SYBASE®

CORBA Components Guide

EAServer

6.0

DOCUMENT ID: DC00547-01-0600-01

LAST REVISED: July 2006

Sybase, Inc., One Sybase Drive, Dublin, CA 94568.

Contents

About This Book

Audience

This book is for application developers who develop C++ or PowerBuilder® clients or components for deployment to EAServer, and developers who must maintain legacy EAServer CORBA/Java clients or components. Developers should be familiar with their chosen programming languages, specifically Java, C++, or PowerScript®.

How to use this book

Chapter 1, "CORBA Component Overview," describes CORBA component concepts and the EAServer component models based on the CORBA model.

Chapter 2, "CORBA Component Life Cycles and Transaction Semantics," explains the EAServer CORBA component life cycle and transaction processing models for CORBA and PowerBuilder components.

Chapter 3, "Using CORBA IDL," describes how CORBA component interfaces are defined in Interface Definition Language (IDL).

Chapter 4, "Managing CORBA Packages and Components," describes how to deploy and configure CORBA components in EAServer.

Chapter 5, "Developing and Deploying PowerBuilder Components," describes EAServer-specific modifications for PowerBuilder components developed and deployed from the PowerBuilder IDE.

Chapter 6, "Developing PowerBuilder Clients," describes how to develop PowerBuilder clients for EAServer components. describes how to develop PowerBuilder clients for EAServer components.

Chapter 7, "CORBA/C++ Overview," provides an overview of things to consider when developing CORBA C++ clients and components for EAServer.

Chapter 8, "Developing CORBA/C++ Components," describes how to implement CORBA components in C++.

Chapter 9, "Developing CORBA/C++ Clients," describes how to implement CORBA clients in C++.

Chapter 10, "Tutorial: Creating C++ Components and Clients," walks you through the creation and deployment of a CORBA/C++ component and a client that calls the component.

Chapter 11, "CORBA/Java Overview," provides an overview of things to consider when developing CORBA/Java clients and components for EAServer.

Chapter 12, "Developing CORBA/Java Components," describes how to implement CORBA components in Java.

Chapter 13, "Developing CORBA/Java Clients," describes how to implement CORBA clients in Java.

Chapter 14, "Tutorial: Creating CORBA Java Components and Clients," walks you through the creation and deployment of a CORBA/Java component and a client that calls the component.

Related documents

Core EAServer documentation The core EAServer documents are available in HTML and PDF format in your EAServer software installation and on the SyBooks™ CD.

What's New in EAServer 6.0 summarizes new functionality in this version.

The *EAServer API Reference Manual* contains reference pages for proprietary EAServer Java classes and C routines.

The *EAServer Automated Configuration Guide* explains how to use Ant-based configuration scripts to:

- Define and configure entities, such as EJB modules, Web applications, data sources, and servers
- Perform administrative and deployment tasks

The *EAServer CORBA Components Guide* (this book) explains how to:

- Create, deploy, and configure CORBA and PowerBuilder™ components and component-based applications
- Use the industry-standard CORBA and Java APIs supported by EAServer

The *EAServer Enterprise JavaBeans User's Guide* describes how to:

- Configure and deploy EJB modules
- Develop EJB clients, and create and configure EJB providers
- Create and configure applications clients
- Run the EJB tutorial

The *EAServer Feature Guide* explains application server concepts and architecture, such as supported component models, network protocols, server-managed transactions, and Web applications.

The *EAServer Java Message Service User's Guide* describes how to create Java Message Service (JMS) clients and components to send, publish, and receive JMS messages.

The *EAServer Migration Guide* contains information about migrating EAServer 5.*x* resources and entities to an EAServer 6.0 installation.

The *EAServer Performance and Tuning Guide* describes how to tune your server and application settings for best performance.

The *EAServer Security Administration and Programming Guide* explains how to:

- Understand the EAServer security architecture
- Configure role-based security for components and Web applications
- Configure SSL certificate-based security for client connections
- Implement custom security services for authentication, authorization, and role membership evaluation
- Implement secure HTTP and IIOP client applications
- Deploy client applications that connect through Internet proxies and firewalls

The *EAServer System Administration Guide* explains how to:

- Start the preconfigured server and manage it with the Sybase Management Console
- Create, configure, and start new application servers
- Define database types and data sources
- Create clusters of application servers to host load-balanced and highly available components and Web applications
- Monitor servers and application components
- Automate administration and monitoring tasks with command line tools

The *EAServer Web Application Programming Guide* explains how to create, deploy, and configure Web applications, Java servlets, and JavaServer Pages.

The *EAServer Web Services Toolkit User's Guide* describes Web services support in EAServer, including:

- Support for standard Web services protocols such as Simple Object Access Protocol (SOAP), Web Services Description Language (WSDL), and Uniform Description, Discovery, and Integration (UDDI)
- Administration tools for deployment and creation of new Web services, WSDL document creation, UDDI registration, and SOAP management

The *EAServer Troubleshooting Guide* describes procedures for troubleshooting problems that EAServer users may encounter. This document is available only online; see the EAServer Troubleshooting Guide at http://infocenter.sybase.com/help/index.jsp?topic=/com.sybase.help.eas_5.2.eastg/html/eastg/title.htm.

jConnect for JDBC documents EAServer includes the jConnect™ for JDBC™ 6.0.5 driver to allow JDBC access to Sybase database servers and gateways. The *jConnect for JDBC 6.0.5 Programmer's Reference* is available on the Sybase Product Manuals Web site at http://infocenter.sybase.com/help/index.jsp?topic=/com.sybase.help.jconnjdbc_6.05.prjdbc/html/prjdbc/title.htm&toc=/com.sybase.help.jconnjdbc_6.05/toc.xml.

Sybase Software Asset Management User's Guide EAServer includes the Sybase Software Asset Management license manager for managing and tracking your Sybase software license deployments. The *Sybase Software Asset Management User's Guide* is available on the Getting Started CD and in the EAServer 6.0 collection on the Sybase Product Manuals Web site at http://infocenter.sybase.com/help/index.jsp?topic=/com.sybase.help.eas_6.0/title.htm.

Conventions

The formatting conventions used in this manual are:

Formatting example	To indicate
commands and methods	When used in descriptive text, this font indicates keywords such as: • Command names used in descriptive text • C++ and Java method or class names used in descriptive text • Java package names used in descriptive text • Property names in the raw format, as when using Ant or jagtool to configure applications rather than the Management Console
variable, *package*, or *component*	Italic font indicates: • Program variables, such as *myCounter* • Parts of input text that must be substituted, for example: `Server.log` • File names • Names of components, EAServer packages, and other entities that are registered in the EAServer naming service

Formatting example	To indicate
File | Save	Menu names and menu items are displayed in plain text. The vertical bar shows you how to navigate menu selections. For example, File | Save indicates "select Save from the File menu."
`package 1`	Monospace font indicates: • Information that you enter in the Management Console, a command line, or as program text • Example program fragments • Example output fragments

Other sources of information

Use the Sybase Getting Started CD, the SyBooks CD, and the Sybase Product Manuals Web site to learn more about your product:

- The Getting Started CD contains release bulletins and installation guides in PDF format, and may also contain other documents or updated information not included on the SyBooks CD. It is included with your software. To read or print documents on the Getting Started CD, you need Adobe Acrobat Reader, which you can download at no charge from the Adobe Web site using a link provided on the CD.
- The SyBooks CD contains product manuals and is included with your software. The Eclipse-based SyBooks browser allows you to access the manuals in an easy-to-use, HTML-based format.

  Some documentation may be provided in PDF format, which you can access through the PDF directory on the SyBooks CD. To read or print the PDF files, you need Adobe Acrobat Reader.

  Refer to the *SyBooks Installation Guide* on the Getting Started CD, or the *README.txt* file on the SyBooks CD for instructions on installing and starting SyBooks.
- The Sybase Product Manuals Web site is an online version of the SyBooks CD that you can access using a standard Web browser. In addition to product manuals, you will find links to EBFs/Maintenance, Technical Documents, Case Management, Solved Cases, newsgroups, and the Sybase Developer Network.

  To access the Sybase Product Manuals Web site, go to Product Manuals at http://sybooks.sybase.com/nav/base.do.

Sybase certifications on the Web

Technical documentation at the Sybase Web site is updated frequently.

❖ Finding the latest information on product certifications

1. Point your Web browser to Technical Documents at http://www.sybase.com/support/techdocs/.
2. Select Products from the navigation bar on the left.
3. Select a product name from the product list and click Go.
4. Select the Certification Report filter, specify a time frame, and click Go.
5. Click a Certification Report title to display the report.

❖ Creating a personalized view of the Sybase Web site (including support pages)

Set up a MySybase profile. MySybase is a free service that allows you to create a personalized view of Sybase Web pages.

1. Point your Web browser to Technical Documents at http://www.sybase.com/support/techdocs/.
2. Click MySybase and create a MySybase profile.

Sybase EBFs and software maintenance

❖ Finding the latest information on EBFs and software maintenance

1. Point your Web browser to the Sybase Support Page at http://www.sybase.com/support.
2. Select EBFs/Maintenance. If prompted, enter your MySybase user name and password.
3. Select a product.
4. Specify a time frame and click Go. A list of EBF/Maintenance releases is displayed.

   Padlock icons indicate that you do not have download authorization for certain EBF/Maintenance releases because you are not registered as a Technical Support Contact. If you have not registered, but have valid information provided by your Sybase representative or through your support contract, click Edit Roles to add the “Technical Support Contact” role to your MySybase profile.
5. Click the Info icon to display the EBF/Maintenance report, or click the product description to download the software.

Accessibility features

EAServer has been tested for compliance with U.S. government Section 508 Accessibility requirements. The online help for this product is also provided in Eclipse help formats, which you can navigate using a screen reader.

The Web console supports working without a mouse. For more information, see "Keyboard navigation" in Chapter 2, "Management Console Overview," in the *EAServer System Administration Guide*.

The Web Services Toolkit plug-in for Eclipse supports accessibility features for those that cannot use a mouse, are visually impaired, or have other special needs. For information about these features see the Eclipse help:

1. Start Eclipse.
2. Select Help | Help Contents.
3. Enter `Accessibility` in the Search dialog box.
4. Select Accessible User Interfaces or Accessibility Features for Eclipse.

Note You may need to configure your accessibility tool for optimal use. Some screen readers pronounce text based on its case; for example, they pronounce ALL UPPERCASE TEXT as initials, and MixedCase Text as words. You might find it helpful to configure your tool to announce syntax conventions. Consult the documentation for your tool.

For additional information about how Sybase supports accessibility, see Sybase Accessibility at http://www.sybase.com/accessibility. The Sybase Accessibility site includes links to information on Section 508 and W3C standards.

If you need help

Each Sybase installation that has purchased a support contract has one or more designated people who are authorized to contact Sybase Technical Support. If you cannot resolve a problem using the manuals or online help, please have the designated person contact Sybase Technical Support or the Sybase subsidiary in your area.

CHAPTER 1 CORBA Component Overview

About CORBA

CORBA is a distributed component architecture defined by the Object Management Group (OMG). EAServer supports many CORBA technologies, including:

- The Internet Inter-ORB Protocol (IIOP) for client-server component invocations.
- CORBA Interface Definition Language (IDL), for defining component interfaces and datatypes used in interfaces.
- Business component models for C++, PowerBuilder, and Java, based on the CORBA specifications.
- Standard CORBA APIs, such as the CosNaming API for naming services.

For information on the CORBA architecture, see the specifications available at the OMG Web site at http://www.omg.org.

CORBA components in EAServer

EAServer provides CORBA component models for these languages and technologies:

- C++
- PowerBuilder
- Java

EAServer hosts CORBA components using generated EJB wrapper components. EJB and CORBA components are fully interoperable. You can call EJB components from CORBA clients and vice-versa.

Java/CORBA versus EJB components
EAServer provides the Java/CORBA component model for backward compatiblity with EAServer 5.*x* and earlier versions. Sybase recommends you create EJB components for new Java development because they are more portable to other application servers.

The CORBA component development process

The high level CORBA development and deployment process for EAServer is:

1. If you are using C++ or Java, define the component interfaces in CORBA IDL and deploy the IDL to the EAServer repository. Chapter 3, "Using CORBA IDL," describes how to do this.

   If you are using PowerBuilder, you can define interfaces with the PowerBuilder IDE. PowerBuilder generates IDL when you deploy to EAServer.

2. Create EAServer entities to define the CORBA packages and components. The package and component properties specify the component interfaces and control interaction between EAServer and your implementation. Chapter 4, "Managing CORBA Packages and Components," describes how to define and configure CORBA packages and components.

3. Develop the component implementation classes and deploy them to EAServer. For more information, see:

   - Chapter 8, "Developing CORBA/C++ Components"
   - Chapter 12, "Developing CORBA/Java Components"
   - The PowerBuilder IDE documentation and online help

4 Run the jaguar-compiler command on the CORBA packages to generate the code and EJB wrapper components required to run the components in EAServer. You can do this several ways:

- From the PowerBuilder IDE, if using PowerBuilder.
- From the Management Console as described in "Refreshing CORBA packages in the Management Console" on page 43.
- Using a configuration script, as described in "Managing CORBA packages with configuration scripts" on page 43.
- Using the jaguar-compiler command-line tool, as described in Chapter 12, "Command Line Tools," in the *System Administration Guide*.

5 Create the client code to invoke the component methods. You can call CORBA components from any other client model, including EJB clients and Web components. For details on CORBA client models, see:

- Chapter 9, "Developing CORBA/C++ Clients"
- Chapter 13, "Developing CORBA/Java Clients"

CORBA component tutorials

EAServer includes tutorials for CORBA/C++ and CORBA/Java components. See:

- Chapter 10, "Tutorial: Creating C++ Components and Clients"
- Chapter 14, "Tutorial: Creating CORBA Java Components and Clients"

CHAPTER 2

CORBA Component Life Cycles and Transaction Semantics

This chapter explains the EAServer CORBA component life cycle and transaction processing models for CORBA and PowerBuilder components.

Transactions allow you to group database updates performed by multiple components into a single atomic unit of work, which greatly simplifies error recovery in component-based applications.

The component life cycle determines how instances of a component are allocated, bound to a client, and destroyed. The EAServer component life cycle is designed to maximize reuse of resources and minimize the possibility that a client application can monopolize a server resource.

The component life cycle and the transaction model are tightly integrated. You must understand both to use transactions effectively in your application.

Component life cycles

The EAServer component life cycle is designed to:

- Maximize sharing and reuse of server resources
- Minimize the possibility that a client application can monopolize server resources

To achieve these goals, EAServer supports the concepts of component instance pooling and early deactivation.

Instance pooling allows a single component instance to service multiple clients. The component life cycle contains activation and deactivation steps: Activation binds an instance to an individual client; deactivation indicates that the instance is unbound. Instance pooling eliminates resource drain from repeated allocation of component instances.

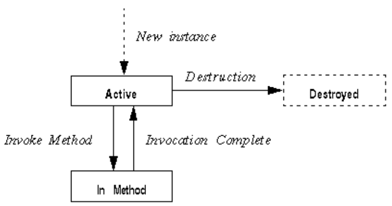
Early deactivation allows a component's methods to specify when deactivation occurs. Early deactivation prevents a client application from tying up the resources that are associated with a component instance and allows the instance to serve more clients in a given time frame. To achieve early deactivation, you can code or configure your component as described in "Supporting early deactivation in your component" on page 9.

A component that is deactivated after each method call and supports instance pooling is said to be a **stateless component** because the component's state is reset across the boundary of a transaction and activation. Early deactivation and instance pooling promotes greater scalability by enabling an increasing number of clients to use a static number of instances. An application design based on stateless components offers the greatest scalability.

States in the component life cycle

EAServer components in any component model follow the state diagram illustrated in this figure:

Figure 2-1: States in the EAServer component life cycle



The state transitions are as follows:

- **New instance** The EAServer runtime allocates a new instance of the component. The instance remains idle in the instance pool waiting for the first method invocation.

- **Activation** Activation prepares a component instance for use by a client. Once an instance is activated, it is bound to one client and can service no other client until it has been deactivated. If a component is transactional, activation also indicates the beginning of the instance's participation in a transaction.
- **In method** In response to a method invocation request from the client, the EAServer runtime calls the corresponding method in the component. The next state depends on which of the transaction state primitives the method calls before returning. (For Java components, the state transition also depends on whether the method returns with an uncaught exception.) See "Using transaction state primitives" on page 16 for more information.
- **Deactivation** Deactivation indicates that the component is no longer bound to the client. Methods can call either the completeWork or rollbackWork transaction state primitives to cause explicit deactivation of the instance. As discussed in "Using transaction state primitives" on page 16, these primitives also affect the transaction's outcome. Deactivation can also occur automatically, under any of the following circumstances:
  - If the instance is participating in a transaction, the instance is deactivated when the transaction commits, rolls back, or times out.
  - If you have configured the component's Instance Timeout property to a finite setting, an instance is deactivated if the time between consecutive method calls exceeds the timeout value. "CORBA component property descriptions" on page 45 describes how to configure this property.

  If an exception occurs in a user transaction, you must call rollbackWork after catching the exception; otherwise, a transaction deadlock may occur in the database, which can cause client applications to fail.
- **Destruction** Destruction occurs if the component instance cannot be recycled. "Supporting instance pooling in your component" on page 9 describes how to ensure instance reuse. If the component cannot be reused, deactivation is followed by destruction of the instance.

The EAServer component life cycle allows component instances to be recycled; idle component instances can be cached when idle and bound to the service of individual clients only as needed. If your component has been coded to support early deactivation, a client holding a reference to the component's stub or proxy object may be serviced by several different instances of the component. After each deactivation, the next method invocation causes an instance to be activated and bound to the client. Overall server scalability is increased because a new instance does not have to be instantiated each time a client invokes a method.

Stateful versus stateless components

A component that can remain active between consecutive method invocations is called a **stateful component**. A component that is deactivated after each method call and that supports instance pooling is said to be a **stateless component**. Typically, an application built with stateless components offers the greatest scalability.

Stateful components A stateful component remains active across method calls. EAServer wraps stateful CORBA components with an EJB stateful session bean. To run a CORBA component as stateful, the Stateful Session Bean (com.sybase.jaguar.component.tx_vote) property must be set to true–see "CORBA component property descriptions" on page 45.

Since deactivation happens at the mercy of client applications, you may wish to configure the Passivation Timeout property for stateful components so that a client cannot monopolize a component instance indefinitely. See "CORBA component property descriptions" on page 45 for more information.

Stateless components A is stateless if you disable the component's Stateful Session Bean property (com.sybase.jaguar.component.tx_vote) —see Table 4-2 on page 46. You can also set the component's com.sybase.jaguar.component.tx_vote property to false in an Ant user configuration file. Alternatively, you can implement the component so that it calls either completeWork or rollbackWork in every method.:

Stateless components cannot use instance-specific data to accumulate data between method invocations. Some situations require that you accumulate data across method invocations. For example, a PurchaseOrder component might have an addItem() method that is called repeatedly to specify the contents of an order. In lieu of instance-specific data, you can use one of these alternatives to accumulate data:

- **Accumulate data in a remote database** Use connection caching and database commands to accumulate data in a remote database. This is the preferred technique. If you deploy your component to a cluster, it may run on multiple servers and the database provides a central location available from all servers.
- **Accumulate data in the client** Create a data structure that is passed to each method invocation and contains all accumulated data. This technique is only practical if the amount of data is small. Sending large amounts of data over the network will degrade performance.
- **Accumulate data in a file** If the accumulated data is small and represented by simple data structures, you can store the data in a local file.

Supporting early deactivation in your component

Early deactivation prevents a client application from tying up the resources (such as connections) that are associated with a component instance.

To support early deactivation in CORBA and PowerBuilder components you can use one of these methods:

- Use a stateless component, which deactivates the component instance after each method invocation—see “Stateful versus stateless components” on page 8.
- In a stateful component, configure the number of seconds an active component instance can remain idle before the client’s proxy becomes invalid—see “Passivation Timeout” in Table 4-2 on page 46.
- Code your component to call one of the completeWork or rollbackWork transaction state primitives to cause explicit deactivation of the instance. This technique is useful when your design requires deactivation to occur after some, but not all, method invocations. If the component is transactional, the completeWork and rollbackWork primitives also affect the outcome of the transaction in which the component is participating. See “Using transaction state primitives” on page 16 for more information.

Supporting instance pooling in your component

Instance pooling eliminates resource drain caused by repeated allocation of new component instances.

For Java components, you can implement a life cycle interface to control whether the component instances are pooled. These interfaces also provide activate and deactivate methods that are called to indicate state transitions in a component instance’s lifetime. See “Set transactional state” on page 158.

For PowerBuilder components, you can enable the Pooling option on the PowerBuilder wizard that you use to create your component. You can then write event scripts that respond to changes in an instance’s life cycle. See the *Application Techniques* manual in the PowerBuilder documentation for more information.

For C and C++ components, you can enable instance pooling using the Management Console. See “CORBA component property descriptions” on page 45. This method also allows you to configure pooling for Java components that do not implement the ServerBean or IObjectControl interfaces, respectively.

To support instance pooling, code that responds to activation events must restore the component to its initial state (that is, as if it were newly created). the Java canReuse interfaces have methods that allow an instance to selectively refuse pooling. For PowerBuilder components, you can script the canBePooled event to selectively refuse pooling.

When the component Pooled option is set in the Management Console, the Java canReuse method is not called, even if the component implements the ServerBean interface.

Long versus short transactions

EASever supports both long and short transactions, which are initially associated with stateful and stateless components, respectively. Both long and short transactions begin when a client calls one of a component’s business methods, as long as the component’s tx_type property is set to neither “not_supported” nor “supports.” Table 4-2 on page 46 describes the allowable values for tx_type. The behavior of short transactions conforms to the J2EE specification. Support for long transactions may be deprecated in future versions of EAServer.

Long transactions

A long transaction is associated with a stateful CORBA component instance the first time a client invokes one of its business methods, subject to the value of tx_type. Clients need not perform any special transaction work. By default, long transactions are enabled for backward compatibility. To disable long transactions, change to the EAServer *bin* directory, and run:

```
configure long-transactions-off
```

If you disable long transactions, short transactions are used instead. To re-enable long transactions, run:

```
configure long-transactions-on
```

In EAServer versions earlier than 6.0, stateful CORBA components, whose tx_vote property was set to true, had to call either JagCompleteWork or JagRollbackWork to end a transaction. And a component timeout resulted in the server rolling back the active transaction..

Short transactions

A short transaction is associated with a stateless component when a client invokes one of its business methods. EAServer automatically ends the transaction upon completion of the business method. If the component calls no APIs, the transaction is committed (as if JagCompleteWork was called). Short transactions are always enabled.

EAServer's transaction processing model

An **EAServer transaction** is a transaction whose boundaries and outcome are determined by EAServer. Components can be marked as transactional in the Management Console. If a component is transactional, the EAServer transaction manager ensures that the component's third-tier database queries execute as part of a transaction. Multiple components can participate in an EAServer transaction; the EAServer transaction manager ensures that all database changes performed by the participating transactions are all committed or rolled back.

Transactions

All transactions are defined by the ACID test:

- **Atomic** If a transaction is interrupted, all changes that the transaction has made are cancelled or rolled back.

- **Consistent** A transaction produces results that preserve invariant properties.
- **Isolated** A transaction's intermediate states cannot be monitored or changed by other transactions; transactions execute their results one after another.
- **Durable** The changes that a transaction completes are permanent.

How EAServer transactions work

In the Management Console, you can declare EAServer components to be transactional. When a component is transactional and uses the EAServer connection management feature, commands sent on a third-tier database connection are automatically performed as part of a transaction. Component methods can call EAServer's transaction state primitives to influence whether EAServer commits or aborts the current transaction.

If long transactions are enabled for the server, the component life cycle is tightly integrated with EAServer's transaction model. Component instances that participate in a transaction are not deactivated until the transaction ends or until the component indicates that its contribution to the transaction is over (that is, its work is done and ready for commit or that its work must be rolled back). An instance's time in the active state corresponds to the beginning and end of its participation in a transaction.

Benefits of using EAServer transactions

The benefits of using transactions to group database updates are clear. You can easily code methods in a single component to implement transactions that run against a single data source. However, those methods may in turn be executed by another component, which itself is defining a transaction. In this situation, error recovery becomes difficult. For example, consider the following scenario in which an Enrollment component calls both Registrar and Billing components:

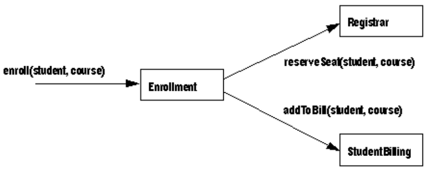
A transaction involving multiple components

In the following figure, the Enrollment.enroll() method calls methods in the Registrar and StudentBilling components:

- Registar.reserveSeat() checks that a seat is available. If so, it decrements the count of available seats and adds the student to the course's enrollment list. If no seats are available, reserveSeat() fails.

- StudentBilling.addToBill() checks that the student has a billable credit record. If so, addToBill() adds the course cost to the student's bill for that semester. If the student has a credit problem (if, for example, she owes money for an overdue book), addToBill() fails.

Figure 2-2: An example EAServer transaction



To be correct, both the database update made by the Registrar and the update made by the StudentBilling components must occur, or neither must occur. In other words, if the student cannot be billed, the course's available seats must not be changed. To handle this case, you could add logic to the enroll() method to undo changes (requiring an unreserveSeat() method in Registrar). However, as more components are added to the scenario, the logic needed to undo previous changes quickly becomes unmanageable. It is much easier to define all the participating components to use EAServer transactions. Then an error in any component can induce a rollback of all changes made by the other participating components before the error occurred.

By defining the participating components to use EAServer transactions, you can be sure that the work performed by the components that participate in a transaction occurs as intended.

Defining transactional semantics

The component and server properties and the component implementation determine how your CORBA component participates in transactions.

❖ **Defining how a component participates in transactions**

1. Specify the component's transaction attribute. Each component has a transaction attribute that determines whether instances of the component participate in transactions. "Transaction type values" on page 52 describes the attribute settings and their meanings.

2 If long transactions are enabled in the server, and your CORBA component is stateful, code methods to call the EAServer transaction state primitives. Each method should call the appropriate transaction state primitive to reflect the state of the work that the component has contributed to the transaction. "Using transaction state primitives" on page 16 describes the state primitives in detail.

   If long transactions are disabled or the CORBA component is stateless, transactions end when each business method returns. Each business method can call completeWork or rollbackWork to influence the transaction outcome. If neither is called, the completeWork behavior is the default.

Transaction coordinator

The Java transaction Service (JTS) transaction coordinator complies with the JTS and the X/Open Architecture (XA) standards. The JTS transaction coordinator integrates the functionality of the shared connection and JTS/JTA transaction modes, and uses two-phase commit to coordinate transactions among multiple databases.

Note To verify that your EAServer edition supports two-phase commit, check the server console or the *$DJC_HOME/logs/<serverName>.log* file.

Transactional component attribute

Components in EAServer have a transaction type property that indicates how a component participates in transactions. You can view and change a component's Transaction Type property using the Management Console. For PowerBuilder components, you can specify the attribute in the PowerBuilder wizards (doing so ensures that it is saved with the PowerBuilder project and not overwritten by redeployment). Allowable values are described in "Transaction type values" on page 52.

Table 2-1 lists design scenarios and the transaction type values that apply to each.

Table 2-1: Deciding on a transaction type value

Design scenario	Applicable transaction type values
Your component interacts with remote databases, and its methods may be called by another component as part of a larger transaction. Multiple updates are issued before calling completeWork, or an update depends on the results of queries that were issued since the last call to completeWork.	Requires Transaction or Requires New Transaction
Updates from your component are performed by a single database update, the update logic is independent of any other query issued by the method, and you call completeWork in each method that issues an update. In other words, your component's updates are already atomic.	Supports Transaction
Your component's methods make intercomponent method calls, and the work done by called components must be included in one transaction.	Requires Transaction or Requires New Transaction
Methods in the component interact with more than one remote database, and updates to different databases must be grouped in the same transaction (this also requires a transaction coordinator that supports two-phase commit to those databases).	Requires Transaction or Requires New Transaction
Transactions begun by your component must not be affected by the outcome of transactions begun by other components that call your component.	Requires New Transaction
Work done by your component must never be done as part of a transaction.	Not Supported

For example, in the scenario illustrated in "A transaction involving multiple components" on page 12, the Enrollment component must be marked *Requires Transaction* or *Requires New Transaction*, since it calls methods in the Registrar and StudentBilling components, and the work performed by the called components must be grouped in a single transaction. Both Registrar and StudentBilling must be marked *Supports Transaction* or *Requires Transaction* so that their database updates can be grouped in the transaction begun by the Enrollment component.

Transaction Not Supported is useful when your component performs updates to a noncritical database. For example, consider a component whose sole function is to log usage statistics to a remote database. Since usage statistics are not mission-critical data, you can choose *Not Supported* as the component's transaction type value to ensure that the logging updates do not incur the overhead of using two-phase commit.

Determining when transactions begin

After a base client instantiates a transactional component, the first method invocation begins an EAServer transaction. This instance is said to be the **root instance** of the transaction. If the root instance invokes methods in other transactional components, those components join the existing transaction.

Use a stub or proxy object for the called component For transactions to occur with the intended semantics, you must perform intercomponent calls using a stub or proxy object for the called component. Do not invoke another component's methods directly. For calls between PowerBuilder NVO components, use a PowerBuilder proxy object rather than calling the other NVO directly.

Using transaction state primitives

EAServer provides transaction state primitives that methods can call to direct the outcome of the current transaction. Each component model provides an interface containing methods for these primitives. Table 2-2 on page 17 lists the API mappings for each component type.

These methods end a component's participation in a transaction (both cause the current instance to be deactivated):

- **completeWork** The component finished its work for the current transaction and should be deactivated when the method returns. This is the default behavior for stateless CORBA components; a component that calls no state primitive behaves as if this method were called. If long transactions are disabled for the server, this is the default behavior for all CORBA components.
- **rollbackWork** The component cannot complete its work. Doom the current transaction and deactivate the instance when the method returns.

These methods are used to maintain state after the method returns (they delay deactivation of the component instance):

- **continueWork** Continue this component's participation in the current transaction after the method returns, and allow the transaction to be committed if the component is deactivated.

  In stateful CORBA components with long transactions enabled in the server, this is the default behavior if a method calls no transaction primitive.
- **disallowCommit** Continue this component's participation in the current transaction after the method returns, but roll back the transaction if the component is deactivated before calling another primitive besides disallowCommit.

These primitives can be used to query the state of the transaction (if any) in which the method is executing:

- **isInTransaction** Query whether the current method is executing in the context of a transaction.
- **isRollbackOnly** Query whether the current transaction is doomed to be rolled back or is still viable.

Table 2-2 describes how the transaction primitives are invoked in Java and PowerBuilder components. For information on the Java methods, see Chapter 1, “Java Classes and Interfaces,” in the *EAServer API Reference*. For information on the PowerBuilder TransactionServer object, see the *Application Techniques* manual in the PowerBuilder documentation and the PowerBuilder online help.

Table 2-2: Java and PowerBuilder transaction primitives

Transaction primitive	Java InstanceContext method	PowerBuilder TransactionServer function
completeWork	completeWork	SetComplete
rollbackWork	rollbackWork	SetAbort
continueWork	continueWork	EnableCommit
disallowCommit	None. You can achieve the same effect by calling, and then raising an exception if deactivate is called before the next method invocation.	DisableCommit
isInTransaction	inTransaction	IsInTransaction
isRollbackOnly	isRollbackOnly	IsTransactionAborted

C and C++ components call the methods and routines in the following table to invoke transaction primitives. See the *EAServer API Reference* for documentation of these methods and routines:

Table 2-3: C and C++ transaction primitives

Transaction primitive	C/C++ routine
completeWork	JagCompleteWork
rollbackWork	JagRollbackWork
continueWork	JagContinueWork
disallowCommit	JagDisallowCommit
isInTransaction	JagInTransaction
isRollbackOnly	JagIsRollbackOnly

Any participating component can roll back the transaction by calling the rollbackWork primitive; Java components can also cause a rollback by returning an unhandled exception. Only the action of the root component determines when EAServer commits the transaction. The transaction is committed when the root component returns with a state of completeWork and no participating component has set a state of disallowCommit.

You can use the transaction state primitives in any component; the component does not have to be declared transactional. Calling completeWork or rollbackWork from methods causes early deactivation. "Supporting early deactivation in your component" on page 9 discusses how this feature can improve application performance.

Example

As discussed in "Benefits of using EAServer transactions" on page 12, EAServer transactions are most useful when your application uses intercomponent calls.

As an example, consider the scenario illustrated in "A transaction involving multiple components" on page 12. The pseudocode below shows the logic used to ensure that the work performed by the Registrar.reserveSeat() and StudentBilling.addToBill() occurs within the same transaction.

In the Registrar component, the reserveSeat() method must check the number of seats. If there is space for the new student, then the method adds the student, decrements the count of available seats, and sets a state of completeWork. If a seat is not an available, the method calls rollbackWork to roll back the current transaction.

Here is the pseudocode for Registrar.reserveSeat():

```
check number of seats
if enough seats
```

```
    decrement number of seats
    add student to enrollment list
    completeWork
  else
    rollbackWork
  end if
```

The transaction attribute for Registrar must be *Requires Transaction* so that the query for available seats and the update of available seats always occur in the same transaction.

In the StudentBilling component, the addToBill() method must verify the student's credit. If the student does not already owe money, the method adds the cost to the semester bill and sets a state of completeWork. If the student owes money, the method calls rollbackWork to roll back the current transaction. Here is the pseudocode for StudentBilling.addToBill():

```
check student's balance
if balance > 0
  add cost to bill
  debit balance
  completeWork
else
  rollbackWork
end if
```

The transaction attribute for StudentBilling must be *Requires Transaction* so that the balance query, the billing calculation, and the debit of the student's balance always occur in the same transaction.

In the Enrollment component, the enroll() method first calls Registrar.reserveSeat(). After Registrar.reserveSeat() returns, the method checks whether the transaction is still viable using the isRollbackOnly primitive. If the transaction is viable, the method calls StudentBilling.addToBill(). Here is the pseudocode for Enrollment.enroll():

```
invoke Registrar.reserveSeat()
if isRollbackOnly returns true
  return
else
  invoke StudentBilling
  completeWork
endif
```

The transaction attribute for Enrollment must be *Requires Transaction* so that the work done by StudentBilling and Registrar occurs as a single transaction.

Dynamic enlistment in bean-managed transactions

EAServer supports dynamic enlistment for bean-managed transactions, which allows you to create a connection in one method, use the connection in another method, and close the connection in a third method.

For a JDBC 2.0 shared connection (PooledConnection), the container manages the single connection’s enlistment and deenlistment in transactions.

For XA connections, the Object Transaction Service libraries need to know all the resources that will participate in a transaction when it starts. If you get an XAConnection before you start a transaction, EAServer enlists the XAConnection in the transaction. If you start a transaction before you create an XAConnection, EAServer creates the connection and enlists it in the transaction.

Dynamic enlistment allows you to do this:

```
connection1 = ds1.getConnection();
// A
user_transaction.begin();
//
connection2 = ds2.getConnection();
connection3 = ds3.getConnection();
// B
connection2.close();
//
user_transaction.commit();
// C
connection3.close();
connection1.close();
```

Where at these points, the following are true:

A – connection1 is not part of any transaction.

B – connection1, connection2, and connection3 are part of the user_transaction.

C – connection1 and connection3 are not part of any transaction.

Earlier versions of EAServer required you to get and release connections within a single component method. In bean-managed transactions, you had to get and release a connection within the scope of a transaction.

You can get only one connection per resource. Each getConnection call for the same database returns the same connection.

Note XA performance diminishes when connections span across methods.

EAServer Transaction Manager

The EAServer Transaction Manager supports the specifications for the Java Transaction API (JTA) 1.0 and the OTS/XA standards. The Transaction Manager supports the integrated functionality of these transaction coordinators: shared connections, OTS/XA, and JTS/JTA, and includes:

- Resource recovery and transaction logging
- Transaction interoperability
- Resource manager

The EAServer Transaction Manager enables EAServer to control the scope and duration of transactions across multiple resource managers. It also provides the ability to synchronize transactions and to communicate with other transaction managers using CORBA OTS. Connections and resources are dynamically enlisted into a transaction when they are requested.

Two-phase commit ensures that all changes to recoverable resources (for example, multiple database servers) occur automically, and the failure of any resource to complete causes all other resources to undo changes. Two-phase commit consists of a prepare phase and an execution phase. In the prepare phase, the transaction coordinator validates that all resources are available. In the execution phase, the transaction coordinator executes all updates to the resources.

You can define components and component methods so that the transaction coordinator automatically handles transactions (implicit control). You can also write component and client code to manage transactions (explicit control).

EAServer implements the javax.transaction.TransactionManager interface, which allows it to control transaction boundaries, and to manage the interaction between Java and Encina transaction objects.

EAServer's implementation of the javax.transaction.Transaction interface enables it to manage a set of javax.transaction.xa.XAResource resources that participate in a transaction. To determine the boundaries and outcome for these transactions, EAServer uses the CosTransaction::Resource interface.

Resource recovery and transaction logging

Resource recovery is a configurable option that provides object persistence and recovery operations. Basic persistence is achieved by writing transactions to a transaction log that contains all the information necessary to re-create the transaction. Persistence is supported for the CosTransactions::Resource and CosTransactions::Synchronization objects that are registered with the transaction. Recovery is supported for JDBC connectors and native type resources that are registered with EAServer. When EAServer starts, the recovery manager is called, which reads the transaction log and starts transaction recovery.

Note Recovery operations can be performed only for transaction logs that were created for EAServer version 5.0 or later.

A transaction log provides enhanced debugging and integrates with the standard EAServer logging functionality. Monitoring functionality is also provided, which allows you to use the Management Console to view statistics, such as the total number of committed transactions and the average duration of transactions.

When EAServer starts, the TransactionLogManager verifies the transaction log's integrity, automatically does necessary repairs, then runs the transaction log defragmenter. This helps to allocate space for new transactions. The recovery manager passes transaction information to the TransactionLogManager, which is responsible for storing and deleting the transaction record from the transaction log.

You can set the following recovery options on the Transactions tab of the Server Properties dialog box:

- Enable Recovery – check to enable.
- Recovery Log File Name – enter the name of the file in which to store the transaction log. You can specify a file name only, or an absolute path to a file. If you specify a file name only, the file is created in the *logs* subdirectory or your EAServe installation.

- Log File Size – enter the maximum file size.

Recovering XA resources registered by user components

In this version of EAServer, you cannot directly recover XA resources that are registered by user components. However, you can enable EAServer to accomplish this task by using the following technique:

1. Create a wrapper DataSource class; for example, WrapperDataSource.
2. WrapperDataSource.getXAConnection() returns an XAConnection class that corresponds to the XA connection with the resource.
3. Create an XA-type data source, and set its class name to the WrapperDataSource class that you created.

Once these steps are implemented, EAServer takes care of the recovery process. This is useful when using a third-party JMS service with XA resources.

Transaction interoperability

EAServer Transaction Manager provides transaction interoperability in accordance with the OTS specifications.

Since EAServer runs in JTS mode, it can share the transaction coordinator across multiple servers. If a transactional component on one server invokes a component method on another server, both components can participate in the same transaction. Also, a client can invoke components on multiple servers that all participate in the same transaction. This feature is useful for load balancing.

Figure 2-3 illustrates a scenario in which a client calls a component method on Server A, which calls a component method on Server B. Server A and Server B use different databases. To ensure that all the database updates occur within the scope of a single transaction, EAServer passes the transaction context between servers.

Figure 2-3: Transaction interoperability

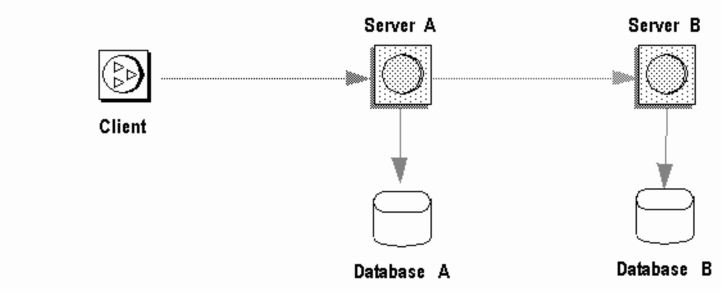
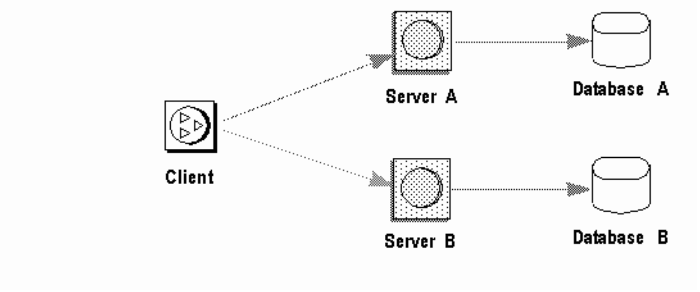


Figure 2-4 illustrates an example where a client calls components on multiple servers, which all participate in the same transaction. The client manages the transaction by calling component methods on each server and passing the transaction context.

Figure 2-4: Server to server



Resource manager

The EAServer Transaction Manager includes an integrated resource manager that supports JDBC 1.0, JDBC 2.0, connectors, and XA resources for both Java and C++. The resource manager allows you to dynamically register resources and synchronize coordinators in accordance with OTS specification for CosTransactions. The resource manager is based on the functionality of both the Java Connection Manager and the Jaguar Connection Manager, which allows you to easily integrate new and existing resources. In future EAServer versions, customers will be able to use the resource manager to create and configure resources that EAServer can use.

Enlisting XA resources with Transaction Manager

When EAServer is running in two-phase commit mode, which is the default for version 5.0 and later, you can enlist XA resources with EAServer Transaction Manager.

❖ Enlisting XA resources

To enlist an XA resource into a current EAServer transaction:

1. Get the instance of Transaction Manager:

```
javax.transaction.TransactionManager tm =
com.sun.jts.jta.TransactionManager.getTransactionManagerImpl();
```

2. Get the instance of the transaction:

```
javax.transaction.Transaction trans = tm.getTransaction();
```

3. Register the XA resource with the transaction:

```
trans.enlistResource(xaresource);
```

EAServer manages this XA resource with respect to its transaction boundaries.

CHAPTER 3 Using CORBA IDL

EAServer stores CORBA component interfaces in Interface Definition Language (IDL) modules.

Learning IDL

IDL is defined by the Object Management Group as a standard language for defining component interfaces. Chapter 3, "OMG IDL Syntax and Semantics," in the *CORBA V2.3 Specification* defines IDL. Printable versions of this document can be downloaded from the following URL:

```
http://www.omg.org/corba/index.html
```

IDL modules

IDL modules form a namespace to group related types and interfaces, similar to C++ namespaces. For example, type Date in module MJD is specified as MJD::Date. Module names must begin with a letter. Modules can be nested by declaring the nested module inside the parent module. For example, to declare interfaces and types in the namespace com::mycompany, use the syntax below and add them to the declaration of module mycompany:

```
module com
{
    module foo
    {
        ...
    };
};
```

Preprocessor directives

The `#include` directive allows you to include code from another file in the current file, using the same syntax and semantics as C++. For example:

```
#include <XDT/DecimalValue.idl>
```

No IDL preprocessor directives other than `#include` are supported.

IDL interfaces

Interfaces define the signatures of CORBA component methods. Each method must be declared as an IDL operation in the IDL interface.

Interfaces are declared as shown below:

```
interface InterfaceName [: BaseInterface1,
BaseInterface2, ...] {
  operations
};
```

where:

- *InterfaceName* is the name of the interface.
- *operations* is a zero or more of IDL operation declarations. See "Operation declarations" on page 29.
- *BaseInterface*, *BaseInterface2*, and so forth form an optional list of existing interfaces from which the new interface inherits definitions. If a new interface inherits from other existing interfaces, the existing interfaces that are inherited from are referred to as *base* interfaces, and the new interface is referred to as a *derived* interface.

For example, this interface, StockComponent, inherits from no other interface:

```
interface StockComponent {
};
```

This interface, C, inherits from interfaces A and B:

```
interface C : A, B {
}
```

Interfaces that inherit definitions from other interfaces are subject to the following constraints:

- *Operations and attributes* cannot be redefined in the new interface.

- *Operation and attribute names* defined in base interfaces must be unique. For example, if a method is defined in both interface A and interface B, you cannot define a new interface that inherits from both B and A.
- *Exceptions, constants, and types* from a base interface can be redefined in the derived interface.
- *References to type names, exception names, and constant names* that are used in multiple derived interfaces must be made unambiguous by prefixing references with the name of the interface that contains the definition of interest. For example, if the constant MAX is defined in both A and B, then A::MAX refers to the definition in A, and B::MAX refers to the definition in B.

Choosing an interface name

Interface names are restricted as follows:

- Interfaces within a module must have unique names, irrespective of case. That is, you cannot define MyInterface and Myinterface in the same module.
- The interface cannot have the same name as the module that contains it.

Sybase recommends that you begin interface names with a capital letter, and operation names with a lowercase letter.

Operation declarations

Operations in an IDL interface become component methods when the interface is assigned to a component. Operations are declared as follows:

```
returnType opName
(
[ ... parameterList ... ]
)
[  raises ( ... exceptionList ... ) ] ;
```

where:

- *returnType* is either a valid IDL datatype or void to indicate that the operation does not return a value. “Datatypes for parameters and return values” on page 32 discusses datatypes in detail.
- *opName* is the name of the operation. Sybase recommends operation names begin with a lowercase letter. Names in the same interface must be unique with respect to case, and capitalization of a name must be consistent wherever it is used.

IDL operation names cannot be overloaded (that is, redeclared with the same return type and different parameter lists). However, you can define IDL operations that map to overloaded C++ or Java methods. To do so, create operation names by appending two underscores and a unique suffix to the method name that will be overloaded. EAServer strips the suffix when generating C++ or Java interface definitions. For example, consider the following IDL:

```
void ov1__double(in double d);
void ov1__string(in long l);
```

When mapped to C++ or Java, these operations translate to the following overloaded methods:

```
void ov1(double d);
void ov1(long l);
```

- *parameterList* is an optional parameter list enclosed in parentheses. The list (but not the parentheses) can be omitted to indicate that the operation takes no parameters. Otherwise, add datatypes and parameter names as shown below:

  ```
  void myMethod
  (
  qual1 type1 param1,
  qual2 type2 param2,
  ...
  );
  ```

  where:

  - *qual1*, *qual2*, and so forth are one of the argument modes in, inout, or out. Use in for parameters that are input-only; no new value is returned when the operation completes. Use inout or out if the operation returns new values for the parameter. An inout parameter's input value is meaningful; an out parameter's input value is not.
  - *type1*, *type2*, and so forth are valid IDL type names (other than the CORBA::Any type). "Datatypes for parameters and return values" on page 32 discusses datatypes in detail.
  - *param1*, *param2*, and so forth are parameter names.

- *exceptionList* is an optional list of user-defined exceptions. If the operation can throw user-defined exceptions, add a raises clause with a list of the IDL user-defined exception names that the operation can throw, as shown below:

  ```
  void myMethod ( in int n )
  ```

```
raises ( Exception1, Exception2, ... );
```

If the operation can throw only CORBA standard exceptions, omit the raises clause. For more information, see “User-defined exceptions” on page 35.

Attribute declarations

Attributes allow you to associate a value with an interface. IDL attributes are similar in concept to structure fields in languages such as C. However, when mapped to a programming language, attribute values can typically be accessed only by generated functions that allow you to set and retrieve the attribute’s value.

Attributes are declared as shown below:

```
[ readonly ] attribute TypeSpec name;
```

where

- readonly is an optional keyword specifying that the attribute can be retrieved but cannot be set.
- *TypeSpec* is the name of a standard or user-defined type. “Datatypes for parameters and return values” on page 32 describes datatypes in detail.
- *name* is the attribute name.

In C++ and Java, a read-only attribute maps to a method with the same name that returns the attribute type. A writable attribute maps to a pair of overloaded methods with the same name as the attribute. For example, consider the following IDL declarations:

```
readonly attribute long days; // readonly
attribute long months;        // writable
```

In a C++ or Java implementation of the interface, these methods must be declared:

```
long days();
long months();
void months(long new_months);
```

Datatypes for parameters and return values

To define parameter and return value datatypes, you can use EAServer's predefined IDL datatypes or your own user-defined IDL types. In addition, EAServer extends IDL to allow the use of Java class names. The sections below describe each option in detail.

- Predefined IDL datatypes
- User-defined IDL datatypes
- Java class names used as IDL datatypes

Predefined IDL datatypes

EAServer ships with predefined datatypes for use in declaring parameter and return value datatypes. Predefined datatypes include all CORBA base types (except for the CORBA::Any type) and equivalents for database result sets and other commonly used database column types such as date, time, and timestamp. Table 3-1 lists these types.

Table 3-1: Predefined EAServer IDL datatypes

CORBA IDL type	Description
boolean	One bit of binary data; a value that is either true or false
short	A 16-bit integer
long	A 32-bit integer
long long	A 64-bit integer
float	Single-precision IEEE floating point numbers
double	Double-precision IEEE floating point numbers
string	A sequence of characters of any length
BCD::Binary	Sequence of bytes
BCD::Decimal	Fixed-point decimal
BCD::Money	Same as decimal
MJD::Date	A date including year, month, day, hour, minute, second, and millisecond values
MJD::Time	Holds the time of day, including hours, minutes, seconds, milliseconds
MJD::Timestamp	Holds the same data as date, plus a nanoseconds value
TabularResults::ResultSet	A single table of relational database rows
TabularResults::ResultSets	A sequence of 0 or more ResultSet objects

For descriptions of the datatypes defined in the BCD, MJD, or TabularResults modules, see the documentation in the *html/ir* subdirectory of your EAServer installation. (Or, load the main EAServer HTML page in your Web browser, and click the Interface Repository link). If you use types from these modules, add an include directive for the appropriate module at the top of the module that defines your interface. For example:

```
#include <TabularResults.idl>
```

Internally, *TabularResults.idl* includes both *BCD.idl* and *MJD.idl*. You need not include *BCD.idl* and *MJD.idl* explicitly if you have already included *TabularResults.idl*.

User-defined IDL datatypes

In addition to EAServer's predefined datatypes, you can define your own datatypes in IDL and use them to declare return types and parameters.

All IDL type definitions are allowed, with these exceptions:

- Fixed sized arrays are supported, but Sybase recommends that you use sequences instead.
- The CORBA::Any type is not supported.
- constant declarations are supported.

EAServer allows forward IDL references
You can create new IDL types that refer to other IDL types that do not yet exist; among other benefits, this feature allows you to create mutually recursive interface definitions. However, you must be sure that all references are resolved before you can generate the package code. EAServer will report errors for any unresolved type references.

For information on defining datatypes, see Chapter 3, "OMG IDL Syntax and Semantics," in the CORBA 2.3 specification.

In some cases, you must use the full scope name. In a parameter list, use a type's full scope name if any of the following is true:

- The type is declared in another interface.
- The type is declared in another module.
- The type has the same local-scope name as a type declared in the interface or module that contains the operation.

For example, consider the IDL:

```
module MyMod {
    typedef string MyType;
    interface MyIntf {
        typedef double MyOtherType;
        ....
    };
};
```

With these declarations, MyMod::MyType is the full scope name for MyType and MyMod::MyIntf::MyOtherType is the full scope name for MyOtherType.

Java class names used as IDL datatypes

EAServer's IDL compiler extends IDL to allow Java class names as parameter and return types for methods. This feature provides functionality that is similar to the proposed Objects by Value CORBA extension (OMG TC Document orbos/98-01-18, *Objects By Value*). Specifically, you can pass a copy of an object rather than passing an interface pointer that refers back to the original object.

You can specify any Java class name for a method input parameter or return type as long as:

- The class containing the type name is in the CLASSPATH environment variable both when the interface is defined and when the server is run.
- At run time, you specify a class instance that is serializable. That is, a class must implement the java.io.Serializable interface or inherit from another class that does so, and an interface must extend the java.io.Serializable interface. If the instance is not serializable, the call fails with a CORBA::MARSHALL exception.

Note the following restrictions for methods that are defined using Java datatypes rather than IDL types:

- Only Java components can implement the method and only Java clients can invoke the method.
- Only in parameters and return values can be declared with Java class names.

- Java datatypes are not marshaled as efficiently as an equivalent IDL datatype. *Marshaling* is the process of reading and writing parameters and return values from the network. More bytes are required to marshal values defined with a Java datatype than to marshal an equivalent IDL type. Consequently, invocations of a method defined with Java datatypes are slower than invocations of an equivalent method defined with IDL datatypes.
- IDL that contains Java class names may not be portable to other CORBA client ORB implementations unless they offer this extension to standard CORBA IDL.

User-defined exceptions

Exceptions can be declared in a module or interface. Exceptions are declared as follows:

```
exception name {
    ... memberList ...
};
```

where *name* is the name of the exception and *memberList* is an optional list of member field declarations. This list has the form:

```
exception MyException {
    type1 member1;
    type2 member2;
    ...
};
```

Where *type1*, *type2*, and so forth are IDL type names (other than CORBA::Any) and *member1*, *member2*, and so forth are the names of the member fields.

Once you have defined an exception, you can use it in the raises clause when defining operations for an interface, as described in “Operation declarations” on page 29.

Interface stub generation directives

For IDL created by deploying EJB and PowerBuilder components, EASerer can embed specially formatted comments in IDL to control the generation of Java stubs for IDL interfaces and structures. These directives appear in a block comment located immediately before the IDL interface or struct declaration.

Imported class name This directive specifies that a structure or interface was imported from a Java class, and that a new version of the imported class must not be generated when stubs are generated. This directive is most commonly used for EJB home and remote interfaces and EJB primary keys that were defined by importing EJB classes or EJB-JAR files.

The format is:

```
** <!-- imported classname -->
```

Where classname is the Java class name, in dot notation. For example, `foo.bar.MyBeanHome` or `foo.bar.MyBeanPrimaryKey`.

Is home interface This directive identifies an interface as a home interface used by EJB clients and components. The format is:

```
** <!-- home -->
```

Finder method return type Applies to multi-object finder methods in an EJB entity bean's home interface. If a finder method's Java form must return java.util.Enumeration, you see a doc comment of this form above the IDL finder method declaration:

```
/*
** <!-- java.util.Enumeration -->
*/
::MyModule::MyRemoteList findByName(in string name);
```

Managing IDL in EAServer

IDL types used by CORBA components must be registered in the EAServer repository. You can register IDL for CORBA components several ways, including:

- Migrating CORBA components from a previous version of EAServer.
- Deploying IDL modules with the Management Console. The Management Console displays IDL modules as folders beneath the top-level IDL folder.
- Deploying IDL modules with the deploy command-line tool.
- By placing IDL files in the *Repository* subdirectory of the server installation and restarting the application server. If the files contains no syntax errors, EAServer registers the types defined in it. If the file does contain syntax errors, the server will log the errors during start-up and the module's declarations will not be added to the IDL repository.

EAServer also creates IDL for EJB and PowerBuilder components upon deployment, allowing interoperability between the CORBA and other component models.

Deploying and viewing IDL with the Management Console

You can import and view IDL in the Management Console.

❖ **Deploying IDL modules in the Management Console**

1. If you haven't already, start EAServer and connect to the preconfigured server with the Management Console as described in Chapter 1, "Getting Started,"in the *System Administration Guide*.
2. In the Management Console, click the IDL Modules folder to display the IDL types in the EAServer repository. Right-click the IDL Modules folder and choose Deploy. The Deploy wizard displays.
3. In the Deploy wizard, specify the IDL file name to be imported.

❖ **Viewing IDL in the Management Console**

1. Highlight and expand the IDL Modules folder in the left pane. A hyperlinked list of modules appears in the right pane, and a tree/folder view of deployed modules appears in the left pane beneath the IDL Modules folder.
2. Use the hyperlinks or tree view to navigate to the interfaces and types defined in each module.

❖ **Deleting IDL in the Management Console**

1. Browse to the IDL module to be removed as described in "Viewing IDL in the Management Console" on page 37.
2. Right-click the module name in the left pane and choose Undeploy.

Warning! Do not delete IDL that is in use by deployed components.

Deploying IDL from the command-line

You can import IDL using the deploy command-line tool. Specify the path and name to the IDL file, as in:

```
deploy MyModule.idl -overwrite true
```

For detailed syntax information, see Chapter 12, “Command Line Tools,” in the *System Administration Guide*.

Specifying Java package mappings for IDL modules

If an IDL module contains datatypes and interfaces (and not just nested modules), EAServer Java classes for the datatypes in a Java package derived from the IDL module name. For example, for IDL types in module foo::bar, the CORBA Java types are in Java package foo.bar, and EJB equivalents are in Java package foo.bar.ejb.

You can override the default Java package name using one of these techniques:

- For CORBA components where the CORBA package name matches the IDL module name, set the Java Package property for the CORBA package (com.sybase.jaguar.package.java.package). See “CORBA package property descriptions” on page 45.
- For stubs generated from other IDL modules, Sybase recommends that you use the default Java package name to simplify coding conventions and avoid redundant Java classes generated from the same IDL module.

  To override the default Java package, specify the -jp option when generating stubs with the idl-compiler command. See the reference page for idl-compiler in Chapter 12, “Command Line Tools,” in the *System Administration Guide*.

Using IDL documentation comments

EAServer includes HTML documentation files for each predefined IDL module in the *html/ir* subdirectory. You can also generate HTML documentation for IDL that you have deployed.

At a minimum, the generated HTML lists the datatypes and interfaces defined in the module. You can embed additional documentation text for a datatype, interface, or method in a C-style comment placed immediately above the declaration. EAServer ignores C++-style line-end comments when generating HTML documentation. That is, text within comments that use double slashes, //, to delineate the comment text is ignored.

Within the C-style comment, add text describing the item to the comment, as in the example below. If desired, you can use HTML codes to format the text. But do not use heading tags such as <H1>, <H2>, and so forth, because they conflict with tags that are already used to structure the sections of the generated output.

The IDL fragment below contains an example of a documentation comment:

```
/**
 ** Example method to demonstrate user-defined
 ** exceptions.
 ** <P>Pass <I>yes_no</I> as <code>true</code>
 ** if you want an exception thrown.
 ** <P>Returns input value of <I>yes_no</I>
 ** parameter.
 */
boolean throwException
(
in boolean yes_no
)
raises
(
myException
);
```

You need not use the spacing conventions illustrated in this example. EAServer treats any C-style comment as an IDL documentation comment. However, when you deploy IDL, EAServer may reformat white space in code and comments.

Stub generation directives in IDL comments
You can embed directives in IDL comments to affect the Java stubs generated for a module or interface. See "Interface stub generation directives" on page 35 for more information.

Refreshing the HTML documentation

HTML documentation is not generated automatically. You must use the EAServer IDL compiler to create or update documentation for new or changed IDL modules. See the reference page for idl-compiler in Chapter 12, "Command Line Tools," in the *System Administration Guide*.

Viewing HTML documentation for IDL modules

EAServer creates HTML documentation for all imported IDL modules in the style of Sun's javadoc tool. At a minimum, this documentation lists the datatypes and interfaces defined in the module, including structure fields, array lengths, parameter names and datatypes, exceptions thrown by methods, and so forth. When editing IDL, you can also create specially-formatted comments that provide descriptions of entities declared in the IDL file, as described in "Using IDL documentation comments" on page 38.

Module documentation can be viewed in a Web browser by connecting to your server with this URL:

```
http://yourhost:yourport/ir/
```

where *yourhost* is the host name and *yourport* is the HTTP port number.

CHAPTER 4

Managing CORBA Packages and Components

What is a CORBA package?

In EAServer, CORBA packages are the unit of deployment for CORBA and PowerBuilder components. A CORBA package allows you to group related components together in the same deployment or export configuration. Packages also provide a means to configure security constraints for related components. You can configure role-based authorization on the package to limit access to all components in the package.

You can create and configure CORBA packages several ways, including:

- Using the Management Console
- Using configuration scripts
- Migrating CORBA and PowerBuilder components from a previous version of EAServer
- By deploying PowerBuilder components from the PowerBuilder IDE

The use of the Management Console and configuration scripts are described in this chapter. For information on migrating components, see the *Migration Guide*. For information on deploying from PowerBuilder, see the PowerBuilder documentation or online help.

Managing CORBA packages in the Management Console

The Management Console provides user-friendly graphical interfaces to manage CORBA packages and components.

❖ Creating a CORBA package in the Management Console

1. If you haven't already, start EAServer and connect to the preconfigured server with the Management Console as described in Chapter 1, "Getting Started,"in the *System Administration Guide*.
2. In the Management Console, right-click the CORBA Packages folder and choose Add. The Add wizard displays.
3. In the Add wizard, specify the new package name. Note the restrictions described in "Restrictions on package names" on page 42.
4. When you finish the Add wizard, the package properties display. Configure the properties described in "CORBA package property descriptions" on page 45. Click Apply to save any changes.

Restrictions on package names
Package names must be unique among other packages in the same EAServer installation, and begin with a letter.

Names are not case sensitive. Your packages must have unique names that differ in ways other than letter case. For example, you cannot define two packages named *MyPack* and *mypack* in the same EAServer installation. You cannot have two packages with the same name, even if one is installed in an application and the other is not.

❖ Creating CORBA components in the Management Console

1. Create the CORBA package as described in "Creating a CORBA package in the Management Console" on page 42.
2. In the Management Console, expand the CORBA Packages folder. Locate the icon for the package in which you are creating the component. Double-click the icon to display the Components folder beneath it.
3. Right-click the Components folder beneath the target package, and click Add. The Add wizard runs and prompts for values for the most commonly configured component properties.

4 When you finish the Add wizard, the component properties display in the right pane. Configure the properties described in "CORBA component property descriptions" on page 45. Click Apply to save any changes.

❖ Refreshing CORBA packages in the Management Console

The Refresh action in the CORBA Package context menu creates (or recreates) the generated code and EJB wrapper components required to run the components in the package. If the components are loaded in the server, the new implementation is loaded to replace the old. Refresh the package as follows:

1 If you haven't already, start EAServer and connect to the preconfigured server with the Management Console as described in Chapter 1, "Getting Started,"in the *System Administration Guide*.

2 In the Management Console, expand the CORBA Packages folder, then right-click the icon for the package to be configured and choose Refresh from the context menu.

3 The Management Console runs the configuration commands to regenerate the component's generated code and reload the implementation. If the operation fails, check the server log file for errors.

Managing CORBA packages with configuration scripts

Configuration scripts allow you to automate the creation and deployment of CORBA components. For a description of the Ant configuration mechanism used in EAServer, see Chapter 2, "Ant-Based Configuration," in the *Automated Configuration Guide*.

The sample script below defines a configure target that shows the commands required to define and configure a CORBA package and component.

```
<?xml version="1.0"?>

<project name="cpptut" default="configure">

  <import file="${djc.home}/config/ant-config-tasks.xml"/>

  <property name="package.name" value="packageName" />

  <target name="configure">
    <setProperties package="${package.name}">
      <property name="com.sybase.jaguar.package.roles" value=""/>
```

```
    </setProperties>

    <setProperties component="${package.name}/compName">
      <property name="propName" value="propValue"/>
        ...
    </setProperties>

    <jaguarJarCompiler package="${package.name}"/>

  </target>

</project>
```

The example defines the CORBA package name as the top-level Ant property package.name. Since the package name appears several places in the script, it is convenient to define it in one place and reference the property with the Ant syntax `${package.name}`.

Inside the configure target, the script runs these commands:

1. The first setProperties command creates and configures the CORBA package. Any modifications to the default package properties must be made with nested property commands in this command. See “CORBA package property descriptions” on page 45.
2. For each component in the package, an additional setProperties command creates and configures the component. To ensure EAServer creates the component in the intended package, the value of the component attribute for the setProperties command must use the syntax:

   ```
   package/component
   ```

   Where package is the CORBA package name, and component is the component name as it should display in the Management Console and output from configuration and status commands.

   Use nested property commands to configure the component properties. See “CORBA component property descriptions” on page 45.
3. The jaguarJarCompiler command generates the EJB wrapper components and other code required to run the components in the server.

CORBA package property descriptions

CORBA package properties affect code generation and security constraints for the components in the package. Table 4-1 lists the properties.

Table 4-1: CORBA package properties

Management Console property name	Configuration script property name	Description
EJB Version	com.sybase.jaguar.package. ejb.version	The EJB specification version to use for the generated EJB wrapper components. Allowable values are 2.0 (the default) and 2.1.
Java Package	com.sybase.jaguar.package. java.package	The Java package name for EJB home and remote interfaces used by the generated EJB wrapper components. If not set, the default Java package mapping for the component's CORBA IDL module is used. For IDL module MyModule, the default Java Package is MyModule.ejb. This property applies only for components in the package that use IDL interfaces defined in a module that matches the CORBA package name. For interfaces defined in a different module, the Java package name is the IDL module name suffixed with .ejb. For example, Java interfaces generated to match IDL module Tutorial use Java package Tutorial.ejb.
Required Roles	com.sybase.jaguar.package.roles	A comma separated list of security role names required for users to invoke components in the package. The package property configures the default role list for components for which the component Roles Required property (com.sybase.jaguar.component.roles) is not set.

CORBA component property descriptions

Table 4-2 describes the CORBA component properties. The first column contains the property names displayed in the Management Console. The second column lists the suffixes for the property name used to configure the property within a setProperties Ant command. The full property name begin with:

```
com.sybase.jaguar.component.
```

For example, the pooling entry in the table must be configured as com.sybase.jaguar.component.pooling.

Table 4-2: CORBA component properties

Management Console name	Configuration file property suffix	Description
Name	name	The component name. This property is read-only once the component has been created.
Component Type	type	The component type. Allowable values are as follows, with configuration script values in parentheses: • CORBA/C++ (cpp) • CORBA/Java (java) • PowerBuilder (pb)
Code Set	code.set	For C++ components, specifies the coded character set name used to encode character and string parameter data. For the list of supported values, list the subdirectories of the charsets directory. Each subdirectory matches the name of a supported character set. Input values for string parameters (and string fields within complex datatype values) are converted to this code set before each method invocation. Upon return, output values are converted from the component's code set to the client's code set. If your C++ component uses Client-Library connection caches, you cannot specify a code set that is different than the server code set. Character data read over a cached Client-Library connection is always in the server's code set. If a component code set is not specified, the default is the server's code set.
Expose as Web Service	web.service	Whether the component is exposed as a Web service. If enabled, the component's EJB remote interface is exposed as a Web service. The default is false.
Roles Required	roles	A comma-separated list of security role names. If set, clients cannot invoke the component unless they connect with a user name that is in one of the assigned roles. If not set, the default is the value of the Roles Required property (com.sybase.jaguar.package.roles) in the CORBA package properties–see "CORBA package property descriptions" on page 45.
C++ Class	cpp.class	For C++ components, the name of the C++ class that implements the component methods.

Management Console name	Configuration file property suffix	Description
C++ Library	cpp.library	For C++ components, the base name of the library file that contains the component implementation.
Copy Library	cpp.copy	For C++ components, specifies whether the server should copy the component library before running it. The default is false. Set this property to true to allow updates to the implementation on operating systems that do not allow overwriting a DLL or shared library while the library is in use.
Debug Library	cpp.debug	For C++ components, specifies whether to catch exceptions. The default is true, which specifies that exceptions are caught in the server. Use the default of true for deployment to production servers to ensure that exceptions thrown by component code do not terminate the server process. When debugging an executing component, set this property to false to allow exceptions to reach your debugger. You must set this property to true to debug an executing C++ component in Microsoft Visual C++. Other C++ debuggers may require the same setting as well.
Java Class	java.class	For CORBA/Java components, specifies the Java class name that implements the component methods.
PowerBuilder NVO Class	pb.class	For PowerBuilder components, specifies the name of the nonvisual object that implements the component's methods. This property is set by deployment from PowerBuilder and should be treated as read-only in EAServer.
PowerBuilder Libarry List	pb.librarylist	For PowerBuilder components, specifies library files that are required to run the object. Set the value to the list of library files separated by semicolons. Prefix library names with a dollar sign ($) if they must be included when the component is included in an export configuration or cluster synchronization. For example: `mylib.pbl;anotherlib.pbl;$utils.pbl` This property is set by deployment from PowerBuilder and should be treated as read-only in EAServer.
PowerBuilder Version	pb.version	For PowerBuilder components, specifies the required version of the PowerBuilder virtual machine. This property is set when deploying from PowerBuilder, and should not be edited in any other way. Components that lack this property setting are run in the version 7.0 VM. This property is set by deployment from PowerBuilder and should be treated as read-only in EAServer.

Management Console name	Configuration file property suffix	Description
IDL Home Interface	home	The name of the IDL home interface, in IDL syntax. For example, Tutorial::CPPArithmeticHome. The home interface allows interoperability with EJB clients. If you specify a home interface when creating a component, the IDL interface must exist already. To have EAServer create a default interface, leave this setting blank when creating new components. If you do not specify an IDL home interface when creating the component, EAServer creates one the first time you refresh the component in the Management Console or run the jaguar-compiler command on the package using a configuration script or the command line.
IDL Remote Interface	remote	The name of the IDL remote interface, in IDL syntax. For example, Tutorial::CPPArithmetic. The remote interface specifies the signatures of the component methods that can be invoked by clients. You must set this property and specify the name of a valid IDL interface that has been deployed to EAServer—see Chapter 3, "Using CORBA IDL."
IDL Component Interfaces	interfaces	Optional. Specifies IDL interfaces that the component supports for client use in addition to the remote interface. If set, specify a comma-separated list of IDL interface names.
Pooled	pooling	Specifies whether component instances should always be pooled. If set to true, lifecycle methods related to instance pooling are not called, such as canBePooled or canReuse.
Instance Pool Timeout	instancePool.timeout	If the component supports instance pooling, specifies how long, in seconds, an instance can remain idle in the pool. The default is 600 (ten minutes). To free resources used by idle component instances, the server may remove instances that remain idle past this time limit.

Management Console name	Configuration file property suffix	Description
Passivation Timeout	timeout	For stateful components, specifies how long, in seconds, an active component instance can remain idle between method calls before EAServer destroys the instance. The default of "0" indicates an infinite timeout. Instance Timeout is useful for ensuring timely deactivation of stateful components. When the timeout period is exceeded, EAServer deactivates the component and invalidates the client's object reference. If the client attempts another method invocation, the client-side ORB throws the CORBA::OBJECT_NOT_EXIST exception. At this point, the client must create a new proxy instance for the component. When specifying timeouts, a resolution of 5 seconds is recommended. Network transport time is not included in the measured timeout period. You may need to configure a larger timeout period if clients connect over slow networks.
Thread Monitor	monitor	Optional. Specifies the name of a thread monitor that constrains component execution. Thread monitors provide a mechanism to govern how many instances of a component can be simultaneously active in the server For more information, see "Monitoring threads" in Chapter 3, "Creating and Configuring Servers," in the *System Administration Guide*.
Transaction Type	tx_type	For components that use connection caches to perform database work, specifies how the database work is scoped in a server managed transaction. "Transaction type values" on page 52 describes the allowable values.
Transaction Outcome	tx_outcome	For components that participate in server managed transactions, determines whether a CORBA::TRANSACTION_ROLLEDBACK exception is thrown to the client when a transaction is rolled back by the component or due to an error in component execution. The default value, `always`, specifies that EAServer sends a CORBA::TRANSACTION_ROLLEDBACK exception to the client when a transaction is rolled back. The value `failed` specifies that EAServer does not send the CORBA::TRANSACTION_ROLLEDBACK exception to the client when a transaction is rolled back. If you use this setting, you can code your components to raise a different exception with a descriptive message after calling the RollbackWork transaction primitive. With this setting in effect, EAServer may still throw a CORBA system exception if unable to commit a transaction at your component's request.

Management Console name	Configuration file property suffix	Description
Transaction Retry	tx_retry	Specifies whether container-managed transactions should be automatically retried after a rollback. The default is false.
Automatic Failover	auto.failover	Specifies whether client proxies can transparently fail over to a different server when the component is deployed to several servers in a cluster. The default of false prevents failover. For more information on failover support, see Chapter 8, “Load Balancing, Failover, and Component Availability,” in the *System Administration Guide*.
Bind Object	bind.object	Specifies whether instances remain bound to client’s object reference after setComplete is called. The default is false. Cannot be set to true unless the component is stateful and thread-safe.
Bind Thread	bind.thread	Specifies whether instances are bound to the thread that created them. If true, the component instance is always called by the same thread. The default is false. Set this property to true if your component uses thread-local storage. Otherwise, use the default of false for best performance. Enabling Bind Thread requires EAServer to create an extra thread for each component instance.
Pooled	pooling	Specifies whether component instances are pooled for reuse by multiple client sessions. The default is false.
Shared	sharing	Specifies whether a single instance or multiple instances of the component implementation serve clients.If set to true, a single instance serves all clients. The default of false means multiple instances are used. If sharing is enabled, and the Thread Safe property is enabled, you must synchronize access to read-write instance variables in the implementation.
Stateful Session Bean	tx_vote	Specifies whether the component is wrapped by an EJB stateful session bean to allow stateful behavior. The default is false, which causes EAServer to wrap the component with a stateless session bean. In business methods, stateful CORBA components must call the transaction state primitive methods to indicate the session state. For example, completeWork or rollbackWork ends the session and deactivates the component instance. For details, see “Using transaction state primitives” on page 16. If long transactions are enabled for the server, server managed transactions depend on the component’s invocation of the transaction state primitive methods. See “Long versus short transactions” on page 10 for more information.

Management Console name	Configuration file property suffix	Description
Thread Safe	thread.safe	Specifies whether multiple component instances can execute concurrently, or whether a shared component can execute simultaneously on multiple threads. The default is true. If set to false, the server serializes all invocations of component methods. **Note** Sharing is true, this property specifies whether it is safe for multiple threads to simultaneously call business methods on a singleton instance of this component.
Debug	debug	Specifies whether the server logs trace information for instance life cycle events such as creation, destruction, pooling, and so forth. The default is false.
MDB Acknowledge Mode	mdb.acknowledge-mode	For CORBA message listener components that have been migrated from EAServer 5.x. Applies only if the remote interface is CtsComponents::MessageListener. Specifies the acknowledgment mode for MDBs that manage their own transactions. Allowable values are: • Auto-acknowledge – The default. The session automatically acknowledges messages. • Dups-ok-acknowledge – Instructs a session to lazily acknowledge messages, which reduces a session’s workload but may lead to duplicate message deliveries.
MDB Topic Name	topic	For CORBA message listener components that have been migrated from EAServer 5.x. Applies only if the remote interface is CtsComponents::MessageListener. For MDBs associated with a message topic, specifies the name of the topic.
MDB Destination Type	mdb.destination-type	For CORBA message listener components that have been migrated from EAServer 5.x. Applies only if the remote interface is CtsComponents::MessageListener. Specifies whether the component listens on a JMS topic or message queue. Allowable values are: • javax.jms.Topic • javax.jms.Queue The default is javax.jms.Queue

Management Console name	Configuration file property suffix	Description
MDB Queue Name	queue	For CORBA message listener components that have been migrated from EAServer 5.x. Applies only if the remote interface is CtsComponents::MessageListener. For MDBs associated with a message queue, specifies the name of the queue.
MDB Message Selector	mdb.message-selector	For CORBA message listener components that have been migrated from EAServer 5.x. Applies only if the remote interface is CtsComponents::MessageListener. If the component listens on a message queue, specifies the message selector. The message service uses the selector to filter the message that it delivers to the queue. Use the syntax: `topic='`*`topicString`*`'` Where *topicString* is the selector to filter messages.
MDB Subscription Durability	mdb.subscription-durability	For CORBA message listener components that have been migrated from EAServer 5.x. Applies only if the remote interface is CtsComponents::MessageListener. For components that listen on a topic, specifies whether the topic is durable or nondurable. Allowable values are: • Durable – Durable topic subscriber; guarantees message delivery. • NonDurable – Nondurable topic subscriber; can receive published messages only while it is connected to EAServer.
MDB Thread Count	mdb.thread-count	For CORBA message listener components that have been migrated from EAServer 5.x. Applies only if the remote interface is CtsComponents::MessageListener. Specifies the number of instances that EAServer creates to respond to incoming messages. Multiple instances can run simultaneously and may improve performance. The default is 1.

Transaction type values

The CORBA component's Transaction Type property (com.sybase.jaguar.component.tx_type) determines how database work is scoped in a server managed transaction.Allowable values are as follows (values for use in setProperty commands are in parentheses):

- Not Supported (not_supported) – The default. The component's methods never execute as part of a transaction. If the component is activated by another component that is executing within a transaction, the new instance's work is performed outside of the existing transaction.
- Bean Managed (bean_managed) – The component implementation explicitly begins and ends transactions. The component can inherit a client's transaction. If called without a transaction, the component can explicitly begin, commit, and roll back transactions by using the CORBA CosTransactions::Current interface.
- Mandatory (mandatory) – Methods may only be invoked by a client that has an outstanding transaction.
- Never (never) – The component's methods never execute as part of a transaction, and the component cannot be called in the context of a transaction. If a client or another component calls the component with an outstanding transaction, EAServer throws an exception.
- Requires (requires) – The component always executes in a transaction. When the component is instantiated directly by a base client, a new transaction begins. If component A is activated by component B, and B is executing within a transaction, then A executes within the same transaction; if B is not executing in a transaction, then A executes in a new transaction.
- Requires New (requires_new) – Whenever the component is instantiated, a new transaction begins. If component A is activated by component B, and B is executing within a transaction, then A begins a new transaction that is unaffected by the outcome of B's transaction; if B is not executing in a transaction, then A executes in a new transaction.
- Supports (supports) – The component can execute in the context of an EAServer transaction, but one is not required to execute the component's methods. If the component is instantiated directly by a base client, EAServer does not begin a transaction. If component A is instantiated by component B, and component B is executing within a transaction, component A executes in the same transaction.

CHAPTER 5

Developing and Deploying PowerBuilder Components

This chapter describes EAServer-specific modifications for developing PowerBuilder components.

For general instructions on developing PowerBuilder components, see Application Techniques at http://infocenter.sybase.com/help/index.jsp?topic=/com.sybase.help.pb_10.5.apptech/html/apptech/title.htm.

EAServer hosts the PowerBuilder virtual machine natively. This means that EAServer can communicate directly with PowerBuilder nonvisual user objects, and vice versa. EAServer components developed in PowerBuilder can take full advantage of the ease of use and flexibility of PowerScript and the richness of PowerBuilder's system objects.

The PowerBuilder IDE runs on Windows platforms, but you can deploy PowerBuilder components to EAServer on any platform for which a compatible PBVM is available, including most UNIX platforms. For more information, see the EAServer *Release Bulletin* for your platform.

PowerBuilder provides full-fledged support for EAServer component technologies, including:

- Instance pooling, by configuring the Pooling setting in the wizards and optionally implementing lifecycle methods to control whether specific instances are pooled.
- Server-managed transactions, by configuring the Transactions settings in the wizards and by calling the methods in the TransactionServer context object.

- Database connection caching, when using DataStore objects or embedded SQL in your implementation code.
- Result sets, by using the PowerScript DataStore, ResultSet, and ResultSets objects. You can use the DataStore object to return result sets that are presented in the client using DataWindow controls. You can also use the ResultSet and ResultSets objects to return tabular results to clients of other types.
- Intercomponent calls, using the CreateInstance method in the TransactionServer object to obtain proxies for components.
- Logging, using the ErrorLogging object to write error or status messages to the server log file.
- Running independent of client interaction, using the EAServer thread manager or service component model.

Developing PowerBuilder components

The PowerBuilder IDE includes wizards to create EAServer components and deployment projects. If you must set additional component properties that cannot be set from the PowerBuilder IDE, consider creating a script or batch file that uses an Ant configuration file or the jagtool set_props command to configure these additional settings. Doing so allows you to maintain an automated deployment mechanism. For more information, see these chapters in the *Automated Configuration Guide*:

- Chapter 2, “Ant-Based Configuration”
- Chapter 6, “Using jagtool and jagant”

Mapping datatypes

Beginning in EAServer version 6.0, PowerBuilder NVOs are wrapped as EJBs. Table 5-1 on page 57 describes the PowerBuilder to EJB datatype mappings, which are applied when an NVO package is wrapped as an EJB module. NVO is a generic term used to describe “custom class user objects,” which inherit directly from the PowerBuilder system type NonVisualObject.

Mappings for datatypes passed by value are valid for in and return parameter modes. Mappings for datatypes passed by reference are valid for out and inout parameter modes.

Table 5-1: PowerBuilder to EJB datatype mappings

PowerBuilder type	EJB parameter type
Blob by value	byte[]
Blob by reference	javax.xml.rpc.holders.ByteArrayHolder
Boolean by value	boolean
Boolean by reference	javax.xml.rpc.holders.BooleanHolder
Byte by value	byte
Byte by reference	javax.xml.rpc.holders.ByteHolder
See "Byte datatype" on page 59.	
Char by value	char – see "Character datatypes" on page 59.
Char by reference	No mapping exists for Char passed by reference (out and inout parameter modes).
Date by value	java.util.Calendar
Date by reference	javax.xml.rpc.holders.CalendarHolder
DateTime by value	java.util.Calendar
DateTime by reference	javax.xml.rpc.holders.CalendarHolder
Decimal by value	java.math.BigDecimal
Decimal by reference	javax.xml.rpc.holders.BigDecimalHolder
Double by value	double
Double by reference	javax.xml.rpc.holders.DoubleHolder
Integer by value	short
Integer by reference	javax.xml.rpc.holders.ShortHolder
	For Java client components that communicate with PowerBuilder server components, the numerical range that this datatype supports is -32768 – 32767.
Long by value	int
Long by reference	javax.xml.rpc.holders.IntHolder
	For Java client components that communicate with PowerBuilder server components, the numerical range that this datatype supports is -2147483648 – 2147483647.
LongLong by value	long
LongLong by reference	javax.xml.rpc.holders.LongHolder
Real by value	float
Real by reference	javax.xml.rpc.holders.FloatHolder

PowerBuilder type	EJB parameter type
String by value	String
String by reference	javax.xml.rpc.holders.StringHolder
Time by value	java.util.Calendar
Time by reference	javax.xml.rpc.holders.CalendarHolder
MyModule_MyArray[] or MyArray[] (return type only)	MyModule.ejb.MyElement[]
MyModule_MyException or MyException	MyModule.ejb.MyException
MyModule_MyComp or MyComp by value	MyModule.ejb.MyComp
MyModule_MyComp or MyComp by reference	MyModule.ejb.holders.MyCompHolder
MyModule_MyStruct or MyStruct by value	MyModule.ejb.MyStruct
MyModule_MyStruct or MyStruct by reference	MyModule.ejb.holders.MyStructHolder
MyModule_MyUnion or MyUnion by value	MyModule.ejb.MyUnion
MyModule_MyUnion or MyUnion by reference	MyModule.ejb.holders.MyUnionHolder
MyModule_MyElement[] or MyElement[] by value	MyModule.ejb.MyElement[]
MyModule_MyElement[] or MyElement[] by reference	MyModule.ejb.holders.ArrayOfMyElementHolder
MyModule_MySequence or MySequence (return type only)	MyModule.ejb.MyElement[]
MyModule_MyElement[n] or MyElement[n] by value	MyModule.ejb.MyElement[]
MyModule_MyElement[n] or MyElement[n] by reference	MyModule.ejb.holders.MyArrayHolder
ResultSet by value	java.sql.ResultSet
ResultSet by reference	TabularResults.SqlResultSetHolder
ResultSets by value	java.sql.ResultSet[ ]
ResultSets by reference	TabularResults.SqlResultSetsHolder
XDT_BooleanValue by value	java.lang.Boolean
XDT_BooleanValue by reference	javax.xml.rpc.holders.BooleanWrapperHolder
	See “XDT datatypes” on page 60.
XDT_CharValue by value	java.lang.Character
XDT_CharValue by reference	XDT.CharacterWrapperHolder
	See “Character datatypes” on page 59.
XDT_ByteValue by value	java.lang.Byte
XDT_ByteValue by reference	javax.xml.rpc.holders.ByteWrapperHolder
XDT_ShortValue by value	java.lang.Short
XDT_ShortValue by reference	javax.xml.rpc.holders.ShortWrapperHolder
XDT_IntValue by value	java.lang.Int
XDT_IntValue by reference	javax.xml.rpc.holders.IntegerWrapperHolder

PowerBuilder type	EJB parameter type
XDT_LongValue by value	java.lang.Long
XDT_LongValue by reference	javax.xml.rpc.holders.LongWrapperHolder
XDT_FloatValue by value	java.lang.Float
XDT_FloatValue by reference	javax.xml.rpc.holders.FloatWrapperHolder
XDT_DoubleValue by value	java.lang.Double
XDT_DoubleValue by reference	javax.xml.rpc.holders.DoubleWrapperHolder
XDT_DecimalValue by value	java.math.BigDecimal
XDT_DecimalValue by reference	javax.xml.rpc.holders.BigDecimalHolder
XDT_IntegerValue by value	java.math.BigInteger
XDT_IntegerValue by reference	javax.xml.rpc.holders.BigIntegerHolder
XDT_DateValue by value	java.util.Calendar
XDT_DateValue by reference	javax.xml.rpc.holders.CalendarHolder
XDT_TimeValue by value	java.util.Calendar
XDT_TimeValue by reference	javax.xml.rpc.holders.CalendarHolder
XDT_DateTimeValue by value	java.util.Calendar
XDT_DateTimeValue by reference	javax.xml.rpc.holders.CalendarHolder

Byte datatype

PowerBuilder version 10.5 introduced a Byte datatype. To use the PowerBuilder Char datatype for backward compatibility, run the following command (once) before deployment:

```
configure idl-octet-to-pb-char
```

To switch back to using the PowerBuilder Byte datatype, run the following command (once) before deployment:

```
configure idl-octet-to-pb-byte
```

Character datatypes

Only characters in the ISO 8859-1 character set can be used for in and return parameter modes. To propagate other characters, use the String datatype.

The char and java.lang.Character datatypes have no defined XML schema mappings for EJB Web services, so you cannot use these as a parameter types or structure field types if you intend to expose a component as a Web service. Use the String datatype instead.

CORBA C++ datatypes

For CORBA C++ datatypes, see CORBA IDL to C++ Language Mapping at http://www.omg.org/technology/documents/formal/c++.htm.

DataStore system object

Sybase recommends that you use the PowerBuilder DataStore system object with the ResultSet return type, especially for NVOs running in an application server. For improved performance, use NVO instance variables, and create the DataStore and assign the DataObject in your NVO constructor.

XDT datatypes

To obtain the PowerBuilder XDT_* datatypes to use as PowerBuilder structure field types or component parameter types, use the EAServer Proxy wizard or the Application Server Proxy wizard in the PowerBuilder IDE to generate proxies for the XDT package. Each of the XDT_* datatypes contains a value field and an isNull field. You must set isNull to true if you want to indicate null values.

Accessing data

From PowerBuilder NVOs, you can access data using either JDBC data sources or Sybase native data sources.

❖ Accessing JDBC data sources in NVOs

1. To set up a JDBC data source in an NVO, use the following PowerScript code, where DefaultDS is the name of an EAServer data source:

   ```
   sqlca.dbms = "JDBC"
   sqlca.dbparm = "CacheName='DefaultDS'"

   connect;    // check error code
   ...         // use embedded SQL or DataStore

   disconnect; // check error code
   ```

2. To assign a JNDI name to your JDBC data source, see “Configuring data sources” in Chapter 4, “Database Access,” in the *EAServer System Administration Guide*.

❖ Accessing Sybase native data sources in NVOs

- Use the following PowerScript code, where JCM_Sybase is the name of an EAServer data source:

   ```
   sqlca.dbms = "SYJ"
   sqlca.dbparm = "CacheName='JCM_Sybase'"
   ...
   ```

Logging errors

The PowerBuilder ErrorLogging class writes errors to *%DJC_HOME%\logs\pb-server.log*. To use PowerScript to create an instance of the class and to log messages, use the following syntax:

```
ErrorLogging logger
```

```
getContextService("ErrorLogging", logger)
logger.log("My Message")
```

Managing transactions

The PowerBuilder TransactionServer class supports the following methods:

- CreateInstance – (for NVO intercomponent calls) use the two-argument form, and specify the full JNDI name of the target component:

  ```
  TransactionServer ts
  getContextService("TransactionServer", ts)

  // generate and use proxies
  pbtest_MyComp comp
  ts.createInstance(comp, "pbtest/MyComp")

  // call methods on comp
  ```

- DisableCommit – prevents the current transaction from being committed, because the component's work has not been completed. The instance remains active after the current method returns.
- EnableCommit – the component should not be deactivated after the current method invocation; allows the current transaction to be committed if the component instance is deactivated.
- IsInTransaction – determines whether the current method is running in a transaction.
- IsTransactionAborted – determines whether the current transaction has been aborted.
- SetAbort – specifies that the component cannot complete its work for the current transaction and that the transaction should be rolled back. The component instance is deactivated when the method returns.
- SetComplete – indicates that the component has completed its work in the current transaction and that, as far as it is concerned, the transaction can be committed and the component instance can be deactivated.

UseContextObject parameter

If you plan to use the TransactionServer context object to work with EAServer transaction service primitives, you may want to set the UseContextObject DBParm parameter for your connection to yes. If a component supports transactions, setting UseContextObject to yes tells PowerBuilder that you will be using the TransactionServer object methods, rather than COMMIT and ROLLBACK, to indicate that the component has completed its work for the current transaction. If your scripts call COMMIT and ROLLBACK, they will generate database errors in the SQLCA.SqlErrText string, which can help you refine your code during development.

If you want to continue to call COMMIT and ROLLBACK on a PowerBuilder Transaction object, set UseContextObject to no. For components that use an EAServer data source, this causes COMMIT and ROLLBACK statements to behave like the TransactionServer object's SetComplete and SetAbort functions, which call the EAServer transaction service's CommitWork and AbortWork methods.This is the default.

For components that do not support transactions, the UseContextObject setting is ignored, and PowerBuilder drivers handle COMMIT and ROLLBACK statements.

Deploying components

The deployment tool wraps your PowerBuilder NVOs as standard EJB session beans. Target-specific deployment descriptors are generated to bind JNDI names and JDBC data source resource references automatically.

PowerBuilder components

You can use the Project painter to deploy PowerBuilder components.

❖ **Deploying PowerBuilder components**

1. In the Project painter Properties dialog box, select the EAServer Host tab, and enter:

   - Host Name – the TCP host name for the server machine. Do not use "localhost" or the IP address.
   - Port – the IIOP port number on the host machine; the default is 2000.
   - Login ID – `admin@system`.

- Login Password – a valid password for the login ID.

Note To override the host name and port number that the server uses for its deployment listener, see “Configuring listeners” on page 41, in the *EAServer System Administration Guide*.

---

2 From the Component type drop-down list on the Components page, select one of:

- Shared component – allows multiple clients to share the same component instance. This provides access to common data that would otherwise need to be retrieved separately by each client connection, and reduces the number of database accesses, allowing the database server to be available for other processing.
- Service component – performs background processing for EAServer clients and other EAServer components. EAServer loads service components at server start-up time. A service component can also be shared.
- Standard component – you can improve performance by allowing multiple instances of a component to handle client requests.

For detailed information about these component types, see “Building an EAServer Component” in the PowerBuilder *Application Techniques* book.

3 In the Standard Options group box on the Components page, to use stateless EJB session beans, select Automatic Demarcation / Deactivation; to use stateful session beans, unselect this option.

Live editing

You can quickly test changes without redeploying, using live editing—see “Testing and debugging the component” in Application Techniques at http://infocenter.sybase.com/help/index.jsp?topic=/com.sybase.help.pb_10.5.apptech/html/apptech/CHDDHCCJ.htm.

To increase the speed at which live editing runs, perform the following configuration:

1 Open *config/deploy-tool-options.xml*, in your EAServer installation.

2 In the com.sybase.jaguar.compiler.JaguarCompiler component description, set the value of ejbDeployIfUnchanged to false.

3 Run:

```
configure deply-tool-options
```

Java packages

For an NVO package called “xyz,” the default Java package name for generated EJB interfaces is “xyz.ejb.”

❖ Changing default Java package names

1 In the Project painter Properties dialog box, select the General tab.

2 In the Comments box, enter the Java package name; for example:

```
javaPackage="com.example.bank";
```

The semicolon is required.

Web services

To deploy a PowerBuilder NVO as a Web service, you must define the Java package name.

❖ Deploying Web services

1 In the Project painter Properties dialog, select the General tab.

2 In the Comments box, enter the names of the components to generate as Web services; for example:

```
javaPackage="com.example.bank";webServices="MyComp1,MyComp2";
```

The final semicolon is required.

3 On the WebService tab, select Expose the Component as a Web Service.

Generated code

The base directory for generated files is *%DJC_HOME%\genfiles\java*, which includes the following subdirectories:

- *applications*
- *classes*
- *ejbjars*
- *src*

Typically, you can delete generated files after deployment, but this causes redeployment to be slower.

Naming conventions

You cannot use hyphens in the names of PowerBuilder components or methods.

PowerScript method and parameter identifiers that contain underscores are mapped to Java names using **lowerCamelCase**; field names are not mapped when using the camel case option. See "Camel case versus default IDL-to-Java mappings" on page 144.

A similar mapping is used for structure names, but the first letter is capitalized; for example, "my_structure" maps to "MyStructure."

Component names are not changed from the names you enter in the Project painter. Sybase recommends that you use the Java class naming conventions; for example, "MyComp."

An NVO implementation class can use any name.

Repository files

The base directory for repository files is *%DJC_HOME%\Repository*, which includes the following subdirectories:

- *IDL* – interface definitions.
- *Component* – component properties and PowerBuilder dynamic libraries (PBDs).
- *Instance* – server and data source properties.
- *Package* – package properties.

The repository files are used during deployment and at runtime.

Security roles

By default, security roles are disabled. To specify security roles:

1. In the Project painter Properties dialog, select the General tab.
2. In the Comments box, specify the security roles required for each component. In the following example, *MyComp1* and *MyComp2* are component names and "manager" is the name of the security role assigned to each:

```
roles:MyComp1="manager";roles:MyComp2="manager";
```

3 Use the Management Console to assign roles to users. See “Managing users” in Chapter 10, “Security Configuration Tasks,” in the *Security Administration and Programming Guide*.

Remote debugging

To start a remote debugging session, in the PowerBuilder debugger, select Start Remote. For details, see the PowerBuilder documentation at http://infocenter.sybase.com/help/index.jsp?topic=/com.sybase.help.pb_10.5.apptech/html/apptech/CEGBEGBH.htm.

Troubleshooting

To troubleshoot runtime problems, check the following:

- The EAServer log file *logs\serverName.log*
- The console window, if available

CHAPTER 6 Developing PowerBuilder Clients

This chapter describes how to develop PowerBuilder clients for EAServer components.

While the PowerBuilder IDE is not included with EAServer, the products are fully integrated and work well together. PowerBuilder allows you to generate non-visual objects (NVOs) that act as proxies for EAServer components. Using a proxy, you can call component methods as if they were implemented as local NVO methods. You can call any type of component from a PowerBuilder client, not just PowerBuilder NVO components.

Developing clients

To create a PowerBuilder client, use the EAServer Proxy wizard to generate PowerScript proxies for the components that the client calls. New PowerBuilder users may find it helpful to run the Template Application wizard to create some of the client-side connection logic. To run the Template Application wizard, select New | Target | Template Application.

Clients can use the PowerBuilder Connection object generated by the Template Application wizard to connect to the server, generate proxies using the EAServer Proxy wizard, then instantiate the proxies and invoke the proxy methods to call the component's business methods.

For more information, see the *Application Techniques* manual in the PowerBuilder documentation.

Component access

For clients—JavaServer Pages (JSPs), servlets, or other EJBs—running in the same application server process, you can use either EJB references or direct JNDI lookups to access components.

When you deploy PowerBuilder components, if the package name is "MyPackage" and the component name is "MyComp":

- The generated EJB home interface is MyPackage.ejb.MyCompHome.
- The generated EJB remote interface is MyPackage.ejb.MyComp.
- The JNDI name is "MyPackage/MyComp."

The PowerBuilder EJBConnection class allows you to call EJBs in EAServer and third-party application servers—see the EJBConnection class description in the PowerBuilder documentation at http://infocenter.sybase.com/help/index.jsp?topic=/com.sybase.help.pb_10.5.apptech/html/apptech/CCJBGAEBA.htm.

Other patterns for proxy instantiation Some patterns for proxy instantiation used in clients written for earlier EAServer releases are not compatible with EAServer 6.0. In particular, clients that use the CosNaming API or SessionManager::Factory::create methods that take parameters should be modified to use the implementation pattern described here. For more information, see "Using the CosNaming interface" on page 121.

Web DataWindow

The Web DataWindow is a thin-client DataWindow implementation for Web applications, which provides most of the data manipulation, presentation, and scripting capabilities of the PowerBuilder DataWindow, requiring the Web DataWindow component on a component server but no PowerBuilder DLLs on the client. The Web DataWindow supports browser-based clients and offers three rendering formats: XML, XHTML, and HTML—see the DataWindow Programmer's Guide at http://infocenter.sybase.com/help/index.jsp?topic=/com.sybase.help.pb_10.5.dwprgug/html/dwprgug/title.htm.

Note the following updates to the *DataWindow Programmer's Guide*:

❖ Configuring a Web DataWindow for generating and deploying JSP targets:

- Change to the EAServer *bin* subdirectory, and run:

```
configure web-data-window
```

❖ Instantiating a Web DataWindow from a JavaServer Faces (JSF) interface, JSP, or servlet

- To instantiate a Web DataWindow, use the following syntax, where *NNN* represents the version of the PowerBuilder VM; for example, use `HTMLGenerator105` for PBVM version 10.5:

```
import com.sybase.pb.datawindow.*;
…
InitialContext nc = new InitialContext();

HtmlGeneratorNNNHome home =
  (HtmlGeneratorNNNHome)javax.rmi.PortableRemoteObject.narrow(nc.lookup
  ("DataWindow/HTMLGeneratorNNN"), HTMLGeneratorNNNHome.class);

HTMLGenerator gen = home.create();
```

CHAPTER 7 CORBA/C++ Overview

This chapter provides an overview of things to consider when developing CORBA C++ clients and components for EAServer.

Overview

CORBA is a distributed component architecture defined by the Object Management Group (OMG). EAServer supports the CORBA Internet Inter-ORB Protocol (IIOP). EAServer also provides a CORBA-compatible C++ client-side interface. These two items allow you to create CORBA EAServer C++ applications. C++ components and clients are also interoperable with clients and components using other technologies.

The dynamic invocation interface (DII) is not supported.

For information on the CORBA architecture, see the specifications available at the OMG Web site at http://www.omg.org.

A tutorial is available
If you are new to EAServer, follow the steps in Chapter 10, "Tutorial: Creating C++ Components and Clients" to get aquainted with the C++ development and deployment cycle.

Requirements

To develop C++ components, you need a C++ development tool such as Microsoft Visual C++, for use on Windows, or the standard C++ compiler for your UNIX platform. All software that is required to run C++ components in EAServer is supplied with the EAServer product.

To develop C++ clients, you need a C++ development tool. To deploy and run C++ clients on end-user workstations, you must install the EAServer C++ client runtime on each workstation.

For detailed system requirements, see the *EAServer Installation Guide* for your platform.

Supported datatypes

EAServer follows the OMG standard for translating CORBA IDL to C++, more specifically, refer to *C++ Language Mapping Specification* (formal/99-07-41). You can download this document from the OMG Web site at http://www.omg.org.

The standard supports all the C++ features in the *Annotated C++ Reference Manual* by Ellis and Stroustrup as implemented by the ANSI/ISO C++ standardization committees. In addition, the namespace construct is supported. Templates are not required but can be used.

IDL modules are mapped to C++ namespaces and IDL interfaces are mapped to C++ classes. All OMG IDL constructs scoped to an interface are accessed through C++-scoped-names. For example, the IDL interface CtsComponents::ThreadManager maps to the C++ class CtsComponents::ThreadManager. If your C++ compiler supports namespaces, you can use the namespace directive and refer to the interface name by itself, as in:

```
using namespace CtsComponents;
...
  ThreadManager threadMan;
```

C++ mappings for predefined IDL datatypes

Table 7-1 lists the CORBA IDL types predefined in EAServer and the equivalent C++ datatypes. You can also define additional types in IDL; when you generate stubs and skeletons, these are translated to C++ types using the standard CORBA IDL to C++ type mappings. For example, The BCD and MJD CORBA IDL modules define types to represent binary data, fixed-point numeric data, dates, and times. For details, see the generated Interface Repository documentation for these IDL modules.

Table 7-1: C++ datatype mappings for predefined CORBA IDL types

CORBA IDL type	Argument mode	IDL C++ type
short	in inout out return	CORBA::Short CORBA::Short& CORBA::Short_out CORBA::Short
long	in inout out return	CORBA::Long CORBA::Long& CORBA::Long_out CORBA::Long
long long	in inout out return	CORBA::LongLong CORBA::LongLong& CORBA::LongLong_out CORBA::LongLong **Define JAG_LONGLONG** Because there is no standard C++ type for an signed 64-bit integer, you must define the JAG_LONGLONG macro as your compiler's type for a signed 64-bit integer.
float	in inout out return	CORBA::Float CORBA::Float& CORBA::Float_out CORBA::Float
double	in inout out return	CORBA::Double CORBA::Double& CORBA::Double_out CORBA::Double
boolean	in inout out return	CORBA::Boolean CORBA::Boolean& CORBA::Boolean_out CORBA::Boolean

CORBA IDL type	Argument mode	IDL C++ type
string	in inout out return	char* char*& CORBA::String_out char*
BCD::Binary	in inout out return	BCD::Binary& BCD::Binary& BCD::Binary_out BCD::Binary*
BCD::Decimal	in inout out return	BCD::Decimal& BCD::Decimal& BCD::Decimal_out BCD::Decimal*
BCD::Money	in inout out return	BCD::Money& BCD::Money& BCD::Money_out BCD::Money*
MJD::Date	in inout out return	MJD::Date& MJD::Date& MJD::Date_out MJD::Date
MJD::Time	in inout out return	MJD::Time& MJD::Time& MJD::Time_out MJD::Time
MJD::Timestamp	in inout out return	MJD::Timestamp& MJD::Timestamp& MJD::Timestamp_out MJD::Timestamp
TabularResults:: ResultSet	return	TabularResults::ResultSet*
TabularResults:: ResultSets	return	TabularResults::ResultSets*

Using mapped IDL types

All EAServer component interfaces are defined in standard CORBA IDL, and C++ stubs and skeletons use the standard CORBA IDL-to-C++ type mappings.

For local variables that map to constructed C++ types and do not represent an IDL interface, use the C++ datatype that is appended with _var. _var variables are automatically freed when they are out of scope. If you do not use the _var type, references must be freed with the C++ delete operator. In Table 7-1, string, binary, decimal, money, date, time, timestamp, ResultSet, and ResultSets have _var types. Other types listed in Table 7-1 map to fixed-length C++ types. For fixed-length types, use the base C++ type.

IDL interfaces map to C++ classes that extend the CORBA::Object class. These object reference types have a _var form for references with automatic memory management, and a _ptr form for references that must remain valid after the reference variable goes out of scope. _ptr references must be freed by calling CORBA::release.

You must pass values in a _var type as follows:

```
MyType_var v;
....
v.in()              // Passes v as an in
                      // parameter.
v.inout()           // Passes v as an inout
                      // parameter.
v.out()             // Passes v as an out
                      // parameter.
return v._retn()    // Passes v as a return value.
```

Note Do not use the C++ _out types for local variables; these types are reserved for method signatures.

For out and inout parameters of IDL type string, use CORBA::string_alloc or CORBA::string_dup to allocate memory for them. For example:

```
ItemName = CORBA::string_dup("Dummy Item Name");
ItemData = CORBA::string_dup("Dummy Item Data");
```

In C++, if you declare string variables as type CORBA::String_var, memory allocated by CORBA::string_dup or CORBA::string_alloc is freed automatically. Otherwise, declare as char * and free the memory explicitly by calling CORBA::string_free.

You can pass a null value as a parameter type only with the object reference type Module::Interface::_nil().

Overloaded methods

Overloading methods is supported for C++ components. When you overload a method, you use the same name for several methods that specify different parameters. When you call an overloaded method, the method with the corresponding parameters is executed. See "Operation declarations" on page 29 for more information.

CHAPTER 8

Developing CORBA/C++ Components

Procedure for creating C++ components

To create a CORBA/C++ compoent, you use the Management Console or a configuration script to define basic information about the component, such as the component name and methods, generate files that are required to write the component's class implementation, then compile the class into a dynamic link library (on Windows) or shared library (on UNIX).

The steps are as follows:

1. Define the component interface in CORBA IDL and deploy the IDL to the EAServer repository. Chapter 3, "Using CORBA IDL," describes how to do this.
2. Create EAServer entities to define the CORBA packages and components. The package and component properties specify the component interfaces and control interaction between EAServer and your implementation. Chapter 4, "Managing CORBA Packages and Components," describes how to do this.

3. Generate the required files by running the jaguar-compiler command on the CORBA package to generate the code and EJB wrapper components required to run the components in EAServer as described in "Generating C++ component files" on page 78.

4. Complete the C++ implementation and compile the component library. For details, see:

   - "Writing the class implementation" on page 80
   - "Compiling source files" on page 81

A tutorial is available
If you are new to EAServer, follow the steps in Chapter 10, "Tutorial: Creating C++ Components and Clients" to get aquainted with the C++ development and deployment cycle.

Generating C++ component files

Run the jaguar-compiler command on the CORBA package to generate the C++ files that you need to compile into a DLL or UNIX shared library as well as a class implementation template in which to write method logic.

You can run the jaguar-compiler command several ways:

- From the Management Console as described in "Refreshing CORBA packages in the Management Console" on page 43.
- Using a configuration script, as described in "Managing CORBA packages with configuration scripts" on page 43.
- Using the jaguar-compiler command-line tool, as described in Chapter 12, "Command Line Tools," in the *System Administration Guide*.

The generated files include sample implementation templates for the component implementation.

The generated C++ files include:

- Method skeletons file – Contains method routines that read the parameters from the network and call the method. The method skeletons also send the return status and output parameter data back to the client.

- Class header file – Contains the method declarations only. This file is an included file in the method skeletons file and the class implementation template.
- Class implementation template – Contains the class, method, and parameter declarations, as well as empty method definitions. You enter any business logic into the empty method definitions.
- Stub interface files – Contain the interface definition for all components in a package, as well as definitions for user-defined types and exceptions used in your component's interface.
- UNIX makefile – You use a makefile to compile the C++ source files into a UNIX shared library.
- Windows makefile and Microsoft Visual C++ module definition file – You use the makefile and a module definition file to compile the C++ source files into a DLL.

C++ file naming conventions and locations

For the component implementation, EAServer generates the following files:

File type	File name
method skeletons file	*package-name_component-name.cpp*
class header file	*class-name.hpp.new* Rename this file to use it as an implementation template.
class implementation template	*class-name.cpp.new* Rename this file to use it as an implementation template. If you have modified the IDL interface, merge modifications from the generated *.new* file to your existing *.cpp* file.

where:

component-name is the name of the component.

class-name is specified by the component's C++ class name property.

The component implementation files are created in the following EAServer subdirectory:

```
cpplib/package_name/component_name
```

where:

package-name is the name of the CORBA package.

component_name is the component name.

Regenerating changed C++ component methods

When you add or delete methods or modify component method prototypes, you must regenerate the method skeletons and class header files. You must manually add, delete, or modify the methods in the class implementation template. Before you regenerate the method skeletons and class header files, make sure that you have moved your modified class implementation template to another directory or renamed it so the generated class implementation template does not overwrite your existing class implementation template.

Writing the class implementation

After you generate the method skeleton file, class header file, and class implementation template, write the code for each method in the class implementation template (you can also write your class implementation from scratch and replace the generated class implementation template).

You must use scoped names to specify the CORBA IDL module, the EAServer SessionManager IDL module, and any component IDL modules that you want to execute methods on. To make using scoped names easier, you can use the C++ using statement for the IDL module namespaces as in the following example:

```
using namespace CORBA;
using namespace SessionManager;
```

If your C++ compiler does not support namespaces, define a compiler macro JAG_NO_NAMESPACE when compiling your source files.

CORBA::is_nil(Object) can be used to verify that a specific interface is implemented by a component.

As with any C++ class, you use the constructor and destructor to initialize and perform any cleanup of objects.

Constructors of class variables in file scope not called
If you declare a class variable in file scope and compile it into a shared object, such as a component, the Solaris C++ compiler doesn't call the constructor of the class variable. If the variables need to be in scope only for a particular function, procedure or module, then declare these variables in the appropriate function, procedure, module; otherwise declare these variables in the class definition.

Each C++ method signature must use the return types and parameter datatypes described in "Supported datatypes" on page 72. In the method implementation, you optionally implement the features below:

- Caching Connections to Third-Tier Database Servers

  You can use a data source to improve performance when connecting to database servers. See "Using data sources" on page 84 for more information.

- Managing explicit OTS transactions

  You can explicitly to manage OTS transactions from your component.

- Setting Transaction State

  Methods in a transactional component should call one of the transaction primitive routines to set the transaction state before returning. See "Setting transaction state" on page 96 for more information.

- Handling Errors

  Use user-defined or CORBA system exceptions to handle errors. See "Handling errors" on page 98 for more information about system and user-defined exceptions.

Compiling source files

Your C++ component code must be compiled and linked into a DLL or UNIX shared library in order to be installed into the EAServer runtime environment. When you generate source files for your component, EAServer creates an example makefile that builds the component library. You may have to edit this file to match your environment, as described in the following sections:

- "Compiling on UNIX platforms" on page 82

- "Compiling on Windows" on page 83

Compiling on UNIX platforms

EAServer generates a *make.unix* file when you generate the component skeleton as described in "Generating C++ component files" on page 78. To build your shared library, run the following command:

```
make -f make.unix
```

On Solaris, you must use a compiler and linker that is compatible with version 6.*x* compilers. The library and binary format is different between version 6.*x* and version 4.*x* compilers.

The generated UNIX make file for C++ components works on other platforms without changes. Platform-specific information is defined in the file *make.include.platform*, where *platform* is the name returned by the command:

```
uname -s
```

The *make.include.platform* includes the necessary settigngs to run the compiler and linker in the component make file. You may need to edit these settings if your compiler and linker are not installed in the standard location, or you use different software.

After building the shared library, copy it to the *cpplib* directory of your EAServer installation.

Note If you do not place the component shared library in the EAServer *cpplib* subdirectory, the directory containing the shared library must be specified in the shared library search path environment variable for your platform (for example, LD_LIBRARY_PATH for Solaris).

Compiling on Windows

For components that run on Windows, you must build a DLL that contains your C++ component methods. After building the DLL, copy it to the *cpplib* directory of your EAServer installation.

Note If you do not place the component DLL in the EAServer *cpplib* subdirectory, the directory containing the DLL must be specified in the PATH environment variable.

"Generating C++ component files" on page 78 describes how to generate C++ component files, including the makefile.

Before compiling your C++ component, verify that the makefile can find the directory containing the ODBC header files and libraries. You must set the ODBCHOME environment variable to the directory containing the ODBC header files and libraries. If you have Microsoft Visual C++ and ODBCHOME is not set, the makefile looks in *C:\msdev* (which is the default installation directory for Microsoft Visual C++) for these files.

To build your DLL, run this command from a command window in your component's source directory:

```
nmake -f make.nt
```

If you make changes to the makefile, rename it so it won't be overwritten when you regenerate the required files.

Visual C++

Visual C++ requires a module definition file that specifies which functions are exported from a DLL and some options that control how the DLL is loaded into memory. Module definition files end with the extension *.def*.

For most projects, you can use the generated module definition file as is. In some cases, you may want to edit settings other than those in the EXPORTS section. For example, your component may perform better with a smaller or larger HEAPSIZE setting.

Note Do not edit the generated function names in the EXPORTS section of the *.def* file for a C++ component. If you do, the EAServer dispatcher will not be able to call your methods.

Using data sources

C++ components can call the C Connection Manager routines to take advantage of connection caching. These routines manage ODBC, Client-Library, and Oracle Call Interface (OCI) data sources.

EAServer C routines are documented in Chapter 2, "C Routines Reference," in the *EAServer API Reference*. The Connection Manager routines have names that begin with JagCm.

Using ODBC data sources

Header files

The header file *jagpublic.h* declares the Connection Manager routines and data structures; the file is located in the *include* subdirectory of your EAServer installation.

Include required ODBC header files before including *jagpublic.h*, for example:

```
#include <sql.h>
#include <sqlext.h>
#include <jagpublic.h>
```

Data structures

Most Connection Manager routines require the address of a CM_CACHE handle as a parameter. The CM_CACHE handle allows your code to refer to a specific data source that is defined in Management Console. The JagCmGetCachebyName routine returns a CM_CACHE handle to the named data source. JagCmGetCachebyUser creates a temporary data source using the specified parameters and returns its CM_CACHE handle.

ODBC uses an HDBC structure to represent a database connection. The JagCmGetConnection routine returns the address of an HDBC structure.

ODBC example

The following example demonstrates program logic that offers improved performance when a matching data source is available and that still functions when no matching data source has been configured. The example first calls JagCmGetCachebyUser to obtain a CM_CACHE handle to a temporary ODBC data source using the values: user name ("myrtle"), password ("secret"), and server name ("tsingtao"). The code sets the *cache* variable to the CM_CACHE handle.

The example then calls JagCmGetConnection, passing the *cache* value as set by JagCmGetCachebyUser, and passing explicit values for the user name, server name, password, and connectivity library. If the *cache* variable contains a valid data source reference, JagCmGetConnection looks directly in the data source for an available connection. If *cache* was set to NULL or the indicated data source has no available connections, JagCmGetConnection creates and opens a new connection.

Code that follows the implementation strategy illustrated here can achieve better performance when there are many configured data sources. Passing the CM_CACHE handle explicitly in JagCmGetConnection eliminates repeated internal table searches.

```
/* ODBC includes */
#include <sql.h>
#include <sqlext.h>

/* Connection Manager includes */
#include <jagpublic.h>

SQLRETURN ret;          /* Return code catcher    */
SQLHDBC *hdbc;          /* ODBC connection handle */
JagCmCache cache;       /* Cache handle           */

/*
** Retrieve a CM_CACHE handle
*/
cache = NULL;
ret = JagCmGetCachebyUser (“myrtle”,“secret”,“tsingtao”,“ODBC”,&cache);

/*
** Ignore the return value. If the call fails, cache is NULL and we can keep
** going.
*/

/*
** Get a connection. If we have a cache handle, use it to get the connection.
```

```
** Otherwise, create a new connection.
*/
ret = JagCmGetConnection (&cache,“myrtle”,“secret”,“tsingtao”,“ODBC”,
   (SQLPOINTER *)&hdbc, JAG_CM_FORCE);

if (ret != SQL_SUCCESS)
{
   ... log the error ...
}

... code that uses the connection goes here ...

ret = JagCmReleaseConnection (&cache, “myrtle”, “secret”, “tsingtao”, “ODBC”,
   hdbc, JAG_CM_UNUSED);

if (ret != SQL_SUCCESS)
{
   ... log the error ...
}
```

You can call JagCmGetCachebyName rather than JagCmGetCachebyUser. For an example, see the reference page for JagCmGetCachebyName in Chapter 5 of the *EAServer API Reference*.

Client-Library data sources

EAServer 6.0 includes a version of Open Client 12.5 adapted to run in EAServer. This version supports high availability, failover, and wide-table features (varchar/varbinary columns more than 255 bytes long and tables with more than 255 columns). You can use Open Client 12.5 only when you are connected to Adaptive Server® Enterprise version 12.5 or later.

Header files

Before including *jagpublic.h*, you must include the Client-Library *ctpublic.c* header file, as in the example below:

```
#include <ctpublic.h>
#include <jagpublic.h>
```

Data structures

Most Connection Manager routines require the address of a CM_CACHE handle as a parameter. The cache handle allows your code to refer to a specific data source that is defined in the Management Console. The routines JagCmGetCachebyName and JagCmGetCachebyUser retrieve CM_CACHE handles.

Client-Library uses a CS_CONNECTION structure to represent a database connection. The JagCmGetConnection routine returns the address of a CS_CONNECTION structure.

Client-Library example

The following example calls JagCmGetConnection to obtain a connection that has a user name of "myrtle," the password "secret," connects to the server "tsingtao," and uses Client-Library:

```
   #include <ctpublic.h>
   #include <jagpublic.h>

   CS_RETCODE ret;
   CS_CONNECTION *connection;
   JagCmCache cache;

   /*
   ** Get a connection.
   */
   cache = NULL;
   ret = JagCmGetConnection (&cache, "myrtle", "secret", "tsingtao",
      "CTLIB",(SQLPOINTER *)&connection, JAG_CM_FORCE);

   if (ret != CS_SUCCEED)
   {
   ... log the error ...
   }

... code that uses the connection goes here ...

   ret = JagCmReleaseConnection (&cache, "myrtle", "secret", "tsingtao",
      "CTLIB", (SQLPOINTER)connection, JAG_CM_UNUSED);

if (ret != CS_SUCCEED)
{
   ... log the error ...
}
```

In the example, the call to JagCmGetConnection looks for a data source that includes matching values for the user name ("myrtle"), password ("secret"), and server name ("tsingtao") and that uses Client-Library. The last parameter value, JAG_CM_FORCE, indicates that the call should open a new connection if no cached connection is available. JagCmReleaseConnection releases control of the connection: a connection that was taken from a cache is returned to that cache; an uncached connection is closed and deallocated.

Note that JagCmGetConnection attempts to open a connection even when no matching data source is configured. In this case, JagCmGetConnection attempts to create a new connection using the specified values.

In this example, JagCmGetConnection and JagCmReleaseConnection return Client-Library return codes since both calls use "CTLIB" as the connection library parameter.

Note Beginning in EAServer 6.0, you can use CTLIB as the connection library for Open Client 11.0, 12.0, and 12.5 connections. Version-specific CTLIB_*x* connection libraries are still provided for backward compatibility.

You can call JagCmGetCachebyName rather than JagCmGetCachebyUser. To see an example, see the reference page for JagCmGetCachebyName in the *EAServer API Reference*.

Client-Library error and message callbacks

EAServer installs default server message and client message callbacks into cached Client-Library connections. The default callbacks write error and message information to the server's log file.

When using Client-Library connections, you can install your own server message and client message callbacks into connections retrieved from JagCmGetConnection. JagCmReleaseConnection reinstalls the default callbacks before placing connections back into the cache.

Oracle OCI data sources

You can define data sources for an Oracle 9i or 10g database, and use OCI as the connection library for both database versions. The OCI_9 and OCI_10 connection libraries are still provided for backward compatibility.

Oracle autocommit setting

EAServer creates Oracle connections with the default autocommit setting, autocommit off. In non-transactional components, you must explicitly issue a commit command to commit update and insert queries. In transactional components, the EAServer transaction manager issues commit and rollback commands for connections used by the components that participate in an EAServer transaction.

Note In a non-transactional component, if you do not explicitly call commit or rollback after sending Oracle commands, the commands may be committed when a transactional component uses the same connection. EAServer issues a commit to clear the connection status before passing Oracle connections to a transactional component.

Header files

Include *oci.h* before *jagpublic.h*, as in the example below:

```
#include <oci.h>
#include <jagpublic.h>
```

Data structures

Most Connection Manager routines require the address of a CM_CACHE handle as a parameter. The routines JagCmGetCachebyName and JagCmGetCachebyUser retrieve CM_CACHE handles.

OCI uses an OCISvcCtx structure to represent a database connection. The JagCmGetConnection routine returns the address of a OCISvcCtx structure.

OCI example

The example below retrieves an OCI connection, executes a statement using the connection, then returns the connection to the cache.

```
#include <jagpublic.h>
#include <oci.h>
#define   USERID  "system"
#define   PASSWD  “manager"
#define   DATASOURCE "OCITEST"

 JagCmCache  cache;
```

```
OCIEnv *envhp;
OCISvcCtx **svcpp, *svchp;
OCIError *errhp;
OCIStmt *stmthp;
sword ociret;

    /* Connect to ORACLE. */
   cache = NULL;
   ociret = JagCmGetConnection(&cache, USERID, PASSWD, DATASOURCE, "OCI",
                               (void*)&svchp, JAG_CM_FORCE);
  ...

    /* Initialize an Env, to allocate stmt and error handles */
  OCIEnvInit( &envhp, OCI_DEFAULT, (size_t) 0, (dvoid **)0 );
  OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp,
                  OCI_HTYPE_ERROR, (size_t) 0, (dvoid **) 0);
  OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &stmthp,
                  OCI_HTYPE_STMT, (size_t) 0, (dvoid **) 0));
  checkerr(errhp, OCIStmtPrepare(stmthp, errhp, sql_statement,
                                (ub4) strlen((char *) sql_statement),
                                (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT));

    /* execute using the service context */
   checkerr(errhp, OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                                 (CONST OCISnapshot *) NULL,
                                 (OCISnapshot *) NULL, OCI_DEFAULT));
  .....

    /* free handles */
   OCIHandleFree(stmthp, OCI_HTYPE_STMT);
   OCIHandleFree(errhp, OCI_HTYPE_ERROR);

    /* release connection */
    ret = JagCmReleaseConnection(&cache, USERID, PASSWD, DATASOURCE, "OCI",
                                  svchp, JAG_CM_UNUSED);
```

Managing explicit OTS transactions

You can code components (and clients) to initiate and complete transactions using the OTS (Object Transaction Service) CosTransactions::Current or CosTransactions::TransactionFactory interfaces.

Note In order to use OTS, you must enable EAServer to use the OTS/XA transaction coordinator. See Chapter 3, “Creating and Configuring Servers,” in the *EAServer System Administration Guide* for more information.

To use the functionality of these interfaces, include CosTransactions.hpp in your source file.

To explicitly use transactions in a component or client, use the CosTransactions::Current interface to perform these tasks.

Task	Call this method	Catch these exceptions
Start a transaction.	begin	SubtransactionsUnavailable
Temporarily stop a transaction.	suspend	None
Resume a suspended transaction.	resume	InvalidControl
Commit a transaction.	commit	NoTransaction, HeuristicMixed, HeuristicHazard
Roll back a transaction.	rollback	NoTransaction
Make the only possible outcome of the transaction a rollback.	rollback_only	NoTransaction
Roll back a transaction after a specified amount of time has elapsed without any response.	set_timeout	None
Retrieve a transaction’s status.	get_status	None
Retrieve a transaction’s name. Use this method when you need to debug transactions.	get_transaction_name	None

---

Using factories
The TransactionFactory interface is included in EAServer only to maintain compatibility with the CORBA OTS specification—Sybase recommends that you use the CosTransactions::Current interface to create explicit transactions.

Note Sybase recommends that you use suspend with caution so as not to conflict with the EAServer component model. For example, do not use suspend to take control of a transaction that it does not control.

Initializing the ORB

To initialize the ORB and retrieve a reference to the CosTransactions::Current interface, specify the TransactionCurrent *ObjectId*, which identifies the CosTransactions::Current interface, to the resolve_initial_references method, and narrow it (using the _narrow method) to the CosTransactions::Current interface. Use the is_nil method to verify that the reference to the CosTransactions::Current interface is valid.

For clients

The following code fragment shows how to initialize the ORB from a client. ORB_init must take the *argumentList* array that specifies the ORBNameServiceURL parameter. You can also set the ORBNameServiceURL using the JAG_NAMESERVICEURL environment variable.

```
int  argumentCnt = 1;
char *argumentList[] = {
   { "-ORBNameServiceURL iiop://<hostnamehere>:2000" },
   { "" }
   };


try {

  CORBA::ORB_var  orb = CORBA::ORB_init(argumentCnt,
    argumentList, 0);
  cerr << "Orb init" << endl;

  CORBA::Object_var  crntObj =
    orb->resolve_initial_references
      ("TransactionCurrent");
  CosTransactions::Current_var CurrentIntf =
    CosTransactions::Current::_narrow(crntObj);
```

```
  if( CORBA::is_nil(CurrentIntf) )
  {
    cerr << "Error getting Current" << endl;
    exit(-1);
  }
cerr << "Got Current" << endl;
```

For components

The following code fragment shows how to initialize the ORB from a component. ORB_init does not need to take any parameters.

```
orb = CORBA::ORB_init(argumentCnt, NULL, 0);
cerr << "Orb init" << endl;

CORBA::Object_var  crntObj =
  orb->resolve_initial_references
    ("TransactionCurrent");
CurrentIntf =
  CosTransactions::Current::_narrow(crntObj);
if( CORBA::is_nil(CurrentIntf) )
{
  cerr << "Error getting Current" << endl;
  /* could be due to:
  **   1. Component not BeanManaged/OTS Style
  **   2. Already in a Txn
  ** 3. not running under OTS
  */
  return CS_FAIL;
}
cerr << "Got Current" << endl;
```

Calling CosTransactions::Current interface methods

After retrieving a reference to the CosTransactions::Current interface, you can call any of the CosTransactions::Current methods on the CosTransactions::Current reference. After executing the begin method, execute the database operations you want to include in the transaction. Depending on whether the database operations succeed or fail, you can execute other appropriate methods, such as commit, rollback, or rollback_only. This code fragment shows how to begin a transaction and commit or roll it back depending on the return codes received from the databases.

```
CurrentIntf->begin();
ret = JagCmGetConnection( &cache,
    (SQLCHAR *) USERID, (SQLCHAR *) PASSWD,
    (SQLCHAR *) xaresource, (SQLCHAR *) "CTLIB_110",
```

```
        (void*) &conn, JAG_CM_UNUSED );

    if (ret != CS_SUCCEED) {
      cerr << "Error getting connection" << endl;
      CurrentInt->rollback();
    }

    CurrentIntf->commit(CS_FALSE);
```

Executing tasks outside of a transaction

To execute a method outside of a transaction, you can write the code to perform either:

- Execute the method before beginning a transaction, or
- Temporarily stop and start execution of the transaction.

❖ Execute tasks outside of a transaction using the *suspend* and *resume* methods

1. Execute suspend to temporarily stop execution of the transaction.
2. Execute the tasks.
3. Execute resume to restart the execution of the transaction from where it stopped.

This code fragment shows how to execute tasks outside of a transaction. The suspend method returns the control context. You specify the control context when you use the resume method to restart the transaction. Catch the InvalidControl exception, which may be raised when a control context is out of scope (and not null).

```
sus_ctrl = CurrentIntf->suspend();

/* The following method is not in the transaction */
component1->method2();

CurrentIntf->resume(sus_ctrl);
/* The following methods are invoked
in the transaction */
    component2->method1();

CurrentIntf->commit(CS_FALSE);

}
```

```
catch(CosTransactions::SubtransactionsUnavailable
  &ex )
{
    cerr << "Exception: SubTxnUnavailable " <<
      ex._jagExceptionCode << endl;
}
catch(CosTransactions::NoTransaction &ex )
{
    cerr << "Exception: NoTransaction  " <<
      ex._jagExceptionCode << endl;
}
catch(CosTransactions::InvalidControl &ex )
{
    cerr << "Exception: InvalidCtrol  " <<
      ex._jagExceptionCode << endl;
}
catch(...)
{
    cerr << "Caught Unexpected exception" << endl;
    exit(-1);
}
```

Exceptions

The CosTransactions module includes these exceptions:

- SubtransactionsUnavailable – raised when the client thread already has an associated transaction and the transaction coordinator does not support nested transactions.
- NoTransaction – raised when there is no transaction associated with the client thread.
- InvalidControl – raised when the specified control is not null and not within the scope of the client thread.
- Inactive – raised when a method such as rollback_only is executed on a transaction has already been prepared.
- InvalidTransaction – raised when a request carries an invalid transaction context, such as if an error occurred when registering a resource.
- TransactionRequired – raised when a request carries a null transaction context but required an active transaction. For example, this could occur when a component specifies the Mandatory attribute.

- Unavailable – raised when the requested object cannot be returned because OTS/XA transaction coordinator restricts the availability of the object.
- TransactionRolledBack – raised when a transaction is marked to roll back or has already been rolled back.

Heuristic exceptions

A heuristic decision is a decision to commit or roll back updates that one or more participants in a transaction make without waiting for the consensus decision from the transaction coordinator. These types of commits and rollbacks are also called heuristic commits and heuristic rollbacks. When a heuristic commit or rollback is made, the transaction can become inconsistent. Therefore, a heuristic commit or rollback is made only in unusual circumstances such as communication failures. When the System Administrator issues a heuristic commit or rollback, a heuristic exception is raised.

- HeuristicMixed – Raised when a heuristic decision is made and some relevant updates are committed and others are rolled back.
- HeuristicHazard – Raised when a heuristic decision may have been made, when not all of the conditions of all relevant updates is known, and for those updates whose condition is known, either all of them were committed or rolled back.
- HeuristicRollback – Raised when a heuristic decision to roll back all of a transaction’s relevant updates has been made.
- HeuristicCommit – Raised when a heuristic decision to commit all of a transaction’s relevant updates has been made.

Setting transaction state

Methods in a transactional component should call one of the transaction state primitive routines listed in Chapter 2, “C Routines Reference,” of the *EAServer API Reference*.

Even if your component is not transactional, you can call one of these methods to explicitly specify whether the instance should be deactivated.

For transactional components, choose the routine that reflects the state of the work that the component is contributing to the transaction, as follows:

- If the work is complete and without error, call JagCompleteWork.

- If the work is not necessarily finished, but not in error, call JagContinueWork.
- If the work is not finished and not ready for commit, call JagDisallowCommit.
- If the work cannot be completed, call JagRollbackWork (you should also log a description of the error and send an error to the client, as described in “Handling errors” on page 98).

For nontransactional components, call either JagCompleteWork or JagRollbackWork to deactivate and destroy the component instance. To keep the instance active, call JagContinueWork or JagDisallowCommit.

If a method does not explicitly set transaction state before returning, the default behavior is JagContinueWork.

Issuing intercomponent calls

To invoke other components, instantiate a stub for the second component, then use the stub to invoke methods on the component.

You must use a stub to issue intercomponent calls. If you call methods in another C++ component directly, EAServer features such as transactions and security will not work.

To invoke methods in other components, create an ORB instance to obtain object references to other components and invoke methods on the object references. You obtain object references for other components on the same server by invoking string_to_object with the IOR string specified as *Package*/*Component*. For example:

```
CORBA::Object_var obj =
    orb->string_to_object("MyPackage/MyComponent");
MyModule::MyInterface_var i =
   MyModule::MyInterface::_narrow(obj);
```

When making intercomponent calls using string_to_object, the user name of the client that executed the component is automatically used for authorization checking. string_to_object returns an instance running on the same server if the component is locally installed; otherwise, it attempts to resolve a remote instance using the naming server.

To components on a non-EAServer ORB

Your component may need to invoke methods on a component hosted by another vendor's CORBA server-side ORB. Sybase recommends that C++ components use the EAServer client-side ORB for all IIOP connections made from EAServer components. See "Connecting to third-party ORBs using the EAServer ORB" on page 123 for more information.

Handling errors

Handle errors by:

1. Writing detailed error descriptions to the server log file using the JagLog C routine.
2. Coding one of these tasks:
   a. If the component is transactional, call JagDisallowCommit or JagRollbackWork (or you can throw the CORBA::TRANSACTION_ROLLEDBACK exception instead of calling JagRollbackWork).
   b. Throw a CORBA system or user-defined IDL exception to be raised by the client stub. See "Handling exceptions" on page 118 for more information.

For more information about these methods, see Chapter 2, "C Routines Reference," in the *EAServer API Reference*.

Debugging C++ components

To debug a component you must run the debug version of the server, and use a debugger running on the same host as EAServer. Chapter 3, "Creating and Configuring Servers," in the *EAServer System Administration Guide* describes how to start the debug server.

To debug a component from Microsoft Visual C++, you must set the component's C++ Debug (com.sybase.jaguar.component.cpp.debug) property under the Advanced tab to `true`.

Follow these steps to attach to the server and step into your component code:

1. Configure the component properties and verify the CPP Debug property is enabled (or set to true). See "CORBA component property descriptions" on page 45.
2. Start your C++ debugger and configure it to launch EAServer using the server-start script.
3. Set a breakpoint on the function jag_dbg_stop. This function executes every time the server loads a component DLL. The jag_dbg_stop prototype is:

   ```
   void jag_dbg_stop(char *compName)
   ```

   The *compName* parameter specifies the name of the library or shared library that was just started. Several components may be started before yours. In the debugger, display the *compName* value when the jag_dbg_stop breakpoint is tripped, and monitor the value to determine when your component is started. Breakpoints on jag_dbg_stop are triggered before the server calls the component's create method.

   Note Make sure the jag_dbg_stop breakpoint is set before running your client application.

   ---

4. When your component's DLL is started, you can specify the component's C++ function names as breakpoints and step into the method's code when it is invoked.
5. When you finish debugging, reconfigure the component properties and verify the CPP Debug property is disabled (or set to true). See "CORBA component property descriptions" on page 45.

CHAPTER 9 Developing CORBA/C++ Clients

Procedure for creating CORBA C++ clients

A CORBA C++ client establishes a connection and session with the EAServer ORB, instantiates a proxy object for the component, and calls methods in the proxy object. When the client calls the methods in the proxy objects, the proxy object methods communicate across the network and execute the corresponding methods in the components.

To create CORBA EAServer C++ clients:

1. Generate C++ header files and CORBA stub implementations for the IDL modules used in the component implementation. See “Generating stubs” on page 102.
2. Implement the C++ client logic. See “Writing CORBA C++ clients” on page 102.
3. Compile the C++ source files as described in “Compiling C++ clients” on page 120.

Generating stubs

The EAServer ORB implementation class requires stub header files in order to invoke component methods. You must include stub header files in your client source files. The stub header files contain as inline all the component functions, which make calls to the C functions in *libjcc.dll*. Inline functions allow EAServer to support multiple C++ compilers without having to include separate link libraries for each compiler.

For CORBA/C++ components, EAServer generates C++ stub header files for deployed C++ CORBA components when you run the jaguar-compiler command—see “Generating C++ component files” on page 78. To generate C++ stubs for components of other types, use the idl-compiler command-line tool. For example:

```
%DJC_HOME%\bin\idl-compiler.bat -v Tutorial\CPPArithmetic.idl -f
%DJC_HOME%\include -cpp
```

For information on idl-compiler syntax, see Chapter 12, “Command Line Tools,” in the *System Administration Guide*.

If you are using another C++ ORB implementation to connect to EAServer, you must export IDL and use the vendor’s IDL compiler to generate stubs that are compatible with that ORB implementation. “Using CORBA ORB implementations other than EAServer” on page 121 describes how to export IDL files for EAServer components.

Writing CORBA C++ clients

These section describes how to code a CORBA C++ client that invokes component methods:

- “Adding required include and namespace declarations” on page 103
- “Instantiating component proxies” on page 104
- “Invoking methods” on page 110
- “Handling exceptions” on page 118

Adding required include and namespace declarations

You must include stub header files for all IDL modules that include interfaces that the component implements. In addition to the stub header files, you must also include *SessionManager.hpp* (which contains the classes and functions that allow a C++ client to create and destroy sessions) in the client source file.

You can also include these optional header files:

- *TabularResults.hpp* – contains the classes and functions that allow C++ clients to receive result sets from components.
- *BCD.hpp* – contains the mappings for binary and arbitrary precision floating point-decimal datatypes.
- *MJD.hpp* – contains the datatype mappings from CORBA to C++ for Modified Astronomical Julian Date (M.J.D.) dates and times.

Note *TabularResults.hpp* already includes *BCD.hpp* and *MJD.hpp*; if you include *TabularResults.hpp*, you do not have to include *BCD.hpp* and *MJD.hpp.*

You must use scoped names to the CORBA IDL module, the EAServer SessionManager IDL module, and any component IDL modules that you want to execute methods on. To make using scoped names easier, you can use the C++ using statement for the IDL module namespaces as in the following example:

```
using namespace CORBA;
using namespace SessionManager;
```

If your C++ compiler does not support namespaces, define the compiler macro JAG_NO_NAMESPACE when compiling your source files.

When you create an object, identify the object reference by appending *_var* to the object name. The *ObjectName_var* reference will be automatically released when it is deallocated or assigned a new object reference.

CORBA::is_nil(Object) can be used to verify that a specific interface is implemented by a component. For an example, see “Creating a Manager instance” on page 108.

If you are returning result sets from components, you should also specify the TabularResults EAServer IDL module with the using statement.

Instantiating component proxies

Before invoking methods on component instances, the client must connect to a server and instantiate the components. Your code must perform these steps to create proxy instances:

Step	What it does	Detailed explanation
1	Initialize the CORBA ORB and create an ORB reference.	"Configure and initialize the ORB runtime" on page 104
2	Use the ORB reference to create a Manager instance.	"Creating a Manager instance" on page 108
3	Use the Manager instance to create a Session.	"Creating sessions" on page 109
4	Use the Session instance to create stub component instances.	"Creating stub instances" on page 109
5	Call the stub methods to remotely invoke component methods.	"Invoking methods" on page 110

Other patterns for proxy instantiation
Some patterns for proxy instantiation used in clients written for earlier EAServer releases are not compatible with EAServer 6.0. In particular, clients that use the CosNaming API or SessionManager::Factory::create methods that take parameters should be modified to use the implementation pattern described here. For more information, see "Using the CosNaming interface" on page 121.

Configure and initialize the ORB runtime

Before you can use any ORB classes, you must call the ORB_init method, which:

- Returns an object reference to the ORB.
- Allows you to pass initialization parameters to the driver class in the form of a string array. You can also set an environment variable (in the System Properties for your machine) for each initialization parameter. If the environment variable and initialization parameter are set, the value of the initialization parameter is used. You can set any initialization parameter to a value of *none*, which overrides the value of the environment variable and sets the value to the default, if any.

You can pass the following initialization parameters to the driver class:

- ORBHttp – this specifies whether the ORB should use HTTP-tunnelling to connect to the server. A setting of of "true" specifies HTTP tunnelling. The default is "false". This parameter can also be set in an environment variable, JAG_HTTP. Some firewalls may not allow IIOP packets through, but most all allow HTTP packets through. When connecting through such firewalls, set this property to "true".
- ORBHttpExtraHeader – An optional setting to specify what extra information is appended to the header of each HTTP packet when connecting through a Web proxy. See Chapter 9, "Deploying Applications Around Proxies and Firewalls," in the *EAServer Security Administration and Programming Guide* for more information.
- ORBHttpUsePost – when using HTTP tunnelling, specifies the HTTP request type used. A value of true indicates that POST requests are to be used. A value of false (the default) specifies that GET requests are to be used. This parameter can also be set in an environment variable, JAG_HTTPUSEPOST.
- ORBLogIIOP – this specifies whether the ORB should log IIOP protocol trace information. A setting of "true" enables logging. The default is "false". This parameter can also be set in an environment variable, JAG_LOGIIOP. When this parameter is enabled, you must set the ORBLogFile option (or the corresponding environment variable) to specify the file where protocol log information is written.
- ORBLogFile – this sets the path and name of the file to which to log client execution status and error messages. This parameter can also be set in an environment variable, JAG_LOGFILE. The default setting is *no log*.
- ORBCodeSet – this sets the code set that the client uses. This parameter can also be set in an environment variable, JAG_CODESET. The default setting is *iso_1*.
- ORBRetryCount – specify the number of times to retry when the initial attempt to connect to the server fails. This parameter can also be set in an environment variable, JAG_RETRYCOUNT. The default is 5.
- ORBRetryDelay – specify the delay, in milliseconds, between retry attempts when the initial attempt to connect to the server fails.This parameter can also be set in an environment variable, JAG_RETRYDELAY. The default is 2000.

- ORBProxyHost – specifies the machine name or the IP address of an reverse proxy server. See Chapter 9, "Deploying Applications Around Proxies and Firewalls," in the *EAServer Security Administration and Programming Guide* for more information.
- ORBProxyPort – specifies the port number of a reverse proxy server.
- ORBforceSSL – force an SSL connection to a reverse proxy server (indicated by the ORBProxyHost and ORBProxyPort properties). Set this property to true if the connection to the reverse proxy must use SSL (HTTPS) tunnelling, but the connection from the proxy to the server does not use SSL tunnelling.
- ORBsocketReuseLimit – specifies the number of times that a network connection may be reused to call methods from one server. The default is 0, which indicates no limit. The default is ideal for short-lived clients. The default may not be appropriate for a long-running client program that calls many methods from servers in a cluster. If sockets are reused indefinitely, the client may build an affinity for servers that it has already connected to rather than randomly distributing its server-side processing load among all the servers in the cluster. In these cases, the property should be tuned to best balance client performance against cluster load distribution. In Sybase testing, a setting of 10 to 30 proved to be a good starting point. If the reuse limit is too low, client performance degrades.
- ORBIdleConnectionTimeout – specifies the time, in seconds, that a connection is allowed to sit idle. When the timeout expires, the ORB closes the connection. The default is 0, which specifies that connections can never timeout. The connection timeout does not affect the life of proxy instance references; the ORB may close and reopen connections transparently between proxy method calls. Specifying a finite timeout for your client applications can improve server performance. If many instances of the client run simultaneously, a finite client connection timeout limits the number of server connections that are devoted to idle clients. A finite timeout also allows rebalancing of server load in an application that uses a cluster of servers.
- ORBWebProxyHost – the host name or IP address of an HTTP proxy server that supports generic Web tunnelling, sometimes called connect-based tunnelling. There is no default for this property, and you must specify both the host name and port number properties. See Chapter 9, "Deploying Applications Around Proxies and Firewalls," in the *EAServer Security Administration and Programming Guide* for more information. You can also specify the property by setting the environment variable JAG_WEBPROXYHOST.

- ORBWebProxyPort – when generic Web tunnelling is enabled by setting `ORBWebProxyHost`, this property specifies the port number at which the HTTP proxy server accepts connections. There is no default for this property, and you must specify both a host name and port. See Chapter 9, "Deploying Applications Around Proxies and Firewalls," in the *EAServer Security Administration and Programming Guide* for more information. You can also specify the property by setting the environment variable JAG_WEBPROXYPORT.
- ORBHttpExtraHeader – an optional setting to specify what extra information is appended to the header of each HTTP packet sent to a proxy server (specified with the `ORBWebProxyHost` parameter). You can also specify the property by setting the property JAG_HTTPEXTRAHEADER. See Chapter 9, "Deploying Applications Around Proxies and Firewalls," in the *EAServer Security Administration and Programming Guide* for more information.

You can pass additional properties to configure secure (IIOPS) connections. See Chapter 5, "Using SSL in C++ Clients," in the *EAServer Security Administration and Programming* guide for more information.

Example: ORB initialization

ORB initialization is demonstrated in this example. You can specify the ORB options as a command line parameters to be passed to the ORB_init method.

```
#include <stdio.h>
#include <iostream.h>
#include <string.h>
#include <SessionManager.hpp>
#include <Jaguar.hpp>
#include <Tutorial.hpp>    // Stubs for interfaces in
Tutorial IDL               // module.

int main(int argc, char** argv)
{
const char *usage =
  "Usage:\n\tarith -ORBNameServiceURL iiop://
  <host>:<iiop-port>/<initial-context>\n";
const char *tutorial_help =
  "Verify that the"
  "Tutorial/CPPArithmetic component exists "
  "and that it implements the "
  "Tutorial::CPPArithmetic IDL interface.";

const char *ior_prefix = "iiop://";
```

```
const char *component_name = "Tutorial/CPPArithmetic";
char *ior = NULL;

try {

cout << "Creating Jaguar session\n\n";

// Initialize the ORB
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv, 0);
```

Creating a Manager instance

The SessionManager::Manager interface is used for client authentication for EAServer connections. To create a Manager instance, you must identify the server by using an IIOP or IIOPS URL to connect to the server.

The server's IIOP port is configured using listeners. In the default configuration, the IIOP port number is 2000. For more information, see Chapter 3, "Creating and Configuring Servers," in the *System Administration Guide*.

Once the client has obtained the server's IOR or URL string, it calls the ORB::string_to_object method to convert the IOR or URL string into a Manager instance, as shown in the following example. You use the Manager::_narrow method to return a new object reference for the existing object, which is the IOR object.

```
  ...
  Object_var object = orb->string_to_object
("iiop://myhost:2000");
  Manager_var manager = Manager::_narrow (object);
  if (is_nil(manager)) {
    cout << "Error: Null SessionManager::Manager
instance. Exiting.";
    return -1;
  }...
```

string_to_object returns an object reference as object. For each reference, the _var form is used because the object will be automatically released when it is deallocated or assigned a new object reference. _narrow converts *object* into object reference for Manager.

_narrow returns a nil object reference if the component does not implement the interface. is_nil(manager) verifies that the SessionManager::Manager interface is implemented and returns an error if the interface is not implemented.

Creating sessions

The SessionManager::Session interface represents an authenticated session between the client application and a server. The Manager::createSession method accepts a user name and password and returns a Session_var object, session, as shown in the example below:

```
...
Session_var session =
  manager.createSession(userName, password);
...
```

Creating stub instances

You call the Session::lookup method to return a factory for proxy object references. The signature of Session::lookup is:

```
SessionManager::Factory_var lookup("name")
```

Session::lookup takes a string that specifies the name of the component to instantiate. A component's default name is the EAServer package name and the component name, separated by a slash as in *calculator/calc*. However, a different name can be specified by changing the name binding properties for EJB components. For example, you can specify a logical name, such as *USA/MyCompany/FinanceServer/Payroll*. For more information on configuring the naming service, see Chapter 5, "Naming Services," in the *EAServer System Administration Guide*.

Session::lookup returns a factory for component proxies. Call the Factory::create method to obtain proxies for the component. This method returns a org.omg.CORBA.Object reference. Call _narrow to convert the object reference into an instance of the stub class for the component.

The code to call Session::factory and Factory::create looks like this:

```
  ...
  // In this example, the component is named
  // Repository and is installed in
  // the EAServer package.

Object_var obj = session->lookup("Jaguar/Repository");
SessionManager::Factory_var repoFactory =
SessionManager::Factory::_narrow(obj);

obj = repoFactory->create();
Jaguar::Repository_var repository =
  Jaguar::Repository::_narrow(obj);
```

```
// Verify that we really have an instance.
if (CORBA::is_nil(repository))
{
  cout << "ERROR: Null instance for component.";
}
```

Calling Session.lookup in server code

When called from server code, Session::lookup resolves the component name by calling the name service, which gives preference to a local component instance if the component is installed on the same server. However, the use of a locally installed component is not guaranteed. To ensure that a local implementation is used, specify the name as `local:`***`package`***`/`***`component`***, where *package* is the package name and *component* is the component name, for example, `local:CtsSecurity/SessionInfo`. When you specify the local: prefix, the lookup call bypasses the name service and returns a local instance if the component is installed in the same server. The call fails if the specified component is not installed in the same server..

Invoking methods

After instantiating the stub class, use the stub class instance to invoke the component's methods. The stub class has methods that correspond to each method in the component. Parameter datatypes are mapped as described in Table 7-1 on page 73. Any parameter datatype can be used as a return type; in addition, user-defined IDL datatypes can be used as return, in, inout, or out parameters.

Processing result sets

To retrieve and process a single result set from a component:

1. Call the component method on the stub instance that returns a result set.
2. Iterate through each row and then each column in a row by using nested for loops.
3. Use the discriminator method (_d) to retrieve the datatype of the column in a row and switch/case syntax to process the column values (such as printing the column values).

To retrieve and process multiple result sets returned from a component method as a TabularResults::ResultSets object:

1. Call the component method on the component reference that returns the result sets.
2. Retrieve the length or number of result sets.
3. Iterate through the result sets using a for loop.

   For each result set, iterate through each row and then each column in a row by using nested for loops.

   You can treat a ResultSets object as an array of ResultSet objects. On each iteration, retrieve a reference to each ResultSet object by using the subscript [ ] operator.
4. Use the discriminator method (_d) to retrieve the datatype of the column in a row and switch/case syntax to process the column values (such as printing the column values).

Example of processing result sets

This example retrieves a single result set. The following code shows the C++ client in its entirety. For detailed explanations, see the sections that explain each result-set processing step.

All of the required header files are included. The IDL module namespaces are specified with the C++ using statement. The printResultSet() method contains the logic for processing a result set. main() contains the logic to initialize and connect to the EAServer ORB, instantiate the stub, call the component method to retrieve the result set object, and call printResultSet() to process the result set.

After the result set has been processed, execution of printResultSet() ends and control is returned to main(). In main(), the screen is kept open with the fprintf statement. Once you press Return, execution ends.

```
#include <stdio.h>
#include <time.h>
#include <iostream.h>
#include <SessionManager.hpp>
#include <TabularResults.hpp>
#include <Test.hpp>
using namespace CORBA;
using namespace SessionManager;
using namespace TabularResults;
using namespace Test;
void printResultSet(const ResultSet& rs)
```

```
{
  ULong nc = rs.columns.length();
  cout << rs.rows << " rows, " << nc << " columns" << endl;
  for (ULong row = 0; row < rs.rows; row++)
  {
    cout << "row " << row << ": ";
    for (ULong column = 0; column < nc; column++)
    {
      if (column > 0)
      {
        cout << ", ";
      }
      BooleanSeq& nulls = ((ColumnSeq&)rs.columns)[column].nulls;
      if (row + 1 <= nulls.length() && nulls[row])
      {
        cout << "null";
        continue;
      }
      Data& values = ((ColumnSeq&)rs.columns)[column].values;
      switch (values._d())
      {
         case TYPE_BIT:
         {
            BooleanSeq& booleanValues = values.booleanValues();
            cout << (booleanValues[row] ? "true" : "false");
            break;
         }
         case TYPE_TINYINT:
         {
            OctetSeq octetValues = values.octetValues();
            cout << octetValues[row];
            break;
         }
         case TYPE_SMALLINT:
         {
            ShortSeq& shortValues = values.shortValues();
            cout << shortValues[row];
            break;
         }
         case TYPE_INTEGER:
             {
            LongSeq& longValues = values.longValues();
            cout << longValues[row];
            break;
         }
         case TYPE_REAL:
```

```
{
   FloatSeq& floatValues = values.floatValues();
   cout << floatValues[row];
   break;
}
case TYPE_DOUBLE:
case TYPE_FLOAT:
{
   DoubleSeq& doubleValues = values.doubleValues();
   cout << doubleValues[row];
   break;
}
case TYPE_CHAR:
case TYPE_LONGVARCHAR:
case TYPE_VARCHAR:
{
   StringSeq& stringValues = values.stringValues();
   cout << stringValues[row];
   break;
}
case TYPE_BINARY:
case TYPE_LONGVARBINARY:
case TYPE_VARBINARY:
{
   BinarySeq& binaryValues = values.binaryValues();
   cout << "(binary)";
   break;
}
case TYPE_BIGINT:
case TYPE_DECIMAL:
case TYPE_NUMERIC:
{
   DecimalSeq& decimalValues = values.decimalValues();
   cout << "(decimal)";
   break;
}
case TYPE_DATE:
{
   DateSeq& dateValues = values.dateValues();
   // Assumption: time_t is seconds from Jan 1, 1970
   time_t t = (time_t)((dateValues[row].dateValue - 40222.0) *
      86400);
   cout << ctime(&t);
   break;
}
case TYPE_TIME:
```

```
         {
            TimeSeq& timeValues = values.timeValues();
            cout << "time: " << timeValues[row].timeValue;
            break;
         }
         case TYPE_TIMESTAMP:
         {
            TimestampSeq& timestampValues = values.timestampValues();
            time_t t = (time_t)((timestampValues[row].dateValue +
            timestampValues[row].timeValue - 40222.0) * 86400);
            cout << ctime(&t);
            break;
         }
       }
     }
     cout << endl;
   }
}
int main(int argc, char** argv)
{
   ORB_var orb = ORB_init(argc, argv, "");
   Manager_var manager = Manager::
       _narrow(Object_var(orb->string_to_object("iiop://myhost:2000")));
   Session_var session = manager->createSession("jagadmin", "");
   Ping_var p = Ping::_narrow(Object_var(session->create("Test/Java")));
   ResultSet_var rs = p->results();
   printResultSet(rs.in());
   {
      char c;
      fprintf(stderr, "Press Return to continue...");
      c = getchar();
   }
   return 0;
}
```

Retrieving the result set

To retrieve the result set, you must instantiate the stub and call the component method that returns a result set to the client. This example instantiates the stub from the Java component in the Test package in a session as an object p of type Ping_var using the _narrow method. The component method, results() is called on p which returns the result set rs.

```
 Ping_var p = Ping::_narrow(Object_var(session-
>create("Test/Java")));
   ResultSet_var rs = p->results();
```

Iterating through the rows and columns

You must process each column value of each row one at a time. In this example, the processing is contained in a method (which you can reuse in other applications) called printResultSet(). printResultSet() takes the result set rs as an input parameter.

```
printResultSet(rs.in());
```

The method uses the length() method to determine how many columns, nc, are in the result set, rs, and displays the number of columns and rows; the number of rows is represented by the variable rows. The method uses a for loop to iterate through each row, row, in the result set; and a nested for loop to iterate through each column, column, in the current row. The method must check for null values before it can process and print the values in each of the columns of the current row. After checking for and printing out null values, the method continues to the next column in the current row.

```
void printResultSet(const ResultSet& rs)
{
  ULong nc = rs.columns.length();
  cout << rs.rows << " rows, " << nc << " columns" <<
endl;
  for (ULong row = 0; row < rs.rows; row++)
  {
    cout << "row " << row << ": ";
    for (ULong column = 0; column < nc; column++)
    {
      if (column > 0)
      {
        cout << ", ";
      }
      BooleanSeq& nulls =
            ((ColumnSeq&)rs.columns)[column].nulls;

      if (row + 1 <= nulls.length() && nulls[row])
      {
        cout << "null";
        continue;
      }
```

Retrieving the column datatype and processing values

In the body of printResultSet(), the _d() method (the discriminator method) is used to retrieve the datatype of the column and switch/case processing is used to process the column value in the current row. values is a reference to a Data object that represents the column value. _d() returns the datatype of the referenced value to the switch statement and the body of the case statement that matches the datatype is executed. In each case, the current row's column value that corresponds to the case's datatype is printed.

For the Date, Time, Timestamp datatypes, some conversion is required to print a value in a standard format (such as "January 5, 1998").

```
 Data& values =
((ColumnSeq&)rs.columns)[column].values;
      switch (values._d())
      {
         case TYPE_BIT:
         {
            BooleanSeq& booleanValues =
values.booleanValues();
            cout << (booleanValues[row] ? "true" :
"false");
            break;
         }
         case TYPE_TINYINT:
         {
            OctetSeq octetValues =
values.octetValues();
            cout << octetValues[row];
            break;
         }
         case TYPE_SMALLINT:
         {
            ShortSeq& shortValues =
values.shortValues();
            cout << shortValues[row];
            break;
         }
         case TYPE_INTEGER:
             {
            LongSeq& longValues = values.longValues();
            cout << longValues[row];
            break;
         }
         case TYPE_REAL:
         {
```

```
            FloatSeq& floatValues =
values.floatValues();
            cout << floatValues[row];
            break;
         }
         case TYPE_DOUBLE:
         case TYPE_FLOAT:
         {
            DoubleSeq& doubleValues =
values.doubleValues();
            cout << doubleValues[row];
            break;
         }
         case TYPE_CHAR:
         case TYPE_LONGVARCHAR:
         case TYPE_VARCHAR:
         {
            StringSeq& stringValues =
values.stringValues();
            cout << stringValues[row];
            break;
         }
         case TYPE_BINARY:
         case TYPE_LONGVARBINARY:
         case TYPE_VARBINARY:
         {
            BinarySeq& binaryValues =
values.binaryValues();
            cout << "(binary)";
            break;
         }
         case TYPE_BIGINT:
         case TYPE_DECIMAL:
         case TYPE_NUMERIC:
         {
            DecimalSeq& decimalValues =
values.decimalValues();
            cout << "(decimal)";
            break;
         }
         case TYPE_DATE:
         {
            DateSeq& dateValues = values.dateValues();
            // Assumption: time_t is seconds from Jan
1, 1970
            time_t t =
```

```
(time_t)((dateValues[row].dateValue - 40222.0) *
               86400);
            cout << ctime(&t);
            break;
         }
         case TYPE_TIME:
         {
            TimeSeq& timeValues = values.timeValues();
            cout << "time: " <<
timeValues[row].timeValue;
            break;
         }
         case TYPE_TIMESTAMP:
         {
            TimestampSeq& timestampValues =
values.timestampValues();
            time_t t =
(time_t)((timestampValues[row].dateValue +
            timestampValues[row].timeValue - 40222.0) *
86400);
            cout << ctime(&t);
            break;
         }
       }
     }
     cout << endl;
   }
}
```

Handling exceptions

The client-side ORB throws two kinds of exceptions:

- CORBA system exceptions – These exceptions are defined in the CORBA specification.
- User-defined exceptions – These exceptions must be defined in the component's IDL definition.

CORBA system exceptions

The CORBA specification defines the list of standard system exceptions. In C++, all CORBA system exceptions are mapped to a C++ class that is derived from the standard SystemException class defined in the CORBA module. You may want to trap the exceptions shown in this code fragment:

```
try
{
... // invoke methods
}
catch (CORBA::COMM_FAILURE& cf)
{
... // A component aborted the EAServer transaction,
    // or the transaction timed out. Retry the
    // transaction if desired.
}
catch (CORBA::TRANSACTION_ROLLEDBACK& tr)
{
... // possibly retry the transaction
}
catch (CORBA::OBJECT_NOT_EXIST& one)
{
... // Received when trying to instantiate
    // a component that does not exist. Also
    // received when invoking a method if the
    // object reference has expired
    // (this can happen if the component
    // is stateful and is configured with
    // a finite Instance Timeout property).
    // Create a new proxy instance if desired.}
}
catch (CORBA::NO_PERMISSSION& np)
{
... // tell the user they are not authorized
}
catch (CORBA::SystemException& se)
{
... // report the error but don't bother retrying
}
```

Note Not all of the possible system exceptions are shown in the example. See the CORBA/IIOP 2.2 Specification (formal/98-02-01) for a list of all the possible exceptions.

User-defined exceptions

In C++, all CORBA user-defined exceptions are mapped to a C++ class that is derived from the standard UserException class defined in the CORBA module. For more information, see "User-defined IDL datatypes" on page 33 and "User-defined exceptions" on page 35.

Note User-defined types must exist in the EAServer IDL repository before you can use them in interface declarations.

Compiling C++ clients

For example C++ client compilation commands, see "Compile the client executable" on page 137.

If the client uses SSL, the following files must also reside on the client machine in a directory specified in the libary search environment variable. In the UNIX column, replace *ext* with the platform extension for shared library files:

Windows	UNIX
libjctssec.dll	*libjctssec.ext*
libjsybscl.dll	*libjsybscl.ext*
libjspks.dll	*libjspks.ext*
libjsentpks.dll	*libjsentpks.ext*
libjintl.dll	*libjintl.ext*

Deploying C++ clients

To deploy a C++ client on another machine:

1. Install the EAServer client runtime if not done already, including C++ libraries. If the client uses SSL, make sure the SSL client runtime support is installed.
2. Copy the client's executable to the machine.
3. Configure the environment as described in "Verify your environment" on page 126.

Using the CosNaming interface

Although EAServer supports the CORBA CosNaming interface to instantiate proxies in client applications, this technique is not recommended. You do not need to use the CosNaming API in clients to realize the benefits incurred by using logical component names. EAServer uses the CosNaming API to resolve component names in the implementation of the Session::lookup and Session::create methods. "Instantiating component proxies" on page 104 describes the recommended technique for stub instantiation.

Unlike earlier releases, clients must be authenticated to perform name service lookups using the default EAServer 6.0 configuration. To enable name service lookups for clients that haven't been authenticated yet, you must set the minimumPasswordLength property for the default security domain to zero and set an empty password for the "guest" user. Sybase does not recommend this configuration, because allowing "guest" access could be a point of security vulnerability.

Use of the CosNaming interface also requires use of deprecated or unsupported SessionManager::Factory methods, in particular the create methods that take parameters. These methods are not compatible with Enterprise JavaBeans components with multiple create methods in the home interface. These methods are not supported for use in C++ or PowerBuilder clients.

Using CORBA ORB implementations other than EAServer

EAServer's IIOP implementation allows you to use any CORBA client ORB to invoke EAServer components. You can also use the EAServer client ORB to execute components that are hosted by another vendor's server ORB.

Connecting to EAServer with a third-party client ORB

In some cases, you may wish to use another vendor's ORB in your client applications. For example, you may have an existing installation of the ORB on client workstations.

Clients that use another ORB can use the same code as the EAServer ORB, except for the following differences:

- You must use stub classes generated by the vendor's IDL-to-C++ compiler rather than stubs generated by EAServer.
- Your code to connect to EAServer and instantiate components may differ.

Generating compatible C++ stubs

Use the IDL-to-C++ compiler that comes with your ORB software to generate compatible stubs, run on the IDL files in the EAServer repository.

For information about which component IDL files and EAServer IDL files you need to use to generate stubs for other ORBs, see "Generating compatible Java stubs" on page 179 (although this section refers to Java clients, it also applies to C++ clients).

EAServer IDL modules

Use the ORB vendor's IDL-to-C++ compiler to generate stubs for the files in the table, "EAServer IDL files" on page 122. All IDL files are installed in the EAServer *include* subdirectory. "Writing CORBA C++ clients" on page 102 describes how these interfaces are used to instantiate EAServer components and call component methods. For additional information, see the comments in each IDL file.

EAServer IDL files

File name	Description
SessionManager.idl	Defines interfaces for session-based creation of EAServer component instances.
BCD.idl	Defines the CORBA datatypes for EAServer's binary and fixed-point numeric datatypes.
MJD.idl	Defines the CORBA datatypes for EAServer's date and time datatypes.
TabularResults.idl	Defines the CORBA datatypes that represent result sets returned by a method invocation.

Performing datatype conversion

EAServer provides C++ header files to convert from the EAServer CORBA datatypes to those commonly used in C++. If you are using another vendor's ORB, use the EAServer header files in your application. For languages other than C++, see the comments in the IDL files for details on how the data is interpreted.

Instantiating components using a third-party ORB

EAServer's naming service cannot be used with other client ORBs, so you must use the EAServer SessionManager::Manager interface to instantiate components from another ORB, as described in "Instantiating component proxies" on page 104.

Also, you must use standard format IORs, not the URL format, as described in "Creating a Manager instance" on page 108.

Connecting to third-party ORBs using the EAServer ORB

You can use the EAServer client-side ORB to execute components hosted by another vendor's server-side ORB, as long as the server-side ORB accepts IIOP connections and the required interfaces are defined in standard CORBA IDL. Implement your client as follows:

1. Import all the required IDL modules into the EAServer repository, as described in "Managing IDL in EAServer" on page 36.
2. Generate stubs for each imported module, as described in "Generating stubs" on page 102. You must generate stubs for each module individually.

CHAPTER 10

Tutorial: Creating C++ Components and Clients

In this tutorial, you will create a C++ component, install it in EAServer, and create a C++ client program that connects to EAServer and calls a method in the component.

Overview of the sample application

In this sample:

1. The client-side executable, developed with C++, instantiates the middle-tier C++ component, CPPArithmetic.
2. The client executable calls the multiply method in CPPArithmetic.
3. The multiply method computes the product of the input values, then returns the result.
4. The client executable displays the result for the end user.

Tutorial requirements

To create this tutorial application, you need:

- The EAServer software. The *EAServer Installation Guide* for your platform describes how to install the software.
- A C++ development environment, such as:

- For Windows, Microsoft Visual C++.
- For UNIX, the system C++ compiler. Some UNIX platforms support multiple C++ versions. The EAServer *Release Bulletin* for your UNIX platform lists compilers that have been tested with EAServer.

Creating the application

To create and run the sample application:

1. Verify your environment.
2. Start EAServer and the Management Console.
3. Import the IDL interface.
4. Define the package and component.
5. Generate server integration code and implementation templates.
6. Write the server-side code.
7. Create a user account.
8. Write the client-side code.
9. Compile the client executable.
10. Run the client executable.

Verify your environment

Before running the tutorial, verify these environment settings:

- For all platforms, the DJC_HOME environment variable must be set to the location of your EAServer installation.
- For Windows, the PATH environment variable must include the EAServer *lib* subdirectory.
- For UNIX platforms, the EAServer *lib* directory must be added to the shared library search path variable listed in Table 10-1 for your platform.

Table 10-1: Shared library search path variables for UNIX platforms

Platform	Variable name
Solaris	LD_LIBRARY_PATH
HP-UX	SHLIB_PATH
AIX	LIBPATH
Linux	LD_LIBRARY_PATH

❖ **Configuring the Windows environment**

- To configure the command line where you are running the tutorials, run these commands, substituting your EAServer installation location for *eas-home*:

```
set DJC_HOME=eas-home
set PATH=%DJC_HOME%\dll;%PATH%
```

  You can also edit these variables in the System dialog for the Windows Control Panel, or create a batch file to configure the settings.

❖ **Configuring the UNIX environment for C shell**

- To configure the C shell session where you are running the tutorials, run these commands, substituting your EAServer installation location for *eas-home*, and the shared-library variable from Table 10-1 for *LIB_PATH*:

```
setenv DJC_HOME eas-home
setenv LIB_PATH $DJC_HOME/lib:$LIB_PATH
```

❖ **Configuring the UNIX environment for Bourne shell**

- To configure the Bourne shell session where you are running the tutorials, run these commands, substituting your EAServer installation location for *eas-home*, and the shared-library variable from Table 10-1 for *LIB_PATH*:

```
DJC_HOME=eas-home export DJC_HOME
LIB_PATH=$DJC_HOME/lib:$LIB_PATH export LIB_PATH
```

Start EAServer and the Management Console

Start the Management Console and connect to EAServer as described in Chapter 1, “Getting Started,”in the *System Administration Guide*.

Import the IDL interface

CORBA component interfaces must be defined using IDL. Your EAServer installation includes a predefined IDL file, *CPPArithmetic.idl* in the *samples/tutorial/cpp-corba* directory. The component interface has one method, multiply.

❖ **Importing the IDL file**

1. If you haven't already, start EAServer and connect to the preconfigured server with the Management Console as described in Chapter 1, "Getting Started,"in the *System Administration Guide*.
2. In the Management Console, click the IDL Modules folder to display the IDL types in the EAServer repository. Right-click the IDL Modules folder and choose Deploy. The Deploy Wizard displays.
3. Browse to the *samples/tutorial/cpp-corba* directory in your EAServer installation and select *CPPArithmetic.idl*.

Define the package and component

This section shows you how to use Management Console to create the package, component, and method for the sample application.

Define a new package

In EAServer, CORBA packages allow you to group CORBA components that perform related tasks. Before a component can be instantiated by clients, it must be installed in a package, and that package must be installed in the server.

❖ **Creating the *cpptut* package**

1. In the Management Console, click the CORBA packages folder under the Local Server folder. This folder displays all packages in the repository for the server that you are connected to.
2. If the cpptut package is displayed, skip to "Define a new component" on page 129.
3. Right-click the CORBA Packages folder, and select Add. The Add wizard displays. For the package name, enter `cpptut`.
4. When you finish the wizard, the package properties display. Leave these properties at their default settings.

Define a new component

You will define a new C++ component, CPPArithmetic.

❖ Defining the component

1. Expand the cpptut package and right-click the Components folder beneath it, then select Add. The New Component Wizard displays. Apply the following settings as you page through the wizard:

   - For component name, enter `CPPArithmetic`.
   - For component type, choose CORBA/C++.
   - For C++ Class Name, enter `CPPArithmeticImpl`.
   - For C++ Library, enter `libCPPArithmetic`.
   - For IDL Home Interface, leave blank. (EAServer generates the default home interface later in the tutorial.)
   - For IDL Remote Interface, enter `Tutorial::CPPArithmetic`.

   When you finish the wizard, the component properties display.

2. In the component properties, select the General tab. Confirm or apply the settings in the table below. Leave the remaining fields at their default settings.

Field	Value
Component Type	CORBA/C++
C++ Class C++ Library	`CPPArithmeticImpl` `libCPPArithmetic` (no extension)
Copy Library	Checked
Debug Library	Checked
IDL Home Interface	Leave blank.
IDL Remote Interface	`Tutorial::CPPArithmetic`
Automatic Failover	Checked
Pooled	Checked
Thread Safe	Checked

3. Click Apply to save changes made to the component properties.

Generate server integration code and implementation templates

Once you have created the package and component, you must generate the files that allow your C++ implementation to run in EAServer and clients to invoke the component. These include the EJB wrapper component that EAServer generates to invoke the C++ library, the client stub interface files, and an implementation template for the component.

❖ Generating the server-side files

1. In the Management Console, expand the cpptut package. Beneath it, right-click the CPPArithmetic component and choose Refresh.
2. The Management Console generates the required files. If generation fails, check the server log file for a description of the problem.

❖ Generating C++ stubs

- If using Windows, run the following command at a prompt:

```
%DJC_HOME%\bin\idl-compiler -v Tutorial\CPPArithmetic.idl -f
%DJC_HOME%\include -cpp
```

If using UNIX, run the following command at a prompt:

```
$DJC_HOME/bin/idl-compiler.sh -v Tutorial\CPPArithmetic.idl -f
$DJC_HOME/include -cpp
```

Write the server-side code

EAServer has generated C++ implementation templates for the component methods. Here we will fill in the implementation template, then build a shared library or DLL file. Finally, we will verify that the shared library or DLL is in the EAServer *cpplib* subdirectory, where EAServer expects to find C++ component library files.

❖ Writing the server-side code

1. Navigate to the *cpplib* directory under your EAServer installation, then navigate to the *cpptut/CPPArithmetic* subdirectory. You should see the following files:
   - **CPPArithmeticImpl.hpp.new** Template for the component header file. Defines the CPPArithmeticImpl class. No changes are required for the tutorial, other than renaming the file as discussed below.

- **CPPArithmeticImpl.cpp.new** Template for the component implementation. Contains the definition of the component methods. Changes you must make to this file are described below.
- **cpptut_CPPArithmetic.cpp** Source for the skeleton. Do not modify the generated skeleton code.
- **make.nt** Microsoft nmake makefile. The nmake utility is included with the Microsoft Visual C++ installation.
- **make.unix** UNIX makefile, for all UNIX platforms.

2 Rename the implementation files to *CPPArithmeticImpl.hpp* and *CPPArithmeticImpl.cpp*. (In other words, remove the *.new* extension from both file names).

3 Open *CPPArithmeticImpl.cpp* in a text editor, then find the definition of the multiply method. Change the definition so that it matches the one below:

```
CORBA::Double CPPArithmeticImpl::multiply
    (CORBA::Double m1,
    CORBA::Double m2)
{
    CORBA::Double result;
    result = m1 * m2;
    return result;
}
```

4 Save your changes.

❖ Building the component on Windows

1 Verify your setup as described in “Verify your environment” on page 126.

2 Rename *make.nt* to *Makefile*, then open *Makefile* in a text editor. Find the definition of the MSVCDIR and ODBCLIB macros:

```
MSVCDIR=c:\msdev
ODBCLIB = "$(MSVCDIR)\lib\odbc32.lib"
```

If you use the standard Microsoft Visucal C++ setup file, *VCVARS32.bat*, no changes are needed to these settings. The Visual C++ installation generates *VCVARS32.bat* to set the MSVCDIR environment variable. If you do not use the generated *VCVARS32.bat* file, or it is incorrect, edit these lines in the makefile to match your system; set MSVCDIR to the location where Microsoft Visual C++ is installed and set ODBCLIB to the full path to the *odbc32.lib* file.

3 Open a command window and change directory to the *cpplib/cpptut/CPPArithmetic* subdirectory of your EAServer installation. Build the DLL as follows:

   a Apply the settings in the EAServer *djc-setenv.bat* file and the Microsoft Visual C++ *VCVARS32.bat* setup file. For example:

```
set DJC_HOME=d:\Sybase\EAS60
call %DJC_HOME%\bin\djc-setenv.bat
call "D:\engapps\Microsoft Visual Studio\VC98\Bin\VCVARS32.bat"
```

   b Run nmake (no arguments are required).

You should see a new file called *libCPPArithmetic.dll*. Verify that the makefile has copied this file to the EAServer *cpplib* subdirectory. If nmake fails, verify that you have renamed the *.cpp* and *.hpp* implementation files with the expected file names, and that you have applied the correct edits to *CPPArithmeticImpl.cpp* and *Makefile*.

❖ Building the component on UNIX platforms

1 Verify your setup as described in “Verify your environment” on page 126.

2 Rename *make.unix* to *Makefile*.

3 Build the shared library by running make (no arguments are required).

You should see a new file called *libCPPArithmetic.ext*, where *ext* is the appropriate shared library extension for your platform. Verify that the makefile has copied this file to the EAServer *cpplib* subdirectory.

If make fails, verify the following:

- You have renamed the *.cpp* and *.hpp* implementation files with the expected file names, and that you have applied the correct edits to *CPPArithmeticImpl.cpp*.
- The compile and link settings in *Makefile* are appropriate for your installation. The settings are defined in the file *cpplib/make.include.plat*, where *plat* is the platform code returned by running `uname -s` on your system. If necessary, edit this file to match your system configuration.

Create a user account

You must have a user account the client application uses to connect to the server. If you don't already have a user account defined, create it as described here. Alternatively, edit the client application source code to use an existing account.

❖ **Creating the Guest user account**

1. In the Management Console, expand the Security folder and right-click the Users folder beneath it. Choose Add from the context menu.
2. In the New User wizard, enter `Guest` as the user name and click Finish.
3. An icon appears for the Guest wizard under the Users folder. Right-click this icon and choose Set Password.
4. In the Set Password wizard, enter `GuestPassword2` for the password and click Apply.

Write the client-side code

Create the source file for the sample C++ client, *arith.cpp*. You can find a copy of *arith.cpp* in the *samples/tutorial/cpp-corba/client* subdirectory of your EAServer installation. Here is the source for *arith.cpp*:

```
/*
** arith.cpp -- Example C++ client for the EAServer C++
**   tutorial.
**
**   This program connects to EAServer,
**   creates an instance of the Tutorial/CPPArithmetic
**   component, and invokes the multiply method.
**
** Usage:
**    arith iiop://<host>:<port>
**
**    Where:
**
**       <host> is the host name or IP address of the server machine.
**
**       <iiop-port> is the server's IIOP port (9000 in the
**         default configuration).
**
*/
```

```
#include <stdio.h>
#include <iostream.h>
#include <string.h>
#include <Jaguar.hpp>
#include <SessionManager.hpp>
#include <Tutorial.hpp>    // Stubs for interfaces in Tutorial IDL
                           // module.

int main(int argc, char** argv)
{
    const char *usage =
"Usage:\n\tarith iiop://<host>:<iiop-port>\n";
    const char *tutorial_help =
"Verify that the"
"cpptut/CPPArithmetic component exists "
"and that it implements the "
"Tutorial::CPPArithmetic IDL interface.";

    const char *component_name = "cpptut/CPPArithmetic";

    try {

        if (argc < 2)
       {
         cout << usage;
          return -1;
        }

        char* manager_url = argv[1];

        cout << "**** Creating session\n";

        // Initialize the ORB
        CORBA::ORB_var orb = CORBA::ORB_init(argc, argv, 0);

       // Create a SessionManager::Manager instance

       CORBA::Object_var obj =
          orb->string_to_object(manager_url);
       SessionManager::Manager_var manager =
         SessionManager::Manager::_narrow(obj);
        if (CORBA::is_nil(manager))
        {
           cout << "Error: Null SessionManager::Manager instance. Exiting. "
                << usage ;
            return -1;
```

```
}

// Create an authenticated session for user Guest
// using password GuestPassword2

SessionManager::Session_var session =
  manager->createSession("Guest", "GuestPassword2");
if (CORBA::is_nil(session))
{
 cout << "Error: Null session. Exiting. " << usage;
  return -1;
}

// Obtain a factory for component instances by
// resolving the component name

cout << "**** Creating component instance for "
     << component_name << "\n" ;

obj = session->lookup(component_name);
SessionManager::Factory_var arithFactory =
SessionManager::Factory::_narrow(obj);

if (CORBA::is_nil(arithFactory))
{
   cout << "ERROR: Null component factory for component "
          << component_name
          << tutorial_help ;
    return -1;
}

// Use the factory to create an instance.

Tutorial::CPPArithmetic_var arith =
   Tutorial::CPPArithmetic::_narrow(arithFactory->create());

// Verify that we really have an instance.
if (CORBA::is_nil(arith)) {
    cout << "ERROR: Null component instance. "
          << tutorial_help ;
    return -1;
}


// Call the multiply method.
```

```
        cout << "**** Multiplying ...\n\n";
        CORBA::Double m1 = (CORBA::Double)3.1;
        CORBA::Double m2 = (CORBA::Double)2.5;
        CORBA::Double result = arith->multiply(m1, m2);

        cout << (double)m1 << " * " << (double)m2
             << " = " << (double)result
             << "\n\n";

    }

    // Explicitly catch exceptions that can occur due to user error,
    // and print a generic error message for any other CORBA system
    // exception.

    // Requested object (component) does not exist.
    catch ( CORBA::OBJECT_NOT_EXIST cone )
    {
        cout << "Error: CORBA OBJECT_NOT_EXIST exception. Check the "
             << "server log file for more information. Also verify "
             << "that the " << component_name
             << " component has been created properly." << tutorial_help ;
    }

    // Authentication or authorization failure.
    catch ( CORBA::NO_PERMISSION npe )
    {
        cout << "Error: CORBA:: NO_PERMISSION exception. Check whether "
             << "login authentication is enabled for your server and "
             << "whether the component has restricted access. If so "
             << "edit the source file to use a valid user name and "
             << "password.\n";
    }

    // Invalid object reference.
    catch ( CORBA::INV_OBJREF cio )
    {
        cout << "Error: CORBA INV_OBJREF exception.";
    }

    // Communication failure. Server could be down or URL's port value
    // could be wrong.
    catch ( CORBA::COMM_FAILURE ccf )
    {
        cout << "Error: CORBA COMM_FAILURE exception. Check that the "
             << "specified host and IIOP port number are "
```

```
            << "correct and that the server is running. "
            << usage;
    }

    // Anything else.
    catch ( CORBA::OBJ_ADAPTER )
    {
        cout << "Error: CORBA::OBJ_ADAPTER \n";
    }
    catch ( CORBA::SystemException cse )
    {
        cout << "Error: CORBA System Exception. Check that the server "
             << "hostname and IIOP port are specified correctly, and "
             << "check the server's error log for more information.\n"
             << usage;
    }

    return 0;
}
```

Compile the client executable

❖ Compiling the client on Windows

1. Verify your setup as described in “Verify your environment” on page 126.
2. Create a batch file with these commands and run it:

```
SETLOCAL
set INCLUDE=.;%DJC_HOME%\include;%INCLUDE%;
set LIB=%DJC_HOME%\lib;%LIB%
cl /W3 /nologo /DWIN32 /Gd /GX -c arith.cpp
set SYSLIBS=kernel32.lib advapi32.lib
link /MAP /out:arith.exe arith.obj libjcc.lib libjutils.lib %SYSLIBS%
ENDLOCAL
```

❖ Compiling the client on UNIX

1. Verify your setup as described in “Verify your environment” on page 126.
2. Create a shell script containing the commands for your platform from Table 10-2, then run the shell script.
3. Change the script file permissions to allow execution, for example, assuming you have named the script *compile.sh*:

```
chmod 777 compile.sh
```

Table 10-2: Client compilation commands for UNIX platforms

<table>
<tr><th>Platform</th><th>Shell script</th></tr>
<tr><td>Solaris</td><td>This shell script works with the Solaris CC compiler, version 6.x:<pre>
#!/bin/sh
. $DJC_HOME/bin/djc-setenv.sh
CC -DJAG_NO_NAMESPACE -z muldefs -I. -I$DJC_HOME/include \
-L$DJC_HOME/lib -ljcc -ljtml_r -lunic -lnsl \
-ldl -lthread -lm -ljutils -o arith arith.cpp
</pre></td></tr>
<tr><td>HP-UX</td><td>This shell script uses the HP-UX ANSI C++ (aCC) compiler:<pre>
#!/bin/sh
. $DJC_HOME/bin/djc-setenv.sh
aCC -c +DA1.1 +DS2.0 +u4 -DNATIVE -D_HPUX -D_POSIX_C_SOURCE=199506L \
  -D_HPUX_SOURCE -I $(DJC_HOME_JDK13)/include -I
$(DJC_HOME_JDK13)/include/hp-ux -I. \
  -I$DJC_HOME/include -L$DJC_HOME/lib -lpthread -ljcc -lnsl -ljtml_r \
  -ljinsck_r -lunic -ljutils -o arith arith.cpp
</pre></td></tr>
<tr><td>HP Itanium</td><td>This shell script uses the HP C++ (aCC) compiler:<pre>
#!/bin/sh
. $DJC_HOME/bin/djc-setenv.sh
aCC -g +DD32 -mt -I. -I$(DJC_HOME)/include \
-L$(DJC_HOME)/lib -lpthread -lunic -ljtml_r -ljinsck_r \
-ljcc -lnsl -ljlog -o arith arith.cpp
</pre></td></tr>
<tr><td>AIX</td><td>This shell script uses the IBM native compiler:<pre>
#!/bin/sh
. $DJC_HOME/bin/djc-setenv.sh
xlC_r -g -c -DDEBUG -DJAG_NO_NAMESPACE -DAIX -D_AIX -qcpluscmt -qnoro \
  -qmaxmem=-1 -qarch=com -qtbtable=full \
  -I. -I$DJC_HOME/include \
  -brtl -L$DJC_HOME/lib -ljcc.so -lunic -ljtml_r.so -ljinsck_r.so \
  -lpthread -lnsl -ljutils -o arith arith.cpp
</pre></td></tr>
<tr><td>Linux</td><td>This shell script uses the g++ compiler:<pre>
#!/bin/sh
. $DJC_HOME/bin/djc-setenv.sh
g++ -c -D_GNU_SOURCE=1 -DLINUX -D_LINUX -D_REENTRANT -fPIC \
  -fwritable-strings -pipe -g -DDEBUG -I $(DJC_HOME_JDK13)/include \
  -I. -I$DJC_HOME/include \
  -L$DJC_HOME/lib -lpthread -ljcc -lnsl -ljtml_r -ljinsck_r \
  -l unic -ljutils -o arith arith.cpp
</pre></td></tr>
</table>

Run the client executable

If you have not refreshed or restarted the server since creating the CPPArithmetic component or adding the `Guest` user account, do so now before running the client program. Make sure your environment is configured as described in “Verify your environment” on page 126.

Run the executable, specifying the server host name and IIOP port number on the command line as follows:

```
arith iiop://host:iiop-port
```

For example:

```
arith iiop://myhost:2000
```

If everything is working, arith prints the results from the invocation of the multiply method. If not, check the error text printed on the console where you ran the client, and check for error messages in the server log file.

CHAPTER 11 CORBA/Java Overview

This chapter provides an overview of things to consider when developing CORBA/Java clients and components for EAServer.

Overview

CORBA is a distributed component architecture defined by the Object Management Group. EAServer supports the CORBA Internet Inter-ORB Protocol (IIOP). EAServer also provides a CORBA-compatible client-side interface that is implemented according to the CORBA specification for IDL-to-Java language mappings. These two items allow you to create CORBA-compliant Java applications and applets that interact with EAServer components.

Java/CORBA versus EJB components
EAServer provides the Java/CORBA component model for backward compatibility with EAServer 5.*x* and earlier versions. Sybase recommends you create EJB components for new Java development because they are more portable to other application servers.

About CORBA Java language bindings

For information on the CORBA architecture, see the specifications available at the Object Management Group (OMG) Web site at http://www.omg.org.

The EAServer Java ORB runtime is implemented according to the CORBA 2.3 specification (specifically, the document *IDL to Java Language Mapping Specification*, formal/99-07-53). You can download this document from the OMG Web site at http://www.omg.org.

EAServer Java ORB runtime

The Java ORB programming interface is defined by the CORBA Java-language bindings specification. The top-level class, org.omg.CORBA.ORB, is an abstract Java class. Each Java ORB vendor must provide an implementation of this class. For example, the EAServer ORB implementation class is com.sybase.CORBA.ORB. You can use the EAServer ORB or any CORBA-compatible ORB to invoke EAServer components.

In this version, EAServer's ORB implementation does not support:

- Method invocation via the Dynamic Invocation Interface (DII)
- The CORBA::Any type

Requirements

All software that is required to compile, deploy, and run Java components in EAServer is supplied with the EAServer product. However, you can use other compilers or Java IDEs such as JBuilder or Eclipse. You must compile components and clients with a JDK version that is compatible with the JDK version used to run the application server.

Java IDL datatype mappings

Java/CORBA components use the type mappings specified by the CORBA document, *IDL to Java Language Mapping Specification* (formal/99-07-53).

The following table lists the CORBA IDL types predefined in EAServer and the equivalent Java datatypes.

Table 11-1: Java types for predefined CORBA IDL types

CORBA IDL type	Java type (input parameter or return value)	Java type (inout or out parameter)
short	short	org.omg.CORBA.ShortHolder
long	int	org.omg.CORBA.IntHolder
long long	long	org.omg.CORBA.LongHolder
float	float	org.omg.CORBA.FloatHolder
double	double	org.omg.CORBA.DoubleHolder

CORBA IDL type	Java type (input parameter or return value)	Java type (inout or out parameter)
boolean	boolean	org.omg.CORBA.BooleanHolder
char	char	org.omg.CORBA.CharHolder
octet	byte	org.omg.CORBA.ByteHolder
string	java.lang.String	org.omg.CORBA.StringHolder
BCD::Binary	byte[]	BCD.Binary
BCD::Decimal	BCD.Decimal	BCD.DecimalHolder
BCD::Money	BCD.Money	BCD.MoneyHolder
MJD::Date	MJD.Date	MJD.DateHolder
MJD::Time	MJD.Time	MJD.TimeHolder
MJD::Timestamp	MJD.Timestamp	MJD.TimestampHolder
TabularResults:: ResultSet	TabularResults.ResultSet	TabularResults.ResultSetHolder
TabularResults:: ResultSets	TabularResults.ResultSet[ ]	TabularResults.ResultSetsHolder

Binary, Fixed-Point, and Date/Time types

The BCD and MJD IDL modules define types to represent common database column types such as binary data, fixed-point numeric data, dates, times. The BCD::Binary CORBA type maps to a Java byte array. The other BCD and MJD types map to data representations that are optimized for network transport.

To convert between the IDL-mapped datatypes and from core java.* classes, use these classes from the com.sybase.CORBA.jdbc11 package:

Class	Description
SQL	Contains methods to convert from BCD.* and MJD.* types to java.* types
IDL	Contains methods to convert from java.* types to BCD.* and MJD.* types

Chapter 1, “Java Classes and Interfaces,” in the *EAServer API Reference* provides reference pages for these classes.

Result set types

The TabularResults IDL module defines types used to represent tabular data. Result sets are typically used only as return types, though you can pass them as parameters.

User-defined IDL types

A user-defined type is any type that is:

- Not in the set of datatypes that is not predefined by EAServer's read-only repository modules and
- Not one of the CORBA IDL base types.

If a method definition includes user-defined types, the Java component method will use a Java type translated from the IDL type definition.

CORBA Any and TypeCode support
EAServer's Java ORB supports the CORBA Any and TypeCode datatypes. Refer to the OMG CORBA 2.3 specification and *IDL to Java Language Mapping Specification* (formal/99-07-53) for information on using these types.

Camel case versus default IDL-to-Java mappings

By default, EAServer uses standard mappings to generate Java classes for user-defined IDL types, as specified by the CORBA Java language mappings specification.

You can configure *camel case* mappings for IDL-to-Java translation. Camel case mappings follow the Java class naming convention rather than the IDL naming convention. When using this option, IDL operation and parameter names such abc_xyz map to abcXyz, and IDL interfaces, sequence, structure, and union type names abc_xyz map to AbcXyz. The camel case mapping is not applied to exception and structure field names.

To enable camel case mapping, run the following command in the EAServer bin directory:

```
configure camel-case-on
```

To disable camel case mapping, run the following command in the EAServer bin directory:

```
configure camel-case-off
```

Note If you intend to expose components as Web services, enable the camel case option. Otherwise you may run into problems with the JAX-RPC identifier mapping rules defined by the JAX-RPC 1.1 specification, Chapter 20, “Appendix: Mapping of XML Names”.

Holder classes for IDL types

All IDL-mapped Java types have an accompanying holder class that is used for passing parameters by reference. Each holder class has the following structure:

```
public class <Type>Holder {
    // Current value
    public <type> value;
    // Default constructor
    public <Type>Holder() {}
    // Constructor that sets initial value
    public <Type>Holder(<type> v) {
      this.value = v;
    }
}
```

This structure is defined by the CORBA Java-language bindings specification.

CHAPTER 12

Developing CORBA/Java Components

Procedure for creating CORBA/Java components

To create a CORBA/Java compoent, you use the Management Console or a configuration script to define basic information about the component, such as the component name and methods, compile and deploy the component implementation classes, then generate files that are required to write the component's class implementation.

The steps are as follows:

1. Define the component interface in CORBA IDL and deploy the IDL to the EAServer repository. Chapter 3, "Using CORBA IDL," describes how to do this.
2. Create EAServer entities to define the CORBA packages and components. The package and component properties specify the component interfaces and control interaction between EAServer and your implementation. Chapter 4, "Managing CORBA Packages and Components," describes how to do this.
3. Develop the component implementation, as described in "Write the Java source file" on page 148.

4. Generate the EJB wrapper components required to host the CORBA component by running the jaguar-compiler command on the CORBA package as described in "Generating EJB wrapper components" on page 160.

A tutorial is available
If you are new to EAServer, follow the steps in Chapter 14, "Tutorial: Creating CORBA Java Components and Clients" to get aquainted with the Java development and deployment cycle.

Write the Java source file

In the component implementation, create a Java method for each IDL operation in the component's client interfaces. When you code the parameters for each method, use the Java types that correspond to the IDL operation parameters. See "Java IDL datatype mappings" on page 142.

In the Java component, component interface methods must be public and cannot be declared static. If the IDL definition of the method has a non-empty raises clause, the Java method must throw equivalent Java exceptions for the IDL exceptions listed in the raises clause.

The component implementation class must be in a Java package. You cannot define components implemented by classes in the default package.

❖ **Implementing the component**

1. Generate Java interface files for IDL types – If your IDL uses types that are not predefined in EAServer, generate Java types from the IDL interface files.
2. Add package import statements – Import the packages that contain the classes that you need to use in your Java class.
3. Code the constructor – Provide a default constructor to be called when EAServer loads the implementation class.
4. Add error handling code – Add code that gracefully handles errors by logging status messages and sending meaningful messages to the client.
5. To finish up, you can use these advanced technique to polish your component implementation:

a. Manage database connections – Connect to databases through connection caches using the Connection Management API.

b. Return result sets – Return result sets using the EAServer Result Sets API.

c. Issue intercomponent calls – Instantiate a Java stub to make intercomponent calls.

Generate Java interface files for IDL types

If the component's definition uses user-defined types for parameters, return values, or exceptions, Java interfaces are required for these types in order to compile your component's implementation file.

The EAServer installation includes Java stubs for the predefined IDL types. To generate Java stubs for other IDL modules and types, use the idl-compiler command-line tool. For example:

```
idl-compiler.bat -v Tutorial\JavaArithmetic.idl
Tutorial\JavaArithmeticHome.idl -f %DJC_HOME%\samples\tutorial\java-
corba\client-src -java
```

For information on idl-compiler syntax, see Chapter 12, "Command Line Tools," in the *System Administration Guide*.

Add package import statements

The packages below are useful if your component is implemented using the standard CORBA IDL-to-Java datatype mappings:

Package(s)	Description
org.omg.CORBA	Contains Java holder and helper classes for each of the core CORBA datatypes. Also defines the interfaces for a standard Java client-side Object Request Broker.
com.sybase.CORBA.jdbc11.*	Contains utility classes for converting between EAServer IDL datatypes and core Java datatypes.
com.sybase.jaguar.server	Contains utility classes for use in server-side Java code.
com.sybase.jaguar.sql	Defines interfaces for defining and sending result sets.

Package(s)	Description
com.sybase.jaguar.jcm	Provides the Java Connection Management (JCM) classes.
com.sybase.jaguar.util.JException	Many of the methods in the EAServer Java classes throw JException. Note that the packages com.sybase.jaguar.util and org.omg.CORBA contain identically named classes, so you can not import all classes from both packages. To avoid compilation problems, import JException explicitly or always refer to this class by its full name.

The fragment below shows the import statements for all of these classes:

```
import org.omg.CORBA.*;
import com.sybase.CORBA.jdbc11.*;
import com.sybase.jaguar.util.JException;
import com.sybase.jaguar.server.*;
import com.sybase.jaguar.sql.*;
import com.sybase.jaguar.jcm.*;
```

Code the constructor

A class constructor is normally used to initialize instance-specific data. However, if your component implements lifecycle methods, then you should use these methods to manage instance-specific data. Otherwise, instance-specific initialization must be done in the constructor.

Any uncaught exception that is thrown within the constructor aborts the creation of the new component instance.

Add error handling code

Errors occurring during component execution should be handled gracefully as follows:

1. Write detailed descriptions of the error to the log. This will help you debug the problem later. You can call any of the System.out.print methods to write to the log (the output is redirected).
2. If the error prevents completion of the current transaction, roll it back as described in “Set transactional state” on page 158.

3 Throw an exception with a brief, descriptive message that is appropriate for display to an end user of the client application.

Java components can record errors or status messages to the server's log file. Writing to the log creates a permanent record of the error, and log messages can be automatically stamped with the date and time that the message was written. Call any of the System.out.print methods to write to the log.

You can also throw an uncaught exception. Ideally, any exception thrown by your component should be a standard CORBA IDL exception or a user-defined IDL exception (the latter must be listed in the raises clause of the IDL method definition and the throws clause of the equivalent Java method declaration). All exceptions are forwarded to the client, but only exceptions that are defined in IDL can be rethrown by the client stub as a duplicate of the server-side exception.

Advanced techniques

After the basic component implementation is in place, you can add code to perform the following advanced tasks:

- "Issue intercomponent calls" on page 151
- "Manage database connections" on page 153
- "Return result sets" on page 153
- "Access SSL client certificates" on page 158
- "Set transactional state" on page 158
- "Retrieve user-defined component properties" on page 159

Issue intercomponent calls

You must use a proxy to issue intercomponent calls. If you call methods in another Java component directly, no server features are available to the called component, such as transaction control, instance lifecycle management, and security.

Using the CORBA ORB to instantiate proxies

To invoke other components, instantiate a proxy (stub) object for the second component, then use the stub to invoke methods on the component.

To invoke methods in other components, create an ORB instance to obtain proxy objects for other components, then invoke methods on the object references. You obtain object references for other components on the same server by invoking string_to_object with the IOR string specified as *Package*/*Component*. For example, the fragment below obtains a proxy object for a component *SessionInfo* that is installed in the *CtsSecurity* package.

```
    java.util.Properties props = new java.util.Propert
ies();
    props.put("org.omg.CORBA.ORBClass",
                    "com.sybase.CORBA.ORB");
    ORB orb = ORB.init((java.lang.String[])null, props
);
    SessionInfo sessInfo =
        SessionInfoHelper.narrow
            (orb.string_to_object(
                "CtsSecurity/SessionInfo"));
```

When making intercomponent calls using string_to_object, the user name of the client that executed the component is automatically used for authorization checking. The exception is when instantiating the system components in the Jaguar package: the ORB automatically switches to the system user priveleges when you specify a component in the Jaguar package. To specify a user name, use this syntax:

```
orb.string_to_object("iiop://0:0:user_name:password/Package/Component"));
```

You can retrieve the system user name and password with these methods in class com.sybase.CORBA.ORB, which both return strings:

- getSystemUser() returns the system user name.
- getSystemPassword() returns the system password.

When called from components, string_to_object returns an instance running on the same server if the component is locally installed; otherwise, it attempts to resolve a remote instance using the naming server.

Connecting to third-party CORBA servers

Your component may need to invoke methods on a component hosted by another vendor’s CORBA server-side ORB. Sybase recommends that Java components use the EAServer client-side ORB for all IIOP connections made from EAServer components. See “Connecting to third-party ORBs using the EAServer ORB” on page 180 for more information.

Manage database connections

If your Java methods connect to remote data servers, you should use EAServer's connection caching feature to improve performance. See the reference pages for the com.sybase.jaguar.jcm classes for more information.

Note EAServer's transactional model works only with connections obtained from the EAServer Connection Manager. Connections that you open yourself will not be able to participate in EAServer transactions.

Return result sets

Using the JDBC API, a Java component can retrieve result sets from a database. Using classes in the *com.sybase.jaguar.sql* package, Java components can also send these result sets to the caller. A Java component can combine the data from several result sets retrieved from databases and send that data as a single result set to a Java client. A Java component can also forward the original result set retrieved from a database.

Java components send results sets with the interfaces in the com.sybase.jaguar.sql package:

- Methods in the JServerResultSetMetaData interface define the format of rows in a result set.
- Methods in the JServerResultSet interface define column values for rows in a result set and send the rows to the client.

The JContext class contains static factory methods to return objects that implement these interfaces.

Chapter 1, "Java Classes and Interfaces," in the *EAServer API Reference* contains reference pages for all classes and interfaces.

You cannot send a result set unless the IDL definition of the component method returns TabularResults::ResultSet or TabularResults::ResultSets. However, you can still use the JServerResultSetMetaData and JServerResultSet interfaces to implicitly return results. Just return null as the method's return value. Alternatively, you can construct the equivalent Java datatypes for the IDL TabularResults::ResultSet and TabularResults::ResultSets types. Call the getResultSet method in the class com.sybase.CORBA.jdbc11.IDL to convert a java.sql.ResultSet instance into a TabularResults.ResultSet instance that can be returned by the method.

Forwarding a ResultSet object

You can use the steps below to forward results from a JDBC query directly to the client:

1. Query the remote server. Use java.sql.Statement or one of its extensions; the appropriate method depends on the query being sent.
2. Handle the results of the query. For each ResultSet returned by the query, call JContext.forwardResultSet(ResultSet) to forward the rows to the client.
3. If your component uses IDL/Java datatypes, return null as the method's return value.

Instead of calling JContext.forwardResultSet(ResultSet), Java components that use IDL/Java datatypes can call the IDL.getResultSet(java.sql.ResultSet) method to convert ResultSet object to TabularResults.ResultSet object, then return the converted object as the method's return value.

Sending results row-by-row

Use the sequence of calls below to define and send a result set row-by-row. Use these calls when building a result set from a non-JDBC source, or when the java.sql.ResultSet returned by a database query cannot be sent as-is to the client.

JServerResultSet sequence of calls

Here are the calls to construct a result set and send it row-by-row:

1. Create a JServerResultSetMetaData object by calling JContext.createServerResultSetMetaData().
2. Call the JServerResultSetMetaData methods to define the format of the result rows, as follows:
   a. JServerResultSetMetaData.setColumnCount(int) to specify the number of columns in each row.
   b. For each column, call JServerResultSetMetaData.setColumnType(int, int) to specify the datatype.
   c. For columns that have a variable length datatype, call JServerResultSetMetaData.setColumnDisplaySize(int, int) to specify the maximum length for column values.
   d. Call other JServerResultSetMetaData methods to specify other column attributes as needed.

3. Create a JServerResultSet object by calling JContext.createServerResultSet().
4. Call JServerResultSet.next() to position the result set's cursor at the first row.
5. For each row to be sent:
   - For each column, call the appropriate JServerResultSet.set<Object>(int, <Object>) method to set the column value.
   - Call JServerResultSet.next() to send the row.
6. If sending a single result set or if using JDBC types, call JServerResultSet.done() to indicate that all rows have been sent in the current result set.
7. If your component uses IDL/Java datatypes, use the com.sybase.CORBA.IdlResultSet class to convert the result set to a TabularResults.ResultSet instance. See Chapter 1, "Java Classes and Interfaces," in the *EAServer API Reference* for details.

You can repeat steps 4 to 6 to send or create another result set that has the same metadata using the same JServerResultSet object. Repeat steps 1 to 6 to send or create another result set that requires different metadata.

You cannot return multiple result sets unless the method's IDL definition returns TabularResults::ResultSets.

JServerResultSet example

The example method below sends three rows with three columns each. Note that exceptions are not caught in the example; the server logs any uncaught exceptions that are thrown in a method call:

```
public void send_rows (IntegerHolder ih) throws
      JException, SQLException
{

  // Declare the constant 'pi'
  final double pi = 3.1414;  // Create the metadata object.
  JServerResultSetMetaData
       jsrsmd = JContext.createServerResultSetMetaData();

  // There will be 3 columns in the result set.
  jsrsmd.setColumnCount(3);
```

```
  // The first column has datatype INTEGER and name 'one'.
  jsrsmd.setColumnType(1, Types.INTEGER);
  jsrsmd.setColumnName(1, "one");

  // The second column has datatype VARCHAR and name 'two'.
  jsrsmd.setColumnType(2, Types.VARCHAR);
  jsrsmd.setColumnName(2, "two");

  // The third column has datatype DOUBLE and name 'three'.
  jsrsmd.setColumnType(3, Types.DOUBLE);
  jsrsmd.setColumnName(3, "three");

  // Create the result set object.
  JServerResultSet jsrs = JContext.createServerResultSet(jsrsmd);

  // Position the cursor.
  jsrs.next();

  // First row values: 1, "first", pi
  jsrs.setInt(1, 1);
  jsrs.setString(2, "first");
  jsrs.setDouble(3, pi);

  // Send the row.
  jsrs.next();

  // Second row values: 2, "second", pi * 2
  jsrs.setInt(1, 2);
  jsrs.setString(2, "second");
  jsrs.setDouble(3, pi * 2.0);

  // Send the row.
  jsrs.next();

  // Third row values: 3, "third", pi * 3
  jsrs.setInt(1, 3);
  jsrs.setString(2, "third");
  jsrs.setDouble(3, pi * 3.0);

  // Send the row.
  jsrs.next();

  // Demarcate the end of the result set by calling done().
  jsrs.done();
}
```

The fragment below shows client-side code to call the stub and print the rows to the console.

```
try {
  ih = new IntegerHolder();
  comp.send_rows(ih);

  ResultSet rs = comp.getResultSet();
  ResultSetMetaData rsmd = rs.getMetaData();

  StringBuffer row = new StringBuffer("");
  for (int i = 1; i <= rsmd.getColumnCount(); i++)
  {
    row.append(rsmd.getColumnName(i));
    if (i < rsmd.getColumnCount())
      row.append("\t");
  }

  System.out.println(row);

  while(rs.next())
  {
    row = new StringBuffer("");
    for (int i = 1; i <= rsmd.getColumnCount(); i++)
    {
      row.append(rs.getString(i));
      if (i < rsmd.getColumnCount())
        row.append("\t");
    }
    System.out.println(row);
  }

  // Discard any remaining results.
  while(comp.getMoreResults())
  {
    rs = comp.getResultSet();
  }

}
catch (Exception e) {
  System.out.println("Exception: " + e.getMessage());
  e.printStackTrace();
}
```

Access SSL client certificates

Clients can connect to a secure IIOP port using an SSL client certificate. You can issue intercomponent calls to the built-in *CtsSecurity/SessionInfo* component to retrieve the client certificate data, including:

- The distinguished SSL user name
- The client certificate fingerprint (MD5 message digest)
- The client certificate data
- The chain of issuing certificates

This component implements CtsSecurity::SessionInfo IDL interface. HTML documentation is available for the interface in the *html/ir* subdirectory of your EAServer installation. You can view it by loading the main EAServer HTML page, then clicking the "Interface Repository" link.

Set transactional state

The transactional state of a component instance determines whether a transactional component's database updates are committed or rolled back.

To set transactional state, you must use the InstanceContext object retrieved by calling Jaguar.getInstanceContext() in each method that sets transactional state (do not save the object across method invocations, because it will not be valid if the component instance has been deactivated and reactivated). See the *EAServer API Reference Manual* for information on this method.

To set transaction state, choose the method that reflects the state of the work that the component is contributing to the transaction, as follows:

- If the work is complete and without error, call setComplete.
- Call setRollbackOnly if the work cannot be completed. Alternatively, throw the exception org.omg.CORBA.TRANSACTION_ROLLEDBACK. If the error indicates an internal inconsistency in the application, log a description of the error to help debug the problem as described in "Add error handling code" on page 150.

Transaction control with the ServerBean control interface
If you use the deprecated control interface JaguarEJB::ServerBean and Auto demarcation/deactivation option is disabled in the Transactions tab in the Transactions properties for your component, the transaction state specified in the method determines whether the instance is deactivated or remains bound to the client.

Retrieve user-defined component properties

You can add user defined properties for your components using the Advanced tab in the Component Properties page in the Management Console. To access these properties at run time, use the Jaguar::Repository API as shown in the example below. For details on this API, see the generated reference documentation in the *html/ir* subdirectory of your installation. The function below returns an array of Jaguar::Property instances that contain the properties defined for the currently executing component:

```
public static Property[] getMyComponentProps() {
    Repository theRep;
    Property[] myProps;
    try {
        java.util.Properties orbProps = new java.util.Properties();
        orbProps.put("org.omg.CORBA.ORBClass",
                  "com.sybase.CORBA.ORB");
        ORB theOrb = ORB.init((java.lang.String[])null, orbProps);
        theRep = RepositoryHelper.narrow
           (theOrb.string_to_object("Jaguar/Repository"));
    } catch (Exception e) {
        System.out.println("Exception instantiating Repository component:"
                           + "\n" + e);
        return null;
    }
    try {
        String myPackage = JContext.getPackageName();
        String myComponent = myPackage + "/" + JContext.getComponentName();
        myProps = theRep.lookup("Component", myComponent);
    } catch (Exception e) {
        System.out.println("Exception getting component properties:"
                           + "\n" + e);
        return null;
    }
    return myProps;
```

```
}
```

Generating EJB wrapper components

EAServer generates EJB wrapper components to host CORBA components in EAServer. Before generating the EJB wrapper components, compile your component implementation to a code base directory that is in the application server's default class path, such as one of the following:

- The *java/classes* subdirectory
- The *genfiles/java/classes* subdirectory

Run the jaguar-compiler command on the CORBA package to generate the EJB wrapper components. You can run the jaguar-compiler command several ways:

- From the Management Console as described in "Refreshing CORBA packages in the Management Console" on page 43.
- Using a configuration script, as described in "Managing CORBA packages with configuration scripts" on page 43.
- Using the jaguar-compiler command-line tool, as described in Chapter 12, "Command Line Tools," in the *System Administration Guide*.

Refreshing Java components

You can refresh a component's implementation classes while the server is running. You do not need to shut down and restart the server. Classes loaded from a different code base directory will not be reloaded. EAServer only reloads the component's implementation class, the skeleton class, and any classes configured in the Java class loader used by the component—see Chapter 10, "Configuring Java Class Loaders," in the *System Administration Guide*.

CHAPTER 13

Developing CORBA/Java Clients

Procedure for creating CORBA/Java clients

A CORBA/Java client establishes a session with the application server, instantiates stub (or proxy) instances for EAServer components, and executes component methods by calling like-named methods on the stub instance.

1 Generate stub classes.

These classes act as a proxy object for a component instance that is executing on the server; there is one stub for each IDL interface that the component implements. "Generating Java stubs" on page 162 describes how to generate stubs.

2 Implement code to instantiate proxy objects.

Your program must obtain proxy objects for the EAServer component and narrow them to the stub interface that you intend to use. See "Instantiating proxy instances" on page 162.

3 Implement code that invokes the component methods.

You execute the component's methods by calling like-named methods on the stub class and passing the necessary input data. Each stub method has a return value and parameter list that is mapped from the corresponding IDL operation definition. "Executing component methods" on page 175 describes return type and parameter type mappings in detail.

4 If desired, you can serialize the component instance reference as an IOR string, then deserialize the reference later.

See "Serializing component instance references" on page 175 for details.

Each of these steps requires appropriate exception handling. "Handling exceptions" on page 176 summarizes CORBA exceptions.

Generating Java stubs

Stub classes allow you to instantiate local Java objects that act as proxies for an instance of the EAServer component. CORBA/Java clients require two types of stub files:

- Java interfaces for types defined in CORBA IDL. To create these stubs, see "Generate Java interface files for IDL types" on page 149.
- Implementation classes for the component proxy interfaces. If you run clients in a full JDK installation (rather than a JRE), EAServer generates these stubs on demand. You can manually generate them with the stub-compiler command. For details, see Chapter 12, "Command Line Tools," in the *System Administration Guide*.

If you are using another ORB implementation class to connect to EAServer, you must export the IDL interface definitions, then use the vendor's IDL compiler to generate stubs. See "Connecting to EAServer with a third-party client ORB" on page 179 for more information.

Instantiating proxy instances

After you have compiled stub classes, you can implement code that uses the stubs to interact with EAServer components.

Your program must obtain proxy objects for the EAServer component and narrow them to the stub interface that you intend to use by following the steps below:

Step	What it does	Detailed explanation
1	Initialize the CORBA ORB classes.	"Configuring and initializing the ORB runtime" on page 163
2	Use an IOR string and the ORB.string_to_object method to obtain the Manager instance for the server.	"Creating a Manager instance" on page 168
3	Use the Manager instance to create a Session.	"Creating sessions" on page 171
4	Call the Session's lookup method to create proxy objects, then narrow them to an interface that the component supports. The lookup method uses the EAServer name service to resolve the requested name to an installed component.	"Creating stub instances" on page 172
5	Call the stub methods to remotely invoke component methods.	"Executing component methods" on page 175

Java exceptions can occur at any step. "Handling exceptions" on page 176 describes common exceptions and their cause.

Other patterns for proxy instantiation
Some patterns for proxy instantiation used in clients written for earlier EAServer releases are not compatible with EAServer 6.0. In particular, clients that use the CosNaming API or SessionManager::Factory::create methods that take parameters should be modified to use the implementation pattern described here. For more information, see "Using the CosNaming interface" on page 121.

Configuring and initializing the ORB runtime

ORB properties define the class name of the ORB driver that will be used, and configure settings required by the driver. Properties can be set externally in HTML parameters for a Java applet or in command-line arguments for a Java application. You can also set them directly in your source code in both applets and applications. Table 13-1 describes the EAServer ORB properties.

Table 13-1: EAServer Java ORB properties

Property	Specifies
org.omg.CORBA.ORBClass	The class that implements interface org.omg.ORB. Specify com.sybase.CORBA.ORB to indicate the EAServer ORB driver class. There is no default for this property.
com.sybase.CORBA. ConnectionTimeout	For applications that run in a cluster, sets a time limit to receive a server response before the connection fails over to try another server in the cluster. Setting this property ensures that failover happens without an unreasonable delay. Specify the timeout period in seconds. The default of 0 indicates no time limit.
com.sybase.CORBA.forceSSL	If set to true when using a a reverse proxy server, forces use of SSL for the connection to the reverse proxy. Set this property to true if the connection to the reverse proxy must use SSL (HTTPS) tunnelling, but the connection from the proxy to the server does not use SSL tunnelling. See Chapter 9, “Deploying Applications Around Proxies and Firewalls,” in the *EAServer Security Administration and Programming Guide* for more information on connecting to EAServer through proxy servers.
com.sybase.CORBA.GCInterval	Specifies how often the ORB forces deallocation (Java garbage collection) of unused class references. Though this property is set on an individual ORB instance, it affects all ORB instances. The default is 30 seconds. The default is appropriate unless you have set an idle connection timeout of less than 30 seconds. In that case, you should specify a lower value for the garbage collection interval, since connections are only closed while performing garbage collection. In other words, the effective idle connection timeout ranges from the idle connection timeout setting to the smallest integral multiple of the garbage collection interval.
com.sybase.CORBA.http	Specify whether the ORB should use HTTP tunnelling without trying to use plain IIOP first. The default is false. With the default setting, the ORB tries to open a connection using plain IIOP, and switches to HTTP tunnelling if the plain IIOP connection is refused. The default is appropriate when some users connect through firewalls that require tunnelling and others do not; the same application can serve both types. If you know tunnelling is required, set this property to true. This setting eliminates a slight bit of overhead that is incurred by trying plain IIOP connections before tunnelling is used.
com.sybase.CORBA. HttpExtraHeader	An optional setting to specify what extra information is appended to the header of each HTTP packet when connecting through a Web proxy. See Chapter 9, “Deploying Applications Around Proxies and Firewalls,” in the *EAServer Security Administration and Programming Guide* for more information.

Property	Specifies
com.sybase.CORBA.http. jaguar35Compatible	When set to true, specifies that HTTP tunnelling must be compatible with servers running EAServer version 3.5 or older installations. The default is false. **Compatibility with version 3.5 or older servers** The default tunnelling model is incompatible with servers older than version 3.6. If you do not set the com.sybase.CORBA.http.jaguar35Compatible property to true, clients using the EAServer 3.6 or later Java client ORB cannot connect to older-version servers using HTTP tunnelling. Note that HTTP tunnelling may happen automatically when clients connect to the server through firewalls.
com.sybase.CORBA. HttpUsePost	When using HTTP tunnelling, specifies the HTTP request type used. A value of true indicates that POST requests are to be used. A value of false (the default) specifies that GET requests are to be used. Some Web browsers cannot handle the long URLs generated when using HTTP tunnelling with GET requests. Setting this property to true can work around the issue.
com.sybase.CORBA. IdleConnectionTimeout	Specifies the time, in seconds, that a connection is allowed to sit idle. When the timeout expires, the ORB closes the connection. The default is 0, which specifies that connections can never timeout. The connection timeout does not affect the life of proxy instance references; the ORB may close and reopen connections transparently between proxy method calls. Specifying a finite timeout for your client applications can improve server performance. If many instances of the client run simultaneously, a finite client connection timeout limits the number of server connections that are devoted to idle clients. A finite timeout also allows rebalancing of server load in an application that uses a cluster of servers. If you specify an idle connection timeout, make sure the garbage collection interval (com.sybase.CORBA.GCInterval) is set to an equal or lesser value.
com.sybase.CORBA.isApplet	Specifies whether the client is a Java applet. The default is `false` unless the ORB is initialized by calling the Orb.init method that takes a java.applet.Applet instance as a parameter. If you call another version of init from a Java applet, you must set this property to `true` in order to connect to EAServer using SSL.

Property	Specifies
com.sybase.CORBA.local	For server-side component use only. Specifies whether the ORB reference can be used to issue intercomponent calls in user-spawned threads. The default is `true`, which means that intercomponent calls are made in memory and must be issued from a thread spawned by EAServer. Set this property to `false` if your component makes intercomponent calls from user-spawned threads. **com.sybase.CORBA.local property is deprecated** This property is not needed when calling components from threads spawned by the the Thread Manager. The Thread Manager is the recommended way to spawn threads in Java components. See Chapter 5, "Using the Thread Manager," in the *Automated Configuration Guide* for more information.
com.sybase.CORBA.ProxyHost	Specifies the machine name or the IP address of a reverse-proxy server. See Chapter 9, "Deploying Applications Around Proxies and Firewalls," in the *EAServer Security Administration and Programming Guide* for more information.
com.sybase.CORBA.ProxyPort	Specifies the port number of a reverse-proxy server. See Chapter 9, "Deploying Applications Around Proxies and Firewalls," in the *EAServer Security Administration and Programming Guide* for more information.
com.sybase.CORBA.RetryCount	Specify the number of times to retry when the initial attempt to connect to the server fails. The default is 5.
com.sybase.CORBA.RetryDelay	Specify the delay, in milliseconds, between retry attempts when the initial attempt to connect to the server fails. The default is 2000.
com.sybase.CORBA.socketReuseLimit	Specify the number of times that a network connection may be reused to call methods from one server. The default is 0, which indicates no limit. The default is ideal for short-lived clients. The default may not be appropriate for a long-running client program that calls many methods from servers in a cluster. If sockets are reused indefinitely, the client may build an affinity for servers that it has already connected to rather than randomly distributing its server-side processing load among all the servers in the cluster. In these cases, the property should be tuned to best balance client performance against cluster load distribution. In Sybase testing, settings between 10 and 30 proved to be a good starting point. If the reuse limit is too low, client performance degrades.
com.sybase.CORBA.WebProxyHost	The host name or IP address of an HTTP proxy server that supports generic Web tunnelling, sometimes called connect-based tunnelling. See Chapter 9, "Deploying Applications Around Proxies and Firewalls," in the *EAServer Security Administration and Programming Guide* for more information. There is no default for this property, and you must specify both the host name and port number properties.

Property	Specifies
com.sybase.CORBA. WebProxyPort	When generic Web tunnelling is enabled by setting com.sybase.CORBA.WebProxyHost, this property specifies the port number at which the HTTP proxy server accepts connections. See Chapter 9, "Deploying Applications Around Proxies and Firewalls," in the *EAServer Security Administration and Programming Guide* for more information. There is no default for this property, and you must specify both the host name and port properties.
com.sybase.CORBA. useJSSE	Use the Java Secure Sockets Extension (JSSE) classes for secure HTTP tunnelled (HTTPS protocol) connections. JSSE provides an alternative to the built-in SSL implementations when secure connections are needed from an applet running in a Web browser. Additional configuration may be required to use this option. See Chapter 4, "Using SSL in Java Clients," in the *EAServer Security Administration and Programming Guide* for more information.

Example: ORB Initialization in an Applet ORB initialization for a Java applet is demonstrated in the example below. This code constructs a java.util.Properties object and sets the required properties. The applet reference and the Properties object are passed to the org.omg.CORBA.ORB.init method.

```
import java.applet.*;
import org.omg.CORBA.*;
public class myApp extends Applet {

  public void init() {
    ...
    java.util.Properties props
        = new java.utils.Properties();
    props.put("org.omg.CORBA.ORBClass",
       "com.sybase.CORBA.ORB");
    ORB orb = ORB.init(this, props);
    ...
```

Rather than property values, you can pass properties to the ORB as parameters in the HTML APPLET tag that loads the applet, as in the example below:

```
<APPLET
codebase=....
<param name="org.omg.CORBA.ORBClass"
  value="com.sybase.CORBA.ORB">
...
</APPLET>
```

A property setting that is passed as an applet parameter supersedes any setting that is specified in the java.utils.Properties parameter to the ORB.init method. If you want to ensure that hard-coded property values are used, pass the Applet parameter as null.

Example: ORB Initialization in an Application ORB initialization for a Java application is demonstrated in the example below. This code constructs a java.util.Properties object and sets the required properties. The command-line parameters are passed to the org.omg.CORBA.ORB.init method.

```
import java.util.*;

public class myApp extends Object {

  public static void main(String[] args)
     throws Exception
{
    ...
    Properties props = new Properties();
    props.put("org.omg.CORBA.ORBClass",
       "com.sybase.CORBA.ORB");
     ORB orb = ORB.init(args, props);
    ...
```

Rather than hard-coding the property values, you can pass them to the ORB as command-line parameters, as in the example below:

```
java yourclass -org.omg.CORBA.ORBClass com.sybase.CORBA.ORB
```

Properties that are specified as command-line parameters supersede values specified in the java.utils.Properties parameter to the ORB.init method. If you want to ensure that hard-coded property values are used, pass the String[ ] parameter to init as null.

Creating a Manager instance

The EAServer authentication service implements the SessionManager::Manager interface. When using CORBA naming services, you can resolve this object by using the special name AuthenticationService. Without using naming services, you must supply a CORBA Interoperable Object Reference (IOR), which is a text string that describes how to connect to the server hosting the object.

Standard CORBA IOR strings are hex-encoded and not human-readable. EAServer supports both standard format IORs and a URL form that is human-readable. For information on standard-format IORs, see "Instantiating components using a third-party ORB" on page 180.

URL format IORs The URL string format offers the benefits of being human-readable. Also, for Java applets, you can create URL strings that connect to the applet's download host by default; this feature simplifies deployment since you do not need to change hard-coded IORs when you move your application to another server. IOR strings in URL format must have the form:

```
protocol://host:iiop_port
```

where

- *protocol* is `iiops` if connecting to a secure port and `iiop` otherwise.
- *host* is the EAServer host address or machine name. In an applet, you can omit the host name to specify that the connection must go to the host from which the applet was downloaded.
- *iiop_port* is the port number for IIOP requests. Your server may accept IIOP connections at several different ports, each of which uses a diffferent security profile. For example, the default server configuration provides listeners at these ports:
    - 2000 accepts unsecure IIOP connections.
    - 2001 accepts IIOPS connections with encryption and server-side authentication.
    - 2002 accepts IIOPS connections with encryption and mutual (client and server) authentication. Mutual authentication requires that your end users have valid digital certificates, and that those certificates are issued by a certificate authority that is trusted by the server.

  The *EAServer Security Administration and Programming Guide* describes how to configure listeners and security profiles.

An example URL-format IOR is `iiop://machina:2000`, which specifies that the server runs on the machine named "machina" and listens for IIOP requests on port 2000. In an applet, you can omit the host name to specify that the connection must go to the host from which the applet was downloaded. For example, `iiop://:2000` specifies a connection to port 2000 on the applet's host.

Standard format IORs Use the standard IOR format if you must have portability to other standard Java ORB implementations. Your server generates IOR strings embedded within text files each time it starts. Several files are generated for each IIOP listener. There are files formatted as an HTML param tag; these can be used to compose HTML applet sections. There are also files that contain the IOR by itself. Additionally, there are different files generated for compatibility with different IIOP protocol versions.

For each listener, the server prints a hex-encoded IOR string with standard encoding to the following files in the EAServer *html* subdirectory:

- *<listener><iiop-version>.ior* – Contains the IOR string by itself, followed by a newline.
- *<listener>_<iiop-version>_param.ior* – Contains the IOR as part of an HTML param definition that can be inserted into an applet section.

where

- *<listener>* is the name of the listener.
- *<iiop-version>* is the version of IIOP and can be either 10 (which represents IIOP version 1.0) or 11 (which represents IIOP version 1.1). Use the file that matches the IIOP version that is supported by your client ORB.

For example, a server will generate the following files for a listener named iiops2. All files are created in the *html* subdirectory:

- *iiops2_10.ior*
- *iiops2_11.ior*
- *iiops2_10_param.ior*
- *iiops2_11_param.ior*

Your applet can retrieve the IOR if you supply it in applet parameters. In this case, you can copy the contents of one of the param format files to the HTML file. Alternatively, you can add code that connects to EAServer via HTTP and downloads one of the generated *.ior* files.

Note If you change a server's host name or port number, you must edit or replace IOR values that contain the host name, including hex-format IORs copied from the server-generated *.ior* files. When using the EAServer ORB, use the URL string format and omit the host name. When using another vendor's ORB, you can download the contents of a generated .ior file, or you can store server IORs in the ORB vendor's name server.

Creating the Manager instance Once the applet or application has obtained the server's IOR string or an equivalent IIOP URL string, it calls the ORB.string_to_object method to convert the IOR string into a SessionManager::Manager instance, as shown in the following example:

```
import org.omg.CORBA.*;
import java.awt.*;
import SessionManager.*;

public class myApplet extends Applet {
  String ior;
  ORB orb;
  ... deleted ORB.init() code and code that
      retrieves IOR from applet parameters ...
  Manager manager = ManagerHelper.narrow(
                     orb.string_to_object(ior));
```

Creating sessions

The SessionManager.Session interface represents an authenticated session between the client application and EAServer. The Manager.createSession method accepts a user name and password and returns a Session object, as shown in the example below:

```
import org.omg.CORBA.*;
import SessionManager.*;
import java.awt.*;

public class myApplet extends Applet {
  Manager manager;
```

```
... deleted code that created Manager instance
    ...
try {
  Session session = manager.createSession(user,
                                         password);
}
catch (org.omg.CORBA.COMM_FAILURE cf)
{
   // The server is likely down or has run
   // out of connections. You can retry the
   // connection if desired.
   ... report the error ...
}
catch (org.omg.CORBA.NO_PERMISSION np)
{
  // Tell the user they are not authorized
  ...
}
catch (org.omg.CORBA.SystemException se)
{
  // Catch-all clause for any CORBA system
  // exception that was not explicitly caught
  // above. Report the error but don't bother
  // retrying.
  ...
}
```

Creating stub instances

A Java stub implements the Java version for one of the EAServer component's IDL interfaces. Call the Session.lookup method to obtain a factory for stub instances. The signature of Session.lookup is:

```
SessionManager.Factory lookup(String name)
```

Session.lookup takes a string that specifies the name of the component to instantiate. A component's default name is the EAServer package name and the component name, separated by a slash as in *calculator/calc*. However, a different name can be specified with the component's com.sybase.jaguar.component.naming property. For example, you can specify a logical name, such as *USA/MyCompany/FinanceServer/Payroll*. For more information on configuring the naming service, see Chapter 5, "Naming Services," in the *EAServer System Administration Guide*.

Session.lookup returns a factory for component proxies. Call the Factory.create method to obtain proxies for the component. This method returns a org.omg.CORBA.Object reference. You must call the narrow method in the IDL interface's generated helper class to convert this to an instance of the stub class for the component's IDL interface. If the component instance does not implement the requested interface, the narrow method returns a null object reference.

Session.lookup can throw these CORBA standard exceptions:

- **NO_PERMISSION** The user is not authorized to instantiate the requested component.
- **OBJECT_NOT_EXIST** The server component cannot be instantiated. Verify that:
  - The specified component is installed in the specified package.
  - The specified package is listed in the server's Start Modules property.
  - The Java class, Windows DLL, or UNIX shared library that implements the component is available.

The code to call Session.lookup and Factory.create looks like this:

```
import org.omg.CORBA.*;
import SessionManager.*;
import java.awt.*;
import Calculator.*;  // Package for Java stubs
                      // for this example, matches
                      // IDL module name for the
                      // component's interface.

public class myApplet extends Applet {

  Session session;

  ... deleted code that created Session instance
      ...

  //
  // In this example, the component is named calc
  // and is installed in the EAServer package
  // calculator. calcHelper.narrow() verifies that
  // the returned object is of the appropriate
  // type, then returns a Calculator.Calc instance
  //
  try {
    Factory fact =
```

```
      FactoryHelper.narrow(
        session.lookup("calculator/calc"));
    Calc c =
      CalcHelper.narrow(fact.create());
  }
  catch (org.omg.CORBA.OBJECT_NOT_EXIST one)
  {
    // Tell the user to contact the server
    // administrator
    ... report the error ...
  }
  catch (org.omg.CORBA.NO_PERMISSION np)
  {
    // Tell the user they are not authorized
    ... report the error ...
  }
  catch (org.omg.CORBA.SystemException se)
  {
    // Catch-all clause for any CORBA system
    // exception that was not explicitly caught
    // above.
    ... report the error ...
  }
```

Calling Session.lookup in server code

When called from server code, Session.lookup resolves the component name by calling the name service, which gives preference to a local component instance if the component is installed on the same server. However, the use of a locally installed component is not guaranteed. To ensure that a local implementation is used, specify the name as `local:`***`package`***`/`***`component`***, where *package* is the package name and *component* is the component name, for example, `local:CtsSecurity/SessionInfo`. When you specify the local: prefix, the lookup call bypasses the name service and returns a local instance if the component is installed in the same server. The call fails if the specified component is not installed in the same server..

Executing component methods

After instantiating the stub class, use the stub class instance to invoke the component's methods. Each method in the stub interface corresponds to a method in the component interface that you have narrowed the proxy object to. See "Java IDL datatype mappings" on page 142 for descriptions of the type mappings.

Serializing component instance references

You can call the ORB.object_to_string() and ORB.string_to_object() methods to serialize and deserialize proxy object references. Assuming that the proxy interface is Payroll, this call serializes a proxy component reference:

```
  Payroll payroll;
... deleted code that instantiates payroll ...

  String payroll_ior = orb.object_to_string(payroll);
```

This call deserializes the reference:

```
  Payroll payroll = PayrollHelper.narrow(
                   orb.string_to_object(payroll_ior));
```

The following restrictions apply when serializing and deserializing component proxy references:

- Unless the proxy is for an Enterprise Java EntityBean, the serialized reference remains valid only as long as the server has not been restarted since the time when proxy was first instantiated. When deserializing, the proxy instance will connect back to the same host and port as was used to create the original instance. An EntityBean proxy can be deserialized at any time, as long as the EntityBean is still installed on the original server.

- If the original proxy instance was created by connecting to a secure port with a client-side SSL certificate, the proxy must be deserialized in a session that connects using the same client certificate and equal or greater security constraints. For example, if you create an object with session that uses 128-bit SSL encryption, serialize the object, then later try to deserialize the object using during a session that uses 40-bit SSL encryption, the ORB will throw the CORBA::NO_PERMISSION exception. Access will be allowed when objects created using less secure session are later accessed using a more secure session.

Handling exceptions

The client-side ORB throws two kinds of exceptions:

- CORBA system exceptions – these exceptions are defined in the CORBA specification.
- User-defined exceptions – these exceptions are defined in the component's IDL definition.

CORBA system exceptions

The CORBA specification defines the list of standard system exceptions. In Java, all CORBA system exceptions extend org.omg.CORBA.SystemException. System exceptions are unchecked exceptions (they extend java.lang.RuntimeException). The Java compiler does not require that you catch CORBA system exceptions. However, some exceptions can occur in a well-behaved program. For example, the Session.loookup call throws a NO_PERMISSION exception when you request a component instance and the user lacks permission to instantiate that component. You may want to trap the exceptions shown in the code fragment below:

```
try
{
    // invoke method(s)
    ...
}
catch (org.omg.CORBA.COMM_FAILURE cf)
{
    // If this occurs when instantiating a Manager
    // instance, the server is likely down or has run
    // out of connections. You can retry the connection
    // if desired.
    //
    // If this occurs after a method call, you
```

```
        // can retry the call (or the transaction call
        // sequence for a stateful component).
        ...
    }
    catch (org.omg.CORBA.TRANSACTION_ROLLEDBACK tr)
    {
        // A component on the server aborted the EAServer
        // transaction, or the transaction timed out.
        // Retry the method call(s) if desired.
        ...
    }
    catch (org.omg.CORBA.OBJECT_NOT_EXIST one)
    {
        // Possibly try to create another instance. Check
        // that the package and component are installed
        // on the server.
        // Received when trying to instantiate a component
        // that does not exist. Also received when invoking
        // a method if the object reference has expired
        // (this can happen if the component is stateful
        // and is configured with a finite Instance Timeout
        // property). Create another instance if desired.
        ...
    }
    catch (org.omg.CORBA.NO_PERMISSSION np)
    {
        // Tell the user they are not authorized
        ...
    }}
    catch (org.omg.CORBA.SystemException se)
    {
        // Catch-all clause for any CORBA system exception
        // that was not explicitly caught above.
        // Report the error but don’t bother retrying.
        ...
```

Note Not all of the possible system exceptions are shown in the example. See CORBA/IIOP 2.3 Specification for a list of all the possible exceptions.

User-defined exceptions

User-defined exceptions are defined in the component’s IDL definition. For example, you might define OverdrawnException to be thrown by methods that withdraw money from a bank account. In Java, all user-defined exceptions extend org.omg.CORBA.UserException.

In Java, IDL user-defined exceptions are checked exceptions; if the IDL definition of a method contains a raises clause, the equivalent Java stub method will have a throws clause that lists the equivalent Java exceptions. For example, consider the IDL definition below:

```
module MyModule {
  exception MyException
  {
    string reason;
  };

  interface MyIntf {
    boolean throwException
    ( in boolean yes_no )
    raises (MyException);
  };
};
```

The equivalent Java throwException method is:

```
boolean throwException (boolean yes_no)
   throws MyModule.MyException;
```

Deploying and running Java clients

Run the Java client in a JDK 1.4 or later Java interpreter.

At run time, the following EAServer JAR files must be in the CLASSPATH for Java applications and included with the class files for applets:

- *lib/eas-client-15.jar* and *lib/eas-server-15.jar* to run in Java 1.5
- *lib/eas-client-14.jar* and *lib/eas-server-14.jar* to run in Java 1.4

The client runtime writes errors to the console by default. In Java applications, you can modify this behavior by specifying the profile name as the Java system property djc.logFile. For example:

```
java -Ddjc.rmiTrace=true "-Ddjc.logFile=%DJC_HOME%\logs\rmiClientTrace.log"
```

For more information, see “Configuring system logging” in Chapter 3, “Creating and Configuring Servers,” in the *System Administration Guide*.

Using other CORBA ORB implementations

EAServer's IIOP implementation allows you to use any CORBA-compliant client ORB to invoke EAServer components. You can also use the EAServer client ORB to execute components that are hosted by another vendor's server ORB.

Connecting to EAServer with a third-party client ORB

In some cases, you may wish to use another vendor's ORB in your client applications. For example, you may have an existing installation of the ORB on client workstations.

Clients that use another ORB can use the same code as for the EAServer ORB, except for the following differences:

- You must use stub classes generated by the vendor's IDL-to-Java compiler rather than stubs generated by EAServer.
- Your code to connect to EAServer and instantiate components may differ.

When executing methods, you may wish to use the EAServer conversion classes to create and interpret the predefined EAServer datatypes. These conversion classes, in packages com.sybase.CORBA.jdbc102 and com.sybase.CORBA.jdbc11, are documented in Chapter 1, "Java Classes and Interfaces," in the *EAServer API Reference*. The classes are compatible with any Java ORB.

Generating compatible Java stubs

You should generate stubs for your third-party ORB using the IDL-to-Java or IDL-to-C++ compiler provided by the vendor. Stubs created by EAServer are not guaranteed to work with another ORB.

Each component's IDL interfaces are specified in the Component Properties window, under the General tab. See "CORBA component property descriptions" on page 45 for more information. All interfaces are defined in IDL modules that are stored as plain text files in the EAServer *Repository* subdirectory. For example, if the component implements the Module1::I1 and Module2::I2 interfaces, you will need to copy the files *Module1.idl* and *Module2.idl* into a working directory for generating stubs for your third-party ORB software. You must also copy any files that are included by these modules, including those listed in *Table 13-2: Predefined EAServer IDL files*.

Table 13-2 lists the names of the predefined EAServer IDL modules that are needed by all client applications.

Table 13-2: Predefined EAServer IDL files

Filename	Description
SessionManager.idl	Defines interfaces for session-based creation of EAServer component instances.
BCD.idl	Defines the CORBA datatypes for EAServer's binary and fixed-point numeric datatypes.
MJD.idl	Defines the CORBA datatypes for EAServer's date and time datatypes.
TabularResults.idl	Defines the CORBA datatypes that represent result sets returned by a method invocation.

Warning! When creating stubs for another ORB, do not overwrite the EAServer Java stubs. Use different package names when creating stubs for third-party ORBs or create the third-party ORB stubs under a different code base.

Instantiating components using a third-party ORB

EAServer's naming service cannot be used with other client ORBs, so you must use the EAServer SessionManager::Manager interface to instantiate components from another ORB, as described in "Instantiating proxy instances" on page 162. Set the org.omg.CORBA.ORBClass property to the name of the class provided by your ORB vendor.

Connecting to third-party ORBs using the EAServer ORB

You can use the EAServer client-side ORB to execute components hosted by another vendor's server-side ORB, as long as the server-side ORB accepts IIOP connections and the required interfaces are defined in standard CORBA IDL.

❖ **Implement your client as follows:**

1. Import all the required IDL modules into EAServer, as described in "Managing IDL in EAServer" on page 36.
2. Generate stubs for each imported module, as described in "Generating Java stubs" on page 162.
3. Implement code to connect to the third-party server and instantiate components, following the vendor' s documentation.

CHAPTER 14

Tutorial: Creating CORBA Java Components and Clients

In this tutorial, you will create a CORBA Java component, install it in EAServer, and create a CORBA Java client that connects to EAServer and calls a method in the component.

Overview of the sample application

The application performs the following steps:

1. The client-side application, developed with Java, instantiates the middle-tier Java component, JavaArithmetic.
2. The client calls the multiply method in JavaArithmetic.
3. The multiply method computes the product of the input values, then returns the result.
4. The client application displays the result for the end user.

Tutorial requirements

To create the tutorial application, you need:

- The EAServer software

  The EAServer *Installation Guide* for your platform describes how to install the software.

- Java development environment

  The tutorial steps use the JDK software and Apache Ant software that is included with your EAServer installation. You can also use Eclipse, JBuilder, or any other development tool that is compatible with JDK 1.4 or later.

Creating the application

To create and run the sample application:

1. Start EAServer and the Management Console.
2. Import the IDL interface.
3. Define the package and component.
4. Compile the component implementation.
5. Generate stubs and skeletons.
6. Create a user account.
7. Create the client program.
8. Run the client program.

Start EAServer and the Management Console

Start the Management Console and connect to EAServer as described in Chapter 1, "Getting Started,"in the *System Administration Guide*.

Import the IDL interface

CORBA component interfaces must be defined using IDL. Your EAServer installation includes a predefined IDL file, *JavaArithmetic.idl* in the *samples/tutorial/java-corba* directory. The component interface has one method, multiply.

❖ **Importing the IDL file**

1. If you haven't already, start EAServer and connect to the preconfigured server with the Management Console as described in Chapter 1, "Getting Started,"in the *System Administration Guide*.
2. In the Management Console, click the IDL Modules folder to display the IDL types in the EAServer repository. Right-click the IDL Modules folder and choose Deploy. The Deploy Wizard displays.
3. Browse to the *samples/tutorial/java-corba* directory in your EAServer installation and select *JavaArithmetic.idl*.

Define the package and component

This section shows you how to use Management Console to create the package, component, and method for the sample application.

Define a new package

In EAServer, CORBA packages allow you to group CORBA components that perform related tasks. Before a component can be instantiated by clients, it must be installed in a package, and that package must be installed in the server.

❖ **Creating the *javatut* package**

1. In the Management Console, click the CORBA packages folder under the Local Server folder. This folder displays all packages installed in the server that you are connected to.
2. Right-click the CORBA Packages folder, and select Add. The Add wizard displays. For the package name, enter `javatut`.
3. When you finish the wizard, the package properties display. Leave these properties at their default settings.

Define and install a new component

You will define a new Java/CORBA component, JavaArithmetic.

❖ **Defining the new component**

1. Expand the javatut package and right-click the Components folder beneath it, then select Add. The New Component Wizard displays. Apply the following settings as you page through the wizard:

- For component name, enter `JavaArithmetic`.
- For component type, choose CORBA/Java.
- For Java Class Name, enter `com.sybase.easerver.tutorials.java.JavaArithmeticImpl`.
- For IDL Home Interface, leave blank. (EAServer generates the default home interface later in the tutorial.)
- For IDL Remote Interface, enter `Tutorial::JavaArithmetic`.

When you finish the wizard, the component properties display.

2 In the component properties, select the General tab. Confirm or apply the settings in the table below. Leave the remaining fields at their default settings.

Field	Value
Component Type	CORBA/Java
Java Class	`com.sybase.easerver.tutorials.java.JavaArithmeticImpl`
IDL Home Interface	Leave blank.
IDL Remote Interface	`Tutorial::JavaArithmetic`
Automatic Failover	Checked
Pooled	Checked
Thread Safe	Checked

3 Click Apply to save changes made to the component properties.

Compile the component implementation

The component implementation classes must be placed in the Java class path for EAServer before we can generate skeletons and the EJB wrapper that integrates the component code into EAServer.

Your EAServer installation includes an Ant project to compile the component in the subdirectory *samples/tutorial/java-corba*. Source for the component is in *JavaArithmeticImpl.java* in the subdirectory *src/com/sybase/easerver/tutorials/java*.

The *build.xml* file defines an Ant project to compile the component and a test client. To ensure the component classes are in the EAServer Java class path, the Ant project compiles them to the EAServer *genfiles/java/classes* subdirectory.

❖ **Compiling the component implementation**

1. At a command prompt, change to the EAServer *samples/tutorial/java-corba* subdirectory.

2. Make sure the DJC_HOME environment variable specifies the location of your EAServer installation, then running the build script or batch file. For example, if on Windows:

```
set DJC_HOME=D:\Sybase\eas60
cd %DJC_HOME%\samples\tutorial\java-corba
build
```

   Or, if running UNIX with C shell:

```
setenv DJC_HOME /opt/Sybase/eas60
cd $DJC_HOME/samples/tutorial/java-corba
build
```

   The build script or batch file runs the EAServer djc-ant command, which invokes Ant on the default build file, *build.xml* in the current directory.

Generate stubs and skeletons

Once you have created the package and component, you must generate the files that allow your C++ implementation to run in EAServer and clients to invoke the component. These include the EJB wrapper component that EAServer generates to invoke the component and client stub interface files that clients use to call the component methods.

❖ **Generating the server-side files**

1. In the Management Console, expand the javatut package. Beneath it, right-click the JavaArithmetic component and choose Refresh.

2. The Management Console generates the required files. If generation fails, check the server log file for a description of the problem.

❖ **Generating CORBA/Java stubs**

- If using Windows, run the following command at a prompt:

```
%DJC_HOME%\bin\idl-compiler -v Tutorial\JavaArithmetic.idl
Tutorial\JavaArithmeticHome.idl -f %DJC_HOME%\genfiles\java\src -java
```

  If using UNIX, run the following command at a prompt:

```
$DJC_HOME/bin/idl-compiler.sh -v Tutorial/JavaArithmetic.idl
Tutorial/JavaArithmeticHome.idl -f $DJC_HOME/genfiles/java/src -java
```

Create a user account

You must have a user account the client application uses to connect to the server. If you don't already have a user account defined, create it as described here. Alternatively, edit the client application source code to use an existing account.

❖ **Creating the Guest user account**

1. In the Management Console, expand the Security folder and right-click the Users folder beneath it. Choose Add from the context menu.
2. In the New User wizard, enter `Guest` as the user name and click Finish.
3. An icon appears for the Guest wizard under the Users folder. Right-click this icon and choose Set Password.
4. In the Set Password wizard, enter `GuestPassword2` for the password and click Apply.

Create the client program

The Ant project for the component

The Ant project is located in the subdirectory *samples/tutorial/java-corba*. Source for the client is in *Arith.java* in the subdirectory *client-src/com/sybase/easerver/tutorials/java/client*.

This is a simple command-line application that:

- Connects to EAServer.
- Creates an authenticated session using the Guest account that we created earlier.
- Creates a proxy for the component.
- Calls the component multiply method.

Here is the source for *Arith.java*:

```
//package com.sybase.easerver.tutorials.java.client;

/**
 * This is a sample command-line Java application that
 * invokes the JavaArithmetic component created in the EAServer
 * CORBA/Java component tutorial. Usage:
 * <pre>
 *    Arith iiop://<host>:<port>
```

```
 *
 *    Where:
 *
 *       <host> is the host name or IP address of the server machine.
 *
 *       <iiop-port> is the server's IIOP port (2000 in the
 *         default configuration).
 *</pre>
 */

import org.omg.CORBA.*;
import SessionManager.*;
import Tutorial.*;  // Package for EAServer stub classes

public class Arith {

    static public final String compName = "javatut/JavaArithmetic";

    static public void main(String options[]) {

        String _usage = "Usage: Arith iiop://<host>:<port>\n";
        String _ior = null;

       try {

           if (options.length >= 1)
           {
          _ior = options[0];
           }
           else
           {
          System.out.println(_usage);
                return;
           }

           //
           // Initialize the CORBA client-side ORB and
           // obtain a stub for the EAServer component instance.
           //
           System.out.println("... Creating session.");

           //
           // Initialize the ORB.
           //
           java.util.Properties props = new java.util.Properties();
           props.put("org.omg.CORBA.ORBClass", "com.sybase.CORBA.ORB");
```

```
    ORB orb = ORB.init(options, props);

    //
    // Create an instance of the EAServer SessionManager::Manager
    // CORBA IDL object.
    //

    Manager manager = ManagerHelper.narrow(orb.string_to_object(_ior));

    //
    // Create an authenticated session with user "Guest" and password
    // "GuestPassword2".
    //
    Session session = manager.createSession("Guest", "GuestPassword2");

    System.out.println("... Creating component instance.");

    //
    // Create a stub object instance for the
    // Tutorial/JavaArithmetic EAServer component.
    //
    JavaArithmetic comp =
     JavaArithmeticHelper.narrow(
       session.create(compName));

    if (comp == null)
    {
      System.out.print("ERROR: Null component instance. ");
      System.out.print(
"Verify that the component " + compName +
"exists and that it implements the " +
"Tutorial::JavaArithmetic IDL interface.");
      return;
    }

    System.out.println("... Created component instance.");

     //
     // Invoke the multiply method.
     //
    System.out.println("... Multiplying:\n");
    double m1 = 3.1;
    double m2 = 2.5;
     double result = comp.multiply(m1, m2);
    System.out.println("    " + m1 + "*" + m2 + "=" + result);
```

```
        // Explicitly catch exceptions that can occur due to user error,
        // and print a generic error message for any other CORBA system
        // exception.

        } catch ( org.omg.CORBA.COMM_FAILURE cfe)
        {
            // The server is not running, or the specified URL is
            // wrong.
            System.out.println(
                "Error: could not connect to server at " + _ior + "\n"
              + "Make sure the specified address is correct and the "
              + "server is running.\n\n" + _usage );
        } catch ( org.omg.CORBA.OBJECT_NOT_EXIST cone )
        {
             // Requested object (component) does not exist.
             System.out.println(
                "Error: CORBA OBJECT_NOT_EXIST exception. Check the "
              + "server log file for more information. Also verify "
              + "that the " + compName
              + "component has been created properly. \n");

        } catch (org.omg.CORBA.NO_PERMISSION npe) {
            // Login failed, or the component requires an authorization role
            // that this user is not a member of.
             System.out.println("Error: CORBA NO_PERMISSION exception. "
                              + " Does the Guest account exist and have"
                              + " you set the password to match this example"
                              + " code?");
              npe.printStackTrace();

        } catch (org.omg.CORBA.SystemException se)
        {
             // Generic CORBA exception
            System.out.println(
                "Received CORBA system exception: "
                + se.toString() );
            se.printStackTrace();
        }

        return;
    } // main()

}
```

❖ **Compiling the client application**

- Compile the application source using the component Ant project, specifying the `client` target. For example, on Windows

```
set DJC_HOME=D:\Sybase\eas60
cd %DJC_HOME%\samples\tutorial\java-corba
build client
```

Or, if running UNIX with C shell:

```
setenv DJC_HOME /opt/Sybase/eas60
cd $DJC_HOME/samples/tutorial/java-corba
build client
```

The build script or batch file runs the EAServer djc-ant command, which invokes Ant on the default build file, *build.xml* in the current directory.

Run the client program

If you have not refreshed or restarted the server since creating the JavaArithmetic component, refresh the server before running the client program.

Create a batch file or UNIX shell script to run the client application, then run it. The batch file or shell script configures the CLASSPATH environment variable, then runs the application using the JDK 1.4 java program included with your EAServer installation.

If necessary, you can run the client on a different machine than the server host, as long as your server uses a real host address and not localhost or 127.0.0.1.

If everything is working, the application prints the results from the invocation of the multiply method. If not, check the error text printed on the console where you ran the client, and check for error messages in the server log file.

❖ **Running the client on Windows**

1 Create a file named *runclient.bat* containing the commands below:

```
setlocal
call %DJC_HOME%\bin\djc-setenv.bat
cd %DJC_HOME%\samples\tutorial\java-corba
set CLASSPATH=%CLASSPATH%;.\client-classes
%JAVA_HOME%\jre\bin\java
com.sybase.easerver.tutorials.java.client.Arith %*
```

2 Run the client by running the batch file and specifying the server's IIOP URL on the command line, for example:

```
set DJC_HOME=D:\Sybase\eas60
runclient iiop://myhost:2000
```

❖ **Running the client on UNIX**

1 Create a file named *runclient* containing the commands below:

```
#!/bin/sh
. $DJC_HOME/bin/djc-setenv.sh
cd $DJC_HOME/samples/tutorial/java-corba
CLASSPATH=$CLASSPATH:./client-classes export CLASSPATH
$JAVA_HOME/jre/bin/java com.sybase.easerver.tutorials.java.client.Arith
$*
```

2 Change the file permissions to allow the script to be executed. For example:

```
chmod 777 runclient
```

3 Run the client by running the batch file and specifying the server's IIOP URL on the command line, for example:

```
setenv DJC_HOME /opt/Sybase/eas60
runclient iiop://myhost:2000
```

Index

D

E

F

G

H

I

P

R

U

W