persistent fields you specified. You then supply the implementations of the required methods.

The logical node is the best place from which to work with an entity bean. All logical nodes appear in the Explorer with this icon: 

- **Adding methods, parameters, and exceptions.** Use the IDE's GUI support as described later in this chapter. You can add a method to a bean by using a dialog box available from the contextual menu or by directly editing the set of required classes.
- **Setting values in a bean's deployment descriptor.** Use the entity bean's property sheet, available at the logical node, to edit properties.

From an entity bean's logical node, you can validate the bean, add methods to its classes, add fields, specify deployment-related properties for the bean, create an EJB module to facilitate the bean's deployment in a production application. You can also use the IDE's testing feature against your bean.

Comparing CMP and BMP Entity Beans

Before you begin creating an entity bean, first consider whether to use CMP or BMP. The IDE's EJB Builder supports either type of entity bean, but you use different processes to create the two types. A more detailed discussion of this choice is in Chapter 2; here, TABLE 4-1 highlights some of the design considerations.

TABLE 4-1 Deciding Between CMP and BMP Entity Beans

Issue	CMP	BMP
Relationship with the database	A CMP entity bean depends on its container to manage its relationship with a database, and is not dependent on any particular data store.	A BMP entity bean handles its own relationship with a specified database.
Persistence	The container manages database access for this and every other CMP entity bean in the application. The bean code does not include calls to the database. The bean's persistent state is represented by virtual persistent fields.	A BMP entity bean contains all the code connecting it to a specified database. A BMP entity bean with persistent data (coded as instance variables) also must contain all necessary calls to the database. All SQL code must be added by hand. If your EJB container doesn't provide adequate persistence mapping to the data store, you must create a BMP entity bean.
Process	The basic structure of a CMP entity bean (the default classes) is simpler and quicker to create. Less coding is needed.	A BMP entity bean requires more coding, which might be an attractive option for experienced JDBC programmers.
Design scope	A single CMP entity bean normally represents only one table, but a bean can be mapped to two or more tables.	A BMP entity bean can be hand-coded to represent one or more tables.
Power and flexibility	A CMP entity bean depends on its container for access to a database, but this bean can be deployed in many different database environments.	An individual BMP entity bean is manually programmed for database access. A BMP entity bean works only in the environment for which it was written.

The rest of this chapter addresses how to create CMP entity beans and issues to consider during development. For the process of creating BMP entity beans, see Chapter 6.

Creating Sets of Related CMP Entity Beans

Many J2EE applications contain related CMP entity beans. That is, two CMP entity beans can have a relationship that is represented by a container-managed relationship (CMR) field. This relationship is analogous to the situation in a database or database schema when two entities or tables contain a related column. For example, a schema might include the tables `Customer`, `Order`, `LineItem`, and `Part`. `Order` has a foreign key to `Customer`, `LineItem` has a foreign key to `Order`, and `LineItem` also has a foreign key to `Part`.

The IDE makes it easy to create a whole set of related CMP entity beans at once. When you use the EJB Builder wizard to generate a set of CMP entity beans to manipulate related database entities, the wizard recreates the entities' relationships in your CMP entity beans, and lets you specify additional relationships between beans. These relationships are represented in a CMP entity bean as CMR fields, and they can be edited for cardinality, type, and cascade-delete capability.

If you want to create a set of related CMP entity beans, or if you want to preserve foreign-key relationships between two entity beans, see Chapter 5.

Defining a CMP Entity Bean

The EJB Builder wizard automates much of the task of creating the minimum classes that your CMP entity bean requires: a bean class, and the interfaces you choose (local, remote, or both local and remote). If you specify a composite primary key, or if a table you chose requires a composite primary key, the wizard also creates a primary-key class for you. To define a CMP entity bean, you take the following steps:

1. Select or create a package to contain the bean.
2. Use the EJB Builder wizard to generate the infrastructure of your CMP entity bean.
3. As appropriate, add create, business, finder, select, and home methods to the bean.
4. Complete the bodies of the methods you added.
5. If necessary, add a primary-key class.

These basic steps are explained in detail next.

Creating a Package

If you need to create a package to house your entity bean, select a filesystem, right-click, and choose New Java Package.

Having a Data Source Ready

A CMP entity bean is modeled on an actual table from a database, and the bean's persistent fields are meant to echo the table's columns. Using the EJB Builder Wizard, you can obtain the table from a live database connection or a database schema object (a snapshot of a database). Or, in one of the wizard's panes, you can manually specify the table's columns as your bean's persistent fields, and then at deployment the fields can be mapped to database columns.

Notice that EJB containers vary in how they treat column-to-field mappings. The PointBase database server is included in the IDE and represented in the following examples. If you're using another database server with the IDE, refer to its documentation for details.

Consider the following when deciding which form of data source to use:

- **Live database connection.** If you plan to build your CMP entity bean from a table in a live database, the database server must be running, and you must be connected to it. You can do this either before or after you start the EJB Builder Wizard. The following instructions pertain to starting the server before starting the wizard. Instructions for starting the server within the wizard are in "Selecting a Table From a Database Connection" on page 77.

First, before you start the IDE, make sure that its `lib/ext` directory contains the driver files for any database that was not automatically loaded when the IDE was installed. This is the only way to ensure that you can select the right database driver when creating your new schema. You can't mount the driver files in the Explorer or place the driver files in the `CLASSPATH` env variable.

Starting: To start the PointBase database, choose Tools → PointBase Network Server → Start Server from the main window.

Connecting: To connect to the running PointBase database, go to the Runtime tabbed pane of the Explorer. Expand the Databases and Drivers nodes. Select the node whose label begins with `jdbc:pointbase` and whose icon appears broken in two. Right-click the node and choose Connect. The PointBase icon becomes whole, and sub-nodes appear for the database tables, views, and procedures.

This is the best way to connect to the database, especially if you're going to be creating more than one entity bean. Alternatively, you can start the database before you start the EJB Builder Wizard, and then you can connect to the database from within the wizard. However, if you do that, you must reconnect for every entity bean you create.

- **Database schema.** If you plan to build your bean from a table in a database schema, you must have the schema available in the Filesystems pane of the IDE's Explorer window.

Starting the EJB Builder Wizard

When you're ready to create a single CMP entity bean, do as follows:

1. **In the IDE's main window, choose View → Explorer to open the Explorer window.**
2. **In the Filesystems pane of the Explorer, select the Java package where you want your CMP entity bean to reside.**
3. **Right-click and choose New → J2EE → CMP Entity EJB.**

The EJB Builder wizard appears. Notice that the panel on the left shows the current step and the steps you still must complete to generate the infrastructure of your CMP entity bean.

Generating a CMP Entity Bean's Infrastructure

In the EJB Builder's CMP Entity Bean Name and Properties pane, as shown in FIGURE 4-1, you name your CMP entity bean. Here, you also make choices about where your bean will get its persistent fields and which kinds of interfaces your bean will have. You can also change the bean's package location, if you like.

CMP Entity Bean Name and Properties

EJB Name:

Package:

Source for Entities and Fields

- ☒ Table from Database Connection
- ☐ Table from Database Schema Object
- ☐ CMP 2.x Bean Class
- ☐ CMP 1.x Bean Class
- ☐ Create from scratch

Component Interfaces

- ☐ Remote Interface Only
- ☒ Local Interface Only (Default)
- ☐ Both Remote and Local Interfaces

FIGURE 4-1 Selections in the EJB Builder Wizard for CMP Entity Beans

The following tables describe these selections and point to your next instructions in this chapter.

In the radio-button box labeled Source for Entities and Fields, consider the following selections.

Table From Database Connection	Select this if your CMP entity bean will represent a table from an existing database.	See “Selecting a Table From a Database Connection” on page 77.
Table From Database Schema Object	Select this if you have a database schema available, and you don’t want to connect to a live database.	See “Selecting a Table From a Database Schema Object” on page 79.

CMP 2.x Bean Class	Select this if your CMP entity bean will be based on an existing bean class that follows the EJB 2.0 specification.	See “Using a CMP 2.x Bean Class” on page 80.
CMP 1.x Bean Class	Select this if your CMP entity bean will be based on an existing bean class that follows the EJB 1.x specification.	See “Using a CMP 1.x Bean Class” on page 81.
Create From Scratch	Select this if you will specify all the CMP fields yourself.	See “Creating Your Bean’s Persistent Fields From Scratch” on page 82.

In the radio-button box labeled Component Interfaces, consider the following selections.

Remote Interface Only	Select this if an external client calls methods on your CMP entity bean, and your bean is never called by local clients.
Local Interface Only (Default)	Leave this selection active if your bean is called only through its local interfaces, never directly by an external client.
Both Remote and Local Interfaces	Select this if your bean is called by both external and local clients (which can also be other beans).

When you have made your selection and clicked Next, the wizard presents appropriate follow-up tasks, which are described next.

Specifying Persistent Fields From a Database Table

If you have chosen to specify persistent fields from a database, you must already be connected to a live database or have available an existing database schema object. For more information, see “Having a Data Source Ready” on page 73 and “Capturing a Database Schema” on page 79. The EJB Builder wizard maps columns from a table of the database (Table from Database Connection) or from a schema (Table from Database Schema Object) to create your entity bean’s persistent fields. Both choices provide the same result in your finished entity bean.

Most application servers let you map a bean’s CMP fields to database columns at deployment time. The server then dynamically generates SQL statements for that mapping within the server process. However, the J2EE reference implementation server (the RI) generates static SQL statements that encode its mapping. For this reason, any changes must be done by editing the generated SQL statements directly. This editing is discussed in Chapter 8.

Selecting a Table From a Database Connection

If you have direct access to the database itself, and if contention among database users is not a problem, you might want to use the direct database connection. (If you need to start and connect to a database, see “Having a Data Source Ready” on page 73.)

You should now be in the wizard’s Table from Database Connection pane. The databases to which you can connect your entity bean appear in the wizard pane’s tree view. Do as follows:

1. Select a database.

Depending on the status of the database, use one of the following approaches.

- **The database is installed but no connection is available.** If you have a database installed but no connection is defined, click Add Connection (or New Connection). In the dialog box that appears or is enabled, do as follows:

- a. Select the database from the Name combination box.
- b. Check the Driver field to make sure the path is correct.
- c. Specify the required information in the Database URL field.
- d. Supply a user name and password if any are needed for your database.
- e. Check the Remember Password During This Session box if appropriate.
- f. Click OK. In the wizard’s Table from Database Connection pane, click Next.

The connection becomes available.

- **The database is installed but the connection is defined.** If you have a database installed and a connection is defined but not active, the database node is shown as a broken icon. Do as follows:

- a. Select the database and click Connect to Database.

You see the broken halves become a whole icon.

- b. Expand the database node until the Tables node appears.

If you get an Unable to connect error message, make sure that the database is up and running.

- **The database is installed and the connection is active.** If your connection to the database is already defined and active (for example, if you are now creating a second entity bean using the same connection as the first), the icon appears whole, not broken. In this case, all you need to do is expand the database node to find the Tables node.

2. **Descend through the selected database's hierarchy until you see a node for the table that you want to map to your bean. Select a table and click Next.**

You see the CMP Fields pane. This pane displays side by side the columns in your database table and the corresponding fields that the EJB Builder Wizard will create in your new CMP entity bean. The wizard will map those database columns to your bean's persistent fields.

3. **Check the Java field names and types, and make any necessary changes.**

The IDE has assigned default names and types to your Java fields. You can change the names and types if necessary, selecting a field and clicking the Edit button to see other permissible data types.

For more information, refer to Chapter 8, "Mapping SQL and Java Types" of *Getting Started with the JDBC API*. You can find the document at:

<http://java.sun.com/j2se/1.3/docs/guide/jdbc/getstart/GettingStartedTOC.fm.html>

4. **Click Next. (Or, if you don't want to examine or change the default classes that the wizard assigns, skip this step and click Finish.)**

The CMP Entity Bean Class Files pane of the wizard appears, listing the parts of your entity bean's infrastructure: the bean class, the interfaces you chose (local, remote, or both), and the type of the primary-key class.

In this pane, you can accept or change your bean's classes. The wizard lets you specify another bean class, interface, or primary-key class if you wish. As you see if you click one of the Modify buttons, you can specify another package for the class or interface to reside in or to come from.

- For example, you can change the package name on one or more of the related objects so that the bean class is in one directory and the home and remote interfaces are in another.

However, first you should find out whether the application server you plan to use supports this distribution of files.

- If you specify an existing class or interface that is missing any required methods or exceptions, you get an error message.
- You must use valid Java identifiers in package and directory names.

5. **Click Finish.**

Your CMP entity bean's infrastructure is generated automatically by the EJB Builder. See "Looking at a CMP Entity Bean's Classes" on page 84 for more discussion.

Capturing a Database Schema

You might need to build your CMP entity bean on a table from a database schema rather than connecting directly to the database. If you don't already have a schema, you can use the IDE's Database Schema wizard to create one. First, if you need to start and connect to a database, see "Having a Data Source Ready" on page 73. Then do as follows:

1. **In the Sun ONE Studio IDE, open the Database Schema wizard in either of the following ways:**
 - Select the package node in the Explorer. Right-click and choose New → Databases → Database Schema.
 - From the main window, choose Tools → Capture Database Schema.
2. **As directed by the wizard, specify the database to be used and select the tables to be included in your schema.**

The progress bar shows the number of tables and views captured.

Selecting a Table From a Database Schema Object

If database access is restricted but schema objects have been made available, you might want to build your CMP entity bean using a table from a schema. (If you don't have a schema available and need to create one, see the foregoing section.)

Having selected Table from Database Schema Object on the first pane of the wizard, you should now be in the Table from Database Schema Object pane. The directories that have been mounted in the Explorer's Filesystems pane appear in the wizard pane. Do as follows:

1. **Locate the database schema that contains the table on which you will build your bean.**

Descend through the selected schema's hierarchy until you see a node for the table that you want to map to your bean.
2. **Expand the schema's nodes until you find the table you want to use. Select the table.**

The Next and Finish buttons become active.
3. **Click Next to review the database columns that will be mapped to your bean's persistent fields. (Or, click Finish to skip this step and the next, and have the wizard generate your bean's infrastructure.)**

The CMP Fields pane appears, displaying side by side the columns in your database table and the corresponding fields that the EJB Builder will create in your entity bean. You can change field names and types if necessary, selecting a field and clicking the Edit button to see other permissible data types.

4. **Click Next to examine or change the default classes that the wizard assigns. (Or, click Finish to skip this step and have the wizard generate your bean's infrastructure.)**

The CMP Entity Bean Class Files pane of the wizard appears, listing the parts of your entity bean's infrastructure: the bean class, the interfaces you chose (local, remote, or both), and the type of the primary-key class.

In this pane, you can accept or change your bean's classes. The wizard lets you specify another bean class, interface, or primary-key class if you wish. As you see if you click one of the Modify buttons, you can specify another package for the class or interface to reside in or to come from.

- For example, you can change the package name on one or more of the related objects so that the bean class is in one directory and the home and remote interfaces are in another.

However, first you should find out whether the application server you plan to use supports this distribution of files.

- If you specify an existing class or interface that is missing any required methods or exceptions, you get an error message.
- You must use valid Java identifiers in package and directory names.

5. **Click Finish.**

Your CMP entity bean's infrastructure is generated automatically by the EJB Builder. See "Looking at a CMP Entity Bean's Classes" on page 84 for more discussion.

Using a CMP 2.x Bean Class

You might want to base your new CMP entity bean on an existing CMP entity bean that was created in the EJB 2.0 environment. In the wizard's CMP Entity Bean Name and Properties pane, you select CMP 2.x Bean Class and click Next. The wizard then presents a navigation list from which you pick a bean class.

You should now be in the Select a CMP 2.x Bean Class pane. Do as follows:

1. **Navigate to the bean class you want to use, and select the class.**

Notice that the IDE presents only the bean class for selection, not the other elements of the bean. When you have selected the class, the Next button is enabled.

2. **Click Next.**

The IDE presents the fields of the bean class you selected. Even if the original CMP entity bean had a primary key, in this pane you must designate one or more fields as the primary key.

3. **Select the field that should be the primary key and click Edit.**

The Edit Persistent Field dialog box appears.

- 4. Make any necessary changes, including checking the Primary Key checkbox, and click OK.**

The CMP Fields pane shows the field you edited as the primary key.

Repeat Step 3 and Step 4 as needed for another field, if your bean needs a composite primary key.

In this pane, you can't remove a field or add a new field, but you can edit an existing one.

- 5. Click Next.**

The CMP Entity Bean Class Files pane lists the elements of the CMP entity bean you are about to create.

If your bean needs to use another interface, use the Modify Interface button to specify it.

If your bean needs a different primary key class, either a new one or an existing one, use the Modify Class button to specify it.

- 6. When you are done, click Finish.**

Your CMP entity bean's infrastructure is generated automatically by the EJB Builder. See "Looking at a CMP Entity Bean's Classes" on page 84 for more discussion.

Using a CMP 1.x Bean Class

You might want to base your new CMP entity bean on an existing CMP entity bean that was created in the EJB 1.0 environment. If you choose this option, your bean will be a version 2.0 Enterprise JavaBean with a CMP version of 1.x, and the bean will not support EJB 2.0 features like local and local home interfaces.

In the wizard's CMP Entity Bean Name and Properties pane, you select CMP 1.x Bean Class. (Notice that, when you've made this selection, only remote interfaces are available.) Click Next. The wizard then presents a navigation list from which you pick a bean class.

You should now be in the Select a CMP 1.x Bean Class pane. Do as follows:

- 1. Navigate to the bean class you want to use, and select the class.**

Notice that the IDE presents only the bean class for selection, not the other elements of the bean. When you have selected the class, the Next and Finish buttons are enabled.

- 2. Click Next.**

The IDE presents the fields of the bean class you selected. Even if the original CMP entity bean had a primary key, in this pane you must designate one or more fields as the primary key.

3. Select the field that should be the primary key and click Edit.

The Edit Persistent Field dialog box appears.

4. Make any necessary changes, including checking the Primary Key checkbox, and click OK.

The CMP Fields pane shows the field you edited as the primary key.

Repeat Step 3 and Step 4 as needed for another field, if your bean needs a composite primary key.

In this pane, you can't remove a field or add a new field, but you can edit an existing one.

5. Click Next.

The CMP Entity Bean Class Files pane lists the elements of the CMP entity bean you are about to create.

If your bean needs to use another interface, use the Modify Interface button to specify it.

If your bean needs a different primary key class, either a new one or an existing one, use the Modify Class button to specify it.

6. When you are done, click Finish.

Your CMP entity bean's infrastructure is generated automatically by the EJB Builder. See "Looking at a CMP Entity Bean's Classes" on page 84 for more discussion.

Creating Your Bean's Persistent Fields From Scratch

In the EJB Builder's Entity EJB Type pane, you might have selected Create From Scratch because your database hasn't yet been created, you don't yet have access to it, or you don't know its location. Or, you might want the application server to create the database when the application that contains the enterprise bean is deployed.

Your CMP entity bean's container might require that your bean be mapped to a database, but not until the assembly and deployment stage. When you select the Create From Scratch option, the fields you specify are marked as persistent in the deployment descriptor, which is later used to notify the container which fields it should map into the database schema. This mapping is done just before the J2EE application is deployed.

The IDE gives you the option of setting up your entity bean's connection by stating your bean's Java field names. Later, during preparation for deployment, you can specify the rest of the database connection information.

In the wizard's CMP Entity Bean Name and Properties pane, do as follows:

1. **In the EJB Name field, type a name for your bean.**
2. **If you want your bean to reside in a different location than shown, use the Browse button to select an existing Java package.**
3. **Select Create From Scratch and click Next.**

The CMP Fields pane appears. The wizard has automatically supplied your bean one default CMP field named `defaultField`.

4. **If you want to name your CMP fields at this time, select the default field and click Edit.**

Name and define your field, following these guidelines:

- Ordinarily, you would specify at least one primary-key field. However, a CMP entity bean is not strictly required to have a primary key.
- If your field needs a type not supplied in the combo box, you can specify another type. Type the fully qualified pathname, for example, `java.lang.Integer`.

5. **Click Add to define each additional persistent field individually.**
6. **Click Next to examine or change the default classes that the wizard assigns. (Or, click Finish to skip this step and have the wizard generate your bean's infrastructure.)**

The wizard's CMP Entity Bean Class Files pane appears, listing the parts of your entity bean's infrastructure: the bean class, the interfaces you chose (local, remote, or both), and the type of the primary-key class.

In this pane, you can accept or change your bean's classes. The wizard lets you specify another bean class, interface, or primary-key class if you wish. As you see if you click one of the Modify buttons, you can specify another package for the class or interface to reside in or to come from.

- For example, you can change the package name on one or more of the related objects so that the bean class is in one directory and the home and remote interfaces are in another.

However, first you should find out whether the application server you plan to use supports this distribution of files.

- If you specify an existing class or interface that is missing any required methods or exceptions, you get an error message.
- You must use valid Java identifiers in package and directory names.

7. **Click Finish.**

Your CMP entity bean's infrastructure is generated automatically by the EJB Builder Wizard. Now let's look at the generated classes.

Looking at a CMP Entity Bean's Classes

The EJB Builder wizard generates the default CMP entity bean classes for you and sets up the relationships between all the classes. FIGURE 4-2 shows how a typical CMP entity bean appears in the Explorer's Filesystems pane. In this example, the default, `Local Interface Only`, is in effect. This bean is called only by other application components running in the same JVM.

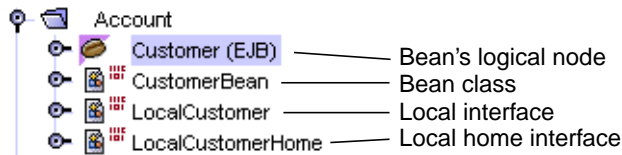
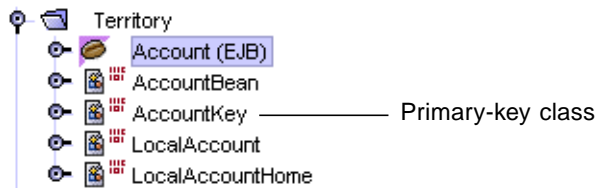


FIGURE 4-2 Default Classes of a Typical CMP Entity Bean

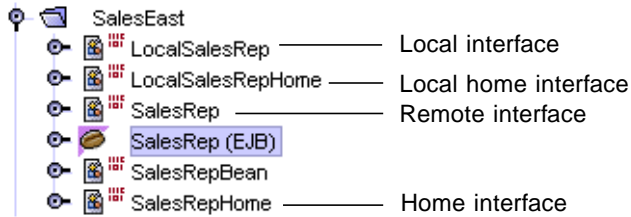
Of the four nodes shown in FIGURE 4-2, three represent actual classes (marked with class icons) and one is a logical node (marked with a bean icon). Do all your editing in the logical node. The example bean's primary nodes are described next.

- The Explorer provides the logical node to group all the elements of your enterprise bean and let you work with them more conveniently.
- The bean class implements the `javax.ejb.EntityBean` interface and implements the entity bean's methods.
- The local interface extends `javax.ejb.EJBLocalObject` and provides a way for beans in the same container to communicate.
- The local home interface extends `javax.ejb.EJBLocalHome` and provides signatures for your create and finder methods.

If you created a primary-key class (for example, if your bean has a composite primary key), the Explorer shows an additional node for your bean.



If you chose Both Remote and Local Interfaces (if your bean might be used both by beans in their own application's JVM and by beans in another JVM), the resulting CMP entity bean has all four interfaces.



- The remote interface extends `javax.ejb.EJBObject` and declares the CMP entity bean's business methods.
- The home interface extends `javax.ejb.EJBHome` and declares the create and finder methods that the client can call on the CMP entity bean.

Expanding the Nodes

When you expand the nodes under your entity bean's package node, you see something like the tree view in FIGURE 4-3. (In this case, the default, Local Interface Only, has been used.)

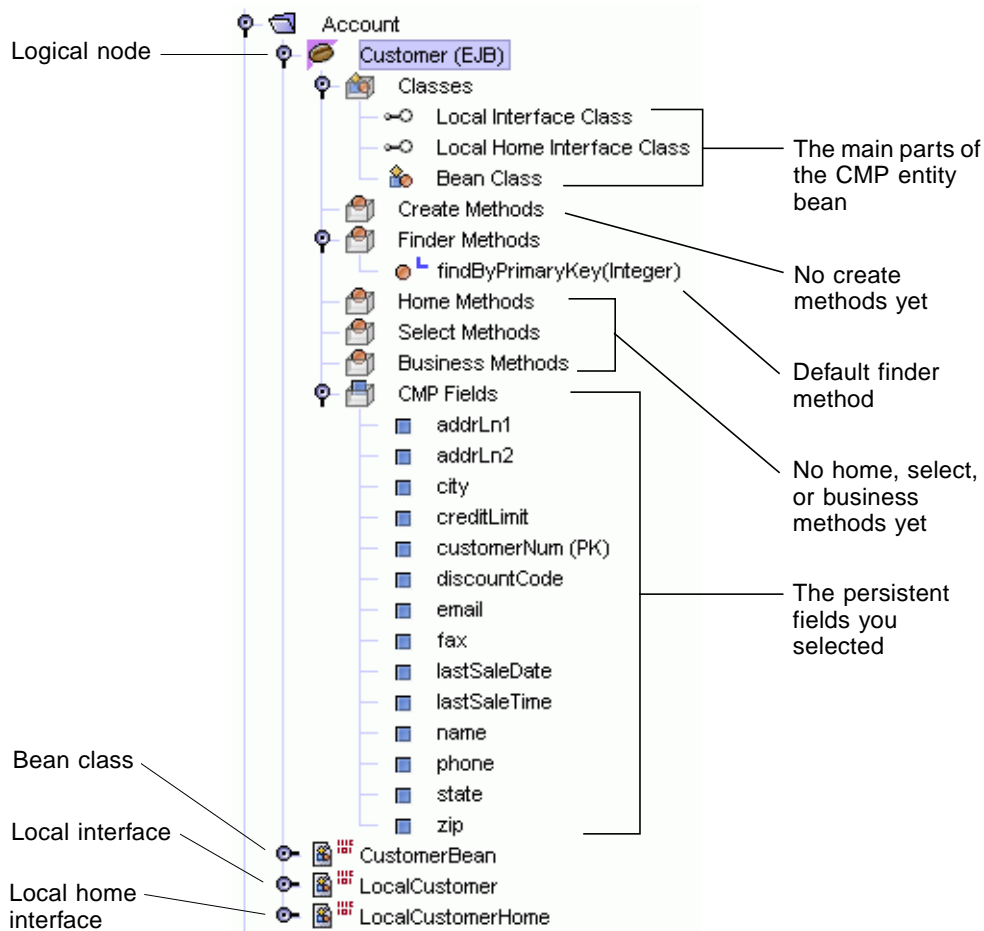


FIGURE 4-3 Explorer's Detailed View of a Typical CMP Entity Bean With Local Interfaces

If you generated a new primary-key class, it appears in the Explorer as shown in FIGURE 4-4.

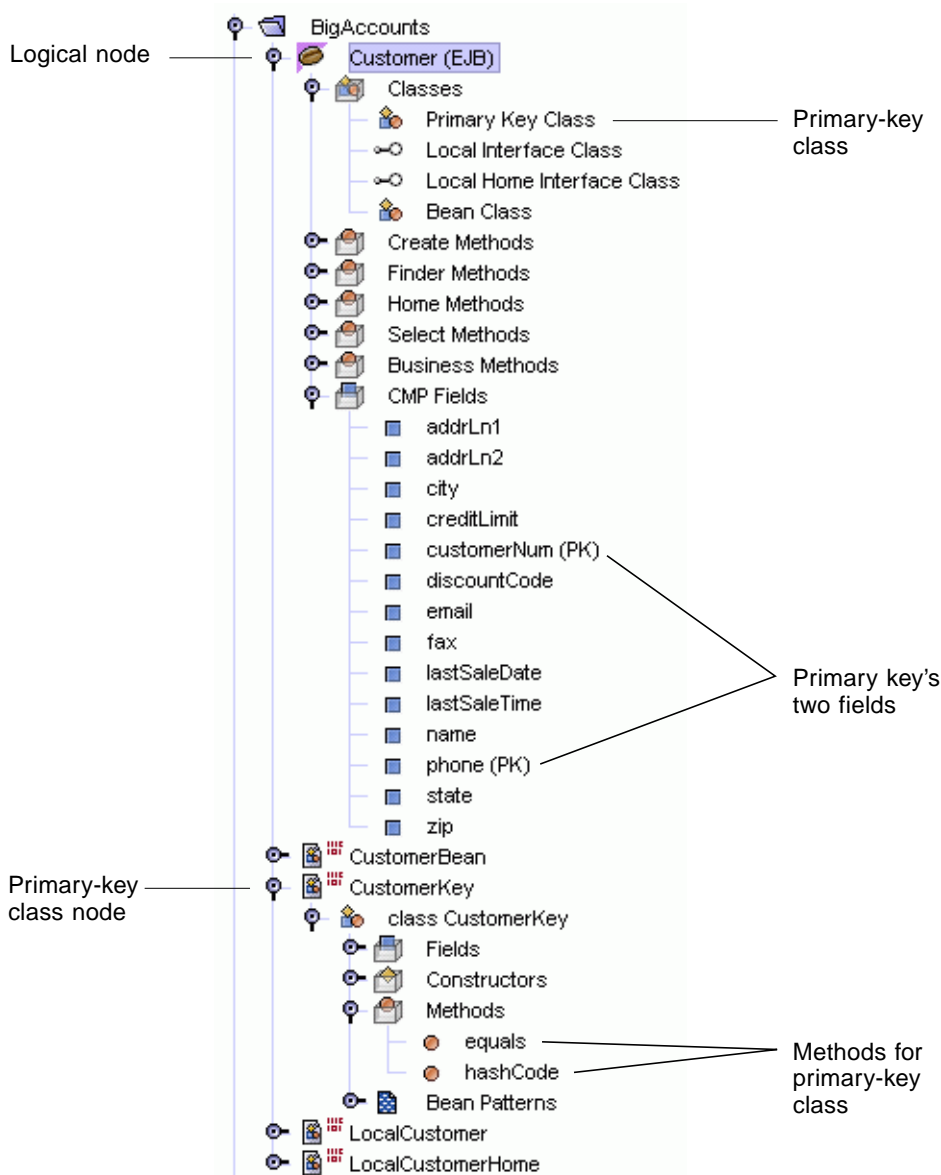


FIGURE 4-4 Explorer's Detailed View of a Typical CMP Entity Bean With a Composite Primary Key

Reviewing the Generated Classes

Any fields that were mapped from database columns appear in your CMP entity bean. In addition, certain default methods are automatically placed in all entity beans.

Default Finder Method

Because the *Enterprise JavaBeans Specification* requires every entity bean to be locatable by its primary key, the method signature `findByPrimaryKey` is added automatically to your entity bean's home interface. In a CMP entity bean, the method signature is enough because your bean's container will implement the `findByPrimaryKey` method.

```
public CustomerLocal findByPrimaryKey(java.lang.Integer aKey)
    throws javax.ejb.FinderException;
```

Persistent Fields and Accessor Methods

The IDE generates and places in the bean class a get method and a set method for every persistent field that you specified for your CMP entity bean. To see these accessor methods, right-click the logical node and choose Open. The Source Editor opens to display the generated bean class source code. Near the end of the code, you see something like the following example:

```
public abstract java.lang.Integer getCustomerNum();
public abstract void setCustomerNum(java.lang.Integer
    customerNum);
public abstract java.lang.String getDiscountCode();
public abstract void setDiscountCode(java.lang.String
    discountCode);
public abstract java.lang.String getName();
public abstract void setName(java.lang.String name);
```


The CMP fields themselves are declared in the deployment descriptor. To see them, select the logical node's bean class, right-click, and choose View Deployment Descriptor. Here is a partial example of a deployment descriptor's XML code that declares the CMP fields `customerNum`, `discountCode`, and `name` for a CMP entity bean whose persistence plan, or abstract schema, is known by the name `Customer`:

```
<abstract-schema-name>Customer</abstract-schema-name>
<cmp-field>
  <field-name>customerNum</field-name>
</cmp-field>
<cmp-field>
  <field-name>discountCode</field-name>
</cmp-field>
<cmp-field>
  <field-name>name</field-name>
</cmp-field>
...
<primkey-field>customerNum</primkey-field>
```

If you have not specified any persistent fields, your CMP entity bean contains one CMP field, called `defaultField`, and the accessor methods on that field. This field is automatically made the primary key.

After you use the wizard to define the CMP entity bean, you can always add CMP fields. Add a field by selecting the bean's logical node, right-clicking, and choosing Add CMP Field. You can also designate a new CMP field as a primary key.

After creating your bean, you can still change the name of a CMP field. Make the change only by selecting the CMP field under the logical node, right-clicking, and choosing Rename. The EJB Builder prompts you for the extent of the change.

The rest of your CMP entity bean's persistence (the actual SQL statements your bean needs for assembly and deployment within the server) is handled later. You make your bean portable across application and database servers by adding select or finder methods with EJB QL statements. These EJB QL statements are kept in the deployment descriptor for the EJB container and application server that you select. During deployment, this EJB QL code is converted to server-specific database-access code. Since most persistence is implemented using relational databases, SQL is the usual target language. In the case of the J2EE reference implementation (RI) server, the converted SQL code is displayed in the server's tabbed pane of the bean's property sheets.

For details on preparing enterprise beans for deployment, see Chapter 8. For details on writing EJB QL code, see the IDE's online help.

Primary-Key Class and Required Methods

The EJB Builder wizard either mapped the database table's primary key to a primary-key field in your CMP entity bean or let you define one or more primary-key fields. If your bean had a composite primary key, the wizard generated a primary-key class. (If not, your bean doesn't contain a primary-key class. Later, when you need to create the primary-key field that maps to the database table's primary key, you must first create the primary-key class. See "Creating a New Primary Key" on page 95.)

The primary-key class contains the set of data needed to uniquely identify an instance of the bean. If the bean has a single primary-key field, the wizard uses the field's class as the bean's primary-key class. If the bean has a composite primary key (one made up of more than one persistent field), the wizard generates a primary-key class with fields of the same name and type.

In addition, if a new primary-key class was generated, the EJB Builder inserted two methods required for the container, as follows:

```
public boolean equals(java.lang.Object otherObj) {  
    ...  
}  
public int hashCode() {  
    ...  
}
```

The `equals` method compares objects with the same `id` value, that is, keys that evaluate to the same hash code. Call this method with a key value as its parameter. The method must ascertain whether the passed key value matches the current key value.

The `hashCode` method converts a key to an integer value so that the key can be looked up quickly in a hash table. Make sure this method returns a hash-code key for the current instance. The value doesn't need to be unique, but your entity bean will have better performance when there is little chance of a duplicate hash value.

The primary-key class must implement the `java.io.Serializable` interface, not the `java.rmi.Remote` interface.

If you plan to use the IDE's testing feature to exercise your CMP entity bean's methods, here are a couple of tips:

- Include either an all-fields constructor in the bean's primary-key class or set methods for the class members.
- Define an appropriate `toString` method to make the test application's display easier to interpret.

For more information on using the testing feature, see Chapter 9.

A CMP Entity Bean's Life-Cycle Methods

The wizard adds the following default life-cycle methods to the bean class of any entity bean:

```
public void setEntityContext(EntityContext context) {
    this.context = context;
}
public void unsetEntityContext() {
    context = null;
}
public void ejbActivate() {
}
public void ejbPassivate() {
}
public void ejbLoad() {
}
public void ejbStore() {
}
public void ejbRemove() {
}
```

Table 4-2 describes the purposes of these methods in a CMP entity bean.

TABLE 4-2 Purpose of Default Life-Cycle Methods in a CMP Entity Bean Class

Method	Purpose
setEntityContext	This method lets you store the EntityContext reference in a field and populate nonpersistent fields. You can use it to allocate resources that are independent of the EJB object and last for the entity bean's lifetime (resources such as a database-connection factory). By default, the EJB Builder wizard generates code that assigns the EntityContext to a nonpersistent field named context.
unsetEntityContext	This method lets you deallocate resources and release memory used by the entity bean instance, before the container destroys the instance. By default, the EJB Builder wizard sets the value of the context field to null.
ejbActivate	This method initializes the bean, prepares it for use, and acquires the resources needed by the instance.
ejbPassivate	Before the bean instance is returned to the generic instance pool, this method releases the resources the bean was using.

TABLE 4-2 Purpose of Default Life-Cycle Methods in a CMP Entity Bean Class (*Continued*)

Method	Purpose
<code>ejbLoad</code>	In a CMP entity bean, this method needs no further coding. The container calls <code>ejbLoad</code> on a bean instance in the ready state and synchronizes the bean instance's state with the state of the entity in the underlying database.
<code>ejbStore</code>	In a CMP entity bean, this method needs no further coding. The container calls <code>ejbStore</code> on a bean instance in the ready state. The container synchronizes the state of the entity in the underlying database with the bean's state.
<code>ejbRemove</code>	In a CMP entity bean, this method does some cleaning up to prepare for the container's data deletions.

Completing Your CMP Entity Bean

To complete your CMP entity bean, do the following:

- Define a create method if you want to let the bean's client insert data into the database. An entity bean can have more than one create method.
- If necessary, add or replace a primary key.
- Define all business methods that your bean needs.
- Define any finder methods that your bean needs in addition to `findByPrimaryKey`.
- Define one or more home methods if a bean needs to perform an operation that does not depend on any given bean instance.
- Define one or more select methods, if you want your CMP entity bean to query other beans within the same EJB module or to query the database and return a local or remote interface.
- Add code, if necessary, to complete your CMP entity bean's `setEntityContext`, `unsetEntityContext`, `ejbActivate`, `ejbPassivate`, and `ejbRemove` methods.

You might need to add one or more CMP fields if, while using the wizard, you didn't specify all the fields that your bean needs.

Make the basic parts of your additions in the Explorer by using the GUI tools that the IDE provides under the logical bean node. You provide the content of these methods as follows:

1. Name the method and completely define the method signature within the appropriate dialog box. Select the logical node, right-click, and choose Add Create Method, Add Business Method, Add Finder Method, Add Home Method, or Add Select Method. The EJB Builder propagates your method to the right classes of the CMP entity bean.
2. Finish coding the method body within the Source Editor.

Using Recommended Approaches When Working With Enterprise Beans

Appendix A discusses the best ways to make changes in your enterprise beans, and the errors and anomalies that you might see if you use other approaches. As a general rule, you should work through the logical node rather than the individual class nodes, use the bean's property sheets or the Customizer dialog box to edit methods, and use the IDE's Source Editor to complete or edit any bean code that isn't available to you through one of the dialog boxes.

Defining Create Methods

Your entity bean can have more than one create method. In each bean, the home interface must have a create method, and the bean class must have corresponding `ejbCreateXxx` and `ejbPostCreateXxx` methods. When you use the recommended process, the IDE ensures that these methods are generated and propagated correctly.

In a CMP entity bean, the `ejbCreateXxx` method typically does the following:

- Validates client-supplied arguments.
- Initializes the instance's variables (in a CMP entity bean, the CMP fields). The container calls `ejbCreate` just before writing the bean's CMP fields to the database.

The `ejbPostCreateXxx` method, which the IDE adds automatically, gives the programmer the opportunity to forward information about the EJB object (such as the home or remote interface) to any other enterprise beans that need to reference it. The method can access the remote interface through `EntityContext`, which the method receives from the container as a parameter. This method is typically used to create dependent beans. For example, the `Order` bean's `ejbCreateLineItem` method might create the given line items in the `ejbPostCreateXxx` method.

Define a new create method as follows:

1. **Select the CMP entity bean's logical node, right-click, and choose Add Create Method.**

The Add New Create Method dialog box appears.

2. **Name your create method, using (if you like) an extension after create.**

Now you need to add parameters to your method.

3. **In the dialog box, click Add.**

4. **In the Enter Method Parameter dialog box, specify the parameter's type and name.**

In a CMP entity bean, the create method must return a primary-key type or the same type as the primary key. As shown in the code example that follows, the method signature in the bean class specifies the primary-key type. However, the method body should return null, because the container manages the primary key of a CMP entity bean.

```
public PrimaryKeyType ejbCreate(param1...) throws excl
```

5. **Click OK.**

The method you added now appears in the bean class code as `ejbCreateXxx` and in the home interface as `create`. The method `ejbPostCreateXxx` also appears in the bean class. If you happen to have the Source Editor open while you are adding the method, notice that the code is immediately updated.

An example follows of `ejbCreate` and `ejbPostCreate` methods generated in the bean class:

```
public String ejbCreate(java.lang.String custname)
    throws CreateException {
}
public void ejbPostCreate(java.lang.String custname)
    throws CreateException {
}
```

6. **Use the Source Editor to add the return statement and all other necessary code to your new create method.**

The create method in CODE EXAMPLE 4-1 is designed for a web application that lets a bank's staff look up customer responses to a survey on service quality in the bank's branch offices. In the code example, an instance of a CMP entity bean is created with the fields `custname`, `branchno`, and `response`.

CODE EXAMPLE 4-1 Example of a Create Method in a CMP Entity Bean Class

```
public CustomerSurveyKey ejbCreateResponse(java.lang.String custName,  
    java.lang.String branchNo, java.lang.String response)  
    throws CreateException {  
    if ((branchNo == null) || (custName == null)){  
        throw new CreateException("Both the branch number and  
            the customer name are required.");  
    }  
    setCustName(custName);  
    setBranchNo(branchNo);  
    setResponse(response);  
  
    return null;  
}
```

Adding or Replacing a Primary Key

If you have deleted your entity bean's primary-key class or if you need to add a primary key to the class, use the property sheet as follows:

- 1. Select the logical node, right-click, and choose Properties.**

The property sheet for your entity bean appears.

- 2. Select the Primary-Key Class field. Click the ellipsis (...) button.**

The Property Editor dialog box appears.

- 3. Select an existing field or a class you have defined. Click OK.**

The Primary-Key Class field now displays the return type of the new field or class.

Creating a New Primary Key

If you need to add a new primary key to an entity bean that has no primary-key class, you must first add the new primary-key field or fields to the bean. Then, you use the EJB Builder wizard to create a new entity bean with a primary-key class, for temporary use. (We'll call this the temporary bean.) Finally, you specify that the original bean is to use the new primary-key class.

Follow these steps:

- 1. In the Explorer, find the package of the entity bean that needs a primary-key class.**
- 2. Right-click the package and choose New → J2EE → CMP Entity EJB.**

3. In the wizard, do the following:

- a. Specify only the field or fields that your existing entity bean needs for its primary key.**
- b. In the next-to-last panel, if you like, you can rename the primary-key class to correspond to the original bean that will be using the class.**

For example, if your bean is named `Account`, you might rename the class `AccountKey`.

4. Click Finish.

Your temporary bean has now been created. Continue as follows:

5. In the Explorer window, if you like, you can delete the temporary bean's classes and logical nodes *except* for the primary-key class.

6. Right-click the original bean's logical node and choose Properties.

7. In the property sheet's Properties tabbed pane, click the Primary Key Class field and then the ellipsis (...) button.

The Primary Key Class dialog box opens.

8. Click this option: Select an existing user-defined class.

A file chooser opens.

9. Navigate to the new primary-key class, select it, and click OK.

In the property sheet, the Primary Key Class field changes to show the name of the new primary-key class.

A warning or error badge might appear on your bean's logical node. Disregard it for the moment. Dismiss the property sheet when you are done.

10. Right-click the original bean's logical node and choose Validate EJB or Error Information.

The IDE points out any errors you must resolve.

11. Fix all errors and revalidate or compile your bean.

The two methods required by the primary-key class, `equals` and `hashCode`, are not regenerated during this process. Typically, therefore, you must change the class names in the `equals` method. You might need to specify a different parameter type in the `findByPrimaryKey` method and a different return type for any `ejbCreate` methods in the bean class.

12. Save your work.

Handling Foreign Keys

If you need to maintain a relationship between CMP entity beans that is implemented as a foreign key, or if your bean needs multiple access to the data store, you should consider creating a set of related CMP entity beans instead of creating the beans separately. The EJB Builder wizard lets you create an entire related set at once, with the source tables' foreign keys intact and represented as container-managed relationships (CMRs) between beans. See Chapter 5 for details.

Defining Business Methods

You add a business method to your CMP entity bean to perform the business logic that needs to be encapsulated within the entity bean. Usually, a business method manipulates one or more persistent fields, but it doesn't access the database directly. The task of the business method is to update the instance variables. The methods `ejbLoad` and `ejbStore` are called by the EJB container as required by the semantics of the transaction, and the variables are thus written to the database.

Note – It is best to keep business logic separate from database access code.

Define a business method as follows:

1. **Select the logical node, right-click, and choose Add Business Method.**

The Add New Business Method dialog box appears.

2. **Type a name for the method, and specify a return type, parameters, and exceptions. Click OK to dismiss the dialog box.**

3. **Finish the method's coding in the Source Editor.**

Or, in the Add New Business Method dialog box, you can simply type a name for your new business method, click OK to dismiss the dialog box, and finish the coding in the Source Editor.

The business method in CODE EXAMPLE 4-2 is designed for the same application mentioned in CODE EXAMPLE 4-1. In this example, the bank customer's phoned-in comments are retrieved from the database.

CODE EXAMPLE 4-2 Example of a Business Method in a CMP Entity Bean

```
public java.lang.String retrieveComments() {  
    return phoneResponse();  
}
```

To see any method you have created for an enterprise bean, expand the bean's logical node and navigate to the sub-node for the kind of method you want to view. Right-click the method's node and choose Open. The Source Editor opens the class directly to the method code.

Adding Finder Methods

The EJB Builder wizard generates a default `findByPrimaryKey` method for you. However, if you want your CMP entity bean to run additional queries, you must define additional finder methods.

Alternatively, if you want a query to return the value of a related entity bean's persistent field, or if you don't need the method to be invocable by a client, you can use a select method. See "Defining Select Methods" on page 100 for more information.

Add a finder method as follows.

- 1. Select the bean's logical node, right-click, and choose Add Finder Method.**

The Add New Finder Method dialog box appears.

- 2. Type a name for the method, starting with `find`.**

- 3. Select one of the following return types:**

- A single object, which is shown with a default name
- A collection
- An enumeration

- 4. Specify parameters and exceptions.**

- 5. Type EJB QL statements into the Select, From, and Where fields.**

For detailed EJB QL syntax and examples, see the IDE's online help. Also see "If Your EJB Module Contains an EJB 1.1 CMP Entity Bean" on page 168 for special instructions about the `WHERE` statement in an older CMP entity bean's finder method.

If you're not ready to input EJB QL statements at this point in the process, you can turn off the compiler's requirement for EJB QL code. See "Compiling and Validating Enterprise Beans" on page 199. However, you must supply the correct EJB QL statements before you deploy the bean to the RI.

A method's EJB QL code goes into the deployment descriptor. You can edit or add EJB QL statements using the Customizer or the method's property sheets. However, normally, you shouldn't edit the bean's deployment descriptor directly.

- 6. Click OK when you are finished.**

Open the Source Editor directly to a finder method as follows: Go to the CMP entity bean's logical node and navigate to Finder Methods. Select the finder method you want, right-click, and choose Open. In the Source Editor, the home interface class opens to the finder method. (Finder methods are not declared in the bean class.)

Here are two tasks a finder method in the `Account` example might do:

- Find an `AccountEJB` instance that holds data for a specific account and returns a remote object for that instance. For that purpose, you select the account by account number.
- Find an `AccountEJB` instance for every overdrawn account and return a collection of their remote objects. For that purpose, you select for accounts with negative balances.

You can edit the finder method either using the Customizer (right-click the method's node and choose Customize) or using the method's property sheet (right-click the method's node and choose Properties).

Defining Home Methods

You can use a home method to perform an operation that does not depend on any given instance of the entity bean. A home method, similar to a static method, contains business logic that applies to all beans of a given class. (A business method, on the other hand, has an identity and logic unique to one instance of the entity bean.) A home method does not access the bean's persistence state (instance variables) or container-managed relationships.

For example, assume that your CMP entity bean `Invoices` reflects customer invoices, and each invoice shows the amount the customer has paid. If you want to see the total of all outstanding invoices, you can add a home method called `getAmountDue` that iterates through the collection of bean instances and invokes a business method to sum the balance due for all active invoices.

Define a home method as follows:

- 1. Select the logical node, right-click, and choose Add Home Method.**
The Add New Home Method dialog box appears.
- 2. Type a name for the method.**
- 3. Specify the return type, parameters, and exceptions.**
- 4. Click OK when you are finished.**

Alternatively, you can simply type a name for your new home method, click OK, and finish the coding in the Source Editor.

If the CMP entity bean has two client views (both kinds of interfaces), the EJB Builder asks you whether to include the home method on the local home interface, the (remote) home interface, or both.

The IDE adds the home method to the home interface or interfaces, and it adds the corresponding `ejbHome` method to the bean class.

Defining Select Methods

You can add one or more select methods if you want your CMP entity bean to query the database and return a local or remote interface (or a collection of interfaces), or if you want the method to return the value of a related entity bean's persistent field (or a collection of those values). A select method is related directly to the `get` method that is created when a relationship between CMP entity beans is defined, and the select method can be invoked only by a method (usually a business method) within the entity bean class. A select method is not exposed in any remote-type interface, and so it can't be invoked by a client.

Add a select method as follows:

1. **Select the bean's logical node, right-click, and select Add Select Method.**
2. **Type a name for the method starting with `ejbSelect`.**
3. **Select one of the following return types:**
 - A single object, which is shown with a default name
 - Any other type on the combination box list
4. **Specify parameters and exceptions.**
5. **Type EJB QL statements into the Select, From, and Where fields.**

For detailed EJB QL syntax and examples, see the IDE's online help.
6. **Click OK when you're finished.**

You can edit the `select` method either using the Customizer (right-click the method's node and choose Customize) or using the method's property sheet (right-click the method's node and choose Properties).

The EJB QL code that you supply goes into the deployment descriptor. You can edit or add EJB QL statements using the Customizer or the method's property sheets. However, you can't edit the bean's deployment descriptor directly.

Defining Additional Fields

After creating your CMP entity bean, you can add CMP fields as follows:

- **Select the logical node, right-click, and choose Add CMP Field.**

Note – Don't use the Source Editor to code the field directly in your bean class. The IDE has no way to identify the field as persistent in the deployment descriptor.

After Creating Your CMP Entity Bean

Your CMP entity bean is now finished except for a few steps that prepare the bean to work in its eventual environment. These final steps are described in Chapter 8.

Recommendations for working with finished enterprise beans are given in Appendix A.

Further Reading

Enterprise beans can be a very powerful and flexible part of your application. Creating the basic parts of an enterprise bean can be very simple, especially with a tool like the Sun ONE Studio IDE. However, completing the bean so that it fulfills the needs of your application can be very complex. For details, refer to the following documents:

- *Enterprise JavaBeans Specification*, version 2.0 at:
<http://java.sun.com/products/ejb/docs.html>
- *Java 2 Platform, Enterprise Edition Tutorial* at:
http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/CMP3.html
- "CMP Example Overview" by Beth Stearns
http://java.sun.com/j2ee/sdk_1.3/techdocs/release/CMP-RI.html

Developing Sets of Related CMP Entity Beans

Many J2EE applications contain related entity beans that use container-managed persistence (CMP entity beans). That is, two CMP entity beans in an application might contain a bean-to-bean relationship field, representing the way different entities or tables in a database or a database schema might contain a related column. For example, a schema might include the tables *Customer*, *Order*, *LineItem*, and *Part*. *Order* has a foreign key to *Customer*, *LineItem* has a foreign key to *Order*, and *LineItem* also has a foreign key to *Part*.

This chapter describes how to use the EJB Builder in the Sun ONE Studio 4 IDE to create a set of related CMP entity beans all at once, along with the necessary interfaces. You can generate CMP entity beans from all entities in a database or schema model, or you can select a subset of the entities. The wizard automatically places the generated beans in an EJB module (see Chapter 8 for more information on modules).

The wizard also considers any relationships between the entities in the data source and preserves the relationships in the resulting CMP entity beans as logical entities called CMR fields. In fact, the IDE does not allow you to accidentally overlook a relationship between two database tables.

The wizard automates or prompts you for all the tasks involved in creating the infrastructure of a set of related CMP entity beans.

When programming a set of related CMP entity beans, you have many options in addition to those described in this chapter. For more information, refer to the resources listed in “Before You Read This Book” on page xviii, or to one of the many excellent texts on programming enterprise beans.

Using the EJB Builder With Sets of Related CMP Entity Beans


The EJB Builder is a collection of wizards, property sheets, and editors with which you can build enterprise beans consistently and easily. You get the most comprehensive support and, in general, the fastest path to bean completion, if you use the EJB Builder's wizards and the approach recommended in this chapter. The methodology described here takes full advantage of the IDE's ability to ensure consistency and its adherence to the J2EE standard.

Creating All Related CMP Entity Beans at Once

For best results, use the EJB Builder to program CMP entity beans by:

- **Creating a set of CMP entity beans and their required classes.** After using the EJB Builder wizard, you have the framework of your set of related CMP entity beans. Each bean, with its necessary class, interfaces, and a logical node, is shown in the Explorer's Filesystems tabbed pane. The wizard generates declarations for the interfaces. Each CMP entity bean's generated bean class contains declarations of required methods, as well as persistent fields matching the columns in the data source.

The wizard prompts you to either include or explicitly exclude related beans. This way, you can't miss seeing a bean-to-bean relationship.

The logical node is the best place from which to work with an entity bean. All logical nodes appear in the Explorer with this icon: 

- **Adding methods, parameters, and exceptions.** Use the IDE's GUI support as described later in this chapter. You can work partly in the GUI and partly in the Source Editor to complete each CMP entity bean's method implementations. For example, you can add a method to a bean by using a dialog box available from the contextual menu or by directly editing the set of required classes.
- **Setting values in a bean's deployment descriptor.** Use the entity bean's property sheet, available at the logical node, to edit properties for deployment purposes.

You can validate an individual CMP entity bean from the bean's logical node. You can also use the IDE's testing feature against an individual bean.

Creating a Set of Related CMP Entity Beans One at a Time

If you wanted to, you could build a set of related CMP entity beans by hand, generating the beans individually, adding them to an EJB module, and using the module's property sheets to declare the relationships (CMRs) between beans. However, the EJB Builder wizard does this work for you automatically, and any relationships between beans are more likely to be complete and accurate if you use the wizard.

If you can't take advantage of the EJB container's persistence management, and you must create a set of entity beans that manage their own persistence (BMP entity beans), then you must manually code all relationships between the beans. See TABLE 4-1 for the differences between CMP and BMP entity beans; see Chapter 6 for details on creating BMP entity beans.

The rest of this chapter addresses how to create a set of related CMP entity beans all at once, using the wizard, and issues to consider during development.

Defining a Set of Related CMP Entity Beans

The EJB Builder wizard automates much of the task of creating the components of your set of related CMP entity beans. For each CMP entity bean, the wizard does the following:

- Creates the minimum required classes, which consist of a bean class and the interfaces you choose: local only, or both local and remote
- Creates a primary-key class in the CMP entity bean, if a table you selected requires a composite primary key or uses a simple Java type for the primary key
- Creates relationships between CMP entity beans. These relationships appear as accessor methods that return collections
- Generates an EJB module to hold your set of related CMP entity beans

To define a set of related CMP entity beans, you take the following steps:

1. Select or create a package to contain the beans and the EJB module.
2. Use the EJB Builder wizard to generate the infrastructure of your set of related CMP entity beans.

3. As appropriate, add create, business, finder, select, and home methods to each CMP entity bean's code.
4. Complete the bodies of the methods you added.
5. Add deployment information in the EJB module's property sheets.

These basic steps are explained in detail next.

Creating a Package

If you need to create a package to house your set of related CMP entity beans, select a filesystem, right-click, and choose New Java Package.

Preparing to Use a Database or Schema

You need to decide whether to model your set of related CMP entity beans on a database or on a database schema (a snapshot of a database). The EJB Builder wizard maps columns from a table of the database or schema to create the persistent fields in your related CMP entity beans. Both choices provide the same result in your finished entity bean.

When you build a set of related CMP entity beans, the EJB Builder also preserves any relationships between the database tables and carries them over into the CMP entity beans in the set.

Consider the following when deciding which form of data source to use:

- **Generating a set of related CMP entity beans from a live database.** Assuming that you have direct access to the database itself, and that contention among database users is not a problem, you might want to use the direct database connection to create your set of related CMP entity beans. If so, you must already have the database up and running before you start the EJB Builder wizard.
- **Generating a set of related CMP entity beans from a database schema.** If database access is restricted but schema objects have been made available, you might want to take a table from a schema. If so, the schema must be available to you from the IDE's Explorer window. If you don't already have a schema, you must capture one from a database.

For details on starting and stopping the PointBase database server that is included in the IDE, and on capturing a database schema, see "Having a Data Source Ready" on page 73.

While you're creating a set of related CMP entity beans, you need access to the database, so the server must be running at least until you finish with the EJB Builder wizard. (Later, when the application that uses the CMP entity beans is being deployed and is executing, the database and application servers must also be running, if you want the plug-in to create tables.)

The EJB Builder wizard provides information to the application server plug-in about how the original database mapping was derived. If appropriate, the plug-in incorporates this information into its default mapping. EJB containers vary in how they treat column-to-field mappings. For details, refer to the documentation for your container and server plug-in.

Starting the EJB Builder Wizard

When you're ready to create a set of related CMP entity beans, do as follows:

1. **In the IDE's main window, choose View → Explorer to open the Explorer window.**
2. **In the Filesystems pane of the Explorer, select the package or filesystem where you want your CMP entity bean to reside.**
3. **Right-click and choose New → J2EE → Related CMP Entity EJBs.**

The EJB Builder wizard appears, displaying New Wizard – Related CMP Entity EJBs in the window's title bar. Notice that the panel on the left shows the current step and the steps you still must complete before your set of related CMP entity beans is created.

Generating the Bean Set's Infrastructure

In the EJB Builder's Specify EJB Module Name and Data Source pane, you must name the EJB module that will contain your set of related CMP entity beans. You must also decide, as shown in FIGURE 5-1, where your CMP entity beans will get their persistent fields and relationships.

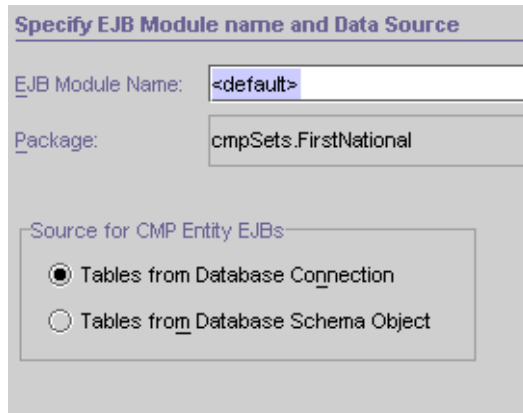


FIGURE 5-1 Selections in the EJB Builder Wizard for a CMP Entity Bean Set

To make these basic decisions about your bean set, do as follows in the Specify EJB Module Name and Data Source pane:

1. Type a name for the module.

If you have used an EJB Builder wizard to create an individual bean, as described in Chapter 4, you remember that the input field was for the bean's name. However, individual beans in a set of related CMP entity beans are named by the wizard a little later in the generation process. Here, type a name for the module that will enclose your set of related CMP entity beans.

2. Select a database source:

- Select Tables from Database Connection if the CMP entity beans you are creating will represent tables from an existing database. (The database must already be up and running.) See the next section.
- Select Tables from Database Schema Object if you want to use an existing schema. (The schema must have already been created, and it must be in a filesystem to which you have access through the IDE's Explorer.) See "Using a Database Schema Object" on page 114.

3. Click Next.

Using a Database Connection

You should now be in the wizard's Specify the Database Connection pane, having selected Tables From Database Connection in the first wizard pane.

Make sure the database you're using is running. If you're using a database that is not supplied with the IDE, make sure that the database driver files are in the Sun ONE Studio 4 lib/ext directory, and start the database server.

In the Specify the Database Connection pane, do as follows:

1. Click either Existing Connection or New Connection:

- Select Existing Connection if you want to use an installed or supplied database and the connection is defined but not active. Select a database from the combo box. A login dialog box appears. Type the information required by your database and click OK.
- Select New Connection if you have a database installed but no connection has been defined. Select a database driver from the combo box. A login dialog box appears. Type the information required to connect to your database and click OK.

The tables that you can use to build your set of related CMP entity beans appear in the next wizard pane, Select Database Tables.

2. From the Available Tables list on the left, select the tables you want and add them to the Selected Tables list on the right.

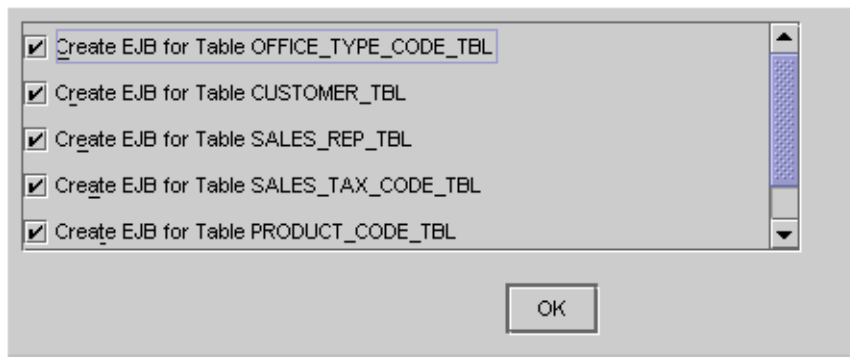
If you like, the EJB Builder can preserve all relationships between tables and replicate them in your set of related CMP entity beans.

Or, you can exclude any tables you don't need, which means that any relationships between those tables and the ones you have selected are gone. In the latter case, the EJB Builder treats foreign-key columns and non-foreign-key columns in the same way.

3. Click Next.

A table that you did select might have a foreign key that references a table you didn't select. If this is the case, a warning dialog box appears, listing all the related tables you didn't select. Consider whether any CMP entity bean in your set will ever need access to data in any of those unselected tables.

For example, part of this dialog box is shown next. It contains a list of tables that you did not select in the previous dialog box. Notice that by default all the tables are checked for inclusion.



(The tables shown in this dialog box are those that are left over from the initial selection list, shown in the previous dialog box, and that are accessible using foreign keys. The set of tables you selected plus the set of tables in this dialog box constitute the entire set of tables that would allow all foreign keys to be included in CMRs.)

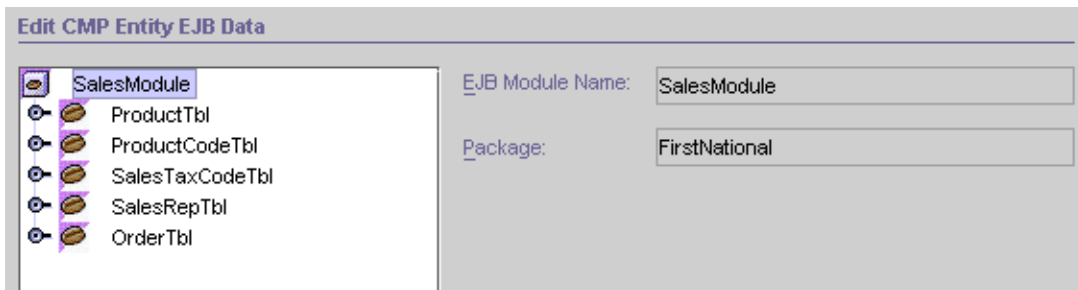
If you don't add a particular table to your set of related CMP entity beans now, you can always do it later. However, at that point, you will also have to specify its relationships with other CMP entity beans. If you think you might need the table and your application's performance isn't likely to be affected, the least complicated way is to add it now by leaving its checkbox selected.

4. In the warning dialog box, deselect the checkbox corresponding to any table you do *not* want represented in your set of related CMP entity beans.

Don't forget to use the scrollbar, if one appears.

When you click OK, all tables that you left selected become CMP entity beans in your set, in addition to any tables you selected explicitly in Step 2.

The Edit CMP Entity EJB Data pane appears, showing the EJB module that is about to be created, the CMP entity beans within that module, the EJB module name, and the package name. The interesting part of this pane is shown next.

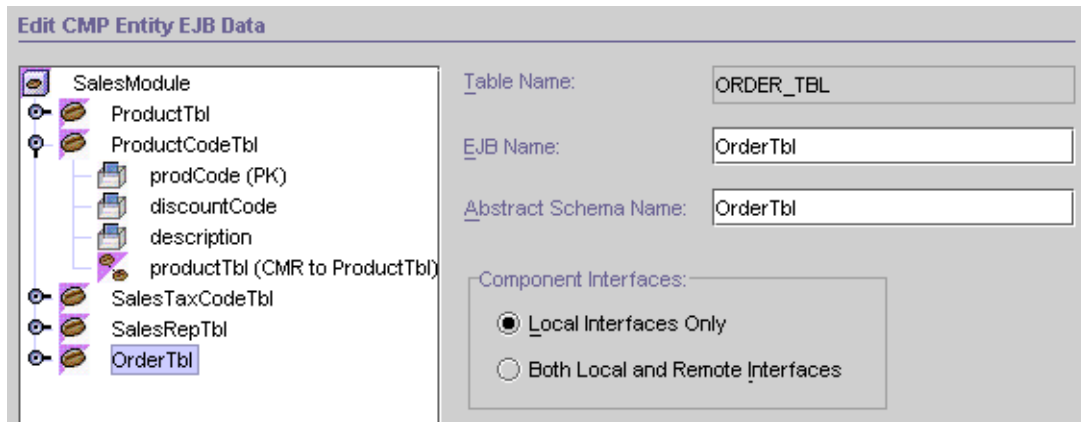




In this pane, if needed, you can select and edit a CMP entity bean, a field, or a container-managed relationship (CMR) between two CMP entity beans. The IDE has assigned default names and types to your beans and their fields, but you can make changes if necessary.

Note – This pane is here for your convenience. If you want, after finishing with the wizard, you can make the same types of changes using the logical nodes of the set of related CMP entity beans and the EJB module.

5. Edit one or more CMP entity beans, if necessary.

Select one of the included bean nodes and notice how the window changes. An example is shown next.



Notice the icons that identify CMP fields () and container-managed relationships ().

You can make changes in the following places:

- (Optional) In the EJB Name field, rename the selected CMP entity bean.
The EJB Builder wizard propagates the change to the bean class, the appropriate interfaces, and the relationships between the beans.
- (Optional) In the Abstract Schema Name field, rename the selected bean's abstract schema.

As you have been specifying your set of related CMP entity beans, you have also been automatically creating part of the set's deployment descriptor, namely, some declarative instructions to the container for handling your beans' persistence. These particular instructions are called the abstract persistence schema or abstract schema. When you add a finder or select method later (see "Adding Finder Methods" on page 98), the EJB QL queries in the method use the abstract schema name.

If you want, you can assign the schema a different name, but the default, which is the bean name, is recommended.

- (Optional) Use the Component Interfaces radio buttons to specify different interfaces for the selected bean.

Unless you select Both Local and Remote Interfaces, each of the CMP entity beans in your module is automatically given only a local interface and a local home interface. If a CMP entity bean will be used by beans in another container (or, to be more specific, in another JVM), then the bean needs both local and remote interfaces.

6. Edit one or more CMP fields, if necessary.

Select a CMP field and notice how the window changes. An example is shown next.

Edit CMP Entity EJB Data

Left Pane (Tree Structure):

- SalesModule
 - ProductTbl
 - ProductCodeTbl
 - prodCode (PK)
 - discountCode
 - description**
 - productTbl (CMR to ProductTbl)
 - SalesTaxCodeTbl
 - SalesRepTbl
 - OrderTbl

Right Pane (Field Configuration):

- Column Name: DESCRIPTION
- Column Data Type: VARCHAR
- Primary Key: ☐
- CMP Field Name: description
- CMP Field Type: java.lang.String

If necessary, make changes in the following places:

- (Optional) In the CMP Field Name field, overwrite the default name with another name.

The EJB Builder wizard propagates the change to the bean class and the appropriate interfaces, maintaining any relationship between this and another field.

- (Optional) In the CMP Field Type field, select another type for the field.

7. Edit a relationship between two CMP entity beans, if necessary.

The EJB Builder wizard shows any relationship between two beans as a separate node. This node does not represent an actual object. It's just a logical node, labeled to show that a container-managed relationship (CMR) exists between the bean you've selected and the bean named in the label. This relationship between CMP entity beans is like the relationship between tables with a foreign key.

In the example shown next, the ProductCodeTbl bean has a CMR with the ProductTbl bean, since the corresponding tables have a foreign key in common.

Select a CMR and notice how the window changes. An example is shown next.

Edit CMP Entity EJB Data

SalesModule

- ProductTbl
- ProductCodeTbl
 - prodCode (PK)
 - discountCode
 - description
 - productTbl (CMR to ProductTbl)
- SalesTaxCodeTbl
- SalesRepTbl
- OrderTbl

EJB Relation Name: ProductTbl-ProductCodeTbl

From ProductTbl to ProductCodeTbl

Role Name: ProductTbl
CMR Field Name: productCode or ☐ None
Multiplicity: Many
Cascade Delete: ☐

From ProductCodeTbl to ProductTbl

Role Name: ProductCode
CMR Field Name: productTbl or ☐ None
CMR Type: java.util.Collection ▼
Multiplicity: One

Make any necessary changes in the following places:

- (Optional) In the EJB Relation Name field, change the name that the wizard has assigned to the relationship between the two beans.
- (Optional) In the Role Name field, change the role name.
Notice the two CMP entity beans described in the right side of the pane. The role name describes the role of the bean shown in the top of the pane as it relates to the bean in the bottom of the pane.
- (Optional) In the CMR Field Name field, change the field name.
The wizard has given each bean's CMR field a name that enables the beans to navigate their relationship. A foreign key, for example, might be mapped to this CMR field. A relationship can be unidirectional (that is, the two related CMP

entity beans have only one CMR field between them) or bidirectional (the pair have two CMR fields). In the example shown here, the relationship is bidirectional, and so each bean has a differently named CMR field.

A CMR field name becomes an abstract method in the bean class. This abstract method doesn't do any work directly on an entity.

If two CMP entity beans have multiple relationships between them, it's a good idea to add semantic meaning to the CMR field names. Do that here.

- (Optional) Select the Cascade Delete checkbox if, when one bean's relationship record is deleted, you also want the other bean's corresponding record to be deleted.

This decision depends on the semantic meaning of the two beans' relationships. For example, an order might have several related line items. The relationship between an order and a line item should always use Cascade Delete because if the order doesn't exist, neither should any of the line items. However, in peer relationships, deleting the referenced side should not cause the referencing bean to be deleted.

The EJB container handles referential integrity when it uses CMR fields to operate on the relationship between two CMP entity beans.

8. When you are done, click Finish.

The infrastructure of your set of related CMP entity beans (the bean classes, the types of interfaces you specified, and the interbean relationships) is generated automatically by the EJB Builder. See "Looking at the Components of a CMP Entity Bean Set" on page 115 for the next step.

Using a Database Schema Object

You should now be in the wizard's Select Database Schema Object pane, having selected Tables From Database Schema Object in the first wizard pane (see FIGURE 5-1).

If you don't already have a schema, you can use the IDE's Database Schema wizard to create one, as discussed in "Capturing a Database Schema" on page 79. Make sure you can get to the schema through the IDE's Explorer.

In the Select Database Schema Object pane, you see the filesystems to which you have access through the Explorer. Your schema should be there. Do as follows:

1. Select the database schema that contains the tables you want represented in your set of related CMP entity beans, and click Next.

You see a side-by-side display that consists of the list of available tables in your database schema and a blank pane labeled Selected Tables.

2. **From the Available Tables list on the left, select the tables you want and add them to the Selected Tables list on the right.**

From this point on, the process and the GUI are just as described in “Using a Database Connection” on page 108:

- a. **When you have selected all your tables, click Next.**
 - b. **In the next dialog, deselect the checkbox of any table you don’t want in your set of related CMP entity beans. Click OK.**
 - c. **In the Edit CMP Entity EJB Data pane, edit beans, fields, or relationships as needed.**
3. **When you are done, click Finish.**

The infrastructure of your set of related CMP entity beans is generated automatically by the EJB Builder.

Looking at the Components of a CMP Entity Bean Set

The EJB Builder wizard generates the basic CMP entity bean classes for you and sets up the relationships between all the beans and their classes. FIGURE 5-2 shows how a typical set of related CMP entity beans and their EJB module appear in the Explorer’s Filesystems pane. In this example, the default, Local Interfaces Only, has been selected for a CMP entity bean whose references are to objects in the same container, while Both Local and Remote Interfaces has been selected for another CMP entity bean.

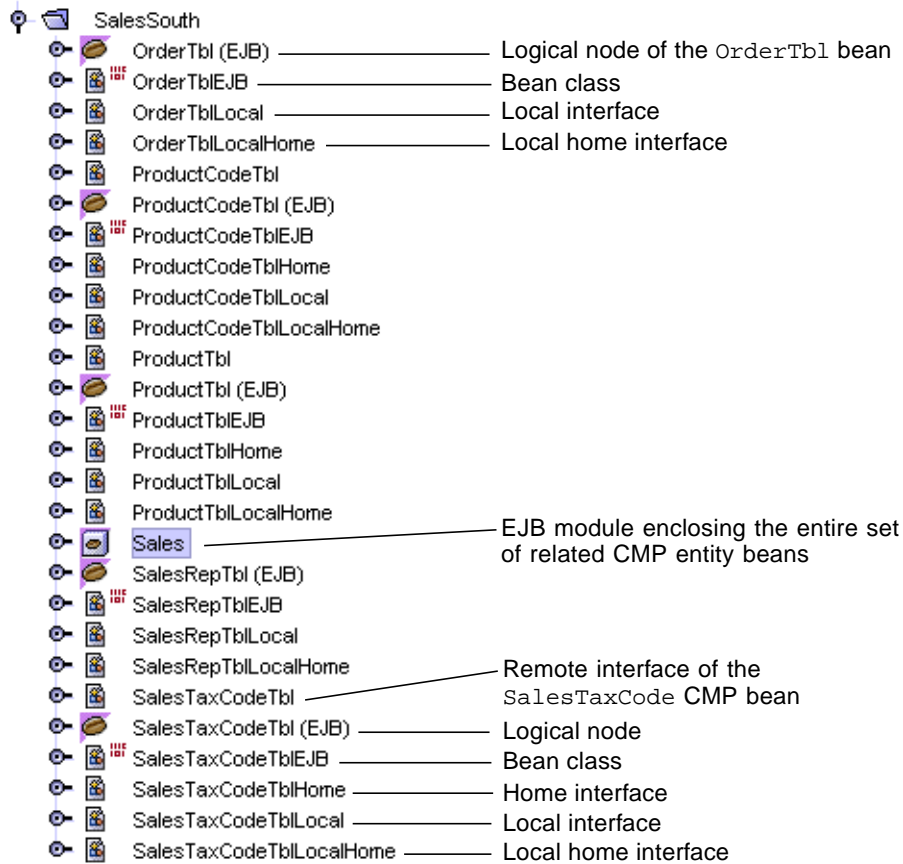


FIGURE 5-2 Default Classes of a Typical Set of Related CMP Entity Beans

With the exception of the EJB module node, the nodes in represent the same components as are described in “Looking at a CMP Entity Bean’s Classes” on page 84. Again, notice each bean’s logical node marked with a bean icon. Do all your editing in the logical node.

Expanding the EJB Module’s Node

The interesting difference lies in the bean-to-bean relationships in a set of related CMP entity beans. Those relationships are stored and displayed at the level of the EJB module, as shown in FIGURE 5-3.

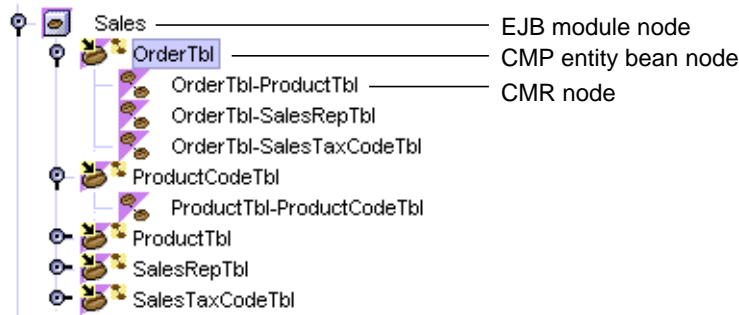


FIGURE 5-3 Expanded Nodes of an EJB Module Containing Related CMP Entity Beans

Notice the module's component bean nodes and relationship nodes. Notice also that the beans displayed in the EJB module are only logical links, not copies of the actual beans.

Reviewing the Generated Classes

Certain default methods, described in Chapter 4, are automatically placed in all CMP entity beans. For details, see the discussion in "Reviewing the Generated Classes" on page 88.

Completing Your Set of Related CMP Entity Beans

To complete your set of related CMP entity beans, do the following:

- Add any other CMP entity beans that your set needs, along with relationships to beans already in the set.
- Edit CMRs as needed.
- Define a create method if you want to let a bean's client insert data into the database. An entity bean can have more than one create method, and adding create methods is the same regardless of whether the CMP entity bean is on its own or in a set of related beans. Follow the instructions in "Defining Create Methods" on page 93.
- If necessary, add or replace a primary key. Again, you do this in the same way for all CMP entity beans. See the instructions in "Adding or Replacing a Primary Key" on page 95.

- Define all business methods that each of your beans needs, as explained in “Defining Business Methods” on page 97.
- Define any finder methods that your beans need in addition to `findByPrimaryKey`. Instructions are in “Adding Finder Methods” on page 98.
- Define one or more home methods if a bean needs to perform an operation that does not depend on any given bean instance. See “Defining Home Methods” on page 99.
- Define one or more select methods, if you want a CMP entity bean to query other beans within the same EJB module or to query the database and return a local or remote interface. Details are in “Defining Select Methods” on page 100.
- Add code, if necessary, to complete a bean’s `setEntityContext`, `unsetEntityContext`, `ejbActivate`, `ejbPassivate`, and `ejbRemove` methods.

You might need to add one or more CMP fields to a bean if all the needed fields were not generated for you.

Make the basic parts of your additions in the Explorer by using the GUI tools that the IDE provides under the logical bean node. Provide the content of a method as follows:

- Name the method and completely define the method signature within the appropriate dialog box. Select the logical node, right-click, and choose Add Create Method, Add Business Method, Add Finder Method, Add Home Method, or Add Select Method. The EJB Builder propagates your method to the right classes of the CMP entity bean.
- Finish coding the method body within the Source Editor.

Using Recommended Approaches When Working With Enterprise Beans

Appendix A discusses the best ways to make changes in your enterprise beans, and the errors and anomalies that you might see if you use other approaches. As a general rule, you should work through the logical node rather than the individual class nodes, use the bean’s property sheets or the Customizer dialog box to edit methods, and use the IDE’s Source Editor to complete or edit any bean code that isn’t available to you through one of the dialog boxes.

Adding a Bean to the Set

After generating your set of related CMP entity beans, you might find that you need another bean to represent a database table you didn't select. If so, follow these steps:

- 1. Decide which CMP entity bean you will add to the set.**

Since the wizard has already generated the CMP entity beans in your existing set, the CMP entity bean you add to the set must already be generated, either as a single bean or as part of another set of related CMP entity beans.

- 2. Select the EJB Module node of your set of related CMP entity beans, right-click, and choose Add EJB.**

The Add EJB to EJB Module dialog box appears. The tree view shows all filesystems mounted in the IDE's Explorer window.

- 3. Find the CMP entity bean you want to add to your set, select it, and click OK.**

The bean is added to your set of related CMP entity beans.

- 4. Expand the EJB module node to see the set of related CMP entity beans, including the CMP entity bean you just added.**

The bean you added appears without any relationships defined to other beans. In the following example, the new bean is `Customer`.



Now you need to add any relationships needed by your new CMP entity bean or by other beans in the set.

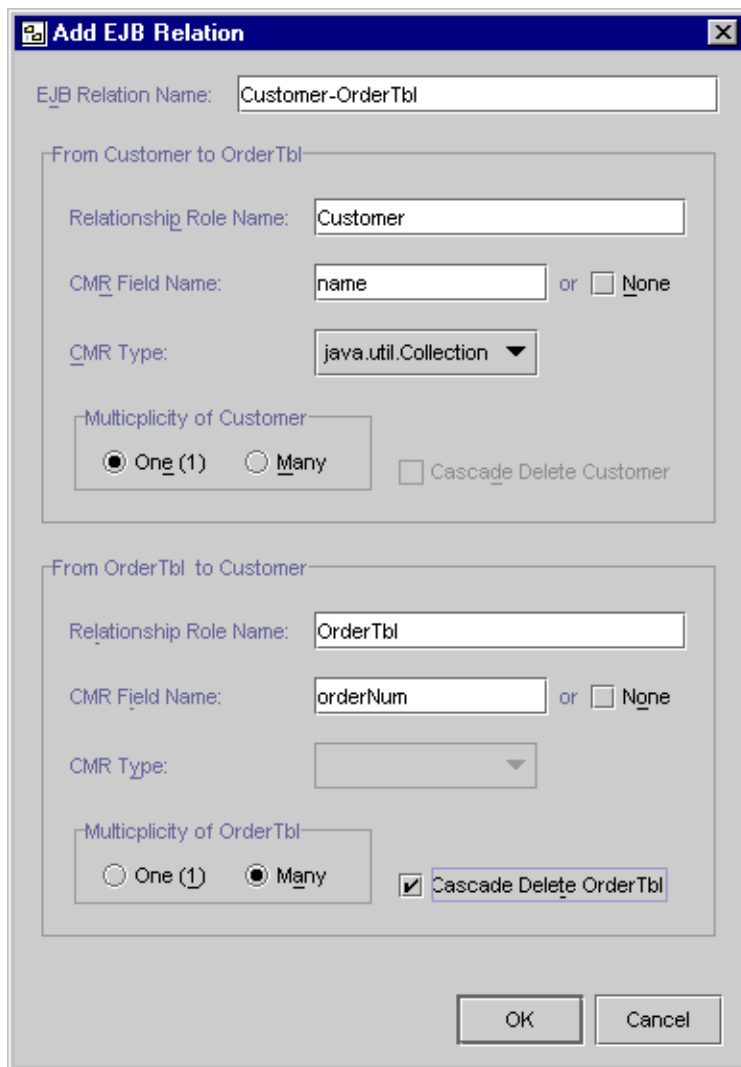
- 5. Under the EJB Module node, select the two CMP entity beans between which you want to add a relationship.**

In this example, the `Customer` and `OrderTbl` bean nodes are selected.



6. Right-click and choose Add EJB Relation.

The Add EJB Relation dialog box appears, as shown next. The dialog box has three main sections: a field that names the relationship between the two CMP entity beans, a section describing the first entity bean, and a section describing the second entity bean.



The dialog box is titled "Add EJB Relation" and contains the following fields and options:


- EJB Relation Name:** A text field containing "Customer-OrderTbl".
- From Customer to OrderTbl:**
 - Relationship Role Name:** A text field containing "Customer".
 - CMR Field Name:** A text field containing "name" with an "or" checkbox and a "None" option.
 - CMR Type:** A dropdown menu showing "java.util.Collection".
 - Multiplicity of Customer:** Radio buttons for "One (1)" (selected) and "Many".
 - Cascade Delete Customer:** An unchecked checkbox.
- From OrderTbl to Customer:**
 - Relationship Role Name:** A text field containing "OrderTbl".
 - CMR Field Name:** A text field containing "orderNum" with an "or" checkbox and a "None" option.
 - CMR Type:** A dropdown menu.
 - Multiplicity of OrderTbl:** Radio buttons for "One (1)" and "Many" (selected).
 - Cascade Delete OrderTbl:** A checked checkbox.
- Buttons:** "OK" and "Cancel" buttons at the bottom right.

7. Define the relationship between the two CMP entity beans.

The wizard has populated the fields with default information based on existing information for the two selected beans. If necessary, you can make changes in the following places:

- (Optional) In the EJB Relation Name field, rename the relationship between the two beans.
- (Optional) In the Relationship Role Name field for either entity bean, give another name to the role that the bean plays in the relationship.
- (Optional) In the CMR Field Name field for either entity bean, specify by another name the field that relates the two beans.
- (Optional) In the CMR Type field for either entity bean, choose another type. However, if you don't change the CMR Field Name, you probably should leave the CMR Type as it is.
- (Optional) In the Multiplicity of *BeanName* section for either entity bean, change the cardinality of that bean in the relationship. However, if you don't change the CMR Field Name or the CMR Type field, you probably should leave the Multiplicity section as it is. Notice that if the radio button labeled Many is selected, the Cascade Delete *BeanName* checkbox becomes available.

8. Click OK when you're done.

Under the EJB Module node, the bean to which you added a relationship now shows a relationship badge added to its main icon: 

After Creating Your Set of Related CMP Beans

Your set of related CMP entity beans is now finished except for a few steps that prepare the set to work in its eventual application environment. These final steps are described in Chapter 8.

Recommendations for working with finished enterprise beans are given in Appendix A.

Developing BMP Entity Beans

The previous two chapters covered the development of entity beans that delegate their persistence management to the EJB container. This chapter discusses how to create and work with entity beans that contain all the code needed to manage their own persistence, that is, bean-managed persistent (BMP) beans. There are many similarities between the development of CMP beans and BMP entity beans; this chapter focuses mainly on the differences.

The Sun ONE Studio IDE provides wizards that let you create the classes required for any BMP entity bean: a bean class, remote or local interfaces or both, and sometimes a primary-key class. To start with, the EJB Builder wizard automates the task of creating a BMP entity bean's infrastructure.

When programming entity beans, you have many options in addition to those described in this chapter. Although the Sun ONE Studio IDE is designed to take care of much of your coding work, the IDE also supports those options flexibly and leaves many decisions up to you. For more information, refer to the resources listed in "Before You Read This Book" on page xviii, or to one of the many excellent texts on programming enterprise beans.

Deciding on an Approach

You can take several approaches to creating entity beans in the IDE. However, you get the most comprehensive support and, in general, the fastest path to bean completion, if you use the approach recommended in "Using the EJB Builder With CMP Entity Beans" on page 69. This methodology takes full advantage of the IDE's ability to ensure consistency and its adherence to the J2EE standard.

If you're not sure whether your entity bean needs to manage its own persistence, look at TABLE 4-1.

Building a BMP Entity Bean

The EJB Builder wizard automates some of the work of creating your BMP entity bean's default classes. The default classes are generated by the wizard. However, the wizard makes no assumptions about how you want your BMP entity bean to interact with a database. The initial process of setting up the default classes can, therefore, be very brief. To create a BMP entity bean, you take the following steps:

1. Select or create a package to contain the BMP entity bean.
2. Use the EJB Builder wizard to generate the infrastructure of your BMP entity bean.
3. As appropriate, add a primary-key class to the bean.
4. As appropriate, add create, business, home, and finder methods to the bean's code.
5. Complete the bodies of the methods you added.
6. Write all persistence code. Complete any methods that affect data in the database.

These basic steps are discussed next.

Creating a Package

If you need to create a package to house your entity bean, select a filesystem, right-click, and choose New Java Package.

Starting the EJB Builder Wizard

When you're ready to create a BMP entity bean, do as follows:

1. **In the IDE's main window, choose View → Explorer to open the Explorer window.**
2. **In the Explorer, select the Java package where you want your BMP entity bean to reside.**
3. **Right-click and choose New → J2EE → BMP Entity EJB.**

The EJB Builder wizard appears.

Generating a BMP Entity Bean's Infrastructure

In the BMP Entity EJB pane of the wizard, do as follows:

1. **Type a name for your BMP entity bean.**
2. **Decide whether to give your BMP entity bean only a local interface (the default), only a remote interface, or both.**

If necessary, you can change the package location of the bean.

3. **Click Next. (Or, if you like, skip to the next step.)**

The BMP Entity Bean Class Files pane shows the class files that will be generated for your BMP entity bean. If necessary:

- You can use the Modify button to change any of the class names, specifying a class that already exists or creation of a new class. For example, you might be implementing a bean whose home and remote interfaces have already been specified, and now you want to generate a new bean class.
- You can click the Modify button for any of the classes shown and change the superclass. If you do, select a class that extends the appropriate interface.



4. **Click Finish when you're done.**

The wizard generates the default classes of your BMP entity bean. These classes are discussed next.

Looking at a BMP Entity Bean's Classes

For a BMP entity bean, the EJB Builder wizard generates all the required entity bean classes and sets up communications between them. However, you must code all the persistence logic yourself.

In the Explorer's Filesystems pane, a BMP entity bean has the same appearance as a CMP entity bean, except that when you pause the cursor over the bean's logical node, the tool tip says `BMP entity bean logical node`.

The nodes marked with the class icon  represent actual classes, while the one marked with the coffee-bean icon  is a logical node. Do all your editing in the logical node.

A BMP entity bean's classes implement the same interfaces as a CMP entity bean's classes. However, a BMP entity bean's class is defined as public and not abstract.

Expanding the Nodes

When you expand the nodes under your BMP entity bean's package node, you see something like the tree view in FIGURE 6-1. In this case, the bean has been assigned local-type interfaces. Notice that a BMP entity bean can have no select methods.

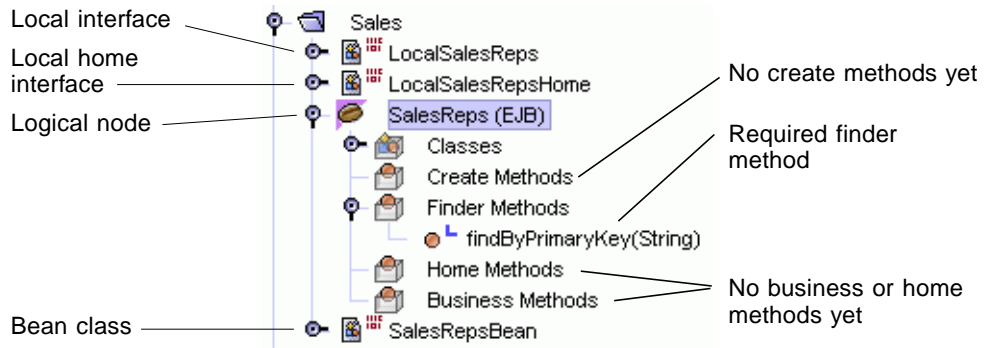


FIGURE 6-1 Explorer's Detailed View of a BMP Entity Bean

If you generated a primary-key class, it shows up in the Explorer as another major node.

Reviewing the Generated Classes

The EJB Builder wizard adds several default methods to every entity bean.

findByPrimaryKey Method

The method signature `findByPrimaryKey` is added automatically to your BMP entity bean's home interface, as shown in the following example:

```
public customer findByPrimaryKey(String aKey)
    throws RemoteException, FinderException;
```

Because this is a BMP entity bean, the wizard adds that method's counterpart, `ejbFindByPrimaryKey`, to the bean class:

```
public String ejbFindByPrimaryKey(String aKey) {
    return aKey;
}
```

A BMP Entity Bean's Life-Cycle Methods

The wizard adds default life-cycle methods to the bean class of your BMP entity bean, as shown in CODE EXAMPLE 6-1.

CODE EXAMPLE 6-1 Default Life-Cycle Methods of a BMP Entity Bean

```
public void setEntityContext(javax.ejb.EntityContext aContext) {
    context=aContext;
}
public void ejbActivate() {
}
public void ejbPassivate() {
}
public void ejbRemove() {
}
public void unsetEntityContext() {
    context=null;
}
public void ejbLoad() {
}
public void ejbStore() {
}
```

The purposes of these methods in a BMP entity bean are described in TABLE 6-1. (For comparison, see TABLE 4-2.)

TABLE 6-1 Purpose of Default Life-Cycle Methods in a BMP Entity Bean Class

Method	Purpose
setEntityContext	This method lets you store the <code>EntityContext</code> reference in a field and populate nonpersistent fields. You can use it to allocate resources that are independent of the EJB object and last for the entity bean's lifetime, resources such as a database-connection factory. By default, the EJB Builder wizard generates code that assigns the <code>EntityContext</code> to a field named <code>context</code> .
ejbActivate	This method initializes the bean, prepares it for use, and acquires the resources needed by the instance.
ejbPassivate	Before the bean instance is returned to the generic instance pool, this method releases the resources the bean was using.

TABLE 6-1 Purpose of Default Life-Cycle Methods in a BMP Entity Bean Class (*Continued*)

Method	Purpose
<code>ejbRemove</code>	In a BMP, this method executes SQL <code>Delete</code> statements and removes data from the underlying data storage. Or, you can call another object, such as a DAO, to remove data.
<code>unsetEntityContext</code>	This method lets you deallocate resources and release memory used by the entity bean instance, before the container destroys the instance.
<code>ejbLoad</code>	In a BMP, this method executes SQL <code>Select</code> statements and loads data into the bean instance from the underlying data source. This happens when the bean is activated or when the entity is referenced within the context of a new transaction. Or, you can call another object, such as a data access object (DAO), to load data.
<code>ejbStore</code>	In a BMP, this method executes SQL <code>Update</code> statements and saves the bean's state (the current values in the persistent fields) to the underlying data storage. This happens when the bean is passivated or when the transaction is committed. Or, you can call another object, such as a DAO, to store data.

Completing Your BMP Entity Bean

To complete your BMP entity bean, do as follows:

- Add all persistence logic.
- Add a primary key class if your BMP entity bean has a composite primary key.
- Define a create method if you want clients of your bean to be able to insert data into the database. An entity bean can have more than one create method.
- Define any finder methods that your BMP entity bean needs besides `findByPrimaryKey`, and code the bodies of all the finder methods.
- Code the `ejbRemove` method to remove the appropriate record from the database.
- Define and code all business and home methods that your BMP entity bean needs.
- Add `private` fields to maintain your entity's state in memory and populate the values of these fields.

Using Recommended Approaches When Working With Enterprise Beans

Appendix A discusses the best ways to make changes in your enterprise beans, and the errors and anomalies that you might see if you use other approaches. As a general rule, you should work through the logical node rather than the individual class nodes, use the bean's property sheets or the Customizer dialog box to edit methods, and use the IDE's Source Editor to complete or edit any bean code that isn't available to you through one of the dialog boxes.

Adding Persistence Logic

To make your BMP entity bean interact with the entity data store, you must write code to access the data, manipulate persistent fields, and transfer data between your bean instance's variables and the data store. Use the Source Editor to write your code. Use resource references (as discussed in Chapter 8) to specify the data source that your bean will use.

Adding a Primary-Key Class

Use the Source Editor to add a primary-key class if:

- You didn't create a primary-key class when you created the BMP entity bean, and your bean needs one.
- You need to specify a primary key that can't be represented by any existing class.
- Your primary key has a type other than `java.lang.String` or an existing primary-key class.
- You must customize the definition of the `equals` and `hashCode` methods.
- You want to wrap the primary key with some extra functionality, such as testing the key for valid values before it is used with the database.

Make sure your primary-key class meets the following requirements:

- The class has the access-control modifier `public`.
- All fields are declared as `public`.
- The class has a `public` default constructor.
- The class implements the `hashCode` and `equals` methods.
- The class is serializable: It implements the `java.io.Serializable` interface.
- The class does not implement the `java.rmi.Remote` interface.

For more information, see the discussion in "Adding or Replacing a Primary Key" on page 95.

Adding Methods

To start defining a new methods, go to the Explorer, right-click the logical bean node, and take advantage of the GUI tools that are available from the contextual menu. Use the dialog boxes to name a method and define its signature. The IDE propagates your method automatically to the correct classes. Then finish coding your method within the Source Editor.

Defining Create Methods

The home interface of your BMP entity bean can have a create method, and, if so, the bean class must have corresponding `ejbCreate` and `ejbPostCreate` methods. When you use the recommended process, the IDE ensures that these methods are generated and propagated correctly.

The `ejbCreate` method in a BMP entity bean typically does the following:

1. Validates client-supplied arguments
2. Initializes the instance's variables
3. Executes SQL `Insert` statements (or you can call another class, such as a DAO, to insert data into the underlying data store)
4. Returns a primary key

In a BMP entity bean, you must provide the code that generates and executes the necessary SQL `Insert` statement.

The `ejbPostCreate` method, which the IDE adds automatically, gives the programmer the opportunity to forward information about the EJB object (such as the home or remote interface) to any other enterprise beans that need to reference it. The method can access the remote interface through `EntityContext`, which it receives from the container as a parameter. This method is typically used to create dependent beans. For example, the `Order` bean's `ejbCreateLineItem` method might create the given line items in the `ejbPostCreate` method.

Your entity bean can have more than one create method. Define a new create method as follows:

- 1. Select the logical node, right-click, and choose Add Create Method.**

The Add New Create Method dialog box appears.

- 2. Name your create method, using any extension after `create`.**

Now you need to add parameters to your method.

- 3. In the dialog box, click Add.**

4. **In the Enter Method Parameter dialog box, specify the parameter's name and type.**
Both the method signature in a BMP entity bean class and the method body return the primary-key type.
5. **Click OK to dismiss the Enter Method Parameter dialog box.**
6. **In the Add New Create Method dialog box, specify any additional exceptions.**
7. **Click OK to dismiss the Add New Create Method dialog box.**
The method you added now appears in the bean class code as `ejbCreate` and in the home interface as `create`. The method `ejbPostCreate` also appears in the bean class.
8. **Use the Source Editor to add the return statement and all other necessary code to your new create method.**

Adding Finder Methods

The EJB Builder has already generated a default finder method for you. In a BMP entity bean, this method shows up in both the home interface (`findByPrimaryKey`) and the bean class (`ejbFindByPrimaryKey`). However, if you want your entity bean to execute additional queries, you must define additional finder methods.

If you follow these steps, your new finder method is automatically propagated to your home interface and bean class:

1. **Select the logical node, right-click, and choose Add Finder Method.**
2. **Type a name for the method starting with `find`. Specify parameters, exception, and a return type. Click OK when you're finished.**

Finish coding your finder method or methods using the Source Editor. To fetch primary keys from the data source, you must write JDBC code or use other means of database access.

Defining Business and Home Methods

To add a business method to your BMP entity bean, do as follows:

- **Under the logical node, select Business Methods, right-click, and choose Add Business Method.**

The Add New Business Method dialog box appears. At this point you can finish coding the method's parameters and exceptions in this dialog box, or you can simply type a name for your new business method, click OK, and finish the coding in the Source Editor.

A business method typically accesses and modifies the values of persistent fields, but it doesn't directly access the database. The EJB container calls the `ejbLoad` and `ejbStore` methods as required by the semantics of the transaction.

Alternatively, you can add a home method to perform an operation that does not depend on any given instance of the entity bean. See "Defining Home Methods" on page 99 for a discussion of home methods.

After Creating Your BMP Entity Bean

Your BMP entity bean is now finished except for a few steps by which you prepare the bean to work in its eventual environment. These final steps are described in Chapter 8.

Recommendations for working with finished enterprise beans are given in Appendix A.

Further Reading

Enterprise beans can be a very powerful and flexible part of your application. Creating the basic parts of an enterprise bean can be very simple, especially with a tool like the Sun ONE Studio IDE. However, completing the bean so that it fulfills the needs of your application can be very complex. For details, refer to *Enterprise JavaBeans Specification*, version 2.0 at:

<http://java.sun.com/products/ejb/docs.html>

Developing Message-Driven Beans

The EJB Builder in the Sun ONE Studio IDE enables you to develop the message-driven beans that you need to support an application client's requests for asynchronous processes. This chapter discusses the process of creating and working with message-driven beans. These beans' transactions are normally managed by the EJB container, but you can provide the transaction-management code yourself, if you prefer.

There are several reasons to use a message-driven bean:

- **Performance and support for multitasking.** The application client can send a message and go on to other tasks without having to wait for a response to the message. That is, the client invokes your message-driven bean asynchronously.
- **Reliability.** If the application uses Java Message Services (JMS), no client requests are lost unless tiers of the application go down at once.

However, message-driven beans aren't always the right answer. For example, an alternative would probably work better in the following cases:

- When the client needs confirmation that a request was received or needs results to be returned
- When the operation is part of a time-sensitive transaction and can't be done during off-peak hours
- When the application is small and uncomplicated, and adding another layer would slow down building, debugging, and execution

For more pros and cons, see "Understanding Message-Driven Beans" on page 36.

The IDE provides a wizard that lets you create the single bean class required for a message-driven bean. Because a message-driven bean merely takes messages from a client and uses them to start other bean processes, no interface classes are needed. The wizard automates much of the task of creating a message-driven bean, and you finish the task using the IDE's Source Editor and property sheets.

When programming message-driven beans, you have options besides those described in this chapter. Although the Sun ONE Studio IDE is designed to take care of much of your coding work, the IDE also supports those options flexibly and leaves many decisions up to you. For more information, refer to the resources listed in “Before You Read This Book” on page xviii, or to one of the many excellent texts on programming enterprise beans.


Using the EJB Builder With Message-Driven Beans

The EJB Builder is a collection of wizards, property sheets, and editors with which you can build enterprise beans consistently and easily. To see if the EJB Builder is installed, go to the main window and choose Tools → Options → IDE Configuration → System → Modules → J2EE Support. If you see EJB 2.0 Builder in the list of modules, and the Enabled field in the property sheet is set to True, the EJB Builder is ready for use.

You can take several approaches to creating message-driven beans in the IDE. However, you get the most comprehensive support and, in general, the fastest path to bean completion, if you use the approach recommended in this chapter. The methodology described here takes full advantage of the IDE’s ability to ensure consistency and its adherence to the J2EE standard.

For best results, use the EJB Builder to program message-driven beans by:

- **Creating a bean’s one required class.** After using the EJB Builder wizard, you have the framework of your message-driven bean, which is made up of the bean class and a logical grouping of the bean’s parts. Nodes for both the class and logical grouping are shown in the Explorer’s Filesystems tabbed pane, along with their subnodes. The wizard generates declarations of the two required methods, `ejbCreate` and `onMessage`, for the bean class. You then supply the method implementations.

The logical node is the best place to do work on a message-driven bean. All logical nodes appear in the Explorer with this icon: 

- **Completing the bean class code as necessary.** Use the IDE’s support as described later in this chapter.
- **Setting values in a bean’s deployment descriptor.** Use the message-driven bean’s property sheet from the logical node to edit properties.

From a message-driven bean’s logical node, you can validate the bean’s code.

Deciding on Transaction Management

Before you begin creating a message-driven bean, first consider whether to have the EJB container manage any transactions that your bean will do, or whether to write that code yourself. You use different processes in the IDE's EJB Builder to create the two kinds of bean. TABLE 7-1 highlights the design considerations.

TABLE 7-1 Deciding Between Container-Managed and Bean-Managed Transactions

Issue	Container-Managed Transactions	Bean-Managed Transactions
Transaction manager	The container itself is the transaction manager.	You write code to manage transactions by using JTA. This can include transactions for other resources such as JDBC.
Setting of transaction boundaries	The EJB container decides when to begin and commit a transaction according to the <i>Java 2 Platform, Enterprise Edition Specification</i> .	The programmer explicitly codes the transaction's boundaries to obtain more granular control over transactions.
Transaction timing	The message-driven bean receives a message and performs its business logic in the same transaction.	The transaction doesn't start until after the message-driven bean receives the message.
Problem handling	The container rolls back the transaction and has the bean acknowledge the message.	The message-driven bean responds according to the acknowledgment mode you specified after you generated the bean.

For more information on these selections, refer to the chapter on transactions in the book *Building J2EE Applications*.

The rest of this chapter addresses how to create message-driven beans of each kind and the issues to consider during development.

Defining a Message-Driven Bean

The EJB Builder wizard automates much of the task of creating the one bean class that your message-driven bean requires. To define a message-driven bean, you take the following steps:

1. Select or create a package to contain the bean.
2. Use the EJB Builder wizard to generate the infrastructure of your message-driven bean.
3. Complete the body of the `onMessage` method and, if necessary, the `setMessageDrivenContext` and `ejbCreate` methods.

These basic steps are explained in detail next.

After you finish the steps covered in this chapter, you must add information to your finished bean's property sheet so that it can interact with other beans, find its resources, and listen for the appropriate messages. These steps, which prepare your finished bean to work in an application, are discussed in Chapter 8.

Creating a Package

If you need to create a package to house your message-driven bean, select a filesystem, right-click, and choose New Java Package.

Starting the EJB Builder Wizard

When you're ready to create a message-driven bean, do as follows:

1. **In the IDE's main window, choose View → Explorer to open the Explorer window.**
2. **In the Filesystems pane of the Explorer, select the package or filesystem where you want your message-driven bean to reside.**
3. **Right-click and choose New → J2EE → Message-Driven EJB.**

The EJB Builder wizard appears, displaying New Wizard–Message-Driven EJB in the window's title bar.

Generating the Basic Message-Driven Bean

In the EJB Builder's Message-Driven Bean Name and Properties pane, name your message-driven bean and decide how to manage any transactions the bean performs. The default is Container-Managed Transactions, but you can decide to provide all transaction management code in the bean class if you wish.

When you have made your selection, you can click Finish. (Or, you can click Next to go to the pane in which you can specify an existing bean class for your message-driven bean. After that, you click Finish.)

Your newly created message-driven bean appears in the Filesystems pane of the IDE's Explorer. The bean's infrastructure (its basic bean class and its two component methods) has been generated automatically by the EJB Builder.

Looking at Your Message-Driven Bean in the Explorer

FIGURE 7-1 shows how a typical message-driven bean appears in the Explorer's Filesystems pane.

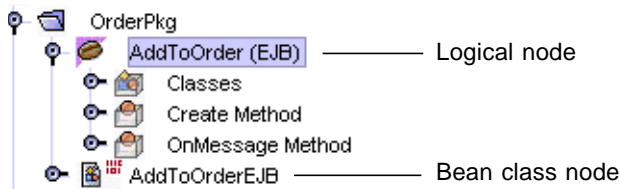


FIGURE 7-1 Default Class and Methods of a Typical Message-Driven Bean

Of the two primary nodes shown, one is a logical node (marked with a bean icon) and one represents the actual class (marked with a class icon). Do all your editing in the logical node. The bean's two primary nodes are described next.

- The logical node is created in the Explorer to group all the elements of your message-driven bean and let you work with them more conveniently.
- The bean class implements the `javax.ejb.MessageDrivenBean` and `javax.jms.MessageListener` interfaces, and exposes the message-driven bean's methods.

The Classes node contains the bean class code, which includes both methods. The Create Method node points to the code that initializes your message-driven bean. The OnMessage Method node points to the method that is invoked when a message is received.

Expanding the Nodes

When you expand the two nodes under your message-driven bean's package node, you see something like the tree view in FIGURE 7-2.

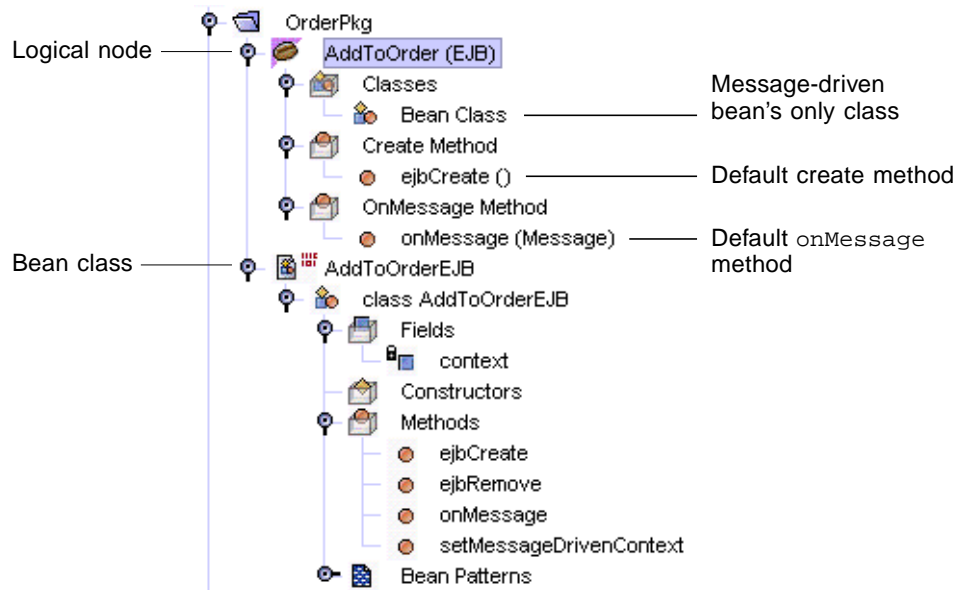


FIGURE 7-2 Explorer's Detailed View of a Typical Message-Driven Bean

Reviewing the Generated Class

The wizard automatically places certain default methods in each message-driven bean: a create method, an onMessage method, and two life-cycle methods. As shown in TABLE 7-2, the create method, `ejbCreate`, behaves much like create methods in other types of enterprise beans, but `onMessage` is a new and different kind of method.

TABLE 7-2 Purpose of `ejbCreate` and `onMessage` Methods in a Message-Driven Bean's Bean Class

Method	Purpose
<code>ejbCreate</code>	This method initializes the message-driven bean, if necessary.
<code>onMessage</code>	This method opens the message the message-driven bean has received, decides what to do with it, and processes it.

The wizard also adds the default life-cycle methods described in TABLE 7-3.

TABLE 7-3 Purpose of Default Life-Cycle Methods in a Message-Driven Bean's Bean Class

Method	Purpose
<code>setMessageDrivenContext</code>	This method is called before <code>ejbCreate</code> , and it associates the message-driven bean with a context object.
<code>ejbRemove</code>	This method is called just before the message-driven bean instance is removed, to free up resources that are no longer needed. In a simple message-driven bean, this method might not even be used.

Completing Your Message-Driven Bean

To complete your message-driven bean, do the following:

- Add code to complete the body of your bean's `onMessage` method.
- Add any code that is necessary to complete your bean's `setMessageDrivenContext` method.

The `ejbCreate` and `ejbRemove` methods are not needed in simple message-driven beans. However, if necessary, `ejbCreate` can be used to allocate resources and `ejbRemove` to let the resources go.

- Use the property sheets (the tabbed interface for the application server to which your bean will be deployed) to specify the type of resource, the resource factory, and the server that the message-driven bean will use. Details are supplied in "Specifying Resources for Client Message-Driven Beans" on page 143 and also in Chapter 8.

Make your additions in the Explorer by clicking bean components under the logical bean node to open the Source Editor.

Using Recommended Approaches When Working With Enterprise Beans

Appendix A discusses the best ways to make changes in your enterprise beans, and the errors and anomalies that you might see if you use other approaches. As a general rule, you should work through the logical node rather than the individual class nodes, use the bean's property sheets or the Customizer dialog box to edit methods, and use the IDE's Source Editor to complete or edit any bean code that isn't available to you through one of the dialog boxes.

Completing the onMessage Method

A single instance of your message-driven bean can handle only one message at a time, and the bean can have only one onMessage method. An example of a completed method follows.

```
public void onMessage(Message inMessage) {
    TextMessage msg = null;

    try {
        if (inMessage instanceof TextMessage) {
            msg = (TextMessage) inMessage;
            System.out.println("MESSAGE BEAN: Message " +
                "received: " + msg.getText());
        } else {
            System.out.println("Message of wrong type: " +
                inMessage.getClass().getName());
        }
    } catch (JMSEException e) {
        System.err.println("MessageBean.onMessage: " +
            "JMSEException: " + e.toString());
        context.setRollbackOnly();
    } catch (Throwable te) {
        System.err.println("MessageBean.onMessage: " +
            "Exception: " + te.toString());
    }
}
```

Completing the setMessageDrivenContext Method

The `setMessageDrivenContext` method stores the message-driven context reference in a field and populates non-persistent fields. You can, if necessary, use this method to allocate resources that are independent of the bean object and last as long as the bean exists. These resources might include a queue-connection or topic-connection factory.

By default, the EJB Builder wizard generates code that assigns the message-driven context to a non-persistent field named `context`. Ordinarily, you don't need to add anything to the generated method. However, if you need to complete it, copy the generated context into the instance variable. For example:

TABLE 7-4 Example of a `setMessageDrivenContext` Method

```
public void setMessageDrivenContext(javax.ejb.MessageDrivenContext aContext ) {  
    this.context=context;  
}
```

After Creating Your Message-Driven Bean

Your message-driven bean is now finished, except for a few steps that prepare the bean to work in its eventual environment. You must specify the following in the bean's property sheet:

- The bean's message-driven destination, that is, whether the bean gets its messages from a queue or a topic
- If the bean listens to a topic, whether its subscription is durable or non-durable
- Whether a message selector (filter) has been applied to the bean to narrow down the messages it gets

If your message-driven bean will receive messages from a client, and you plan to deploy your bean to the J2EE reference implementation application server (RI), you must specify the destination in the J2EE RI tabbed interface of the bean's property sheet.

If your message-driven bean will act as a client itself, sending messages to a destination, you must specify the following in the References tabbed interface of the bean's property sheet:

- The bean's resource references (the connection factories it uses to access its message-driven destinations)
- The bean's resource environment references (the actual destinations: queues or topics)

These property settings are discussed next.

Specifying a Message-Driven Destination

To specify whether the message-driven bean will be a queue listener or a topic listener, do as follows:

- 1. In the IDE's Explorer window, right-click the message-driven bean's logical node and choose Properties.**

The property sheet for the bean appears.

- 2. In the Properties tabbed interface, click the Message-Driven Destination field and then the ellipsis (...) button.**

The property editor appears.

- 3. Select Queue, Topic, or (Not Set).**

- Select Queue if clients will send messages only to this particular bean and you need to use the point-to-point model.
- Select Topic if you need to allow multiple clients to send messages to this bean, using the publish-subscribe model. If you choose Topic, you must also specify whether the bean's subscription is durable or non-durable.
 - Select Durable if messages should be persisted until the bean consumes them. This way, even if the bean's application server crashes, the messages are available when the bean is next available.
 - Select Non-durable if the bean should get only messages published while the bean is available. All other messages are deleted.
- Leave the Destination field blank (using the Not Set value) if you will set this property later.

- 4. Click OK to dismiss the property editor.**

Specifying a Message Selector

If you want to filter your bean's incoming messages, do as follows:

1. **Click the Message Selector field and then the ellipsis (...) button.**
A property editor appears.
2. **Specify a filter if you wish to reduce the number of messages for which your bean must listen.**
3. **Click OK to dismiss the property editor.**

Specifying Resources for Client Message-Driven Beans

The References tabbed interface of a message-driven bean's property sheet contains the Resource Reference and Resource Environment Reference fields. These fields are completed on behalf of the client that sends messages. For example, your message-driven bean might be part of an application in which a web module sends messages to a queue for consumption by your bean. In that case, this Resource Reference and this Resource Environment Reference should be specified by the provider of the web module.

Or, if your message-driven bean is meant to act as a client within its own module, sending messages to a queue or topic, you specify the resource factory and the resource here.

Specifying Resource Factories

To associate the message-driven bean with a factory object that will create the destination object, do as follows:

1. **In the References tabbed interface, click the Resource References field and then the ellipsis (...) button.**

In the property editor are fields for specifying the connection factory that the client (or message-driven bean as client) will use to gain access to its messaging resource.

2. Click the Add button.

The Add Resource Reference dialog box appears with two tabbed interfaces.

- In the Standard tabbed interface:
 - Type the name of the object that will create your bean's connection to its queue or topic.
 - In the Type combo box, select the type of resource factory your bean will use. This type should correspond to the choice you made in the Message-driven Destination field of the Properties tabbed interface. See "Specifying Resource Environment References" on page 156 for an explanation of the various types of resource factories.
 - In the Authorization field, specify whether the EJB container or the application client will authorize the bean to use the resource.
 - In the Sharing Scope field, specify whether the connection to this resource can be shared by another enterprise bean in the same application. If two or more beans can use the same resource in the same transaction context, the container can carry out transactions locally and save time.

If you plan to deploy your message-driven bean to the RI, complete the following fields.

- In the J2EE RI tabbed interface:
 - In the JNDI Name field, type the name by which the resource factory is located.
 - In the User Information field, provide any information needed to gain access to the resource.
 - In the Mail Configuration field, if appropriate to your situation, provide the information that will be needed to use a JavaMail session factory.

3. When you're finished, click OK to dismiss the dialog box.

Specifying Resources

To associate the message-driven bean with a particular destination object, do as follows:

1. In the References tabbed interface, click the Resource Environment References field and then the ellipsis (...) button.

In the property editor are fields for specifying the actual resources to which the client (or message-driven bean as client) will send messages.

2. Click the Add button.

The Add Resource Environment Reference dialog box appears with two tabbed interfaces.

- In the Standard tabbed interface:
 - Type the name of the queue or topic to which your client or bean will send messages.
 - In the combo box, select a resource type.

If you plan to deploy your message-driven bean to the J2EE reference implementation application server (RI), complete the following field.

- In the J2EE RI tabbed interface, type the JNDI name by which the message resource (the queue or topic) is located.

3. When you're finished, click OK to dismiss the dialog box.

Specifying Resources for Listener Message-Driven Beans Deployed to the RI

If your bean will be deployed to the RI, you must not only say whether it listens to a queue or a topic. You must also specify the JNDI name of the resource so that the RI can locate it. Do as follows:

1. **In the J2EE RI tabbed interface, click the Destination JNDI Name field and then the ellipsis (...) button.**
2. **In the property editor, specify the JNDI name of your bean's messaging resource.**

Use the form *type/resource*. For example, a messaging queue might be specified as `jms/myQueue`.

Read more about message-driven destinations in Chapter 2 and more about setting properties in Chapter 8.

Recommendations for working with finished enterprise beans are given in Appendix A.

Avoiding Pitfalls of Message-Driven Beans

The messaging tier of your application will run into fewer problems if you understand the following possible complications.

- **Order of Messages.** Your message-driven beans should be prepared to handle messages that arrive out of sequence. A JMS server might deliver messages in any order to a pool of message-driven beans.
- **Dropped `ejbRemove` Invocations.** A simple message-driven bean doesn't need to use an `ejbCreate` or `ejbRemove` method. However, if your bean is more complex and does use those methods, be aware that under certain circumstances (such as a system or container crash), `ejbRemove` might not be called. In this case, you should provide for the bean to do its own clean-up. This depends on the behavior of your application server; for details, see your server's documentation.
- **Poison messages.** When you're using the EJB Builder wizard to generate the infrastructure of a message-driven bean that manages its own transactions, you can set the bean property Acknowledge Mode to Auto. This setting makes the bean automatically acknowledge each message it gets. This way, you avoid the situation in which a transaction fails, the message destination never hears that the message was received, and the destination keeps sending the message over and over.

For detailed design considerations, refer to *Enterprise JavaBeans Specification*, version 2.0 and to texts on programming enterprise beans.

Further Reading

Enterprise beans can be a very powerful and flexible part of your application. Creating the basic parts of an enterprise bean can be very simple, especially with a tool like the Sun ONE Studio IDE. However, completing the bean so that it satisfies the needs of your application can be somewhat more complex. For details, refer to *Enterprise JavaBeans Specification*, version 2.0 at:
<http://java.sun.com/products/ejb/docs.html>

Preparing Enterprise Beans for Deployment

The foregoing chapters have focused on creating individual enterprise beans. However, before a finished bean can be assembled into an application and deployed in a production environment, three tasks remain to be done:

1. Furnish certain information about the bean's external dependencies and operating requirements. This information becomes part of the bean's deployment descriptor, which is discussed in the next section.
2. Form an EJB module around a group of beans that need to work cooperatively in an application. Also, if necessary, add an EJB JAR file around the EJB module.
3. Test your beans and module, using the automated testing feature of the Sun ONE Studio IDE.

Ordinarily, enterprise beans are assembled into an EJB module, one or more EJB modules are assembled into an application, and the application is deployed to an application server. However, an individual EJB module can also be deployed.

Another document in this series, *Building J2EE Applications*, discusses in detail the design and assembly of applications and their deployment to servers.

Understanding Deployment Information

The deployment information you furnish in an enterprise bean's property sheets becomes part of the bean's deployment descriptor. This is an XML-based text file that captures information about the bean's structure, its relationships to other beans, and its other external dependencies. The deployment descriptor contains all the instructions an application server needs when the bean's application is deployed. Any change in this descriptor can change the bean's behavior in the application.

You automatically start a deployment descriptor for an enterprise bean when you use the EJB Builder wizard to create the bean, as described in the foregoing chapters. The wizard generates the bean's basic descriptor.

When you place enterprise beans in an EJB module (as explained in "Creating an EJB Module" on page 161), the IDE automatically generates a deployment descriptor for the module as well. This descriptor file captures:

- Each bean's declarative meta-information (that is, information about the beans, but not in the beans' code) from the bean's deployment descriptor
- A higher level of information about how the beans fit into the application and the deployment for which the module is designed
- Security and transaction information that allows the EJB module to override such information specified at the bean level
- Instructions to the container about the data source on which an entity bean is based

By changing the EJB module's descriptor, you can change the application's behavior without touching the source code of the component beans.

The contents of a deployment descriptor are accessible through the property sheets of the corresponding bean or module. If necessary, you can also edit an EJB module's descriptor directly. The following three sections describe how you can see and affect the deployment descriptor.

Looking at a Generated Deployment Descriptor

To see the XML-based file that constitute the deployment descriptor of an enterprise bean:

- **In the Explorer window, select a bean's logical node, right-click, and choose View Deployment Descriptor. (Or, select an EJB module's logical node, right-click, and choose Deployment Descriptor → View.)**

The Source Editor displays the file as read-only.

Editing an EJB Module's Deployment Descriptor

Ordinarily, you should edit a bean's or EJB module's deployment descriptor only through property sheets on the bean or EJB module (as described in the next section). When you edit this file using Properties, all you have to do is specify file names and values (and possibly table and column names, if you're working on an entity bean). You don't have to write any XML. You make your selections using a dialog box, and the IDE automatically synchronizes the changes across the appropriate classes of your enterprise bean.

However, if you need to, you can edit the deployment descriptor directly, and after editing you can revert to the generated descriptor.

Editing an EJB Module's Deployment Descriptor Directly

If you need to make a direct edit in the deployment descriptor file of an EJB module, you can do it as follows:

- **Right-click the EJB module's node and choose Deployment Descriptor → Final Edit.**

After directly editing an EJB module's deployment descriptor, you can still use the property sheets to make changes that don't affect the deployment descriptor. For example, you can still go to the J2EE RI tabbed pane of the property sheet and specify or change the Data Source JNDI Name, Data Source Password, or Data Source User Name fields. (Be careful not to edit those fields in both the property sheets and the deployment descriptor.) However, fields that represent items in the deployment descriptor are closed to edits in the property sheets.

Reverting to the EJB Module's Last Generated Descriptor

If you have used the Final Edit feature, but now want to go back to the last generated version of the deployment descriptor and continue from there to make changes in the property sheets, do as follows:

- **Right-click the EJB module's node and choose Deployment Descriptor → Revert to Generated.**

Note – If you choose Revert to Generated, any edits you made directly in the deployment descriptor file will be lost.

Using Properties to Edit a Deployment Descriptor

To use property sheets to add to, edit, or complete an enterprise bean's or an EJB module's descriptor, do as follows:

- **Select the bean's or EJB module's logical node, right-click, and choose Properties.**

In the case of an enterprise bean, the Properties dialog box appears, showing at least three tabbed panes:

- Properties
- References
- J2EE RI (the reference-implementation server from the Java™ 2 Platform, Enterprise Edition)

You also see a tabbed pane for any other application server's plugin module that has been installed with the IDE.

In the case of an EJB module, there are two default tabbed panes: Properties (which includes reference fields) and J2EE RI.

The three default tabbed panes for enterprise beans are described next.

Specifying Bean Properties

Before you create an EJB module around enterprise beans, specify the individual beans' properties. These properties are discussed pane by pane in the following sections: "Using the Properties Tabbed Pane" on page 150, "Using the References Tabbed Pane" on page 153, and "Using the J2EE RI Tabbed Pane" on page 159.

Using the Properties Tabbed Pane

Since you're familiar with your bean's code, you'll probably recognize most of the fields in the Properties tabbed pane. Also notice the following:

- In the Properties pane for any kind of enterprise bean, certain fields are read-only. These fields were automatically completed by the EJB Builder wizard when you created the bean. The properties named in these fields are so intrinsic to the bean type that you shouldn't need to change them. The only practical way to change them would be to go back to the wizard and recreate the bean.

- The Name field and the fields naming the bean's interfaces were completed by the EJB Builder wizard when you created the bean (or you overrode the names in those fields in the wizard, at creation time).

If you want your bean to use another existing class, you can override one of the classes shown in these Properties fields.

If you want to change the name of a class but retain the same class content, you must either make the change in the property editor for that class (not the property editor for the logical bean node) or directly edit the class code in the Source Editor.

- The Large Icon and Small Icon fields are for any icons that you want to accompany the enterprise bean and be available to tools like application servers. For a large icon, the file must be a JPEG or GIF image 32x32 pixels in size, or 16x16 pixels for a small icon. The file must have the suffix `.jpg` or `.gif`.
- The Security Identity field lets you specify the identity the enterprise bean will use when it makes calls.
 - If you select Run As Specific Security Role and name a role, for example, a role found in a related bean, your bean executes under that security role, and all methods called by the bean carry that security role. In this way, the bean can access data ordinarily reserved for the other bean. Or, you can specify the bean's default security role here.
 - If you select Use Caller's Security Identity, the executing bean assumes the security identity of the caller.
 - If you leave the Not Set value in this field, the bean executes under the security identity defined at the application-server level or under the caller's security identity.

Some property types in this tabbed pane are unique to specific bean types, and these properties are discussed next.

Properties of Entity Beans

In the Properties pane for any entity bean, you also see the following fields:

- **Primary Key Class.** If you want to change what the EJB Builder wizard put here, you can select an existing field, declare an unknown primary-key class, or select an existing user-defined class.
- **Reentrant.** If you want to avoid unintentional multithreading problems, leave this field marked False. This way, if an instance of your bean executes a client request in a given transaction context, and a second request for the same context arrives for the same entity object, the container throws an exception to the second request. However, if you really need to let your bean use another bean to make a call to itself, mark this field True.

In the Properties pane for a CMP entity bean, you also see the following fields:

- **Abstract Schema Name.** The abstract schema defines the bean's persistent fields and relationships. Your CMP entity bean's EJB QL queries are modeled on the bean's abstract persistence schema and its dependent object classes.
- **CMP Version.** This read-only field reflects whether the CMP bean has a version 1.x bean class or a version 2.x bean class.

Properties of Session Beans

In the Properties pane for a session bean, you also see the following fields:

- **Bean Type.** If you want, you can change this value from Stateless to Stateful (or the opposite).
- **Transaction Type.** This field reflects whether the container or the bean manages its transactions. If you change this field, the EJB Builder will look for corresponding changes in your bean class code.

Properties of Message-Driven Beans

In the Properties pane for a message-driven bean, you also see the following fields:

- **Transaction Type.** This field reflects whether the container or the bean manages its transactions. The EJB Wizard populated this field, but you can change it if necessary.
- **Acknowledge Mode.** If the message-driven bean's transaction type is Bean, you see the Acknowledge Mode property in this tabbed pane. To make sure all messages consumed by the bean are acknowledged, set this property to `Auto`. Or, if you want the bean to acknowledge duplicate messages at its convenience, set this property to `Duplicates Allowed`.
- **Message Selector.** Use this field if you want to apply one or more filters to reduce the number of incoming messages your message-driven bean must listen for. For example, the following string causes the bean to receive only messages whose `AccountStatus` property is set to `Late` or `Delinquent`.

```
AccountStatus = 'Late' OR AccountStatus = 'Delinquent'
```

The syntax for message selectors, which is based on a subset of the SQL92 conditional expression syntax, is described in the JMS specification and the JMS tutorial.

- **Message-Driven Destination.** In this field, specify whether your message-driven bean listens to a message queue, or whether it subscribes to a topic (and, in that case, whether the subscription is durable or non-durable). The data in this field is related to the data in certain fields on the property sheet's tabbed interfaces for application servers. Whatever you put in this field must correspond to what you

put in the J2EE RI tabbed pane's Connection Factory Name, Destination JNDI Name, and Durable Subscription Name fields. For details, see "Setting J2EE RI Properties for Message-Driven Beans" on page 160.

Using the References Tabbed Pane

To prepare your enterprise bean for assembly and deployment, you complete this tabbed pane's fields, which represent many of the enterprise bean's external dependencies. Some of the information can be stated at the bean level and then overridden at the module or application level.

One example of a References tabbed pane is shown next.

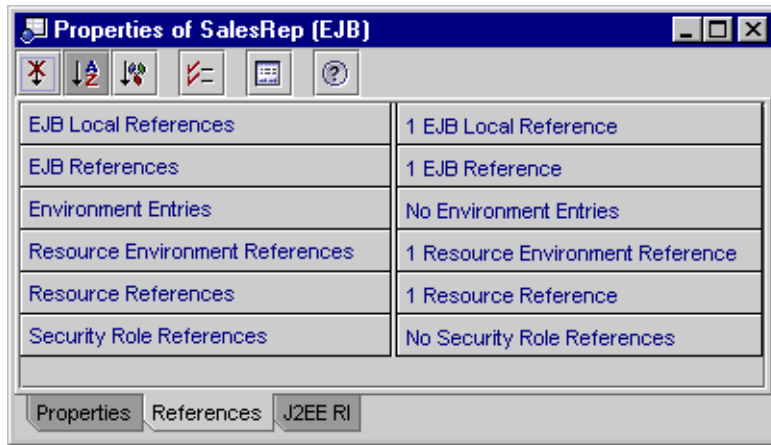


FIGURE 8-1 References Tabbed Pane of the Properties Dialog Box for a CMP Entity Bean

Each field in this pane is described next, with instructions for completing the field.

Specifying EJB Local References

This field and the following one contain information about any other enterprise beans whose methods your bean calls. You use the EJB Local References field to specify references to beans that reside within the same JVM. An EJB local reference accesses the local interface of a bean running in the same JVM, although the two beans might be in different EJB modules. (For contrast, see "Specifying EJB References" on page 155.)

You can specify the name of the implementing enterprise bean at development time, and, if necessary, that setting can be overridden when your bean is assembled into an EJB module.

In the bean's code, you use the JNDI interface to look up the home interface of another bean. Before enclosing the bean in an EJB module, you also link the beans by specifying those same references in the bean's property sheet. The person who assembles the application looks at the EJB References field to see which other beans your bean must have access to in order to work as designed. The references specified here at the bean level can be used or overridden at the module level, as needed in the target environment.

Before multiple beans are enclosed in an EJB module, the references should be coded in the bean class and the EJB references should be specified in the property sheet.

To specify EJB local references in the property sheet, do as follows:

1. **Click the EJB Local References field, then the ellipsis (...) button.**

The EJB Local References property editor appears.

2. **Click Add.**

The Add EJB Local Reference dialog box appears.

3. **Complete the fields.**

The following fields are mandatory.

Note – The easiest way to fill in these fields is to start with the Referenced EJB Name field, selecting a local enterprise bean from the Browse list. The IDE then automatically fills in the type field and the two interface fields. However, you can change the interface fields if you need to.

- **Reference Name.** The referenced bean's name, as it is mentioned in the `context.lookup` method call in your bean class code. Notice that the field already contains `ejb/`. The name is relative to the `java:comp/env` context under the `ejb/` subcontext. After the slash, type the name of the bean to which your bean refers.

For example, the bean `Account` might look up a reference to the home interface for the bean `DiscountCodeTbl`. The full reference name in this case would be `ejb/DiscountCodeTblHome`, or you can supply another reference name that is consistent with the name you used in the JNDI lookup code.

- **Referenced EJB Name.** The name of the enterprise bean that implements the local home and local interfaces specified in those fields. You can do any of the following, depending on your situation:
 - Click Browse and select a bean. In this case, the IDE completes the Local Home Interface and Local Interface fields for you.
 - Type over data in this field to specify another bean that implements those same interfaces, and change the reference to another bean.
 - Leave this field blank until the bean has been assembled into an EJB module or an application.

- **Type.** Whether the referenced bean is a session or entity bean.
- **Local Home Interface.** The referenced bean's local home interface.
- **Local Interface.** The referenced bean's local interface.

The description field is optional. That information might be helpful to the person who assembles your EJB module into an application.

- **Description.** The purpose of the referenced bean, or why your bean needs to reference it.

Specifying EJB References

In the EJB References field, you specify links to any enterprise beans that reside outside your bean's JVM but whose methods your bean calls. You use the field as you would the EJB Local References field, except that the reference you name here points to the remote interface of a bean running in the another JVM.

Specifying Environment Entries

An environment entry, stored in your bean's runtime environment, is a named data value that depends on policy or procedure at the deployment site. Environment entries can alter the behavior of an enterprise bean at deployment time without modifying the source bean's code. Any values that you set here in the property sheet can be overridden at deployment time in the deployment descriptor of an EJB module or an application.

For example, the `Account` bean can use the environment entry `overdraftAllowed` (of type `boolean`). This variable might indicate whether or not a particular bank that uses the `Account` bean allows customers to overdraw their account balances. The `Account` bean looks up the value of `overdraftAllowed` to decide what should happen if a customer's request causes an overdraft.

To add an environment entry, do as follows for each environment:

1. **Click the Environment Entry field and then the ellipsis (...) button.**

The Environment Entry property editor appears.

2. **Click Add.**

The Add Environment Entry dialog box appears.

3. **Complete the fields.**

The following two fields are mandatory.

- **Name.** The name of the environment variable.
- **Type.** The data type of the variable.

You can also complete the other two fields.

- **Description.** The purpose of the variable, and any other information that the assembler or deployer should know when using your bean in that environment.
- **Value.** An initial value.

Specifying Resource Environment References

Use this field to specify any administered object your bean needs to use, such as a JMS destination (a queue or topic). The resource environment reference is the queue's or topic's logical name. This logical name must map to the name that appears in your bean class's `InitialContext.lookup` method. When your bean needs the resource that you specify in this field, an instance of the resource is created by the factory mentioned in the next section.

To add a resource environment reference, do as follows for each object that you want your bean to use:

1. **Click the Resource Environment Reference field and then the ellipsis (...) button.**

The Resource Environment References property editor appears.

2. **Click Add.**

The Add Resource Environment Reference dialog box appears.

3. **Complete the fields.**

The following fields are mandatory.

- **Name.** The name that appears in the `InitialContext.lookup` method of your bean class code.
- **Type.** The type of resource factory. Specify your own type, or select one of the following:
 - `javax.jms.Queue`, a Java Message Service queue.
 - `javax.jms.Topic`, a Java Message Service topic.

Specifying Resource References

This field contains the name of the factory that creates a connection to a resource that your bean needs. This resource might be a datasource (such as a relational database), an administered object (such as a queue or topic), a JavaMail session, a URL, or a J2EE connector (which lets you connect your bean to another application system or EIS). The information in the Resource Reference field must correspond to a JNDI lookup method call in your bean class code.

To add a resource factory reference, do as follows for each resource that your bean needs:

1. Click the Resource Factory Reference field and then the ellipsis (...) button.

The Resource Factory References property editor appears.

2. Click Add.

The Add Resource Reference dialog box appears.

3. Complete the fields.

The following fields are mandatory.

- **Name.** The name that appears in the `InitialContext.lookup` method of your bean class code. For example, for a JDBC resource factory named `myDataBase`, your lookup method might be coded as follows:

```
javax.naming.InitialContext myContext =  
    new javax.naming.InitialContext();  
javax.sql.DataSource mySource = (javax.sql.DataSource)  
    myContext.lookup("java:comp/env/jdbc/myDataBase");
```

The corresponding name for the resource reference in this case would be `jdbc/myDataBase`. Notice that the environment's subcontext is represented by `jdbc/`.

- **Type.** The type of resource factory. Specify your own type, or select one of the following:
 - `javax.sql.DataSource`, a JDBC connection factory.
 - `javax.jms.QueueConnectionFactory`, a Java Message Service connection factory.
 - `javax.jms.TopicConnectionFactory`, a Java Message Service connection factory.
 - `javax.mail.Session`, a JavaMail session factory.
 - `javax.resource.cci.ConnectionFactory`, a connection factory that you declare if you want your bean to connect to another application system or EIS. An enterprise bean can use the Connector architecture Common Client Interface (CCI) API and a resource adapter to get access to outside data. For more information on implementing the CCI API in your application, refer to the *Java 2 Platform, Enterprise Edition Tutorial*.
 - `java.net.URL`, a URL connection factory.
- **Authorization.** How the user is authenticated and authorized to use the resource.
 - **Container:** The EJB container signs on to the resource manager, using information supplied by the application deployer at deployment time.

- **Application:** The enterprise bean code causes the resource manager sign-on to happen programmatically.

The following field is optional.

- **Sharing Scope.** Whether the connection to this resource can be shared by another enterprise bean in the same application. If two or more beans can use the same resource in the same transaction context, the container can carry out transactions locally and save time.

Specifying Security-Role References

If your enterprise bean does its own security checking, that is, if it checks to see if the user has the authority to use your bean to perform a task, you must provide security-role references in this field. (A message-driven bean needs no security role; it merely propagates whatever security information came from a client in a message. Any needed security checking is done later by either the container or the enterprise bean from which the message-driven bean requests work.)

To use this field, you must also have provided corresponding code (programmatic security) in your bean class. For example, your code might include the following method from the `javax.ejb.EJBContext` interface:

```
isCallerInRole(rolename)
```

In that case, you also need to add all applicable role names as security-role references in this property sheet.

Security roles can also be defined at the module level. For more information, refer to *Building J2EE Applications*.

To add a security role at the bean level, do as follows:

1. **Click the Security Role Reference field and then the ellipsis (...) button.**

The Security Role References property editor appears.

2. **Click Add.**

The Add Security Role Reference dialog box appears.

3. **Complete the fields.**

- **Name.** The name of the security role, as it appears in your bean class code. This field is mandatory.
- **Description.** An explanation of the role. This field is optional.
- **Security Role Link.** A link to a security role in the deployment environment. (This field is optional at the bean level. The data might not be available until deployment time; the field typically is completed by the deployer.)

Using the J2EE RI Tabbed Pane

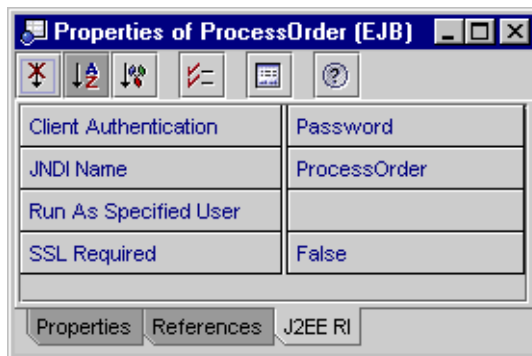
The tabbed pane for the J2EE reference-implementation server (the J2EE RI, or simply the RI) shows properties that are automatically assigned when you use the EJB Builder to create an enterprise bean. These properties contain default values that are appropriate for developing and testing your beans. However, before an entity bean is deployed as part of an application, some of the values must be changed and some blank fields must be filled.

If you're looking at the J2EE RI-related properties of a message-driven bean, a session bean, or an entity bean that manages its own persistence (a BMP bean), you see little data. However, a great deal of information is needed for the J2EE RI-related property fields of an EJB module that contains one or more CMP beans. There, you see the IDE's generated instructions to the container on how to intercede between the bean and its underlying data storage.

First, let's consider the property-field values that should go into the J2EE RI tabbed pane for individual enterprise beans. Then, starting in "Creating an EJB Module" on page 161, we will explore the process of creating an EJB module around enterprise beans, and then, in the case of CMP beans, setting their database-related properties at the EJB-module level.

Setting J2EE RI Properties for Individual Session and Entity Beans

For a session bean, a CMP bean, or a BMP bean, this tabbed pane displays the Client Authentication, JNDI Name, Run As Specified User, and SSL Required properties. The J2EE RI properties tabbed pane for a session bean called `ProcessOrder` is shown next.



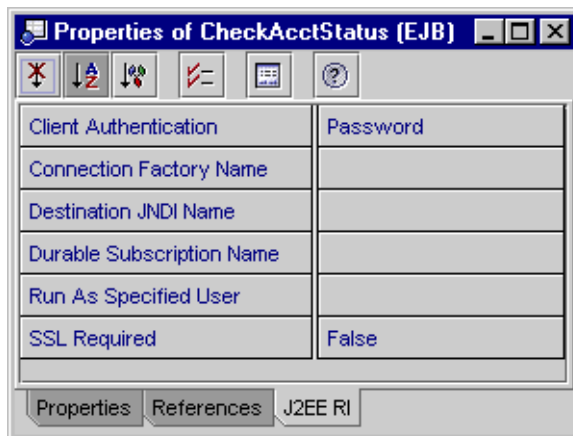
Supply values in these fields as follows:

- **Client Authentication.** Specify whether the client will be authenticated through a password (that is, the user must provide a user ID and a password) or a certificate (that is, a public key certificate must be used, for greater security). Or, specify that the bean must let the client choose between password or certificate authentication.
- **JNDI Name.** The RI plugin populates this field with a default value for the name by which a JNDI lookup call locates your bean before creation. You can change the name if you want, but you can't leave it blank.
- **Run As Specified User.** If you want the bean to use a specific security identity, not the caller's, specify the user identity here. If you selected Run-As Role in the Security dialog box, you can select the user identity from that role.
- **SSL Required.** The default is False. If the bean requires the Secure Socket Layer protocol, change it to True.

When deploying a CMP entity bean to the RI, you first create an EJB module, then you handle the bean's relationships with its underlying data storage at the level of the EJB module. See "Setting Database-related Properties for CMP Entity Beans" on page 163 for details.

Setting J2EE RI Properties for Message-Driven Beans

A message-driven bean's J2EE RI tabbed pane also contains fields that describe what kinds of messages the bean consumes.



Supply values in these fields as follows:

- **Client Authentication.** As for other types of beans, specify here whether the client will be authenticated through a password or a certificate, or that the bean will let the client provide the means of authentication.
- **Connection Factory Name.** Specify the name by which the message-driven bean's queue or topic connection factory is located.
- **Destination JNDI Name.** Specify the name by which the message-driven bean's queue or topic is located.
- **Durable Subscription Name.** If the message-driven bean subscribes to a topic and has a durable subscription, specify the name here.
- **Run As Specified User.** If you want the message-driven bean to use a particular security identity, specify the identity here. Otherwise, the bean uses the caller's identity.
- **SSL Required.** The default is False. If the bean requires the Secure Socket Layer protocol, change it to True.

To set database-related properties for CMP beans, you must first have or create an EJB module around the CMP beans. EJB modules are discussed next.

Creating an EJB Module

Perhaps you have designed an enterprise bean to work on its own, or perhaps you have designed several beans to work cooperatively. In either case, you might want to package your enterprise beans in an EJB module to be sure that the right beans are together, and that all necessary information about the beans' operating requirements is packaged with them.

Enterprise beans don't have to be in the same module to work together; they can be in different modules as long as they are in the same JVM. However, sometimes it's convenient to keep cooperating beans together in the same module.

An EJB module is a logical entity in the IDE, a symbolic representation of its physical counterpart, which is an EJB JAR (a Java archival file with the extension `.jar`). The EJB module tracks the list of beans that need to be included in an EJB JAR, along with the connections between the beans and the properties that need to be set in the deployment environment. An EJB module is the smallest unit of enterprise beans that can be deployed to an application server.

As you can see in the IDE's Explorer window, an EJB module node also represents the module's deployment descriptor, which lists the enterprise beans that compose the module and specifies where the beans' source code is loaded. The beans in an EJB module can be in the same directory or in several directories, even in different filesystems. Neither the beans nor copies of them actually reside in the EJB module.

Deciding What Should Go Into an EJB Module

Here are some general guidelines for deciding how many enterprise beans to package in a single EJB module. (For details, refer to the documentation for the Java 2 Platform, Enterprise Edition.) You might want to package your beans for any of the following results.

- **Maximum reusability.** If one enterprise bean is highly reusable, you might package it in its own EJB module. When the application is assembled, this module can be combined freely with other modules, to supply only the functionality that the application needs and to keep the size of the application down.

You should group into one module any beans that are most likely to be used together. For example, all CMP beans that were built in the EJB 2.0 environment and that have relationships between them must be in the same module.

- **Maximum ease of assembly.** The application assembler has less work to do if you package in one module all the enterprise beans an application needs, or at least all the beans in a chunk of the application. This can be an effective approach if reusability isn't an issue.
- **A balance between reusability and ease of assembly.** For a J2EE application of moderate size, you can probably group any related or closely coupled enterprise beans in one module, and any singly reusable beans each in its own module. Good candidates for grouping are beans with related functionality, dependencies on each other, circular references, or common security profiles.

Putting Enterprise Beans in an EJB Module

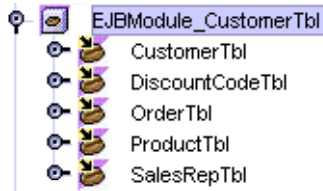
To create an EJB module around a single enterprise bean, do as follows:

1. In the Explorer window, right-click the bean's logical node and choose **Create New EJB Module**.
2. In the **New EJB Module** dialog box, rename the module if necessary.
3. Select a location for the module from the tree view of your filesystem.
4. Click **OK**.

To create an EJB module around multiple cooperating enterprise beans, follow the steps above, but use the Control or Shift key to select at the same time all the beans you want in the module.

Another way to create an EJB module around several beans is to right-click the beans' Java package, and choose New → J2EE → EJB Module. The IDE creates a new, empty EJB module in the package. Right-click the module node and choose Add EJB. In the file chooser that appears, use the Control or Shift key to select all the beans you want to include.

See the contents of the module by expanding its node in the Explorer.



If you need to have multiple CMP beans working together within the EJB module, it would have been best to start by using the EJB Builder wizard to create a set of related CMP beans from a database or database schema. At the same time, the wizard creates the surrounding EJB module. When you approach the task that way, the IDE automatically preserves all the relationships between the beans.

However, if you need to add one enterprise bean to a module you've already created, do as follows:

1. **In the Explorer window, right-click the EJB module's node and choose Add EJB.**
A file chooser appears, showing a tree view of your filesystem.
2. **Select an enterprise bean from the tree view.**
3. **Click OK.**

Setting Database-related Properties for CMP Entity Beans

The RI-related properties for a CMP entity bean are provided at the level of the EJB module. FIGURE 8-2 shows the J2EE RI tabbed pane for an EJB module called EJBModule_AccountsSouth. This EJB module, as you will see a little later, contains several related CMP beans. Notice that the fields in this pane address the relationship between the beans and their underlying data storage, and the role that the container is to play in managing that relationship.

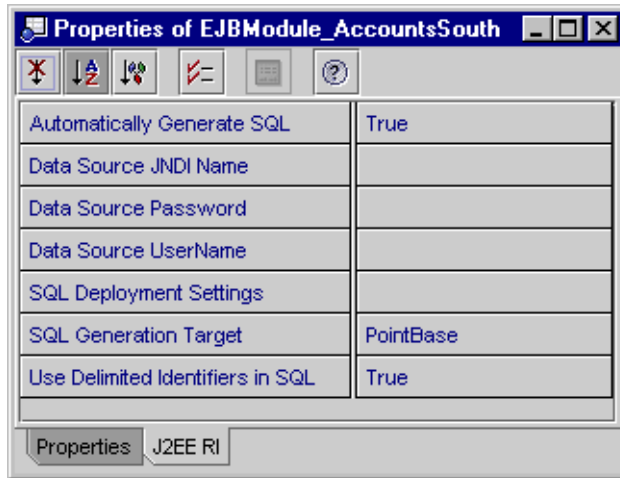


FIGURE 8-2 J2EE RI Tabbed Pane Showing Properties for an EJB Module Containing CMP Beans

Note – If you are going to use the IDE’s testing feature on a CMP bean, you might want to wait until later to create that CMP bean’s EJB module. The testing process creates an EJB module for you. This is not a module you can deploy in production, but one that lets you test your CMP bean’s code and all its deployment settings. You might save time by running your CMP bean through the testing process, finding out how the properties you set there work, and afterward replicating those settings in a “real” EJB module that you generate for production purposes, following the instructions in this section. For information on testing enterprise beans, see Chapter 9.

For all CMP beans, complete the following fields:

- **Data Source JNDI Name.** The JNDI name of the data store that is to be updated with the entity’s state. For example: `jdbc\Pointbase`, where `jdbc` indicates the type of data storage and `Pointbase` indicates the particular data store.
- **Data Source Password.** The password needed for access to the database.
- **Data Source UserName.** The user name needed for access to the database.

Note – If you’re using the PointBase database server included in the IDE, type its JNDI name as shown above, with initial capital only. The default user ID and password are the same: `pbpublic` (remember to press the Enter or Return key after typing the password).

Depending on how you created the CMP bean or beans in the EJB module, and on whether you want your CMP beans to use an existing database table, you treat the rest of the RI-related property fields differently.

Understanding the RI's Generated SQL

When you have used the EJB Builder wizard to create a CMP bean, and you leave the default value `True` in the Automatically Generate SQL field, the J2EE RI plugin provides the SQL statements that it will need to manage the bean's associated persistent records in the underlying database. The plugin uses the bean's methods and fields, and also any EJB QL code that you supplied for certain methods, to produce SQL specifically for the server's own use. This is true of all the servers installed in the IDE: Each uses the bean's code and your EJB QL to generate its own server-specific SQL.

The plugin also can regenerate its SQL statements whenever you add a CMP field, delete a CMP field, change the fields that make up the primary key, or change the bean's container-managed relationships. However, you can choose whether or not to use this SQL regeneration feature. If you do use it, that is, if you leave the value set to `True`, you shouldn't edit the SQL statements unless your EJB module contains a CMP bean with a version 1.x bean class. (In this case, see "If Your EJB Module Contains an EJB 1.1 CMP Entity Bean" on page 168 for the one statement you will need to modify.)

The generated statements are designed to do as follows:

- Create a table when the bean is deployed. The table is named *BeanTable* (where *Bean* is the name of your CMP bean). The table columns are named for their corresponding CMP fields.
- Drop the table when the bean is undeployed.
- Delete, insert, select, and update records as the bean executes.

Guidelines follow for setting the database-related properties of CMP beans.

If Your CMP Beans Don't Need to Use an Existing Database Table

Perhaps your CMP beans don't need to use an existing database table, and you want the RI plugin to create a table for their use, for example during testing. In that case, you can leave the default settings as they are in the Automatically Generate SQL field and the Use Delimited Identifiers in SQL field.

Note – Turn off delimited identifiers only if you are sure that your bean name, CMP field names, and CMR field names are reserved words in the SQL database specified as the data source.

Use the default setting in SQL Generation Target, or specify another database.

To see the default settings for table creation and drop, click the SQL Deployment Settings field and then the ellipsis (...) button. The SQL Deployment Settings property editor appears, as shown in FIGURE 8-3.

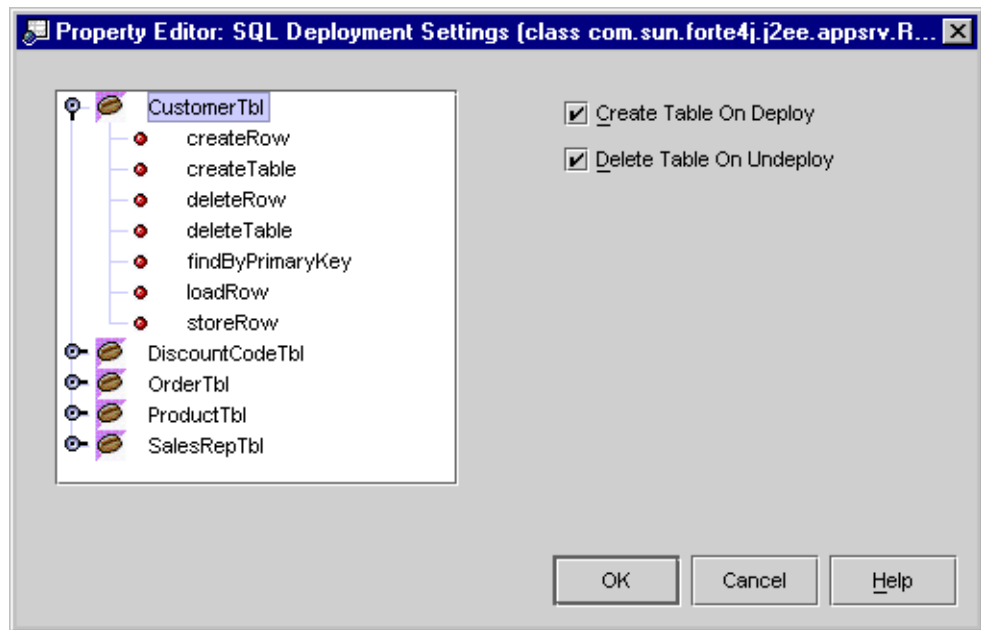


FIGURE 8-3 Table-related Settings for a CMP Entity Bean in an EJB Module

Notice that the table-related settings appear when a bean node is selected.

To see the default SQL statements that have been generated by the RI plugin for your CMP beans, expand a bean node in this property editor and select one of the bean's methods. The method's SQL statements appear, as shown in the example in FIGURE 8-4.

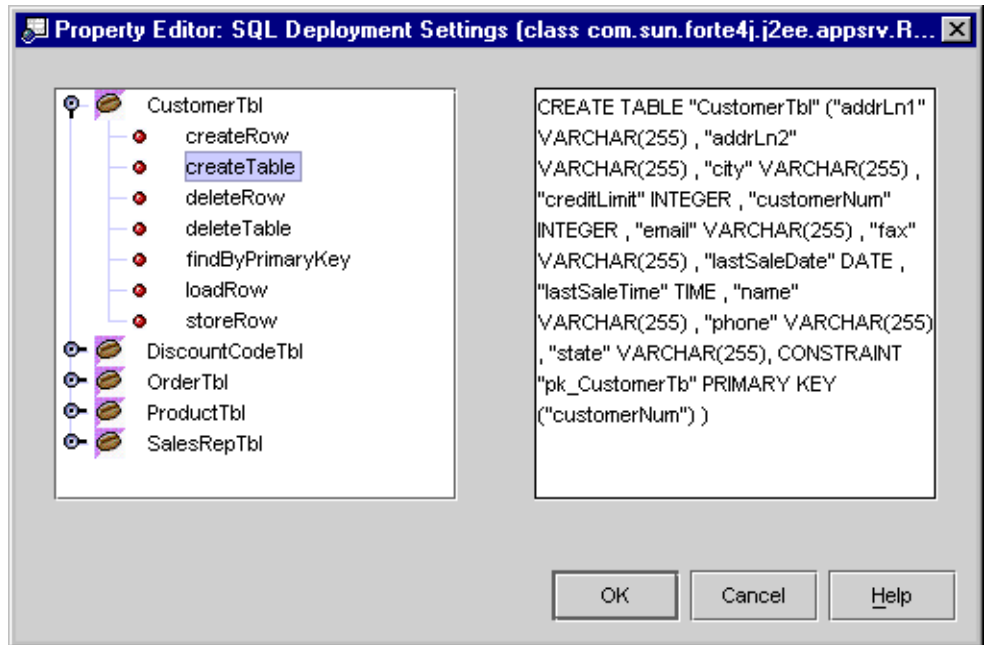


FIGURE 8-4 Example of SQL Code Generated by the RI Plugin for a CMP Bean's `createTable` Method

You should leave the server's generated SQL statements as they are unless your EJB module contains a CMP bean with a version 1.x bean class. In that case, see "If Your EJB Module Contains an EJB 1.1 CMP Entity Bean" on page 168.



Caution – The IDE doesn't prevent you from changing generated SQL statements in a bean's property sheets. However, wherever possible, you should instead make your changes through the bean's EJB QL statements, so that the plugins of any and all application servers you use can propagate your changes automatically. A bean's server-specific SQL code is meant to be generated. You can lose changes that you make in the generated SQL code.

If Your CMP Beans Need To Use an Existing Database Table

If your CMP beans need to use an existing database table (for example, if you generated the beans' infrastructure from a table, and you want to test the beans against that table), use the following settings:

Automatically Generate SQL	True (default)
SQL Generation Target	(The database containing your table. This should be the same database that the Data Source JNDI Name is mapped to.)
Use Delimited Identifiers in SQL	False

The value `True` in the Automatically Generate SQL field means that when you change your CMP beans' persistent fields and container-managed relationships, the RI plugin automatically regenerates your SQL statements and keeps them up to date. (For the one exception to this rule, see the next section.)

Note – Be sure that none of your beans (tables) or their fields (columns) have names that are reserved words in SQL.

Click the SQL Deployment Settings field and then the ellipsis (...) button. In the property editor that appears, select each CMP bean that you will want to test against the existing table. For each of those beans, deselect the checkboxes for Create Table on Deploy and Delete Table on Undeploy. Deselected checkboxes mean that the RI plugin won't create any table for your bean to use, and that it won't drop any table when the bean's application is no longer running.

Again, you shouldn't have to adjust the generated SQL statements unless your EJB module contains an EJB 1.1 CMP bean.

If Your EJB Module Contains an EJB 1.1 CMP Entity Bean

In an application, you might include one or more CMP entity beans that were generated according to the previous EJB specification, using an earlier version of the Sun ONE Studio IDE. Or, a CMP bean in your application might be based on a bean class of version 1.x (for details on creating such a CMP bean, see Chapter 4). If so, you can treat these CMP beans the same as you do EJB 2.0 CMP beans, with one important change. You must go to the generated SQL and adjust the `WHERE` clause in the finder method.

To make this edit, you should be in the EJB module's property sheet, the J2EE RI tabbed pane, and the SQL Deployment Settings field. Click that field and then the ellipsis (...) button, and the SQL Deployment Settings property editor appears.

Edit the WHERE clause as follows:

1. Find the EJB 1.1 CMP bean node and expand it.
2. Select the node for the bean's finder method.

You see an SQL statement something like the one in FIGURE 8-5.

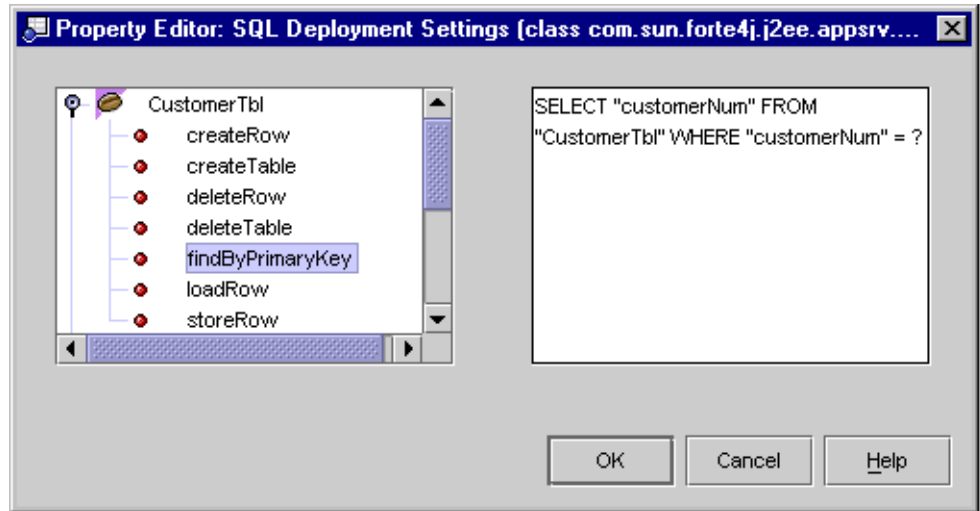


FIGURE 8-5 Example of SQL Code Generated for a CMP Bean's Finder Method

3. Complete the WHERE clause.

- For the `findByPrimaryKey` method, if you are not using the plugin's generated SQL, you can change the column name. Otherwise, leave this method as it is.
- For other finder methods, you must supply the constraint. For example, a finder method called `findByY2000Accounts` might read as follows:

```
findInCustIDRange(double low, double high)
```

In that case, the SQL statement might read as follows:

```
SELECT "custID" FROM "CustomerTbl" WHERE "custID" > ?2000 AND  
"custID" < ?2001
```

The number after the question marks indicates the corresponding parameter in the method's parameter list.

Your edits in the WHERE clause are preserved when the RI plugin regenerates the SQL statements.

See Appendix B for more details on handling EJB 1.1 CMP entity beans in the current version of the IDE.

Understanding the Order of CMP Field Values

If you have occasion to change any of the SQL that the RI plugin generates for your CMPs, you might need to understand how the values of the CMP fields are ordered.

The container sends the values of the CMP fields to the database in alphabetic order. If your table columns do not have the same names as the CMP field names, you must put the columns in the same order as the alphabetic order of their mapped fields. For example, assume that your CMP bean class is named `EmployeeEJB`.

- The source database table is named `Employee`.
- Your CMP fields are named `address`, `city`, `id`, and `name`.
- Your mapped table columns are named `Mail_Address`, `City`, `Emp_ID`, and `Emp_Name`.
- The default generated SQL for the `INSERT` statement is as follows:

```
INSERT INTO "EmployeeEJBTable" ("address", "city", "id", "name")
VALUES (?, ?, ?, ?)
```

In this example, you would need to change the SQL code as follows:

```
INSERT INTO "Employee" ("Mail_Address", "City", "Emp_ID",
"Emp_Name") VALUES (?, ?, ?, ?)
```

The table column names are in the same alphabetic order as the mapped CMP fields (not the alphabetic order of the table columns). This is because the `VALUES` parameters are sent in that order.

Adding Transaction Attributes to an EJB Module

Transaction attributes tell the EJB container how to control a CMT bean's transactions. In a session bean that manages its own transactions (a BMT bean), you must explicitly code the bean's transactions. In a CMT bean, you include no explicit code for transactions; instead, you let the container control them based on transaction attributes you assign to your bean's methods.

By default, the IDE sets all bean methods to use the `Required` attribute. You can assign different transaction attributes at the bean level or at the method level. For example, if you don't want a particular method to be included in the context of a transaction, you can change that method's attribute from `Required` to `Not Supported`.

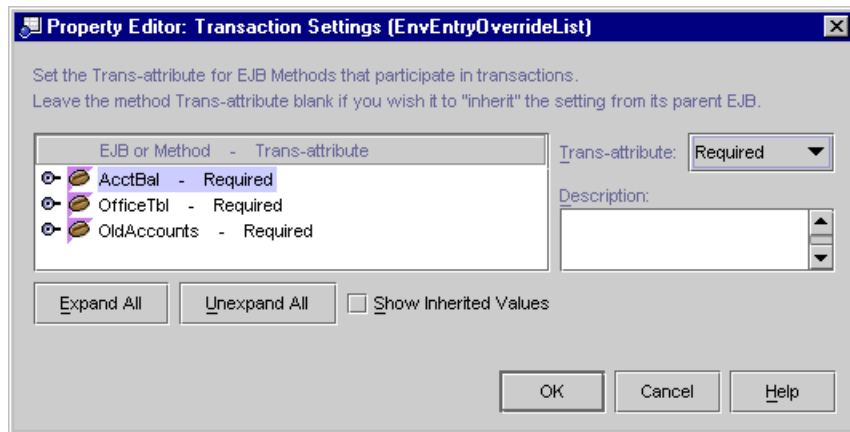
The transaction attributes are stored in the deployment descriptor, and they can be edited using the EJB module's property sheet. Before an EJB module that contains CMT beans is assembled into an application, you should make sure that the appropriate transaction attributes are specified for that module.

When you change a transaction attribute for an individual bean or for a method in a bean, the attribute is changed only for execution within that EJB module. The bean's source code has not been changed. If you reuse that bean within another EJB module, you can apply a different set of transaction attributes.

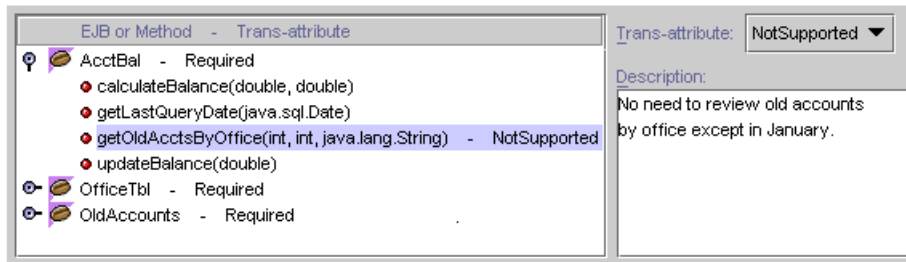
To change a CMT bean's transaction attributes within an EJB module, do as follows:

1. **In the Explorer window, right-click the bean's EJB module and select Properties.**
In the Properties tabbed pane, notice that the Transaction Settings field displays the value `Container-transaction`.
2. **In the Properties tabbed pane, click the Container-transaction field, and click the ellipsis (...) button.**

The Transaction Settings dialog box appears. Notice that the larger pane displays the module's enterprise beans that use CMT, each bean with a transaction attribute that applies to the entire bean.



- To change a bean's transaction attribute, select the bean and select another item from the Trans-attribute combo box. The new attribute appears beside the bean's name in the larger window.
- To change a particular method's transaction attribute, expand the bean, select the method, and select another item from the Trans-attribute combo box. The new attribute appears beside the method's name in the larger window, as shown next. Notice that the bean's other methods display no transaction attribute unless the default attribute is overridden.



While you're working in the Transaction Settings dialog box, you can take the opportunity to provide the application assembler a description of the transaction settings for individual beans or methods in the module. For example, you might want to explain why you've changed a particular transaction attribute in this EJB module.

3. Click OK when you're finished with transaction attributes.

Changing EJB References Within the EJB Module or Application

In "Specifying EJB Local References" on page 153 and "Specifying EJB References" on page 155, you saw how references between enterprise beans are declared at the level of the individual bean. Those references also can be overridden as follows:

- If the beans are in the same EJB module, you can override the references at the level of the EJB module.
- If the beans are in different EJB modules but within the same application, you can override the references at the level of the application.

This feature is handy if, for example, you want to package a particular enterprise bean in three different EJB modules to be used variously by one or more applications.

To use this feature, you must provide the EJB module (or application) two or more enterprise beans whose interfaces are the same. The feature works for local, remote, or local and remote interfaces, but the beans' interfaces must be identical, while the bean classes or properties (deployment information) can differ.

Overriding a Reference at the Module Level

To override a bean reference within the EJB module, do as follows:

1. In the Explorer window, right-click the bean's EJB module and select Properties.
2. Click the applicable references field (EJB Local References or EJB References) and then click the ellipsis (...) button.

The appropriate property editor appears.

3. Find the enterprise bean whose reference you want to override and click the Override checkbox.
4. Click the Override Value field.

Below the Override Value field appears a combo box containing the names of the enterprise beans you can select from. An example is shown in FIGURE 8-6.

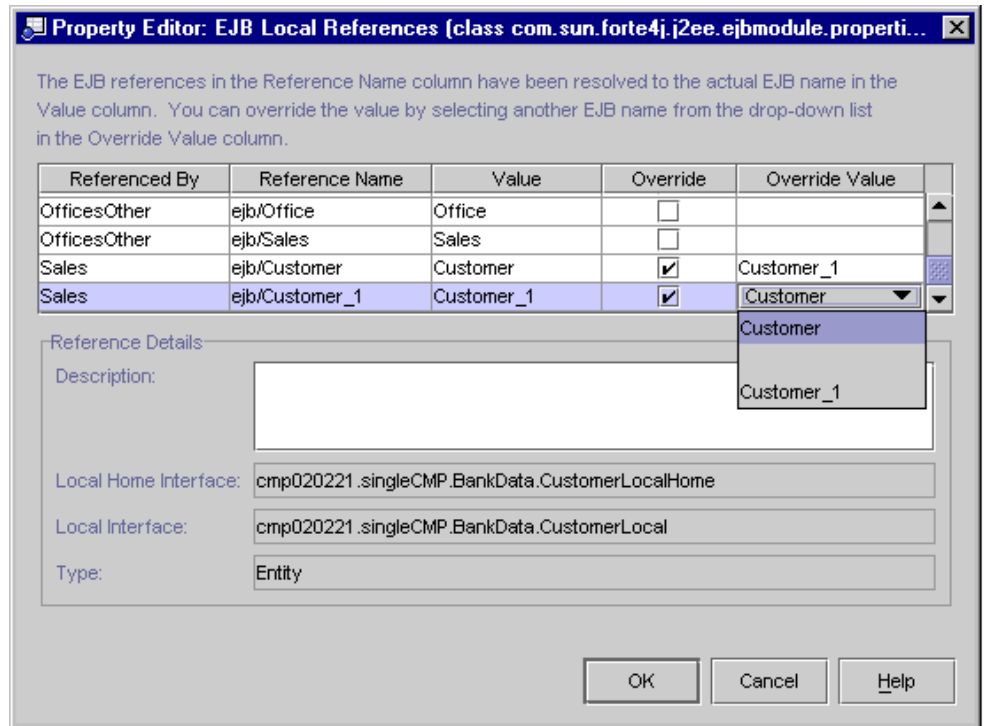


FIGURE 8-6 EJB Local References Property Editor, Showing an Example of Override Selections for an Enterprise Bean's Local References

5. Select the reference you want the enterprise bean to use, and click OK.

Overriding a Reference at the Application Level

To override a bean reference at the application level, do as follows:

1. **In the Explorer window, right-click the bean's application and select Properties.**
2. **Click the applicable references field (EJB Local References or EJB References) and then click the ellipsis (...) button.**

The appropriate property editor appears.

The property editors for EJB references and EJB local references are the same at the module and application level. Continue with the same steps as described in the previous section.

Creating an EJB JAR

After creating the EJB module, if you will be handing off the module to another part of the organization for application assembly, you might want to create an EJB JAR. This file can be used to export the module and its contents for use in an application. Create the EJB JAR file as follows:

1. **In the Explorer window, select the EJB module.**
2. **Right-click and select Export EJB JAR File.**

The Output Window appears and shows you the progress as the module and its contents are compiled. Notice any error messages in this window.

After the EJB JAR file has been created, you can still adjust the module or its contents for use in this or another EJB JAR file.

One or more EJB modules can be placed in a J2EE application and deployed. For details on assembly and deployment, refer to *Building J2EE Applications*.

Adding a JAR File to an EJB Module

You might want to let the enterprise beans in an EJB module take advantage of the functionality offered by beans in another container. For example, if you have a session bean that needs to authenticate a user, you can add a JAR file containing beans that implement security checks. Add an existing EJB JAR file to your EJB module as follows:

1. **In the Explorer's Filesystems pane, mount the filesystem that contains the JAR file.**
2. **Right-click your EJB module's mode and choose Properties.**

3. In the module's property sheet, click the Extra Files field and then the ellipsis (...) button.
4. In the Source pane, navigate to the JAR file you want to add, select the file, and click Add.

The JAR file appears in the Files To Be Added pane.

5. Click OK to add the JAR.

When you deploy the EJB module, this JAR file is included in the module's EJB JAR file.

Testing Enterprise Beans

As you develop enterprise beans, you might find it expedient to test them before doing a full-scale application assembly and deployment to a production application server. Using the Sun ONE Studio IDE, you generate a J2EE application for this purpose, including a web module with JavaServer Pages™ (JSP™) test pages and an EJB module for your bean. You then use the IDE's test feature to display the JSP page's resulting HTML page in a web browser. In the HTML page, you can create instances of an enterprise bean and exercise the bean's methods.

The objects that the IDE creates for you are designed for use during the test process. They are not intended for deployment in a production environment.

You can use the IDE's testing feature with any supported database and application server. The following instructions and example use PointBase as the test database, the J2EE RI application server (the RI) as the test server, and Netscape Navigator as the web browser.

Fulfilling the Prerequisites for Testing

Set-up can vary according to bean type. Your testing set-up can vary depending on which type of enterprise bean you want to test. Some preparation pointers follow.

- **Consider how you want to set up the EJB module.** To test a session bean, you can simply use the EJB module generated by the testing feature. However, an entity bean's EJB module can be handled two different ways. See "If You Want to Test a CMP or BMP Bean" on page 189.

Message-driven beans are not supported by the testing feature in this release of the IDE.

- **Have all referenced beans in the same module.** With the testing feature, you exercise one bean at a time. However, the bean being tested must have any bean it references available for its use. See “If You Want to Test a Bean With EJB References” on page 190.
- **Remote references are required.** Any enterprise bean you want to test must have remote interfaces; it can also have local interfaces, but the testing feature requires remote ones. A bean referred to by the bean you’re testing can have either or both kinds of interfaces. For details, see “Adding Remote Interfaces to a Bean” on page 190.
- **Your bean’s resources must be available.** You must have the RI and the required database server up and running. See the following section for details.

Variations on the set-up for enterprise bean tests are described in “If You Want to Test a CMP or BMP Bean” on page 189.

Preparing to Deploy to the J2EE RI

To use the IDE’s testing feature on an enterprise bean, you must be able to deploy your bean’s application to the RI, which must be installed locally on your machine. At least one server instance of the RI must be running.

The RI server installation and instantiation should have been done automatically when you installed the IDE. If you need detailed instructions, refer to the RI’s online help.

Preparing to Test Beans Against the PointBase Database

You can use the PointBase database, which is included in the IDE, to test any enterprise bean that needs access to a database. (Normally, this means only entity beans.)

- If you have decided to let the testing feature create an EJB module for your use, apply the following instructions when you reach that point in the process.
- If you have decided to use an existing EJB module to test entity beans, be sure the module’s properties are set so that your test application will be able to find and log into the database.

Set these properties as follows:

1. In the IDE's Explorer window, right-click the EJB module node and choose **Properties**.

The module's property sheet appears.

2. In the J2EE RI tab of the Properties window, specify the connection to the database as follows:

Field	Your Input
Data Source JNDI Name	jdbc/Pointbase
Data Source Password	PBPUBLIC
Data Source UserName	PBPUBLIC

Note – In this window, type **Pointbase** with an initial capital only. After you type the password, press Enter.

Two of the RI's default settings make the server create a table for the bean being tested, and drop the table when testing is done. If your bean must have live data from a particular database table, make the following change:

3. Click the **SQL Deployment Settings** field and then the ellipsis (...) button.

The SQL Deployment Settings property editor appears.

4. Select the bean you want to test, and clear the following two checkboxes so that they are blank:

- Create table when deployed
- Drop table when undeployed

5. Click **OK** and dismiss the EJB module's property sheet.

6. Save your work with **File** → **Save All**.

Starting the Database Server and Web Browser

To start the PointBase database server, choose **Tools** → **PointBase Network Server** → **Start Server** from the main menu.

Launch your web browser as you usually do.

You might want to minimize both of those windows, but don't close them down until you're finished with your testing activities.

Generating Test Objects

Now you're ready to use the IDE's wizard to provide the EJB module, the web module, and the application with which you will test your enterprise bean.

Follow these steps to generate test objects for your enterprise bean:

1. **In the Explorer window, select the bean's logical node, right-click, and choose Create New EJB Test Application.**

A wizard appears, showing default values for all the components needed to test your application.

Note – If you right-click one bean and see that the Create New EJB Test Application menu item is disabled, the bean probably has no remote interfaces. Follow the directions in “Adding Remote Interfaces to a Bean” on page 190 and try again.

Notice that the Package field displays the current package name where your bean resides. You can move the EJB module and test application objects that the wizard is about to create by typing other package names into those fields.

In the next fields, you can also specify other package and module names if necessary. However, if you plan to substitute an existing EJB module for the one that this feature will generate, don't specify the existing module here. Instead, you'll use the application's property sheets later in the process.

Notice the selections in the Application Server combo box. You can use the RI as your testing server, or you can use another server that you have specified as your default server.

2. **Mark the Auto Deploy checkbox or leave it blank.**

If you mark the checkbox, the IDE automatically deploys your bean's testing module to the server as soon as you've finished creating it in the wizard. You might want to use this approach if the bean you're testing is stand-alone and needs no other beans to do its work.

If you leave the checkbox blank, you will handle the deployment yourself in a later step. Use this approach if your bean needs to be tested in conjunction with other beans.

3. Click OK to generate the EJB module, the web module, and the application, and (if applicable) to deploy the application automatically.

A progress monitor tells you how the module generation and deployment are going. When deployment is complete, a message appears in the IDE's log window.

Assuming that you have let the wizard place what it produced in the bean's package, the generated objects for a bean named `Account` in a package named `Accounts_North` would look like the example in FIGURE 9-1.

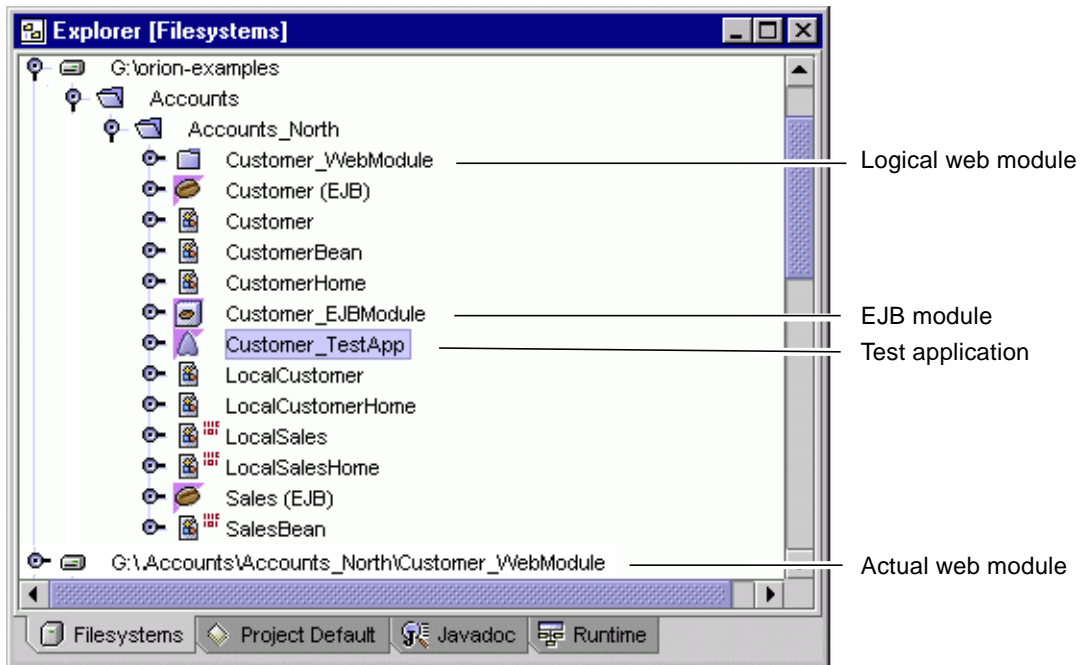


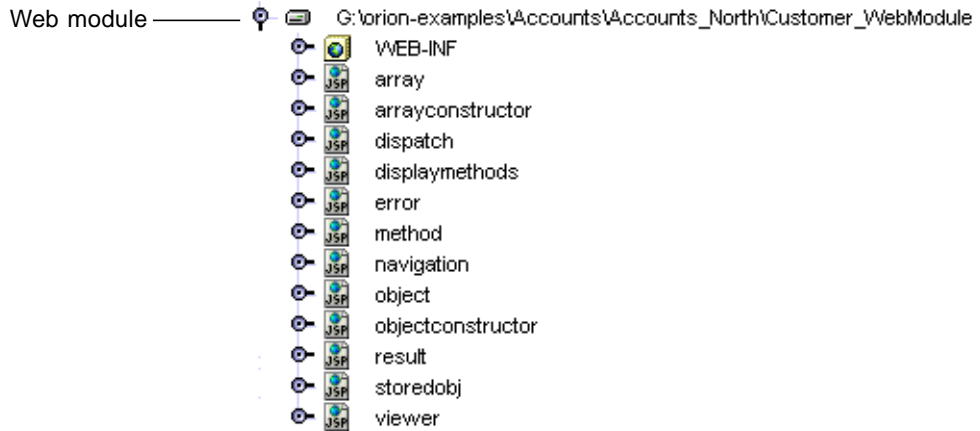
FIGURE 9-1 Example of Test Objects Generated for Enterprise Beans

Notice the generated objects that have been added to the bean's package:

- A logical web module
- An EJB module containing the enterprise bean to be tested
- A test application containing the EJB module and the web module

Unless you specified otherwise in the wizard, the IDE has placed the actual web module (containing JSP pages and helper Java classes) in a new filesystem of its own.

Notice also that the wizard created a new filesystem to contain the web module and other objects for use in testing. When you expand the filesystem node, you see the web module as the first subnode. This web module, shown next, contains the JSP pages and helper Java classes.



You have now generated the basic test application. If your bean references any other beans, add them to the EJB module as follows:

4. In the Explorer window, select the generated EJB module node, right-click, and choose Add EJB.
5. In the tree view, navigate to the referenced enterprise bean, select it, and click OK.

The referenced bean is added to the EJB module, and a reference to the bean is added to the test application.

Repeat Step 4 and Step 5 for each referenced bean.

Now, unless you marked the Auto Deploy checkbox in Step 2, you're ready to deploy your test application to a server. Or, if you prefer, you can deploy and execute the application in one step. The following sections describe each approach.

Deploying the Test Application to a Server

To deploy your test application to the RI, do the following:

1. In the Explorer window, select the J2EE application node, right-click, and choose Deploy.

A progress monitor and output window tell you how the deployment is going. You see a message, for example, indicating that the test application has been deployed on the RI (which is referred to as localhost).

2. Verify that your test application has been deployed by opening the J2EE command window and checking for the statement “Application *name_TestApp* deployed.”

Tip – If your deployment failed, check to see that you set the IDE to run the J2EE RI properly. Especially, notice whether the `RI_Home` property is set to your `J2EE_HOME` value.

When the deployment is successful, you can follow the steps under “Executing the Test Application” on page 184.

Deploying and Executing the Test Application in One Step

To deploy your test application to a server and start it executing at the same time, do the following:

- **In the Explorer window, select the J2EE application node, right-click, and choose Execute.**

A progress monitor and output window tell you how the deployment and execution are going. You see messages, for example, indicating that the application server has been contacted, the enterprise bean or beans have been deployed on the server (notice that the RI is referred to as `localhost`), the wrapper and RMI-IIOP code have been compiled, the JAR or JARs for the server and client have been made, the web server has been contacted and asked to run the test application, and all the generated code has been saved.

When this double step is complete, a web browser appears and opens to the test client, a JSP page that contains the GUI for testing your enterprise beans. An example is shown in FIGURE 9-2.

Continue at “Using the Test Client to Test Your Beans” on page 184.

Executing the Test Application

If you didn't choose Execute to deploy and execute your test application in one step, you can do the following to test your enterprise bean or beans:

- **Open a web browser and type in the appropriate URL.**

This URL has the following format if you are using the J2EE RI:

`http://localhost:port/application_name/`

port is the port number you indicated when you installed the RI.

application_name is the name of the application.

The test application's client, a JSP page, appears in the browser.

Using the Test Client to Test Your Beans

You follow the instructions on the test client's JSP page to create instances of your enterprise bean and call its business methods. The following section describes how you might test a very simple session bean called `dollarToYen`, which converts amounts in U.S. dollars to the equivalent values in Japanese yen. (This session bean happens to reside in the Java package `Converter`.)

Understanding the Test Client Page

FIGURE 9-2 shows an example of a JSP page that the IDE has created for the application client that was generated to test the example session bean.

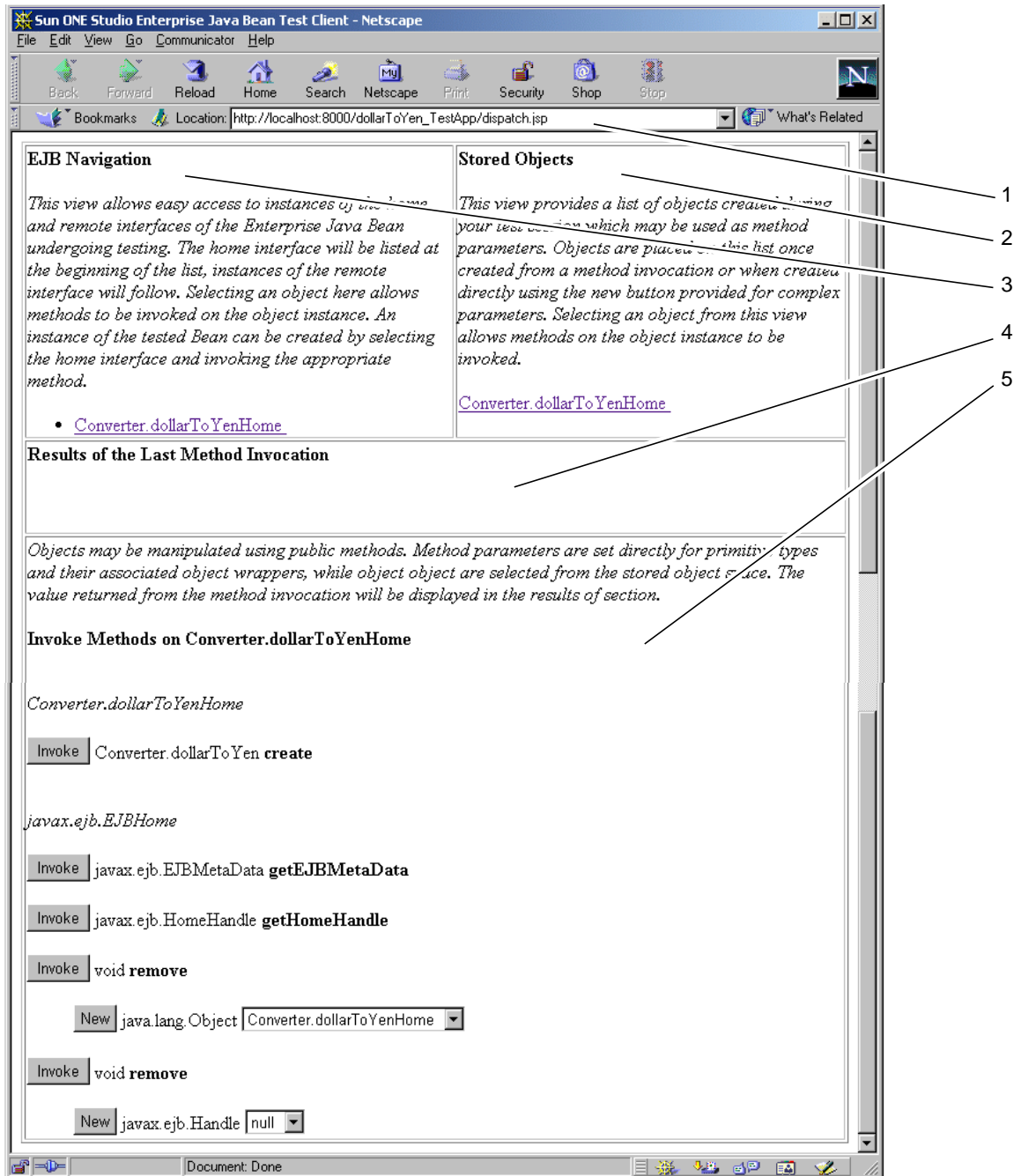


FIGURE 9-2 Client JSP Page Generated to Test Simple Session Bean dollarToYen

As shown in FIGURE 9-2, the parts of the testing window are as follows:

1. The browser's URL field shows the location of the test client's JSP page. Your client's URL is generated automatically by the IDE's testing feature. If you like, you can use this URL to return to this testing window.
2. The Stored Objects frame shows the stack of objects created during your testing sequence by the IDE or by your own actions, for example, when you have called a method on an interface or on a bean class. Right now, only the home interface is shown.

You can remove objects from the Stored Objects stack by using the Remove Selected or Remove All button.

3. The EJB Navigation frame shows the objects that the IDE has generated so that you can test your bean. If the bean you're testing has references to other beans and your EJB module contains more than one bean, this frame lists the created objects in logical order, that is, in the order that one bean in the module calls another.

In the EJB Navigation frame in FIGURE 9-2, you see `Converter.dollarToYenHome`, which shows that the IDE has created the session bean's home interface. Later, you will click this home interface to create and initialize a new instance of the session bean.

When you see more than one object listed in this frame, you click an object to change focus to the bean component you want to test. When you click an object, notice how the other frames change.

4. The Results of the Last Method Invocation frame shows, for example, the last method you called and its parameters. Right now, since we haven't yet begun testing the session bean, nothing appears in this frame.
5. The bottom frame shows the methods that are now available for you to test. This frame changes as you place focus on different components of the bean (listed in the EJB Navigation frame).

Now let's test the `dollarToYen` bean's home interface and business method.

Testing the Sample Bean's Home Interface

To verify that `Converter.dollarToYenHome` correctly creates an instance of the session bean, do as follows:

1. **In the EJB Navigation frame, click the home interface name.**
2. **In the bottom frame, click the Invoke button under the home interface name.**

In this case, you click the Invoke button under `Converter.dollarToYenHome`.

Notice the following changes in the page's frames:

- In the EJB Navigation frame, an instance of the bean has been added. In this particular case, the instance is called `Converter.dollarToYen` and followed by a process number.
- In the Stored Objects frame, the bean instance has been added to the top of the stack.
- In the Results frame, the bean instance is reflected, along with the fact that the `create` method was invoked with no parameters.

Now let's test the `dollarToYen` bean's business method.

Testing the Sample Bean's Business Method

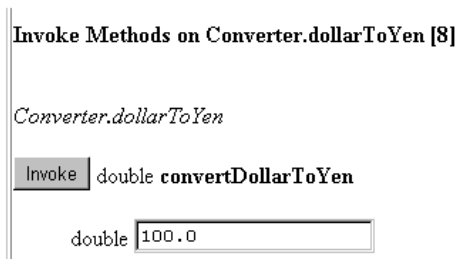
To verify that the instance of `Converter.dollarToYen` correctly converts U.S. dollar amounts to Japanese yen, call the bean's business method as follows:

1. **In the EJB Navigation frame, click the bean name (under the home interface name).**

Notice that the Results frame is cleared and that the bottom frame's list of invocable methods now starts with the bean's business method.

2. **In the bottom frame, under the business method, type a parameter into the input field and then click the Invoke button.**

As shown next, we use the parameter 100 (and the IDE adds the `.0`).



After clicking Invoke, notice the following changes in the page's frames:

- In the Results frame, as shown next, are the result of the method invocation (the result is 12755 yen, because the business method included a calculation of 127.55 yen to the dollar) and the parameter we input for testing purposes (100 dollars).

Results of the Last Method Invocation

12755.0

Method Invoked: *convertDollarToYen (double)*

Parameters:

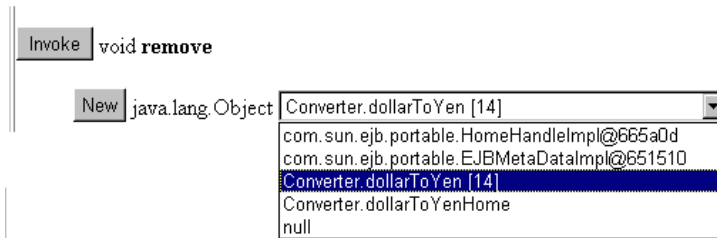
100.0

- In the Stored Objects frame, the stack includes the result object and the parameter object.

You can call any other bean methods that are shown in the bottom frame.

Creating New Testing Classes

When you create an object by invoking the bean's other methods, that object can be used to test the bean further. For example, the objects created so far by testing `Converter.dollarToYen` also appear in a combo box, as shown in the following example. You can select one of the objects and create a new class to use for testing.



Making Changes After Deployment

You can make changes in an enterprise bean and retest it without generating another test application. However, you must redeploy the test application before testing a bean any of whose components you have modified. Do this by closing the testing window, going to the Explorer window, right-clicking the test application module, and choosing **Execute**. Don't try to redeploy a bean during the same testing session.



Caution – The IDE generates an EJB module, a web module, and a J2EE application that are designed only to be used to test your enterprise bean. These generated objects are not meant to be modified. If you modify any of them, you might not be able to deploy the J2EE application again.

Preparing to Test: Some Variations

As mentioned in “Fulfilling the Prerequisites for Testing” on page 177, if you’re not testing something like a simple session bean that already has its remote interfaces, you need to consider the preparatory steps described next.

If You Want to Test a CMP or BMP Bean

You can test an entity bean in one of the following ways:

- You can test your bean in an EJB module that is automatically created for you during the testing process and in which you declare data-source-related properties before going on to test the bean.
- You can create the bean’s EJB module outside the testing process (see “Putting Enterprise Beans in an EJB Module” on page 162) and make all your properties declarations there. Then, before you begin testing the bean, you can substitute the EJB module you created for the one that the testing feature generates.

Either way, when you’re testing an entity bean, you must use an EJB module in which the bean’s properties are declared. Since the EJB module generated by the testing feature is designed only for test use, you will also have to create a “real” EJB module for your production entity beans. You can do this before or after testing.

If You Want to Test a Bean With EJB References

If you want to test an enterprise bean that is designed to interact with another bean, (for example, if you're testing a session bean that manages work done by an entity bean), you must make sure the referenced bean is included in the EJB module.

You can take either of the following approaches:

- **Use an existing EJB module** that has all the necessary properties specified. To do this, you specify the EJB module in the Create a New EJB Test Application wizard, using the Modify button.
- **Build the module around the referencing bean and add the referenced beans.** To do this, first create a test application around the bean you want to test. Then, in the Explorer window, find the EJB module that the IDE created for you. Right-click the EJB module node and choose Add EJB.

Before you generate the test application, make sure that all necessary EJB references are specified in the beans' property sheets, and that any necessary overrides are specified in the EJB module's property sheet. Otherwise, the test might fail or give mixed results.

See "Specifying EJB References" on page 155 for details.

Adding Remote Interfaces to a Bean

If the bean you're testing has only local interfaces, you can prepare it for testing by copying the local interfaces to create remote ones. Briefly, you use existing interfaces as follows:

- Use the local interface (`LocalBean_name`) to create the corresponding remote interface (`Bean_name`).
- Use the local home interface (`LocalBean_nameHome`) to create the corresponding home interface (`Bean_nameHome`).



Caution – Don't try to use this process to add EJB 2.0 features (such as local interfaces or references) to an EJB 1.1 CMP entity bean that has not been fully updated to conform to the current version of the *Enterprise JavaBeans Specification*. The resulting bean will be invalid and irreparable. See "Avoiding the Use of New Features in an Old Bean" on page 210 for details.

Add remote interfaces one at a time. (These instructions start with the local interface and the corresponding remote interface. Later you will repeat the process with the local home interface and the home interface.)

Do as follows:

1. In the Explorer window, right-click the interface node labeled `LocalBean_name` and choose **Copy** from the menu.
2. Select the bean's Java package, right-click, and choose **Paste → Copy**.
A copy of the interface appears in the folder, labeled `LocalBean_name_1`.
3. Select the copy, right-click, and choose **Rename**. In the **Rename** dialog box, name the copied interface according to the J2EE convention: *Bean_name*.
4. In the Explorer, right-click the bean's logical node and choose **Customize**.

The Customizer dialog box appears.

Notice that the Customizer deals with many of the properties also available in the property sheet that appears when you right-click the bean's logical node and choose **Properties**. You can change many of the bean's properties in either dialog box. However, the Customizer does deals only with properties on the bean itself, not with properties relating to application servers. Also, if you want to add remote interfaces to a bean, you can do so only in the Customizer.

5. In the Customizer dialog box, find the empty **Remote Interface** field and click the **Browse** button.

The Select a Class file chooser appears, as shown in FIGURE 9-3.

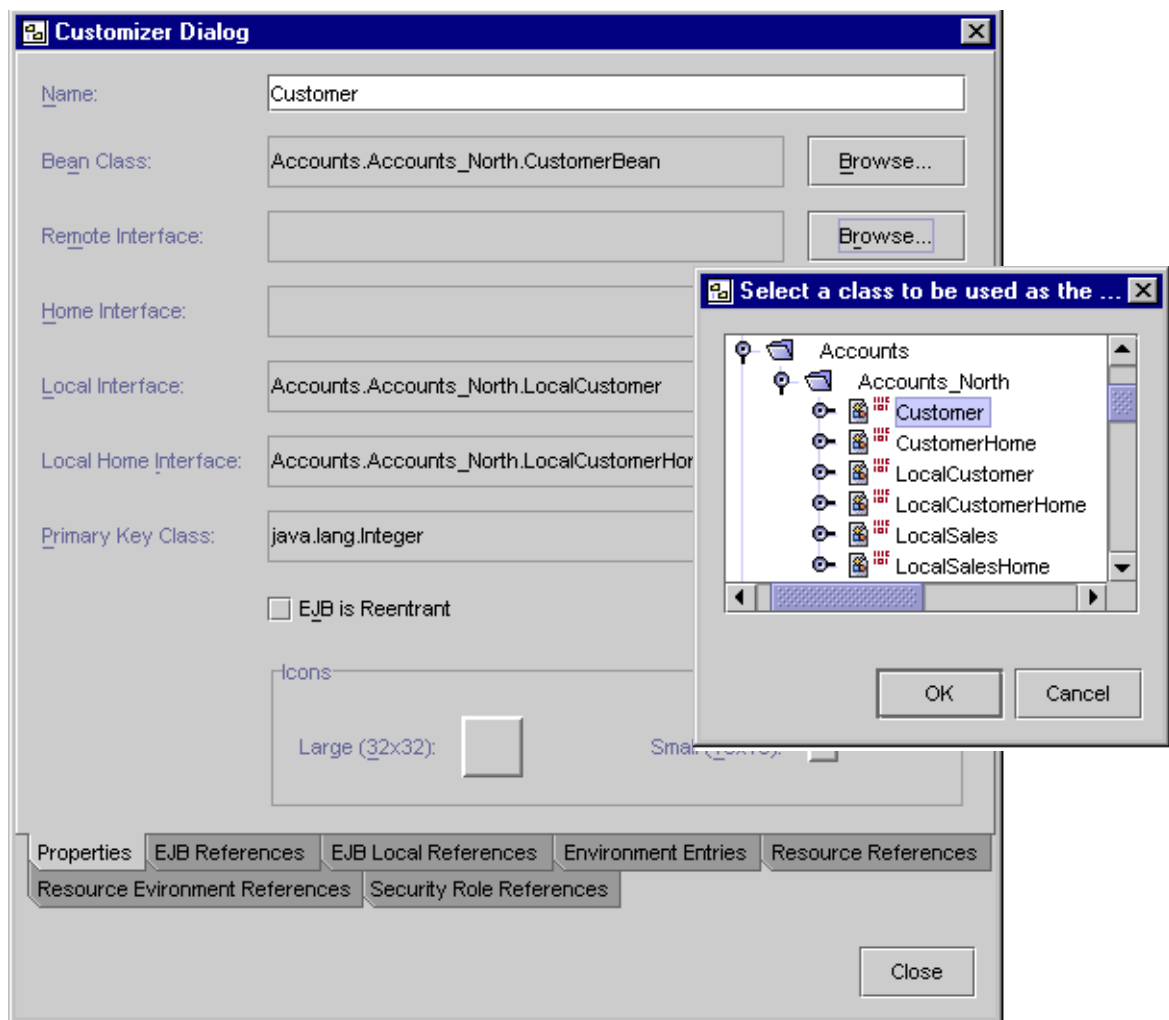


FIGURE 9-3 Customizer for Adding an Interface Class to a Bean

6. Navigate to the interface class you created by copying and pasting. Select the interface node and click OK.

The class appears in the Remote Interface field.

(You will see later that the Remote Interface field in the bean's property sheet has automatically been updated.)

7. Dismiss the Customizer dialog box.

8. In the Explorer, expand the logical bean node. Expand the Classes node, select the new remote interface (labeled Remote Interface Class), and open it in the Source Editor.

Edit the remote interface class to extend `javax.ejb.EJBObject`.

9. For each method in the remote class, add the exception

`java.rmi.RemoteException`.

10. For each `ejbCreate` method, change the return type to the new remote interface.
11. For each finder method that finds a single object, change the return type to the new remote interface.
12. Compile your enterprise bean and resolve any errors.

13. Use the same process (Step 1 through Step 12) to create a home interface from a copy of the local home interface.

However, in Step 8, edit the home interface class to extend `javax.ejb.EJBHome`.

Note – Be sure to remove from the two new interfaces any methods that cannot be used as remote methods.

Working With Enterprise Beans


The relationships between the elements of an enterprise bean can be intricate and complex. The Sun ONE Studio IDE makes certain assumptions to preserve the integrity of your beans, but also flexibly supports various options for reusing beans. This appendix prescribes the best practices for working with existing enterprise beans.

Using Recommended Approaches When Editing Beans

To be certain that changes take place as you intended, make your edits through the enterprise bean's logical node and property sheets. If you use these recommended approaches, the IDE can ensure that the standards in the J2EE specification are followed.

These approaches are explained next.

Working Through the Logical Node

As a general rule, you should go through the logical node of your enterprise bean to make changes in the bean's code. This node's icon looks like a coffee bean: . The node was designed to organize all the elements of your enterprise bean.

When you work through the logical node, the IDE can most easily propagate the changes correctly throughout the bean.

All the classes in your bean are represented under this one node, as if the bean were a single object. There you can edit your enterprise beans without having to think about which class must receive each change. For example:

- When you add a new method to the Create Methods node under a bean's logical node, the body of that method (`ejbCreateXxx`) and its related method (`ejbPostCreateXxx`, required for entity beans) are added to the bean class. Depending on which types of interface the bean has, the corresponding method signature (`createXxx`) is added to the local home interface, the home interface, or both.
- When you add a new method to the Finder Methods node under an entity bean's logical node, the Add New Finder Method dialog box prompts you for the correct name of the method. The method signature is added to the local home interface, the home interface, or both.
 - In a CMP entity bean, the Add New Finder Method dialog box also prompts you for EJB QL Select, From, and Where statements. When you deploy your bean to a supported application server, this EJB QL code is automatically converted into the kind of SQL that the server needs.

If you want to define a new finder or select method but you aren't ready to provide the EJB QL code yet, you can disable the EJB Compiler's requirement for EJB QL. See "Compiling and Validating Enterprise Beans" on page 199.
- In a BMP entity bean, the corresponding `ejbFind` method is also added to the bean class.)
- When you add a new method to the Business Methods node under a bean's logical node, the method body is added to the bean class. Depending on which types of interface the bean has, the method signature is added to the local interface, the remote interface, or both.
- When you add a new method to the Home Methods node under the logical node, the method body is added to the bean class, and the method signature is added to the appropriate interface or interfaces.
- When you add a new method to the Select Methods node under a CMP entity bean's logical node, the Add New Select Method dialog box prompts you for all the information needed to complete the method, including EJB QL statements. The method body is added to the bean class.

In many cases, the IDE can propagate your changes and synchronize your bean's classes and interface even when you go through other nodes of your bean to edit the Java code. However, you get the most consistent results by working through the logical node.

Using the Customizer or Property Sheet

When you need to modify a method's name or return type, or when you need to edit or add a parameter or exception, the best place to do so is in the method's Customizer dialog box or property sheet. Under the logical node, select the method, right-click, and choose Customize or Properties.

- The Customizer has the same format as the dialog box in which you created the method.
- The property sheet shows the method's parts in tabbed interfaces that correspond to the classes that the method inhabits.

Any change you make in one of these two places is validated and propagated in the right form to the right classes.

When you need to add code to complete a method, use the IDE's Source Editor.



Caution – If you make changes outside the logical node, by working within the bean class node or one of the interface nodes, the EJB Builder still tries to propagate your changes. However, in certain cases, you might need to manually ensure that your code matches Sun's J2EE specification. See the specific examples that follow.

Using the Source Editor to Edit Beans

You can create or modify any part of an enterprise bean by using the IDE's Source Editor exclusively. However, the IDE's wizards and other GUI tools are designed to save you work and to help prevent inconsistencies, so that you can produce standard, J2EE-compliant enterprise beans quickly.

In some cases, if you bypass the assistance offered by the EJB Builder, you might get mixed results. The EJB Builder tries to keep the changes you make in one class synchronized appropriately with the other classes, but the tool might not always be able to understand your intent and might not apply all the changes needed. Therefore, if you make direct changes to class code, the result might be an enterprise bean with errors that you must correct by hand.

A few examples follow:

- You open a bean's interface class (home, remote, local home, or local) in the Source Editor, and you add code for a new method.

To be valid, a method must have a name and the correct return type, and the method must throw the appropriate exceptions. If the new method is valid, it is automatically added to the bean class. If not, the method remains as you coded it, in the interface only.

Afterward, you might find and fix the problem in the method. However, the EJB Builder might not always be able to add the repaired method to the bean class; you might have to add it yourself. Until you make the manual addition, the method's node in the Explorer displays an error badge with a red X. (See "Understanding the IDE's Error Information" on page 199.)

Here are two examples of how changes are propagated between classes:

- If you correctly add a create method to the home interface, the EJB Builder automatically adds the corresponding `ejbCreateXxx` method to the bean class.
- If you correctly add an `ejbCreateXxx` method to the bean class, the EJB Builder automatically adds the corresponding create method to the appropriate interface or interfaces.
- You open a bean class in the Source Editor, and you add a finder method.
The compiler validates the code. Assuming that the method is valid, it is automatically added to the correct interface or interfaces.
- You open a bean class in the Source Editor, and you add a business method.
The compiler validates the code as far as is possible. However, it's possible that you might have intended your addition merely as a helper or utility method within the bean class, and therefore the method is not propagated to the remote or local interface.
- You add a business method with the correct exception to a bean's remote or local interface.
The method is automatically propagated to the bean class.
- You use the Source Editor to modify a create method in a bean's home (or local home) interface or a business method in a bean's remote (or local) interface.
The EJB Builder propagates your changes to the bean class.
- You modify the `ejbCreateXxx` method in the bean class.
The compiler validates the code as far as is possible, but does not propagate the change to the home or local home interface. (Relationships between Java interface classes are treated similarly throughout the IDE.)
- You modify a method in a bean's home interface. In the process, you make the method invalid by removing a required exception.
The compiler validates the code and provides error information, but does not propagate the change to the bean class.
- You enter a new create method in the bean class and give it a name other than `ejbCreate` or `ejbCreateXxx`. Or, you enter a new finder method in a home or local home interface and name it other than `findByXxx`. (Or, you modify such a method.)

The compiler verifies that the declaration is syntactically correct and that the return type and parameter types are valid Java classes that can be resolved.



Caution – Once you start working on a class in the Source Editor, your enterprise bean is not saved until you explicitly save it.

Understanding the IDE's Error Information

If you create code that is inconsistent with the J2EE specification, a warning badge or an error badge appears on a node's icon in the Explorer. To see a description of the problem, select the offending node, right-click, and choose Error Information or Validate EJB.



This yellow triangle warning badge on the logical node means that the bean or one of its classes might have a validation problem. Expand the logical node to see where the problem lies. For instance, a method defined in the remote interface might not be in the bean class, or a class might extend the wrong Java superclass. Even if you can compile the bean, it will encounter problems.



This red X error badge on the logical node means that the bean or one of its classes might have a severe problem. For instance, an entire class could be missing. A bean bearing this error badge will not execute or even allow interaction.

Compiling and Validating Enterprise Beans

The EJB Builder contains a custom compiler that validates your enterprise beans against the *Enterprise JavaBeans Specification*. You can decide whether to have compilation and validation done separately. However, when you select an enterprise bean node, right-click it, and choose Validate EJB, the default action is to compile the bean and then immediately validate it.

Notice that compilation alone does not catch all errors, and that if a bean has compilation errors, it is not validated.

Validation and compilation in the IDE serve different purposes. When you only compile an enterprise bean, the IDE compiles the various classes that constitute the bean. If these individual classes have Java code that is syntactically correct, the bean can compile without errors, even if one class is not consistent with another or with the J2EE specifications. To ensure consistency between the elements of an enterprise bean, you must also validate the bean.

The EJB Compiler conforms to the IDE's definitions of `build`, `compile`, and `clean`.

- If the bean is out of date (that is, the bean's deployment descriptor has been modified, or a directly referenced Java class has been modified since the last successful compilation), a compilation both compiles and validates the Java class according to the *Enterprise JavaBeans Specification*.
- A `clean` removes the directly referenced Java class files and the time stamp of the last EJB compilation.
- A `build` first performs a `clean` and then compiles the referenced classes.

In most cases, you can save considerable time by using the EJB Compiler with its default options during your iterative development cycle. However, under certain conditions (for example, if you must change a superclass after a successful compilation), compilation does not catch Java or EJB validation errors. In such cases, you might need to perform a `build` (possibly traversing several directories, depending on the location of all your bean's Java files) to detect or resolve compilation errors.

The compiler also does some semantic checking of the EJB QL in the select and finder methods of a CMP entity bean. Again, you can choose whether or not to have the compiler require EJB QL statements. Sometimes, you might find it convenient to turn off that requirement, so that you can develop, compile, and validate your enterprise bean without receiving error messages about missing EJB QL statements. However, you must add the EJB QL (and turn the compiler's EJB QL requirement option back on) before you deploy the bean to the IDE's embedded application server, the J2EE Reference Implementation (the RI). This server and some others require EJB QL, which the server plugin converts to the SQL it needs in order to run queries in your finder and select methods.

To set compilation and validation options for your enterprise bean, do as follows:

1. **From the main menu, choose Tools → Options.**
2. **Expand the Options node, the Building node, and the Compiler Types node.**
3. **Select EJB 2.0 Compiler.**

4. In the property sheet, make your selections.

The default setting for both fields is `True`. Change the setting in the following cases:

- Change the `Require Compilation` field to `False` if you want to be able to validate your bean in a separate step before you compile it.
- Change the `Require EJB QL` field to `False` if you want your bean to pass validation without EJB QL statements in its finder and select methods.

If you have chosen to separate validation from compilation, validate your bean as follows:

- **In the Explorer's Filesystems pane, select the bean's logical node, right-click, and choose `Validate EJB`.**

Depending on the size of your bean, validation might take a few moments. When it's finished, an output window opens and displays messages about your bean.

When you compile an EJB module or do anything to a module that involves a compile (such as exporting an EJB JAR file, or calculating the class files for export), the module and all its component beans are validated automatically.

Saving Your Changes

In many cases, your work is saved automatically. However, this is not always true. For example, compiling a bean does not always save it. When you exit from the IDE, notice the confirmation dialog box and save the work you want to. For best results, use `File → Save All` periodically while you work in the IDE.

Renaming an Enterprise Bean

When you rename a bean, you don't have to manually rename all the bean's related objects and their internal references. Use the IDE's GUI support as explained next, and the IDE synchronizes all interfaces automatically (both their external object names and related references within the source code). Do as follows:

1. **Select the bean's logical node, right-click, and choose `Rename`.**

The `Rename` dialog box appears. As soon as you begin typing into the `New Name` field, the checkbox options are activated.

2. **Use the checkboxes to rename all the bean's related objects at once.**

However, if you obtained one of your bean's objects from elsewhere, carefully consider whether or not you want to rename it. For example, if two or more entity beans share the same home and remote interfaces, you might want to keep similar interfaces named the same wherever they occur.



Caution – If you rename any one of the related objects independently, connections between the objects can be lost.

Modifying a Bean Based on Another Bean

An enterprise bean's class can be based on a class from another bean. For example, you can create an enterprise bean that uses another bean's remote interface. When you modify a class that is based on another class, you are actually modifying the original class. The more recent bean points to the earlier bean's class; you are not working with a separate copy of the class file. This is part of the IDE's design to promote easy reuse of enterprise bean elements.

Copying and Pasting an Enterprise Bean

When you copy and paste a bean into another package, the IDE creates a node in the new package that points to classes and interfaces in the original package. The IDE does not make the assumption that you want everything in one package; some developers need to reuse bean elements more flexibly. Therefore, if you want an exact copy of one bean in another package, you must also ensure that the paths of all the bean's classes and interfaces are changed to the new package.

To copy and paste an enterprise bean from one package to another, do as follows:

1. **In the Explorer's Filesystems pane, right-click the logical node of the bean you want to copy and choose Copy.**
2. **Right-click the package to which you want to copy the bean and choose Paste → Copy.**

3. Expand the **Classes** node under the original bean's logical node, right-click the node for a class or interface, and choose **Copy**. Then paste the class or interface into the destination package by right-clicking the destination package and choosing **Paste** → **Copy**.

Repeat this step for each of the bean's classes and interfaces.

4. Right-click the logical node of the copied bean and choose **Properties**.
5. In the property sheet, update each class to point to the destination package. Find the following fields:
 - bean class
 - home interface (if any)
 - remote interface (if any)
 - local home interface (if any)
 - local interface (if any)
 - primary-key class (if any)

a. Click each field, then click the ellipsis (...) button.

b. In the property editor, navigate to the destination package, select the copy of the class or interface you pasted into the package, and click **OK**.

The property value changes to *PackageName.classname*.

When you're done, each class and interface property has the package name appended to the original name. The destination package contains a complete copy of the bean.

Exchanging Bean Classes or Interfaces

After you create an enterprise bean, you might need to change it to use an element from another bean, such as a home or remote interface. To do this, use the bean's property sheet as follows:

1. In the Explorer window, select the bean that needs a different class or interface, right-click, and choose **Properties**.
2. In the Properties tabbed pane, click the appropriate property (**Bean Class**, **Home Interface**, **Primary Key Class**, or **Remote Interface**), and click the ellipsis (...) button.
3. Navigate to the class or interface you want to use, select it, and click **OK**.

The property field shows the fully qualified path name of the new class or interface. Your bean does not contain a copy of the new element. It merely points to the original element.

Editing a Bean's Methods

When you have added a method using the GUI support available from the Explorer window, you can edit the method in the Source Editor. If all you need to do is complete the body of the method in the bean class, and your edits don't affect any other class or interface, you should use the Source Editor. However, for changes that might have a ripple effect on other classes, you might need to synchronize the changes in the bean's related objects. For examples, see "Using the Source Editor to Edit Beans" on page 197.

Or, you can simply use the Customizer dialog box as follows:

1. **In the Explorer window, expand the logical node of the bean whose method you want to edit. Navigate to the method.**

2. **Select the method, right-click, and choose Customize.**

The Customizer dialog box appears with the same fields as the New Method dialog box.

3. **Edit the fields as needed. Click Close when you're done.**

The IDE propagates your changes throughout the bean.

Viewing a Method

To see any method you have created for an enterprise bean, expand the bean's logical node and navigate to the sub-node for the kind of method you want to view. Right-click the method's node and choose Open. The Source Editor opens the class directly to the method code.

Changing an Entity Bean's Fields

Depending on whether your entity bean uses container-managed or bean-managed persistence, you use different methods to rename a field and to change a field's type.

Renaming a Field

In a CMP bean, use the GUI support available from the Explorer window. Do as follows:

1. **Expand the bean's logical node and select the CMP field. Right-click, and choose Rename.**
2. **Use the checkboxes to specify the extent of your change.**

In a BMP bean, use the Source Editor to rename a persistent or a nonpersistent field.

Changing the Type of a Field

In a CMP bean, use the Explorer's GUI support. Change a field's type as follows:

1. **Expand the bean's logical node and select the CMP field. Right-click, and choose Customize.**
2. **In the Customizer dialog box, select another type.**

In a BMP bean, use the Source Editor to change the type of a persistent or a nonpersistent field.

Deleting an Enterprise Bean

Regardless of the type of enterprise bean, delete a bean *only* as follows:

1. **Select the bean's logical node, right-click, and choose Delete.**
The Confirm EJB Deletion dialog box appears.
2. **Use the checkboxes to confirm that you want to delete all the bean's related objects at once.**

If one of your bean's related objects is used elsewhere, carefully consider whether or not you want to delete it. For example, if several entity beans use the same primary-key class, you should deselect its checkbox before deleting the rest of the bean's classes.



Caution – Don't choose Edit → Delete from the menu bar when deleting beans; the IDE simply deletes the selected class without trying to synchronize your bean's constituent classes.

Migrating and Upgrading EJB 1.1 Enterprise Beans

If you used the previous version of the Sun ONE Studio IDE to build enterprise beans, that is, if your enterprise beans were built according to the *Enterprise JavaBeans Specification* version 1.1 (EJB 1.1), you can migrate your beans to the current version of the IDE and to *Enterprise JavaBeans Specification* version 2.0 (EJB 2.0). Depending on the type of EJB 1.1 enterprise bean and what you want to do with it, the bean might be converted automatically by the IDE. Or, you might have to make some manual changes, which are described in this appendix.

Understanding Updates in the Current Release

Some of the differences between the current and previous IDE release involve container-managed persistent (CMP) entity beans, their properties, and the validation of enterprise beans in general. Notice the following points.

- **Most bean conversions are automatic.** In most cases, when you import an EJB 1.1 enterprise bean into the current version of the IDE, the IDE automatically converts it to an EJB 2.0 enterprise bean. However, CMP entity beans created in the EJB 1.1 environment are a special case.

You can easily identify these beans by their version tag, a feature that was added in the current IDE. In the property sheets for a CMP entity bean, in the CMP Version field, you see that a bean created in the current IDE has the value 2.x, while the older CMP entity bean has the value 1.x. (We'll refer to the older type of bean as a CMP 1.x entity bean.)

- **EJB validation now catches more.** In the current release of the IDE, validation has been tightened up. You might find that this upgraded feature points out failures in Java code that seemed to succeed in the previous release. An example is the `transient` modifier, described next.

Use EJB Validate (from the contextual menu on the enterprise bean in the IDE's Explorer) or Build All (from the contextual menu on the package) to see whether an application containing that bean can be deployed in the current version of the IDE. If validation uncovers errors in your code, see the following sections.

Once the bean is validated successfully, assemble it into a module and an application, and try to deploy it. If deployment fails, check to see whether changes in the bean's property sheets could be causing the problem.

- **Get rid of `transient` in session beans.** The previous version of the IDE seemed to allow the use of the `transient` modifier, but validation in this version catches it and issues an error message. See "Avoiding the `transient` Modifier" on page 211 for instructions on finding and changing the modifier.
- **Old CMP entity beans are mainly OK to use in development, but update them before deployment.** If you have changed the bean's `transient` modifier, you shouldn't experience any problems using CMP 1.x entity beans in your development work with the current IDE. However, when you're ready to deploy a CMP 1.x entity bean in this IDE, you must make a few changes by hand, as follows:
 - **Shift certain bean properties to the EJB module for RI deployment.** A significant change in the current release is the way data source references are declared for CMP entity beans that are deployed to the J2EE reference-implementation application server (RI). The RI-related properties for such a bean must now be set in the EJB module. For details, see "Shifting a Bean's RI Properties to the EJB Module Level" on page 212.
 - **Edit generated SQL for RI deployment.** If you're deploying a CMP 1.x entity bean to the RI, you must edit one of the SQL statements that the RI plugin generates. Specifically, you must edit the `WHERE` clause in the `SELECT` statement of the finder method. See "If Your EJB Module Contains an EJB 1.1 CMP Entity Bean" on page 168 for details.

When you have made these changes, the CMP entity bean still has the `CMP Version` value 1.x; it hasn't been converted to a CMP 2.x entity bean. However, the bean and its existing interfaces should operate and deploy without problems in the current IDE, assuming you're deploying the beans to an application server that complies with EJB 2.0.

- **Convert old CMP entity beans before adding new features.** You can't add new EJB 2.0 features such as local interfaces to a CMP 1.x entity bean. If you need new features in an old bean, you must first convert the bean manually to the CMP 2.x level. These instructions are given in "Converting a CMP 1.x Entity Bean" on page 209.

Making Specific Changes

In the sections that follow are instructions for updating enterprise beans that were created using the previous version of the IDE.

Converting a CMP 1.x Entity Bean

Sometimes you can't recreate a CMP 1.x entity bean from scratch, but you want the bean to be able to use new EJB 2.0 features such as local interfaces, local references, select and home methods. In this case, you will probably need to manually upgrade the bean. Convert it to a CMP 2.x bean as follows:

1. **In the Explorer window, create a new Java package.**
2. **Copy the Java files from the CMP 1.x entity bean's old package. Paste them into the new package (as copies, not as links).**
3. **Use the EJB Builder Wizard as discussed in Chapter 4 to create a new CMP entity bean.**

On the last pane of the wizard, specify the copied classes as the bean's remote and home interfaces. Also specify the bean class and primary-key class.

For the moment, ignore the IDE's warning that no CMP (persistent) fields have been found.

4. **Add fields as needed and make other edits to correct EJB Validation errors.**

When you import any other kind of EJB 1.1 enterprise bean into the current version of the IDE, the bean is automatically updated to conform to EJB 2.0.

Avoiding the Use of New Features in an Old Bean

If you try to use new features in an enterprise bean that was created using the previous version of the IDE, the results cannot be predicted. Two examples follow.

Don't Add Local Interfaces to a CMP 1.x Entity Bean

When you use the product as intended, EJB 1.1 beans have a different contextual menu (the menu that appears in the Explorer window when you right-click the bean's logical node), and the menu options are limited. However, if you choose Customize from the contextual menu, you see a window that seems to allow a local home and a local interface file.

These fields are not editable directly, and the file paths of the local interface do not appear on the property sheet of an EJB 1.1 CMP entity bean.



Caution – Do not add local interfaces in this way. If you do, you will not be able to remove the interfaces again. The CMP entity bean will become invalid and cannot be repaired.

Don't Add Local EJB References, Either

The same caveat applies to local EJB references. They cannot be added to a CMP 1.x entity bean, even though the Customizer window might appear to allow it. If you have done so, you can go to the Customizer dialog box and delete the references you added.

The EJB 2.0 standard is quite large, and its new coding practices are strongly encouraged or enforced by the current version of the IDE. If possible, you should recreate your old enterprise beans using the current IDE's EJB Builder Wizard. Carefully evaluate your code and upgrade it where necessary to conform to the current EJB standard.

Note – You can add local interfaces and local EJB references, if needed, to another type of EJB 1.1 enterprise bean, because the IDE converts those beans automatically to conform to the current EJB standard. Either the Customizer dialog box or the property sheet can be used to modify these enterprise beans.

Changing the PointBase User Name and Password

The default database user ID and password for the PointBase database have changed from `public` to `pbpublic`. All property sheets must be modified to reflect the change. Remember to press the Enter or Return key after typing the password.

Avoiding the `transient` Modifier

The previous version of the IDE seemed to allow statements of this form in session beans:

```
public class HelloBean implements javax.ejb.SessionBean {  
    private transient SessionContext context;  
}
```

In the previous version, validation did not catch the use of the `transient` modifier. In the current version, validation catches and disallows that modifier.

A simple EJB Validate command on an EJB 1.1 session bean might not catch the use of this modifier. To be sure your bean is validated completely, do the following:

1. In the Explorer window, select the bean.
2. In the main menu, choose **Build** → **Build**.

The IDE builds all the bean's Java classes and performs a validation check. The Compiler output window shows the result of the build, including any error messages.

If the `transient` modifier is present in your session bean, you see an error when building the bean or when a client executes against the bean. To correct the problem, remove the word `transient` from the EJB Bean class, and then validate or build the bean again. If validation or building is successful, deploy the bean and run the client again.

Version 2.0 of the *Enterprise JavaBeans Specification* says:

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

Shifting a Bean's RI Properties to the EJB Module Level

For any CMP entity bean that is deployed to the RI, you must move its RI-related properties from the bean level to the level of the EJB module.

The property sheets for enterprise beans have changed slightly. When one enterprise bean calls another, a reference is set on the calling bean. Two changes must be made so that this type of application can execute. If these changes are not made, you might see a `Cannot save deployment error`. This type of error is probably due to database-related information being in the wrong property sheets of a CMP entity bean. To correct the problem, do as follows:

1. **In the Explorer window, right-click the bean's EJB module node and choose Properties.**
2. **Go to the J2EE RI tabbed interface of the Properties window.**

Three properties that were on the bean's J2EE RI property sheet in the previous version of the IDE now appear here. They represent the database connection that the CMP entity bean will use.

3. **Specify each property for the database.**

If you're using the PointBase database, type the following values:

- In the Datasource JNDI Name field: `jdbc/Pointbase` (the JNDI name that exists in the application server. Type `Pointbase` with only an initial capital)
- In the Datasource Password field: `pbpublic` (the new password default. Press the Enter or Return key after typing the password)
- In the Datasource User Name field: `pbpublic` (the new user-name default)

Changing CMP Entity Bean Properties Before Testing the Bean

In the previous release of the IDE, if a CMP enterprise bean had the correct information for database connection, then the test application that was generated for the bean could be deployed to the RI with no further changes.

In the current release, you don't have to make any changes in the EJB test application for session beans or bean-managed persistent (BMP) entity beans.

However, for CMP entity beans that you intend to deploy to the RI and test against PointBase, a change is required. As mentioned earlier, the properties for the Data Source JNDI Name, user ID, and password have been moved from the enterprise

bean level to the EJB module level, the PointBase default user ID and password have changed, and the JNDI name on an EJB reference's J2EE RI tabbed interface has only one property. A CMP bean's property sheet is not updated automatically.

To change these properties, do as follows:

- 1. In the Explorer window, right-click the node for the EJB module that the testing feature generated. Choose Properties.**
- 2. In the property sheet, click the J2EE RI tab.**
- 3. In the J2EE RI tabbed interface, update the three data source fields as explained in "Shifting a Bean's RI Properties to the EJB Module Level" on page 212.**

If you try to deploy your application before setting those properties, you get an error message, with the details of the error in the Deploying Application tab of the Output Window.

Index

A

- abstract accessor methods, 36
- abstract schema name, 89, 111, 121, 152
- accessor methods, 36
- acknowledgement mode for message-driven beans, 146
- activating
 - entity bean instances, 32
 - message-driven bean instances, 39
 - stateful session bean instances, 28
- afterBegin method
 - in a stateful CMT session bean, 29, 59, 65
- afterCompletion method
 - in a stateful CMT session bean, 29, 59, 66
- anonymous instance
 - entity bean, 33
 - message-driven bean, 39
 - stateless session bean, 28
- application assembly, *refer to Building J2EE Applications*
- application examples, *refer to Sun ONE Studio 4 tutorials and examples at*
<http://forte.sun.com/ffj/documentation/>
- application server
 - EJB container services, 30
 - requirements, 55
 - RI provided with IDE, 44
 - RI requirements, 159 to 161, 163 to 170
 - tabbed pane on property sheets, 150
- application-level problems, *See exceptions*
- applications, configuring, 41

- assembling beans into an EJB module, 162 to 163
- attributes, *See transaction attributes*
- avoiding message-driven bean problems, 146

B

- back-end tier, *See data store*
- base classes, *See bean classes and interfaces*
- bean
 - class, 12
 - entity bean, 84, 115
 - message-driven bean, 138
 - session bean, 55
 - classes and interfaces, 8, 55 to 56, 84, 126
 - methods, introduction to, 8
 - properties, 42, 150 to 161
 - session bean types, 49
 - types, entity bean, 71, 105
- bean home name, *See abstract schema name*
- bean-managed persistence (BMP), 30
 - comparing with CMP, 71
 - completing the generated code, 128 to 132
- bean-managed transactions (BMT), 25, 51, 53, 135
- beforeCompletion method
 - in a stateful CMT session bean, 29, 59, 65
- business methods, 9, 196
 - in entity beans, 34
 - in a BMP entity bean, 131
 - in a CMP entity bean, 97
 - in a message-driven bean, 39
 - in a session bean, 27, 63
 - compared to home methods, 99

C

- cardinality of a CMR, 36
- cascade-delete functionality in a CMR, 36
- changing
 - a primary-key class, 95
 - an entity bean's fields, 204
 - beans, general rules, 195 to 199
 - field types, 205
 - to another bean class or interface, 203
- checking security, 42
- class files
 - of a session bean, 54
 - of an entity bean, 78
- clean-up after server crash, 146
- clients
 - relationships with enterprise beans, 22, 29
 - supported by the IDE, 2
- CMP fields
 - adding, 101
 - and CMRs, 111
 - from database table columns, 76, 111
 - in a set of related CMP entity beans, 111
 - in a single CMP entity bean, 76
 - initializing values, 33
 - specifying individually, 82 to 83
- CMRs (container-managed relationships)
 - introduction to, 36
 - adding, 119 to 121
 - editing, 112 to 114
 - in a set of related CMP entity beans, 112
 - managed by the EJB module, 117
- code, finishing
 - entity beans, 31, 92 to 101, 117 to 121
 - message-driven beans, 139 to 141
 - session beans, 26, 59 to 67
- coding security into enterprise beans, 43
- commit method, 64
- compiler options
 - include validation or not, 200
 - require EJB QL or not, 201
- compiling compared to validating, 199
- configuring an EJB application, 41
- connection factories
 - for enterprise beans in general, 156
 - for message-driven beans, 142
- connections to resources

- for enterprise beans in general, 156
 - for message-driven beans, 142
- consistency through validation, 199
- container, *See* EJB container
- container-managed persistence (CMP), 30, 71
- container-managed transactions (CMT), 25, 51, 53, 135
- contracts within J2EE architecture, 6
- conversational session, 22 to 29
- copying and pasting a bean, 202 to 203
- create methods, 9, 196
 - in entity beans, 93 to 95, 130
 - in message-driven beans, 139
 - in session beans, 57, 60 to 61
 - to insert data into a data store, 33
- creating
 - a new entity bean instance, 33
 - a new message-driven bean instance, 39
 - a new session bean instance, 27
 - an EJB module around enterprise beans, 161
 - testing objects, 180
- customized exceptions, 42
- customizer
 - adding interfaces to a bean, 190 to 193
 - modifying methods, parameters, and exceptions, 197

D

- data access object (DAO), 128
- data store in the J2EE application model, 4
- data synchronization, 35
- database connections, 77 to 78
 - specifying in property sheets, 156 to 158
 - when generating a CMP entity bean, 73 to 74, 106
- database mapping, 18
 - with CMP fields, 76 to 78, 108 to 114
- database schema
 - capturing, 79
 - using to generate CMP fields, 79 to 80, ?? to 81, ?? to 82, 114 to 115
- database server
 - included in the IDE, 44
 - using to generate CMP entity beans, 73

- data-storage connections, 156 to 158
- declaring
 - runtime information, 42, 141, 147 to 170
 - security, 43, 158
 - transaction attributes, 25, 170 to 172
- deleting an enterprise bean, 205
- deployment descriptor, 13, 42, 147
- descriptor, *See* deployment descriptor
- design recommendations, 45
- destination, message-driven, 141
- destroying, *See* removing
- developer roles in the J2EE model, 5
- development life cycle of an enterprise bean, 15
- difference between
 - business and home methods, 99
 - container-managed and bean-managed persistence, 30
 - container-managed and bean-managed transactions, 25
 - enterprise beans and JavaBeans, 3
 - finder and select methods, 98
 - session and entity beans, 22, 29
 - stateless and stateful session beans, 23
 - using JTA and the JDBC API, 26
- directionality of a CMR, 36
- dropped ejbRemove invocations, 146
- duplicate messages, 146

E

- editing
 - bean methods, 204
 - beans, 195 to 199
 - CMRs, 112 to 114
 - EJB QL statements, 98
 - SQL statements in CMP beans' property sheets, 163 to 170
- EJB 2.0 specification supported, 1
- EJB Builder Wizard, 16, 48 to 67
 - defining a BMP entity bean, 123 to 132
 - defining a CMP entity bean, 69 to 101
 - defining a message-driven bean bean, 134 to 146
 - defining a session bean, 52 to 55
 - defining a set of related CMP entity beans, 104 to 115
 - generating CMP entity bean classes, 107 to 115

- generating exceptions, 42
- generating method signatures, 27, 32, 39
- generating session bean classes, 55 to 59
- propagating changes through bean classes, 195 to 199
- EJB container
 - managing persistence, 30
 - managing transactions, 25
 - pooling entity bean instances, 31
 - pooling message-driven bean instances, 40
 - pooling stateless session bean instances, 28
 - role within a J2EE application, 4
 - services provided to entity beans, 30
- EJB group, *See* set of related CMP entity beans
- EJB JAR file, 5, 161 to 174
- EJB module, 5
 - creating, 161 to 174
 - for testing purposes, 177
 - properties, 42
 - transaction attributes, 170 to 172
- EJB QL, 196
 - editing, 98
 - errors, 201
 - foreign keys, 36
 - in finder methods, 9, 98
 - in select methods, 9, 100
 - required or not by compiler, 201
 - table-to-table joins, 36
- EJB references, 155
- ejbActivate method, 58, 91, 127
 - completing in a stateful session bean, 61 to 62
 - on entity bean instances, 32
 - on stateless session bean instances, 29
- ejbCreate method, 130, 196
 - in a BMP entity bean, 130 to 131
 - in a CMP entity bean, 93 to 95
 - in a message-driven bean, 39, 139
 - in a session bean, 27, 60 to 61
 - in entity bean instances, 33
 - pooling stateless session bean instances, 28
- ejbFind method, 196
- ejbLoad method, 92, 128
 - on BMP entity bean instances, 32
 - on CMP entity bean instances, 92
 - to synchronize with the data store, 35
- ejbPassivate method, 58, 91, 127
 - completing in a stateful session bean, 61 to 62
 - on entity bean instances, 32

- on stateless session bean instances, 29
 - ejbPostCreate method, 33, 130, 196
 - in a BMP entity bean, 130 to 131
 - in a CMP entity bean, 93 to 94
 - in a session bean, 60 to 61
 - ejbRemove method, 58, 92, 128, 139
 - pooling stateless session bean instances, 28
 - removing a database entity, 34
 - ejbStore method, 35, 92, 128
 - on BMP entity bean instances, 32
 - empty EJB group, *See* set of related CMP entity beans
 - enterprise beans
 - classes, 8
 - deleting, 205
 - design recommendations, 45
 - development life cycle, 15
 - different from JavaBeans, 3
 - elements of a bean, 8
 - methods, 8
 - persistence, 18
 - relationship to EJB container
 - security, 18, 42, 158
 - testing, 177 to 189
 - transactions, 18
 - updating, 195 to 205
 - used in applications, 41
 - workflow, 14
 - entity
 - context method, 32
 - mapping bean to database, 18
 - represented by a session bean, 23
 - represented by an entity bean, 29
 - entity beans
 - introduction to, 29
 - bean class, 84, 115
 - completing the code for a BMP, 128 to 132
 - completing the code for a CMP, 92 to 101, 117 to 121
 - generating BMP classes, 124 to 125
 - generating CMP classes, 74 to 83, 107 to 115
 - home interface, 84, 115
 - life cycle, 31
 - locating instances, 33
 - methods, 8 to 9
 - pooled state, 32
 - primary keys, 33
 - primary-key class, 84
 - ready state, 32
 - relationship to EJB container, 30
 - remote interface, 84, 115
 - type, 71, 105
 - environment
 - entry on property sheet, 155 to 156
 - information for runtime, *See* deployment descriptor
 - equals method, 90
 - error information, 199
 - evicting
 - a message-driven bean instance from memory, 40
 - a session bean instance from memory, 28
 - an entity bean instance from the pool, 32
 - example applications, where to download, xxii
 - exceptions
 - customized, 42
 - java.rmi.RemoteException, 42
 - javax.ejb.CreateException, 42
 - javax.ejb.EJBException, 41
 - predefined, 42
 - remote, 41
 - system-level and application-level, 41, 65
 - executing business logic
 - in a message-driven bean, 39
 - in a session bean, 27
 - in entity beans, 34
 - Explorer window of the IDE, 52, 74, 107, 124, 136
 - external dependencies, *See* deployment descriptor
- ## F
- features of J2EE architecture, 2
 - filter for messages, *See* message selector
 - findByPrimaryKey method, 33, 126
 - finder methods, 8, 33, 88, 98 to 99, 131, 196
 - foreign keys, 97
- ## G
- generated code
 - CMP entity bean set classes, 104
 - deployment descriptor, 147
 - entity bean classes, 70

- exceptions, 42
- message-driven bean class, 134
- method signatures in entity beans, 32
- method signatures in message-driven beans, 39
- method signatures in session beans, 29
- session bean classes, 48
- generating
 - testing objects, 180
- getCallerPrincipal method, 43
- getRollbackOnly method, 65
- getter and setter methods, 36
- getUserTransaction method, 64
- gotchas for message-driven beans, avoiding, 146

H

- hashCode method, 90
- home interface, 11, 12
 - entity bean, 84, 115
 - session bean, 55
 - See also* local home interface
- home methods, 9, 196
 - compared to business methods, 99

I

IDE

- best practices, 195 to 205
- completing a BMP entity bean, 128 to 132
- completing a CMP entity bean, 92 to 101, 117 to 121
- completing a deployment descriptor, 147 to 170
- completing a message-driven bean, 139 to 141
- completing a session bean, 59 to 67
- EJB compiler, 199
- error information, 199
- Explorer window, 52, 74, 107, 124, 136
- saving changes, 201
- Source Editor, 197
- validating beans, 199
- initializing
 - a message-driven bean instance, 39
 - a session bean instance, 27
 - an entity bean instance, 33
 - persistent fields, 33
 - state in stateful session beans, 50

- inserting data into a data store
 - using a create method, 33
- instance pool
 - entity beans, 31
 - message-driven beans, 40
 - stateless session beans, 28
- isCallerInRole method, 43

J

J2EE

- application architecture, 2
- contracts, 6
- developer roles, 5
- documentation list, xviii
- specification, Blueprints, xviii
- J2EE reference-implementation server, *See* RI
- JAR, *See* EJB JAR file
- Java Message Service (JMS), 36
- Java Transaction API, 26
- Java Transaction Service (JTS), 26
- java.io.Serializable, 90
- java.rmi.Remote, 90
- java.rmi.RemoteException, 42
- java.security.Principal, 44
- java.sql.Connection, 64
- JavaBeans, different from enterprise beans, 3
- Javadoc, using in the IDE, xxii
- javax.ejb.CreateException, 42
- javax.ejb.EJBContext, 64
- javax.ejb.EJBException, 41
- javax.ejb.EJBHome, 55, 84
- javax.ejb.EJBObject, 55, 85
- javax.ejb.EntityBean, 84
- javax.ejb.MessageDrivenBean, 137
- javax.ejb.MessageListener, 137
- javax.ejb.SessionBean, 55
- javax.transaction.UserTransaction, 64
- JDBC API, 4, 26, 30 to 31
 - don't mix with JTA code, 64
- JNDI, 4, 154
- JSP pages as clients, 2
- JTA, 26, 64

L

- large icon, 151
- life cycle
 - methods, 10
 - in a BMP entity bean, 127 to 128
 - in a CMP entity bean, 91 to 92
 - in a session bean, 61 to 62
 - of a message-driven bean, 39
 - of a session bean, 26
 - of an entity bean, 31
- local home interface
 - introduction to, 12
 - See also* home interface
- local interface
 - introduction to, 11
 - See also* remote interface
- locating entity bean instances, 33
- logical node, 55, 84, 116, 137, 195 to 196

M

- maintaining enterprise beans, 195 to 205
- maintaining state across method calls, 24
- making changes to beans, 195 to 199
- message order, 146
- message selector, 141
- message-driven beans
 - bean class, 138
 - completing the code, 139 to 141
 - developing, 133 to 146
 - methods, 8 to 9
 - onMessage method, 140
 - setMessageDrivenContext method, 141
 - transaction management, 135
- message-driven destination, 141
- message-oriented middleware, 36
- methods on enterprise beans, 8
 - afterBegin, 29
 - afterCompletion, 29
 - beforeCompletion, 29
 - business, 9
 - create, 9, 33
 - editing, 204
 - ejbActivate, 29
 - ejbCreate, 27, 33, 39
 - ejbLoad, 32, 35

- ejbPassivate, 29, 32
- ejbPostCreate, 33
- ejbRemove, 28, 34
- ejbStore, 32, 35
- equals, 90
- findByPrimaryKey, 33
- finder, 8, 33
- getCallerPrincipal, 43
- hashCode, 90
- home, 9
- isCallerInRole, 43
- life-cycle, 10
- newInstance, 27, 32, 39
- onMessage, 10
- permission to execute, 43
- security, 18
- select, 9
- setEntityContext, 32
- setMessageDrivenContext, 39
- setSessionContext, 27
- unsetEntityContext, 32
- modifying a bean based on another bean, 202
- modifying bean methods, 197 to 199
- modifying beans in general, 195 to 205
- module, *See* EJB module
- multithreading
 - approximating with message-driven beans, 36
 - not needed in enterprise beans, 29

N

- nested transactions, 26
- newInstance method
 - in entity beans, 32
 - in message-driven beans, 39
 - in session beans, 27
- nodes
 - entity bean, 84, 116, 137
 - logical, 55, 84, 116, 137
 - message-driven bean, 137
 - session bean, 55

O

- onMessage method, 10, 140
- optimizing enterprise beans, 45

order of messages, 146
out-of-sequence messages, 146

P

package (folder) node in Explorer, 52, 73, 106, 124, 136
parallel processing, approximating with message-driven beans, 36
passivating
 entity bean instances, 32
 stateful session bean instances, 28
performance in enterprise beans, 45
permission to execute a method, 43
persistence, 18
 completing BMP entity beans, 129
 managed by the EJB container, 30
 setting properties, 159 to 170
 wizard selections for CMP entity beans, 74 to 75, 107 to 108
persistent fields, 88
 specifying individually, 82 to 83
poison messages, 146
pooling
 entity bean instances, 31
 message-driven bean instances, 40
 session bean instances, 24
 stateless session bean instances, 28
predefined exceptions, 42
primary keys, 33
 adding a new one to an entity bean, 95 to 96
 adding more to an entity bean, 95
primary-key class, 129
 in an entity bean, 84
 required methods, 90
private fields, 128
problem handling with exceptions, 41
problems
 error information, 199
 system-level or application-level, *See* exceptions
 working outside the logical node, 195 to 196
programmatic security, 43
project pane, *See* Explorer window
propagating changes, 195 to 199
properties of a bean, 150 to 161

property sheets, 42, 150 to 161
prototyping on the RI, 44
publish model for message-driven beans

Q

queue, 141
quick reference
 to entity bean types, 71
 to session bean types, 50 to 51

R

ready state, entity bean instances, 32
recommendations for enterprise bean design, 45
related objects
 of an entity bean, 78
 renaming all at once, 201
remote exceptions, 41
remote interface, 10, 11
 entity bean, 84, 115
 session bean, 55
 See also local interface
remote object, 33
remotely referenced enterprise beans, 155
removing
 a database entity, 34
 a message-driven bean instance, 40
 a session bean instance, 28
 an entity bean instance, 32
renaming
 an enterprise bean, 201
 bean fields, 205
repeated messages, 146
resource environment references
 to queues or topics, 142 to 145
resource references to connection factories, 142
resource-factory references, 156 to 158
resources
 in a stateful session bean, 28
responsibilities of the bean provider
 when coding entity beans, 31
 when coding message-driven beans, 39
 when coding session beans, 26

- reuse
 - through declarative runtime information, 42, 147
 - through wizard selections, 55
- RI, 44
 - declaring properties, 159 to 170
 - properties for CMP entity beans, 163 to 170
 - finder methods, 168 to 169
 - CMP field order, 170
 - properties for session beans and BMP entity beans, 159
- roles
 - security, 42, 158
- runtime information, 42, 147

S

- sample applications, where to download, xxii
- saving changes, 201
- schema, *See* database schema or abstract schema
- security, 18, 42
 - getCallerPrincipal method, 43
 - isCallerInRole method, 43
 - roles in deployment descriptor, 158
- security checking
 - in a message-driven bean, 39, 158
 - in a session bean, 27
 - in an entity bean, 33
- select methods, 9, 196
- sequence of messages, 146
- server crash, entity bean state survives, 29
- server, *See* application server or database server
- services
 - provided by EJB container
- services provided by EJB container, 4, 30
- servlets as clients, 2
- session beans
 - introduction to, 22
 - bean class, 55
 - completing the code, 59 to 67
 - home interface, 55
 - life cycle, 26
 - methods, 8 to 9
 - pooling, 24
 - remote interface, 55
 - representing entities, 23

- stateful, 24
- stateless, 23
- synchronizing state during a session, 29
- type, 49
- session-synchronization interface, 29, 65 to 67
 - classes, 58 to 59
- set of related CMP entity beans, 17, 103
- setAutoCommit method, 64
- setEntityContext method, 32, 91, 127
- setMessageDrivenContext method, 39, 139, 141
- setRollBackOnly method, 65
- setSessionContext method, 27, 58
- small icon, 151
- Source Editor, 197 to 199
- specification supported
 - EJB 2.0, 1
- specifying security, 43
- SQL, 4, 30
 - generated from EJB QL statements, 196
 - setting properties in the RI tab, 168 to 169
- SQL Insert statements, 130
- state, maintaining across method calls, 24
- stateful session beans, 24, 49
 - passivating and activating, 28
 - selecting in the wizard, 53
- stateless session beans, 23, 49
 - selecting in the wizard, 53
- subscribe model for message-driven beans
- superclass, 54, 199
- supported EJB specification, 1
- synchronizing
 - an entity bean instance with the data store, 35
 - state during a session, 29, 58 to 59
- system exception, 65

T

- table mappings, 76 to 80, ?? to 81, ?? to 82, 108 to 114
- techniques for working with enterprise beans, 195 to 205
- testing on the RI, 44, 177 to 189
- threading, approximating with message-driven beans, 36
- tiers in J2EE architecture, 2

topic, 141

transaction control

in a message-driven bean, 39

in a session bean, 27

in an entity bean, 33

transactions, 18

attributes, 25

on an EJB module, 170 to 172

on individual beans, 171

on individual methods, 171

bean-managed, 25, 51, 135

boundaries, 64

container-managed, 25, 51, 135

in entity beans, 30

in message-driven beans, 135

in session beans, 24, 51, 63 to 67

nested, not allowed in JTA, 26, 64

rollbacks, 64

using JTA, 26

using the JDBC API, 26

type of

entity bean, 71, 105

session bean, 49

X

XML deployment-descriptor file, 42, 147

U

unique identifiers in entity beans, 29

unsetEntityContext method, 32, 91, 128

updating enterprise beans, 195 to 205

user security roles, 42

user transaction (UT) methods, 64

V

validating beans, 199 to 201

verifying, *See* validating

W

wizard, *See* EJB Builder Wizard

workflow of an EJB application, 14

wrapping legacy code using the JDBC API, 26, 31

