WebSphere® Application Server V4.0.1 for z/OS and OS/390

Assembling CORBA Applications

SA22-7848-02

WebSphere® Application Server V4.0.1 for z/OS and OS/390

Assembling CORBA Applications

SA22-7848-02

Note

Before using this information and the product it supports, be sure to read the general information under "Appendix D. Notices" on page 179.

Third Edition (October 2001)

This is a major revision of SA22–7848–01

This edition applies to WebSphere Application Server V4.0.1 for z/OS and OS/390 (5655-F31), and to all subsequent releases and modifications until otherwise indicated in new editions.

The most current versions of the WebSphere Application Server V4.0.1 for z/OS and OS/390 publications are at this Web site: `http://www.ibm.com/software/webservers/appserv/`

Contents

Figures

Tables

About this book

WebSphere Application Server V4.0.1 for z/OS and OS/390: Assembling CORBA Applications, SA22-7848 describes how to create, assemble, and deploy Common Object Request Broker Architecture (CORBA) applications to run in a WebSphere for z/OS server. WebSphere for z/OS includes two types of application servers: One for Java 2 Enterprise Edition (J2EE) applications, the other for CORBA applications. The server for CORBA applications is known as the managed-object framework (MOFW) server. For information about J2EE applications and the WebSphere for z/OS server in which they are installed, see *WebSphere Application Server V4.0.1 for z/OS and OS/390: Assembling J2EE Applications*, SA22-7836.

CORBA applications may consist C++ or Java business objects, which are reusable pieces of code that represent specific business functions or entities, such as a checking account, or a customer. WebSphere for z/OS servers provide an application environment that allows these objects to be highly managed and integrated with databases and transactional systems on z/OS or OS/390. Object-oriented client applications, written in a variety of programming languages, may drive requests against these server applications from any operating system (for example, Windows NT and AIX) that runs a WebSphere Application Server for z/OS and OS/390 product.

Before using the instructions in this book, programmers must be familiar with the WebSphere programming and component models, which are described in detail in *WebSphere Application Server for OS/390 Component Broker: Programming Guide*.

Who should read this book

This book is intended for programmers who want to learn how to enable object-oriented, distributed applications to access new or existing data or programs that reside or run on z/OS or OS/390 or its subsystems, such as DB2, CICS, and IMS. To enable such access, programmers develop, assemble, and deploy server applications that run in WebSphere for z/OS servers. These tasks each require different skills:

- To **develop** the portable components of a server application (that is, the component objects that are platform independent), programmers need to know:
  - The C++ or Java programming languages

  - The WebSphere for z/OS common programming and component models, which are described in detail in *WebSphere Application Server for OS/390Component Broker: Programming Guide*
  - The Object Management Group's Common Object Request Broker Architecture (CORBA)
  - Application development tools, including Object Builder and other tools that comprise VisualAge Component Development (formerly known as the CBToolkit). Instructions for using these tools are described in *WebSphere Application Server for OS/390 Component Broker: Application Development Tools Guide*.

  The bulk of information for client and server CORBA application programmers appears in the common WebSphere for z/OS books; in other words, those books that contain platform-independent information. This book, however, contains some information, such as design considerations and z/OS or OS/390-specific guidelines, that might be helpful if you know that you will deploy your server application on z/OS or OS/390. This information appears in "Chapter 2. Developing CORBA applications for WebSphere for z/OS" on page 15.
- To **assemble** the portable and platform-specific components of a CORBA server application, programmers need some expertise with both workstation and z/OS or OS/390 application development. In other words, they need some understanding of the process and tools required for developing those portable components (as described in the preceding list), and some familiarity with working in the UNIX System Server (USS) environment. Working in the USS environment includes using the hierarchical file system (HFS), setting environment variables, and using the `make` command to compile code.

  General information about working in the USS environment appears in *z/OS UNIX System Services User's Guide*, SA22-7801. Information about the assembly process for CORBA applications appears in "Chapter 3. Assembling CORBA applications on z/OS or OS/390" on page 39.
- To **deploy** a CORBA server application on z/OS or OS/390, one needs some knowledge of the application to be deployed and knowledge of z/OS or OS/390 and the subsystems that the server application requires. z/OS or OS/390 system programmers and database administrators are the most likely personnel to have the skills required for deploying an application. In this book, personnel who assemble and deploy server applications are called **application assemblers**. In addition to some details about the server application to be deployed, these people need to know:
  - How to work in the USS environment (as described in the preceding list), and in TSO/E to access z/OS or OS/390 components and data sets.

  - How to set up z/OS or OS/390 subsystem resources that the application requires. For example, these resources might include databases, transaction managers, and security products.

    The degree of expertise in each area depends on the type of server application to be deployed. For example, if your CORBA application accesses DB2 data directly instead of accessing DB2 through an IMS transaction, you do not need to know anything about IMS.
  - How to define and activate a WebSphere for z/OS application server (that is, the run-time environment for the application to be deployed).

  Information about the deployment process appears in "Chapter 4. Deploying CORBA applications in WebSphere for z/OS MOFW servers" on page 71.

Although the primary emphasis of this book is on server applications that run in WebSphere for z/OS servers, several topics also address the client applications that drive methods against CORBA applications, to obtain the services of z/OS or OS/390 and its subsystems. WebSphere for z/OS supports Java, C++, and other client applications that run on a variety of platforms, including Windows NT and z/OS or OS/390.

Client application programmers require similar workstation skills and resources as programmers who develop server applications. Again, the degree of expertise in each area depends on the type of client application and its run-time environment. One key difference between the skill requirements for client and server application programmers, however, is that client programmers do not need to know platform-specific details about the resource managers or data stores, such as IMS and DB2, that server applications use. Client applications deal only with the server application interfaces, not the internal workings of the server deployment platform.

To develop client applications that run on z/OS or OS/390, however, programmers need to have some z/OS or OS/390 skills, including familiarity with the USS environment. The bulk of development information for client application programmers appears in the common WebSphere for z/OS books; this book addresses the requirements for preparing and running client applications on z/OS or OS/390.

How this book is organized

This book is organized in the following chapters:

- "Chapter 1. Getting started with CORBA applications for WebSphere for z/OS" on page 1 is the place to start if you have little or no experience working with CORBA applications on WebSphere for z/OS. This chapter walks you through the process of developing and deploying a CORBA server application, from writing code through running the server application in a WebSphere for z/OS server.

This chapter serves as an introduction to the topics that are discussed in detail in the next three chapters:

– "Chapter 2. Developing CORBA applications for WebSphere for z/OS" on page 15
– "Chapter 3. Assembling CORBA applications on z/OS or OS/390" on page 39
– "Chapter 4. Deploying CORBA applications in WebSphere for z/OS MOFW servers" on page 71

- "Chapter 2. Developing CORBA applications for WebSphere for z/OS" on page 15 provides the z/OS or OS/390-specific guidelines that you need to know when you are designing and writing code for server applications that you plan to deploy on z/OS or OS/390. For a complete picture of the development process and guidelines, depending on the type of CORBA application you are creating, you also might need to refer to one or more of the common books for WebSphere for z/OS application development:
  – *WebSphere Application Server for OS/390 Component Broker Programming Guide*, SC09–4442, describes the programming model including business objects, data objects, and information about the managed object framework, IDL, and C++ CORBA programming.
  – *WebSphere Application Server for OS/390 Component Broker Programming Reference, Volumes 1 and 2*, SC09–4446 and SC09–4447, contain information about the application programming interfaces (APIs) available to Component Broker application developers.
  – *WebSphere Application Server for OS/390 Component Broker Advanced Programming Guide*, SC09–4443, describes the Component Broker implementation for the CORBA Object Services and the Component Broker Object Request Broker (including remote method invocation and the dynamic invocation interface (DII) procedures), among other topics.
  – *WebSphere Application Server for OS/390 Component Broker Application Development Tools Guide*, SC09–4448, explains how to create and test Component Broker applications using the tools provided in the CBToolkit with a focus on common development scenarios such as inheritance and team development.
  – *WebSphere Application Server for OS/390 Component Broker Glossary*, SC09–4450, defines commonly used terms.
- "Chapter 3. Assembling CORBA applications on z/OS or OS/390" on page 39 lists the steps required to transform application source code developed on the workstation into executable code that can run in a WebSphere for z/OS server. These assembly steps include packaging the component objects that constitute a server application, and compiling code on z/OS or OS/390.

- "Chapter 4. Deploying CORBA applications in WebSphere for z/OS MOFW servers" on page 71 provides guidelines and procedures for creating and activating a WebSphere for z/OS server (that is, the run-time environment) for your server application.
- "Chapter 5. Developing, assembling, and deploying client applications on z/OS or OS/390" on page 87 provides guidelines for setting up client applications that use the CORBA applications you develop for WebSphere for z/OS servers.
- "Chapter 6. Working with CORBA applications in a production system" on page 97 provides background information alternative methods of installing server applications in a production-level, rather than a test-level, WebSphere for z/OS server.
- "Chapter 7. Collecting data about CORBA application activity" on page 101 covers various methods of collecting information, such as trace data, about CORBA applications that run in WebSphere for z/OS servers.

Where to find related information, tools, and supplements

This is a list of books that are in the WebSphere for z/OS library. They can be found at the following Web site:

`http://www.ibm.com/software/webservers/appserv/`

- *WebSphere Application Server V4.0.1 for z/OS and OS/390: Program Directory*, GI10-0680, describes the elements of and the installation instructions for WebSphere for z/OS.
- *WebSphere Application Server V4.0.1 for z/OS and OS/390: License Information*, LA22-7855, describes the license information for WebSphere for z/OS.
- *WebSphere Application Server V4.0.1 for z/OS and OS/390: Installation and Customization*, GA22-7834, describes the planning, installation, and customization tasks and guidelines for WebSphere for z/OS.
- *WebSphere Application Server V4.0.1 for z/OS and OS/390: Messages and Diagnosis*, GA22-7837, provides diagnosis information and describes messages and codes associated with WebSphere for z/OS.
- *WebSphere Application Server V4.0.1 for z/OS and OS/390: Operations and Administration*, SA22-7835, describes system operations and administration tasks.
- *WebSphere Application Server V4.0.1 for z/OS and OS/390: Assembling J2EE Applications*, SA22-7836, describes how to develop, assemble, and install J2EE applications in a WebSphere for z/OS J2EE server.
- *WebSphere Application Server V4.0.1 for z/OS and OS/390: Assembling CORBA Applications*, SA22-7848, describes how to develop, assemble, and deploy CORBA applications in a WebSphere for z/OS (MOFW) server.

- *WebSphere Application Server V4.0.1 for z/OS and OS/390: System Management User Interface*, SA22-7838, describes the system administration and operations tasks as provided in the Systems Management User Interface.
- *WebSphere Application Server V4.0.1 for z/OS and OS/390: System Management Scripting API*, SA22-7839, describes the functionality of the WebSphere for z/OS Systems Management Scripting API product.
- *WebSphere Application Server V4.0.1 for z/OS and OS/390: Migration*, GA22-7860, describes migration procedures for WebSphere for z/OS.

Here are some other WebSphere Application Server books on that Web site that you might find particularly helpful:
- *WebSphere Application Server for OS/390 V3.5 Standard Edition Planning, Installing, and Using*, GC34-4835, provides information about running the WebSphere for z/OS plug-in within the HTTP Server address space. You can use this configuration if you want to continue running non-J2EE-compliant Web applications in the V4.0.1 WebSphere for z/OS plug-in within the HTTP Server address space while migrating to the full WebSphere for z/OS run time.
- *Getting Started with WebSphere Application Server*, SC09-4581, provides an overview of the WebSphere Application Server family of products.
- *Building Business Solutions with WebSphere*, SC09-4432

The integrated WebSphere Application Server Advanced Edition and WebSphere Application Server Enterprise Edition InfoCenter includes CORBA (MOFW) information you need to code CORBA (MOFW) components. Go to:

```
http://www.ibm.com/software/webservers/appserv/infocenter.html
```

For additional WebSphere for z/OS tools and supplements, go to the following Web site:

```
http://www.ibm.com/software/webservers/appserv/zos_os390/download.html
```

You might also need to refer to information about other z/OS or OS/390 elements and products. All of this information is available through links at the following Internet locations:

```
http://www.ibm.com/servers/eserver/zseries/zos/
http://www.ibm.com/servers/s390/os390/
```

How to send your comments

Your feedback is important in helping to provide the most accurate and highest quality information. You can e-mail your comments to:

```
wasdoc@us.ibm.com
```

or fax them to 919-254-0206.

Be sure to include the document name and number, the WebSphere Application Server version, and, if applicable, the specific page, table, or figure number on which you are commenting.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Summary of changes

Summary of changes
for SA22-7848-02
WebSphere for z/OS V4.0.1
as updated, October 2001

This book contains information previously presented in for SA22-7848-01, which supports WebSphere for z/OS. The following is a summary of changes to this information:

- The information about migrating applications has been removed from this book. Information about migrating applications now appears in *WebSphere Application Server V4.0.1 for z/OS and OS/390: Migration*, GA22-7860.
- "Debugging and tracing distributed applications" on page 101 describes how to use the IBM Distributed Debugger and Object Level Trace, which provides debugging and tracing capabilities for J2EE application components and their Java clients, which may reside on platforms other than z/OS or OS/390.

Summary of changes
for SA22-7848-01
WebSphere for z/OS
as updated, June 2001
service level W400018

This book contains information previously presented in for SA22-7848-00, which supports WebSphere for z/OS. The following is a summary of changes to this information:

- The information in "Logging messages and trace data for Java applications" on page 106 has been changed to reflect the following behavior, introduced through APAR PQ47682 (PTF UQ53715, service level W400010):

  All messages that your application issues will appear in the CTRACE data set for WebSphere for z/OS. Some messages also will appear on the master console or in the error log, depending on the message type:
  - `TYPE_INFORMATION` (or `TYPE_INFO`) will appear on the master console.
  - `TYPE_ERROR` (or `TYPE_ERR`) will appear in the error log.

  Note that comments in the sample code in section "Steps for coding your Java application to issue messages and trace requests" on page 112 also have changed to reflect the changed destinations for messages.

- New environment variables have been added to “Appendix A. Environment files” on page 125.

Technical changes or additions to the text and illustrations are indicated by a vertical line to the left of the change.

Chapter 1. Getting started with CORBA applications for WebSphere for z/OS

This chapter contains an overview of the processes for developing and deploying CORBA applications on WebSphere for z/OS. This overview walks you through the programming and deployment tools you will use for CORBA applications.

The following table shows the tasks and associated background information or procedures discussed in this chapter:

Task:	**Associated background information or procedure (See ...)**
Learning how to develop CORBA applications for z/OS or OS/390	"Background on developing CORBA applications"
Learning how to deploy CORBA applications in a WebSphere for z/OS server	"Background on deploying server applications" on page 8
Setting up the application development and deployment environments	"Background on setting up the development and deployment environments" on page 11

Background on developing CORBA applications

The first step in the development process is to create the component objects that constitute the CORBA application. The types of component objects that a CORBA application requires may vary depending on a number of factors. For example, the objects you need to access CICS or IMS resources are different from the objects you need to access DB2 resources. Because of this variety in component objects, the development process also varies somewhat, as figures in the following topics illustrate.

The deployment process, however, is the same regardless of the type of CORBA application. Depending on the CORBA applications you want to develop, review one or more of the development topics, along with "Background on deploying server applications" on page 8:

- "Developing CORBA applications that access relational databases" on page 2
- "Developing CORBA applications that access CICS or IMS resources" on page 5

If you have a basic understanding of the WebSphere for z/OS client programming and component models, you already know about these component objects: the business object, managed object, data object, and so on. If you are unfamiliar with these terms, see the *Component Broker Programming Guide* for an overview and detailed object descriptions.

Developing CORBA applications that access relational databases

To develop applications that access relational databases, such as DB2, you first use Object Builder to define the components illustrated in Figure 1.

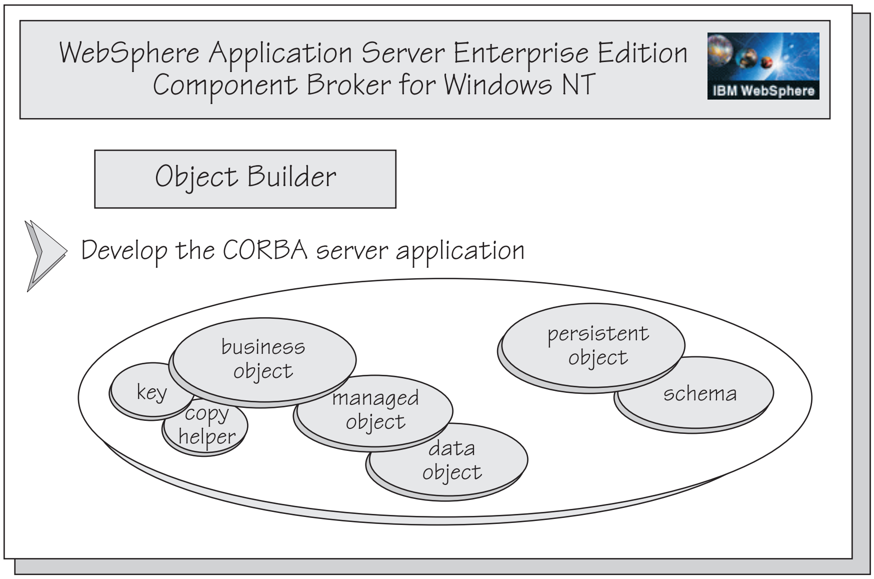
WebSphere Application Server Enterprise Edition
Component Broker for Windows NT
IBM WebSphere
Object Builder
Develop the CORBA server application
business object
persistent object
key
copy helper
managed object
schema
data object

Figure 1. Developing component objects for a CORBA application

From those component object definitions, you instruct Object Builder to generate source code and define the dynamic load libraries (DLLs) that are necessary for client applications that use the CORBA server application, and for the WebSphere for z/OS server in which the CORBA application will run. Figure 2 on page 3 illustrates those steps in the process. At this point, Object Builder also generates make files that you use later, on z/OS or OS/390, to generate executable code.

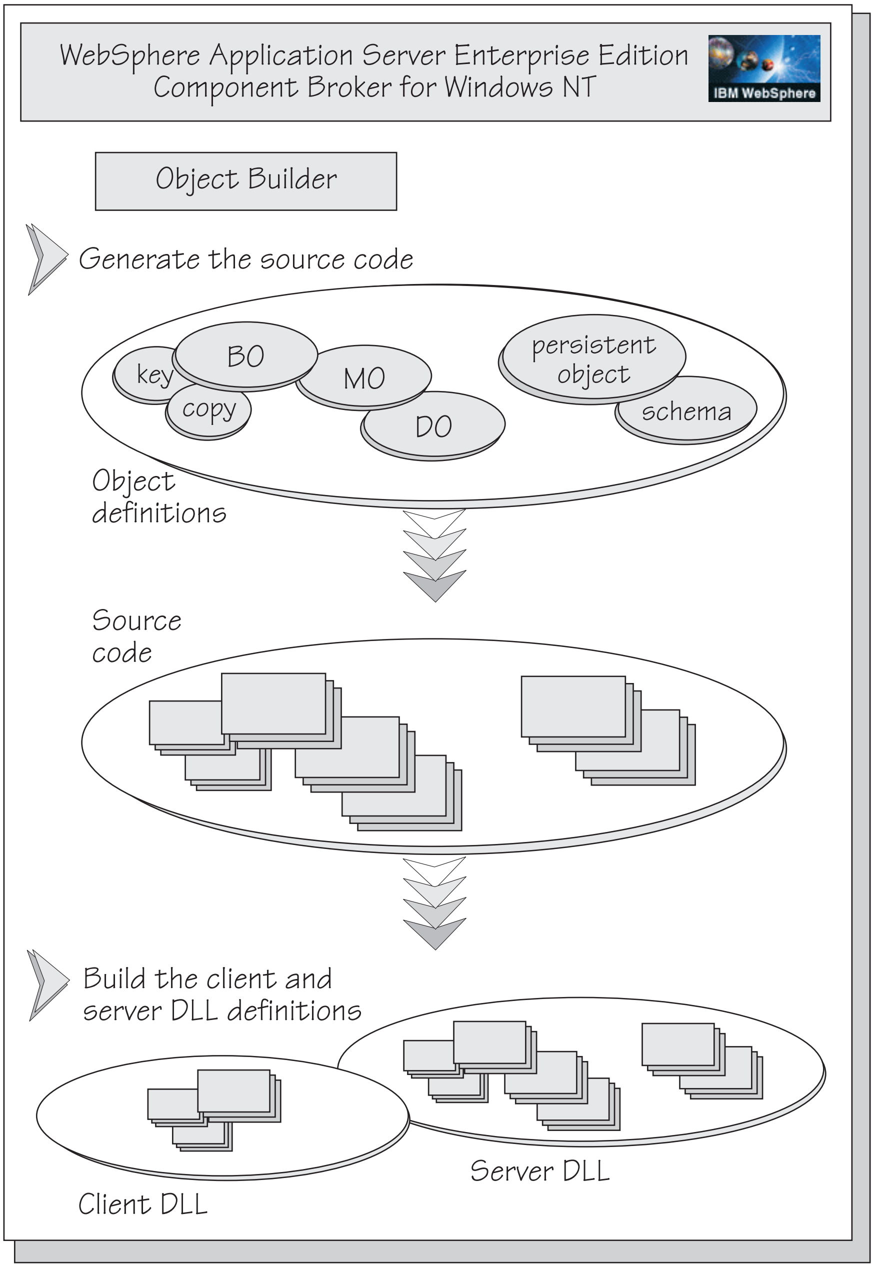

Figure 2. Generating source code for the component objects

After generating source code, client and server DLLs, and make files, you need only one more artifact from Object Builder: the metadata that a WebSphere for z/OS server needs to understand the structure and identity of the CORBA application, its home and container. Object Builder generates this metadata in data definition language (DDL) when you complete the steps for packaging the CORBA application, as illustrated in Figure 3.

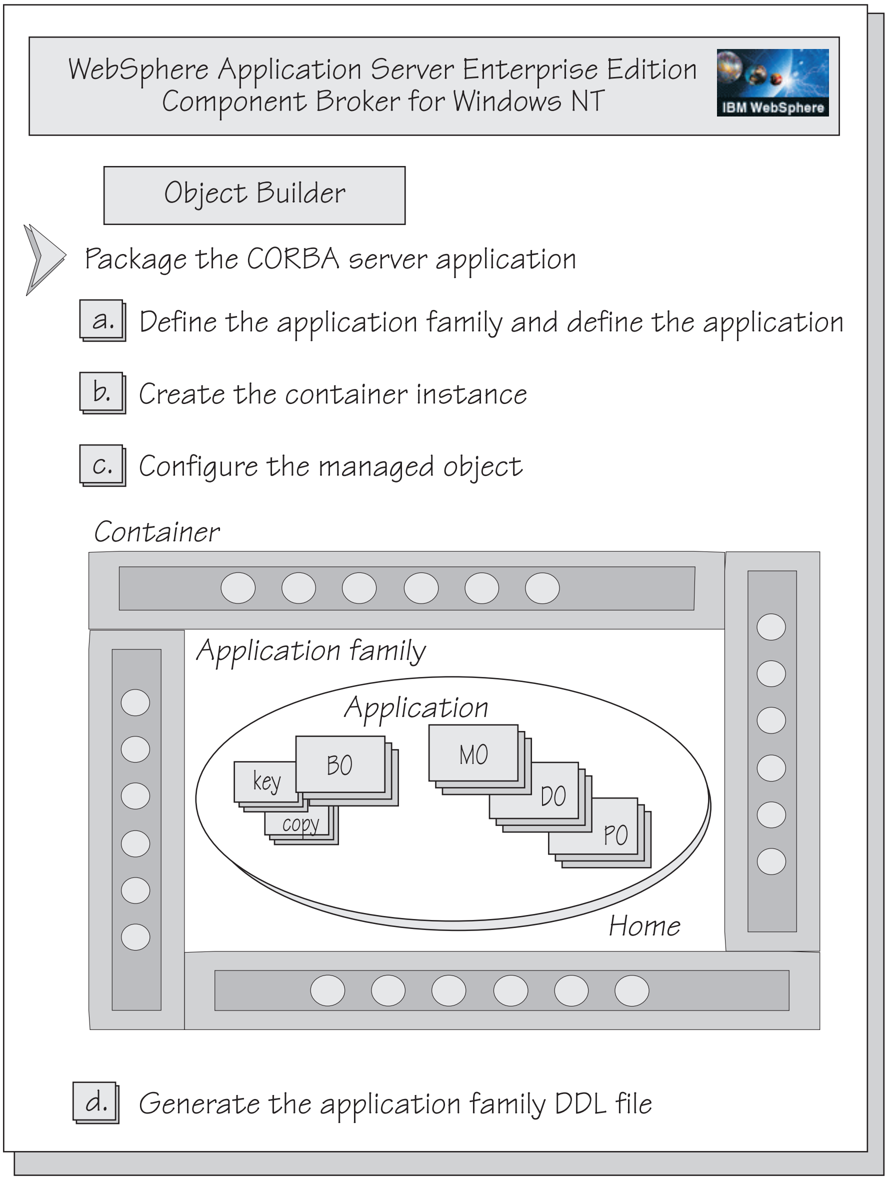
WebSphere Application Server Enterprise Edition
Component Broker for Windows NT
IBM WebSphere
Object Builder
Package the CORBA server application
a. Define the application family and define the application
b. Create the container instance
c. Configure the managed object
Container
Application family
Application
key
BO
copy
MO
DO
PO
Home
d. Generate the application family DDL file

Figure 3. Packaging the component objects into an application family

At this point, you transfer the artifacts from the development process to a z/OS or OS/390 system on which WebSphere for z/OS is installed. On that system, you can run the make files to generate executable code. The instructions in this book provide the details you need to complete the transfer and code generation. Once you have generated the code, you are ready to deploy your CORBA application in a WebSphere for z/OS application server, as illustrated in “Background on deploying server applications” on page 8.

Developing CORBA applications that access CICS or IMS resources

If you are developing a CORBA application that uses CICS or IMS, the beginning of the development process is a bit different than that shown in “Developing CORBA applications that access relational databases” on page 2. The CICS and IMS CORBA applications use the procedural application adapter (PAA) support that WebSphere for z/OS provides. For such server applications, you first must create a procedural adapter object (PAO) using VisualAge for Java. Through VisualAge for Java, you package this object and its associated classes into a PA bean. You then import the PA bean into Object Builder, which converts the PAO and its classes into two component objects: the persistent object and its schema. Figure 4 on page 6 illustrates this process:

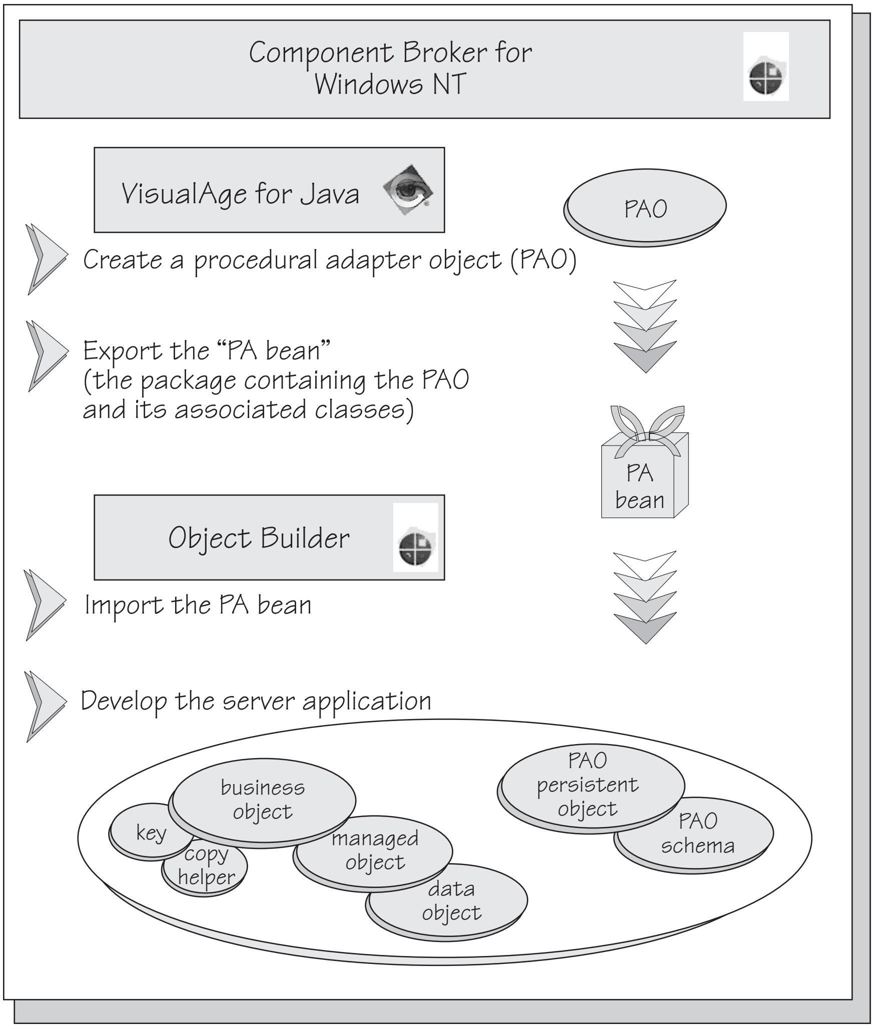

Figure 4. Developing component objects for a CORBA application that uses PAA support

Figure 5 on page 7 gives you a closer look at the PAO and its beans, or classes, and their function:

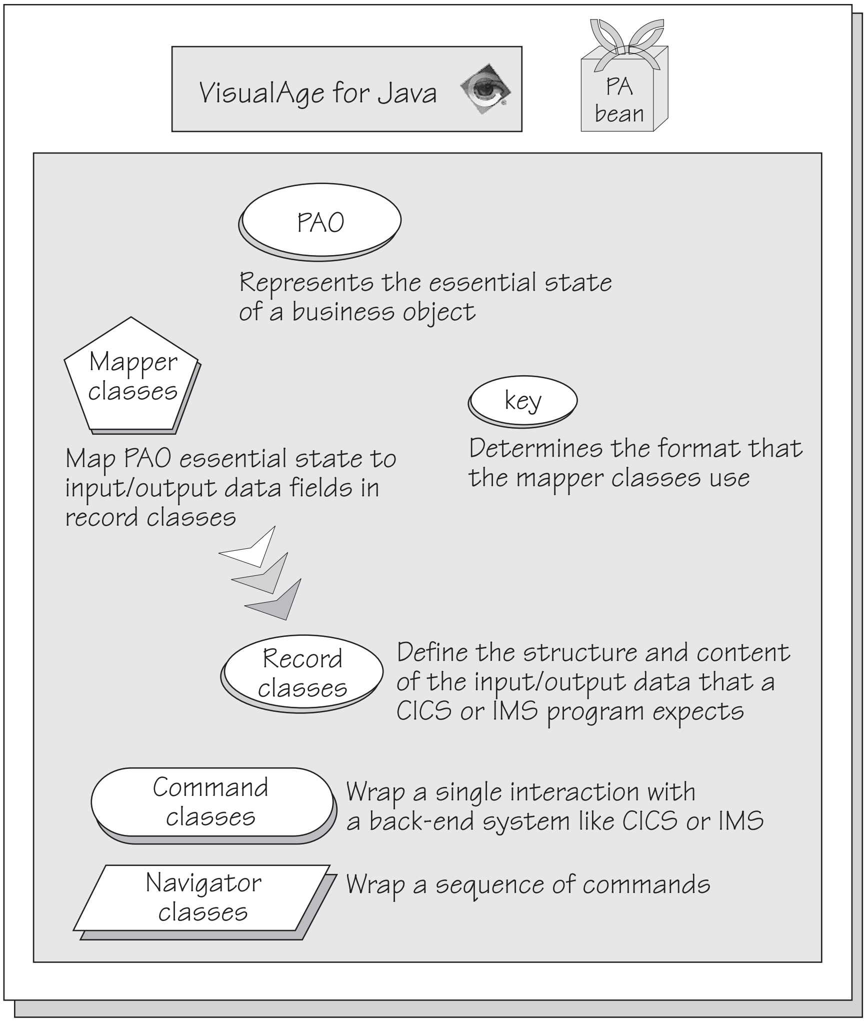
VisualAge for Java
PA bean
PAO
Represents the essential state of a business object
Mapper classes
Map PAO essential state to input/output data fields in record classes
key
Determines the format that the mapper classes use
Record classes
Define the structure and content of the input/output data that a CICS or IMS program expects
Command classes
Wrap a single interaction with a back-end system like CICS or IMS
Navigator classes
Wrap a sequence of commands

Figure 5. The PA bean package: the PAO and its associated classes

The remainder of the development process is the same as illustrated in Figure 2 on page 3 and Figure 3 on page 4.

After generating source code and packaging the application family, you transfer the artifacts from the development process to a z/OS or OS/390 system on which WebSphere for z/OS is installed. On that system, you can run the make files to generate executable code. The instructions in this book provide the details you need to complete the transfer and code generation. Once you have generated the code, you are ready to deploy your CORBA application in a WebSphere for z/OS application server, as illustrated in “Background on deploying server applications” on page 8.

Background on deploying server applications

Once you have an executable CORBA application, you need to set up its run-time environment. Depending on the sample, you might have additional tasks, such as DB2 tables to set up, specific client applications to set up, and other tasks that are unique to one type of CORBA application. In such cases, the instructions in this book outline those additional tasks.

In all cases, however, you will use the WebSphere for z/OS Administration application, which runs on Windows NT, to define an application server. When you define an application server, you create a model that consists of the elements illustrated in Figure 6 on page 9.

Note: WebSphere for z/OS includes two types of application servers: One for Java 2 Enterprise Edition (J2EE) applications, the other for CORBA applications. The server for CORBA applications is known as the managed-object framework (MOFW) server. When you use the WebSphere for z/OS Administration application to define servers for your CORBA applications, you will notice that the Administration application uses two labels for servers: J2EE server and Server (for the MOFW server type). Make sure you use Server for your CORBA applications.

The steps to define a server include:
1. Starting a new conversation
2. Adding a MOFW application server
3. Adding a MOFW server instance
4. Adding a container
5. Adding a logical resource mapping (LRM), instance, and connection
6. Importing a CORBA server application

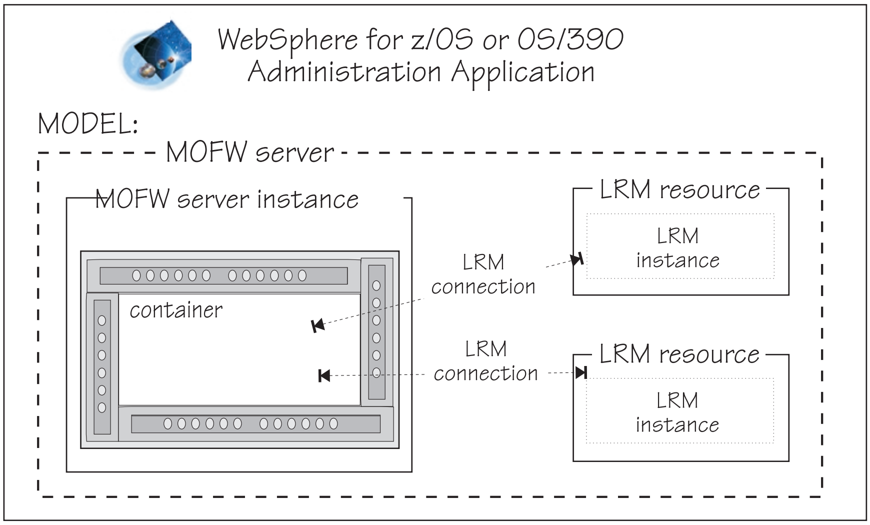

Figure 6. Defining the run-time environment, or MOFW application server

The server model you define includes environment variables that control the attributes of the server, or run-time environment for your application. The instructions in this book contain information about what environment variables to set for each type of CORBA application. When you start creating WebSphere for z/OS servers for your own applications, see "Background on setting environment variables for the WebSphere for z/OS MOFW server" on page 73 for more details.

Also as part of this server definition, you import the DDL (the metadata) that defines the structure of the CORBA application. Figure 7 on page 10 illustrates the result of the server definition and DDL import process: a model of the application server, the container, home, and the CORBA application.

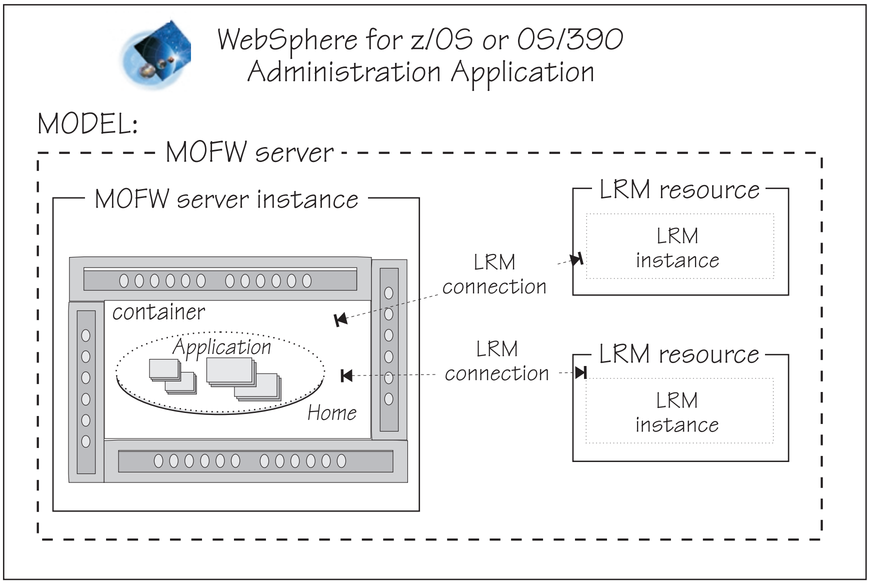

Figure 7. Defining the application family to its MOFW server

Once you have a model of the run-time environment for your CORBA application, you start the last phase of this process: converting the model into an active run-time environment on z/OS or OS/390, as illustrated in Figure 8 on page 11.

First, you use the WebSphere for z/OS administration application to commit the server model, which is analogous to permanently saving the definition. You cannot make any changes to the model after committing it. Then, on z/OS or OS/390, you must complete some manual tasks such as security definitions. The instructions in this book provide you with advice to help you with these tasks. If you are unfamiliar with these aspects of z/OS or OS/390 systems, you might need the help of a system programmer or security administrator as well.

Finally, through the WebSphere for z/OS Administration application, you can activate the server.

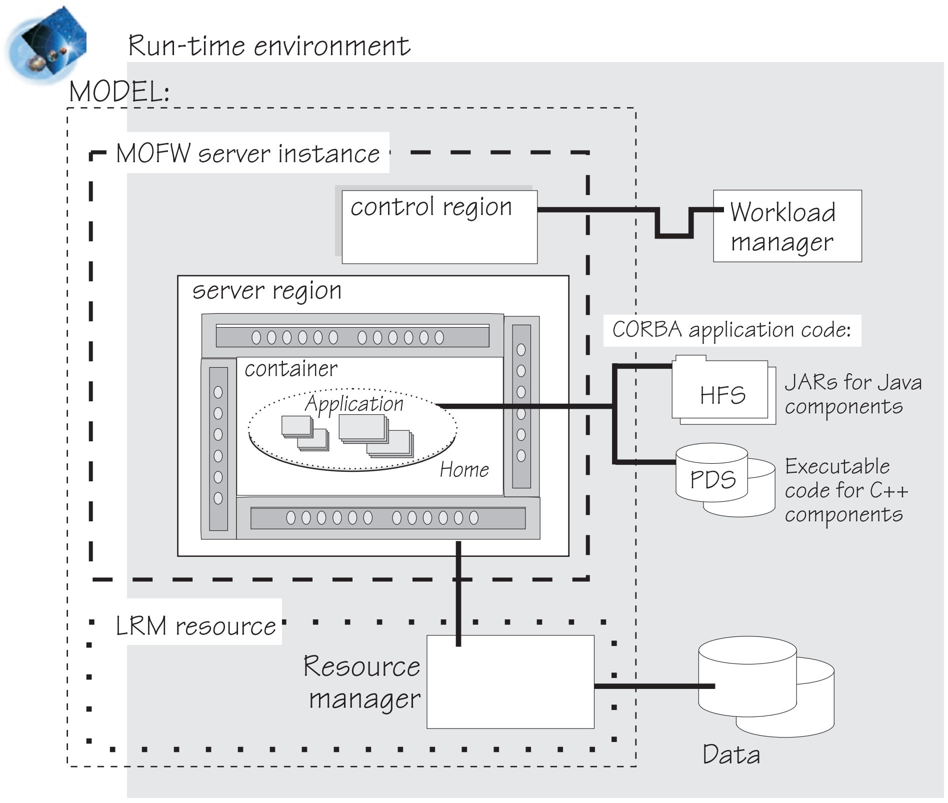

Figure 8. Activating the MOFW application server

Once the server is activated, you may run client applications that use the CORBA application you have just developed. Depending on the type of client application, you might have to complete a few set-up tasks for the client application. The instructions in this book provide you with models or advice to help you with these tasks.

Background on setting up the development and deployment environments

Depending on the roles and responsibilities at your site, your system programmer might have already set up both the application development and deployment environments for you. In this case, you might only have to check that everything is set up correctly. Otherwise, you might have to do some installation work yourself. In either case, this section reviews the minimum requirements and indicates where to find more detailed information, if you need it. You might need the help of your system programmer, security administrator, or database administrator to complete some of these set-up tasks.

The instructions in this book for developing the CORBA applications assume that you are working on a Windows NT workstation with WebSphere for Windows NT and the VisualAge Component Development tools installed. You may, however, develop applications in other workstation environments that WebSphere for z/OS supports. To deploy CORBA applications in a WebSphere for z/OS application server, you will also need to work in the z/OS or OS/390 UNIX environment. For more information about the application development environment, read "Setting up the application development and assembly environment".

For simplicity, the instructions in this book also assume that you will deploy and run CORBA applications and their client programs on the same z/OS or OS/390 system on which you perform application development tasks. (In other words, you compile client and server application code on the same system in which the WebSphere for z/OS application server will run.) When you develop and deploy your own server applications, this client-server configuration might be too simplistic for your needs. When you use more complex configurations, you have to make sure that both the WebSphere for z/OS application server that manages the sample application, and all of the sample application's clients, have access to the sample application code that they need.

To deploy a server application in a WebSphere for z/OS application server, you use the WebSphere for z/OS Administration application, a program that runs on Windows NT, to complete most of the setup required for that application server (or run-time environment). Although the instructions for deploying the sample applications list all required tasks, they do not prescribe how to perform some tasks, such as setting up security, because these tasks vary for each customer installation. For more information about these tasks, read "Setting up the server application run-time environment" on page 13.

Setting up the application development and assembly environment

When system programmers at your site install WebSphere for z/OS, they also have the option of setting up the application development environment. The instructions they receive in *WebSphere Application Server V4.0.1 for z/OS and OS/390: Installation and Customization*, GA22-7834 are listed in "Steps for setting up the application development environment" on page 40, so you may verify or install the correct environment yourself.

To create CORBA applications, you must use the VisualAge Component Development tools that are available with WebSphere for Windows NT 3.5. If you find that you must install some software products, check the information about software requirements in *WebSphere Application Server V4.0.1 for z/OS and OS/390: Installation and Customization*, GA22-7834, to ensure you install the correct product versions.

To create CORBA applications, you might have to perform the following tasks:

Task:	Associated background information or procedure (See ...)
Setting up the workstation environment to generate application artifacts for the z/OS or OS/390 platform.	"Setting workstation tools to generate application artifacts for z/OS or OS/390"
Changing some environment variables for compiling source code on z/OS or OS/390.	"Changing make environment variables to compile source files on z/OS or OS/390"

Setting workstation tools to generate application artifacts for z/OS or OS/390

Depending on the CORBA application you are developing, you will use one or more of the following tools to develop component objects and generate source files for them. For components and code to be deployed on z/OS or OS/390, make sure the tools are set properly, as noted:

VisualAge for Java

Make sure that you:

- Select **Window** → **Options**. Select **Visual Composition** and clear **Inherit BeanInfo of bean superclass**.
- Add the appropriate features for the bean you are creating. Individual sample instructions list the correct features to add.

Object Builder

When you begin a new model, select **Platform** and set:

- **View** to 390
- **Generate** to 390
- **Constrain** to 390

Changing make environment variables to compile source files on z/OS or OS/390

Environment variables for the make process are set in the CB390make.env file that is shipped with the WebSphere for z/OS product. As part of the installation process, your system programmer may have tailored this file for use at your site.

Environment variable values that you should use depend on guidelines for your installation or your individual development environment. If you need further detail about the meaning of specific variables, see the descriptions in "Background on make processing" on page 55.

Setting up the server application run-time environment

The series of figures in "Background on deploying server applications" on page 8 illustrate the steps you complete through the WebSphere for z/OS Administration application to define the server configuration, also known as a conversation. Before you can activate that conversation, however, you also need to complete some manual tasks to set up the run-time environment for that server.

Specifically, you need to create JCL procedures to start the control and server regions. You may copy and modify sample JCL procedures that are shipped with the WebSphere for z/OS product. The instructions in this book tell you where to find the JCL samples and what changes are required; see “Coding JCL procedures to start the WebSphere for z/OS MOFW server” on page 73.

Setting up client applications

When you begin to develop client applications that drive the CORBA applications installed in a WebSphere for z/OS server, see:

- “Background on designing and coding clients for your server applications” on page 91 for information on designing and coding client applications, and
- “Chapter 5. Developing, assembling, and deploying client applications on z/OS or OS/390” on page 87 for information about preparing client applications that use server applications on z/OS or OS/390.

Setting up security controls

Make sure you have set up the appropriate security definitions before you import your server application’s DDL file. Because the input and output files for the import process are associated with the Systems Management server region identity, that user ID must have access to the files:

- If you use data sets for the DDL to be imported, the user ID must have read access to the input data set, and alter access to the output data set.
- If you use HFS files for the DDL, the user ID must have the ability to search the directories to find the input file, the ability to read the input file, and the ability to write to the output file.

Chapter 2. Developing CORBA applications for WebSphere for z/OS

Because of the Application Server's common programming model and the capabilities of its tool set, you can develop reusable and portable business objects, concentrating on business objectives rather than learning the intricacies of programming for a specific platform. When you deploy these objects in WebSphere for z/OS servers, they run in environments that are highly integrated with databases and transactional systems on z/OS or OS/390. Because of this integration, some knowledge of z/OS or OS/390 and its subsystems can help you design efficient applications that are optimized for the z/OS or OS/390 environment.

When you are ready to develop your own CORBA applications to run in WebSphere for z/OS servers, read the topics in this chapter, which contain design considerations, high-level functional descriptions, z/OS or OS/390-specific guidelines, and restrictions or limitations of the current level of WebSphere for z/OS. This information can help you develop applications that get the most out of the unique qualities of service that z/OS or OS/390 provides.

Before you begin: Before you start designing and coding CORBA applications for WebSphere for z/OS servers:

- Consider reviewing the information in "Chapter 1. Getting started with CORBA applications for WebSphere for z/OS" on page 1, which walks you through the entire development, assembly, and deployment processes that you will perform for your own applications. That chapter also contains some introductory information to help you learn the tasks you must complete to develop, assemble, and deploy an application in a WebSphere for z/OS server.
- Have one or more of the following resources available, depending on the type of CORBA application that you are developing:
  - *WebSphere Application Server for OS/390 Component Broker Programming Guide*, SC09–4442, describes the programming model including business objects, data objects, and information about the managed object framework, IDL, and C++ CORBA programming.
  - *WebSphere Application Server for OS/390 Component Broker Programming Reference, Volumes 1 and 2*, SC09–4446 and SC09–4447, contain information about the application programming interfaces (APIs) available to Component Broker application developers.
  - *WebSphere Application Server for OS/390 Component Broker Advanced Programming Guide*, SC09–4443, describes the Component Broker

implementation for the CORBA Object Services and the Component Broker Object Request Broker (including remote method invocation and the dynamic invocation interface (DII) procedures), among other topics.

– *WebSphere Application Server for OS/390 Component Broker Application Development Tools Guide*, SC09–4448, explains how to create and test Component Broker applications using the tools provided in the CBToolkit with a focus on common development scenarios such as inheritance and team development.
– *WebSphere Application Server for OS/390 Component Broker Glossary*, SC09–4450, defines commonly used terms.

The following table shows the subtasks and associated information for developing your own CORBA applications:

Subtask	**Associated information (See . . .)**
Learning how the z/OS or OS/390 environment might affect the design of your application	• "Background on the OS/390 Component Broker transactional environment" on page 17 • "Restrictions for CORBA applications and their clients" on page 24 • "Guidelines for getting the best performance from CORBA applications" on page 25 • "Background on designing and coding clients for your server applications" on page 91
Creating CORBA applications for z/OS or OS/390, using workstation tools	• "Background on setting workstation tools to generate application artifacts for z/OS or OS/390" on page 25 • "Background on required component objects for CORBA applications" on page 26 • "Guidelines for developing CORBA business objects in C++ or Java" on page 26 – "Guidelines for developing CORBA applications that use IMS resources" on page 27 – "Guidelines for developing CORBA applications that use CICS resources" on page 34 • "Background on creating a container" on page 37

Background on the OS/390 Component Broker transactional environment

As part of developing a CORBA application, you define one container for each business object that is part of the that application. You also may define container policies, which dictate how a specific container manages its object, during the development process (through Object Builder). These policy settings, however, have no effect on containers in a OS/390 Component Broker server. To define container policies for server applications deployed on z/OS or OS/390, you use the WebSphere for z/OS Administration application.

One container policy that you set defines the transactional scope for the object that the container manages. OS/390 Component Broker currently supports four transaction policies for CORBA applications, but most applications will use one: Required. With Required, the container either uses the client application's global transaction, or begins a global transaction on behalf of the client. In a global transaction, the OS/390 Component Broker server, z/OS or OS/390 resource recovery services (RRS), and other involved resource managers (such as DB2 and IMS) work together to ensure that the CORBA application's processing is coordinated and treated as an atomic operation. In other words, the application's updates to distributed resources are either all made (committed) or not made (rolled back).

Note: This limitation applies only for WebSphere for z/OS MOFW servers. WebSphere for z/OS J2EE servers, in which only J2EE applications (Enterprise beans, servlets and JavaServer Pages) may be installed, do not require container definitions. The J2EE servers use the attributes set in the deployment descriptors for a given J2EE application or its individual components. For information about J2EE servers, see *WebSphere Application Server V4.0.1 for z/OS and OS/390: Assembling J2EE Applications*, SA22-7836.

The other three policies improve the performance of only specific types of server applications, because these policies simulate the absence of a global transactional environment. In other words, the container does not represent and manage the transaction for an object; instead, the container allows RRS and other z/OS or OS/390 resource managers to manage transactional context. To safely use these three alternative policies, you must thoroughly understand your server application's processing, and must abide by the rules for each policy. Otherwise, your application might not behave as you want or expect it to behave.

Recommendation: Use Required, unless you can guarantee that your server application abides by the rules for one of the other policies, as summarized in Table 1 on page 18. These other policies improve performance only for server applications with specific characteristics and processing. As the WebSphere for z/OS product matures, performance should improve for all applications.

Table 1 presents a summary of the container transaction policies that WebSphere for z/OS MOFW server currently supports:

Table 1. Summary of container transaction policies for MOFW servers, and their usage rules

For this container transaction policy:	Your CORBA application must abide by these rules:
Required (default container policy) The WebSphere for z/OS MOFW server either uses the client application's global transaction, or begins a global transaction on behalf of the client. See Figure 9 on page 20 for an illustration. The server, RRS and other involved resource managers all participate in managing the global transaction. See "Rules for using the Required container policy" on page 19 for more details about server management of the transactional context.	Because the server either uses the client application's global transaction or begins a global transaction, the object cannot issue `begin` to start a new transaction without first issuing `suspend` to suspend the existing global transaction. If you want the business object to manage transactional context, additional rules apply. See "Rules for using the Required container policy" on page 19.
TX MOFW Merged Hybrid Global The WebSphere for z/OS MOFW server disregards the client transaction, if any, and allows RRS and other involved resource managers to manage the hybrid-global transaction. All methods executed in the same container share the same hybrid-global transaction, but methods against objects in other containers might require a different transaction context, depending on the transaction policy specified for the other containers. Also, if methods are driven against objects in different servers, the server does not propagate the hybrid-global transaction to those remote servers. See "Rules for using the TX MOFW Merged Hybrid Global container policy" on page 21 for more details about server management of the transactional context.	If the business object is designed to first issue `begin` to start a transaction, the object cannot require the container to access a protected resource, such as a DB2 database, to obtain initial object state. In this case, the server automatically starts a hybrid-global transaction to retrieve initial state from the persistent data store. Once the server starts a hybrid-global transaction, the server does not permit the object to manage the transactional context. Additional rules apply; for further details, see "Rules for using the TX MOFW Merged Hybrid Global container policy" on page 21.

Table 1. Summary of container transaction policies for MOFW servers, and their usage rules (continued)

For this container transaction policy:	Your CORBA application must abide by these rules:
TX MOFW Isolated Hybrid Global As with the TX MOFW Merged Hybrid Global policy, the WebSphere for z/OS MOFW server disregards the client transaction, if any, and allows RRS and other involved resource managers to manage the global transaction. Also, if methods are driven against objects in different servers, the server does not propagate the hybrid-global transaction to those remote servers. Unlike the TX MOFW Merged Hybrid Global policy, however, each object method executed in the server has its own hybrid-global transaction. See "Rules for using the TX MOFW Isolated Hybrid Global container policy" on page 22 for more details about server management of the transactional context.	If the business object is designed to first issue `begin` to start a transaction, the object cannot require the container to access a protected resource, such as a DB2 database, to obtain initial object state. In this case, the server automatically starts a hybrid-global transaction to retrieve initial state from the persistent data store. Once the server starts a hybrid-global transaction, the server does not permit the object to manage the transactional context. Additional rules apply; for further details, see "Rules for using the TX MOFW Isolated Hybrid Global container policy" on page 22.
TX MOFW Supports Merged Hybrid Global As with the policies other than Required, the WebSphere for z/OS MOFW server allows RRS and other involved resource managers to manage the global transaction. In this case, however, the server honors the client application's transactional context. If the client has a global transaction, all methods executed in the same server instance share that same global transaction; the transactional environment is the same as that for the Required policy.	All objects that the server application uses must be configured to run in the same server instance. Additional rules apply; for further details, see "Rules for using the TX MOFW Supports Merged Hybrid Global container policy" on page 23.

Rules for using the Required container policy

With the Required container policy, the WebSphere for z/OS MOFW server either uses the client application's global transaction, or begins a global transaction on behalf of the client, as illustrated in Figure 9 on page 20. The server propagates this global transaction to remote servers, if the object drives methods against another object that resides in a different WebSphere for z/OS server.

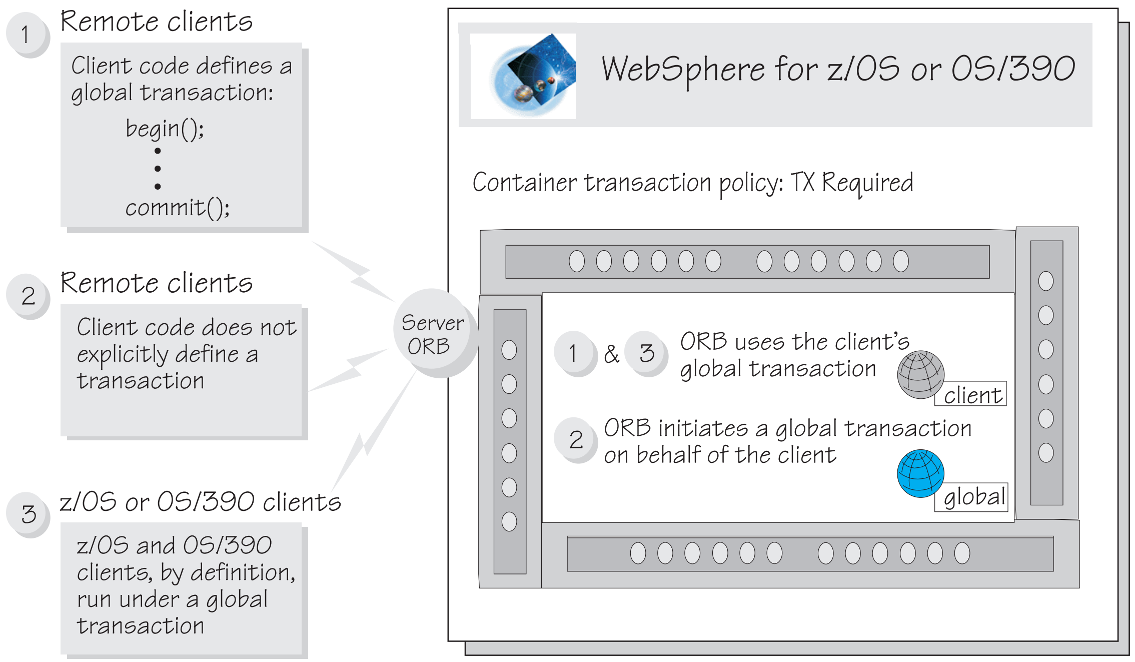
1
Remote clients
Client code defines a global transaction:
begin();
commit();
2
Remote clients
Client code does not explicitly define a transaction
3
z/OS or OS/390 clients
z/OS and OS/390 clients, by definition, run under a global transaction
Server ORB
WebSphere for z/OS or OS/390
Container transaction policy: TX Required
1 & 3 ORB uses the client's global transaction
client
2 ORB initiates a global transaction on behalf of the client
global

Figure 9. The OS/390 Component Broker MOFW server transactional environment for most CORBA applications

If you want the business object to manage transactional context, you must design the object to first suspend the current transaction. When you do so:

- You cannot invoke any managed-object methods before either coding `begin` to start a new transaction, or coding `resume` to resume an existing transaction. If you do not start a new or resume an existing transaction, processing results will be unpredictable.
- You must ensure that the object does not end its processing without first resuming the transaction that was active when the object began executing. If you do not end object processing under the first active transaction, the server region abnormally ends, and the first active transaction is forced to roll back.

Transactional objects are available for the business object to use for managing transactional context. For example, the business object may use the CosTransactions::Control object, which enables an application to represent the transaction context. For more information about transactional objects, see *Component Broker Advanced Programming Guide*.

Because the Required container policy does not permit transactional nesting, you cannot design your server application to perform specific actions when either system or user exceptions occur:

- If a system exception occurs, the OS/390 Component Broker MOFW server rolls back the transaction.
- If a user exception occurs, the server either:
  - Commits the transaction only if you defined this user exception in the interface definition language (IDL) for your server application; or
  - Converts the user exception to a system exception, and rolls back the transaction.

Rules for using the TX MOFW Merged Hybrid Global container policy

With the TX MOFW Merged Hybrid Global container policy, the WebSphere for z/OS MOFW server disregards the client transaction, if any, and allows RRS and other involved resource managers to manage the hybrid-global transaction, as illustrated in Figure 10 on page 22. Note that the initial transaction context set for an object is the same for both the TX MOFW Merged Hybrid Global and TX MOFW Isolated Hybrid Global container policies.

All methods executed in the same container share the same hybrid-global transaction, but methods against objects in other containers might require a different transaction context, depending on the transaction policy specified for the other containers. If methods are driven against objects in different servers, the server does not propagate the hybrid-global transaction to those remote servers.

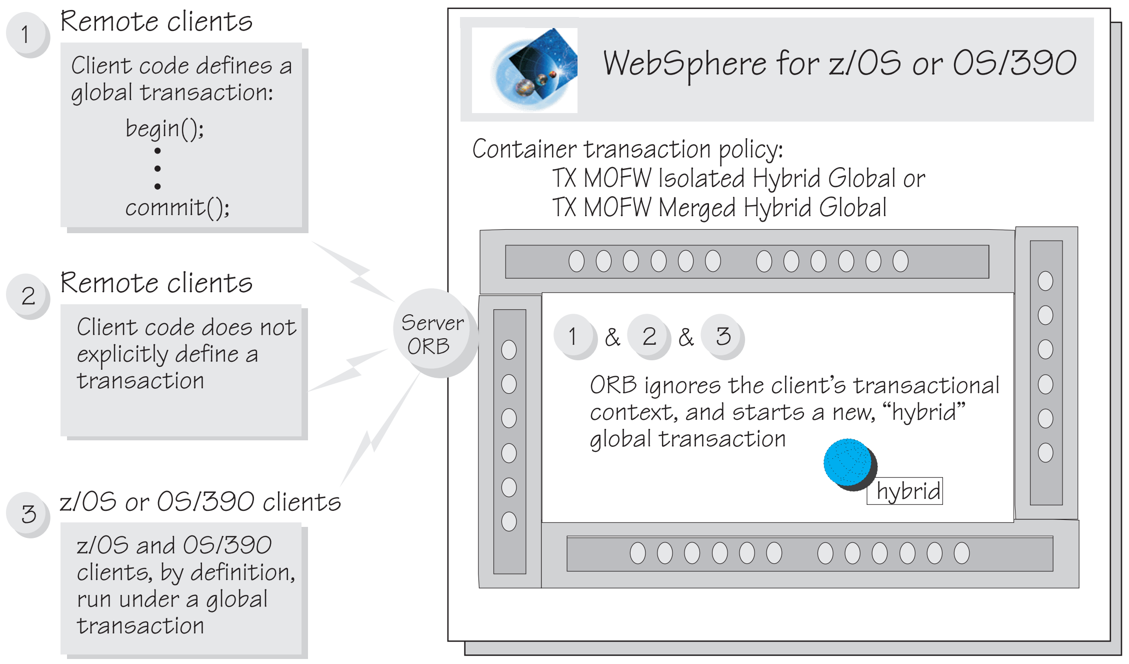
1
Remote clients
Client code defines a global transaction:
begin();
commit();
2
Remote clients
Client code does not explicitly define a transaction
3
z/OS or OS/390 clients
z/OS and OS/390 clients, by definition, run under a global transaction
Server ORB
WebSphere for z/OS or OS/390
Container transaction policy:
TX MOFW Isolated Hybrid Global or
TX MOFW Merged Hybrid Global
1 & 2 & 3
ORB ignores the client's transactional context, and starts a new, "hybrid" global transaction
hybrid

Figure 10. The OS/390 Component Broker MOFW server transactional environment for the TX MOFW Merged Hybrid Global container policy

If the business object is designed to first issue `begin` to start a transaction, the object cannot require the container to access a protected resource, such as a DB2 database, to obtain initial object state. In this case, the server automatically starts a hybrid-global transaction to retrieve initial state from the persistent data store. Once the server starts a hybrid-global transaction, the server does not permit the object to manage the transactional context.

Additionally:

- The server application must be designed to assume that it is running under a global transaction.
- The server application must not attempt to manage its transactional context.
- All objects that interact must be deployed in the same server.

Rules for using the TX MOFW Isolated Hybrid Global container policy

As with the TX MOFW Merged Hybrid Global policy, the WebSphere for z/OS MOFW server disregards the client transaction, if any, and allows RRS and other involved resource managers to manage the global transaction. Also, if methods are driven against objects in different servers, the server does not propagate the hybrid-global transaction to those remote servers. See Figure 10 for an illustration of this initial transactional context, which is the same for both the TX MOFW Merged Hybrid Global and TX MOFW Isolated Hybrid Global container policies.

Unlike the TX MOFW Merged Hybrid Global policy, however, each object method executed in the server has its own hybrid-global transaction.

This policy ensures local-remote transparency; in other words, objects behave the same way regardless of where they are deployed. Using TX MOFW Isolated Hybrid Global provides performance improvements along with maximum flexibility for deploying server application objects.

The TX MOFW Isolated Hybrid Global policy, however, does not provide as much performance improvement as the TX MOFW Merged Hybrid Global policy provides for applications that involve multiple objects. With TX MOFW Merged Hybrid Global, the server can use the same hybrid-global transaction for most objects. In contrast, with the TX MOFW Isolated Hybrid Global policy, the server has additional overhead because it must suspend the current hybrid-global transaction and create a new one, for each object method request.

If the business object is designed to first issue `begin` to start a transaction, the object cannot require the container to access a protected resource, such as a DB2 database, to obtain initial object state. In this case, the server automatically starts a hybrid-global transaction to retrieve initial state from the persistent data store. Once the server starts a hybrid-global transaction, the server does not permit the object to manage the transactional context.

Rules for using the TX MOFW Supports Merged Hybrid Global container policy

As with the policies other than Required, the WebSphere for z/OS MOFW server allows RRS and other involved resource managers to manage the global transaction. In this case, however, the server honors the client application’s transactional context. If the client has a global transaction, all methods executed in the same server instance share that same global transaction. Figure 11 on page 24 illustrates this initial transactional context.

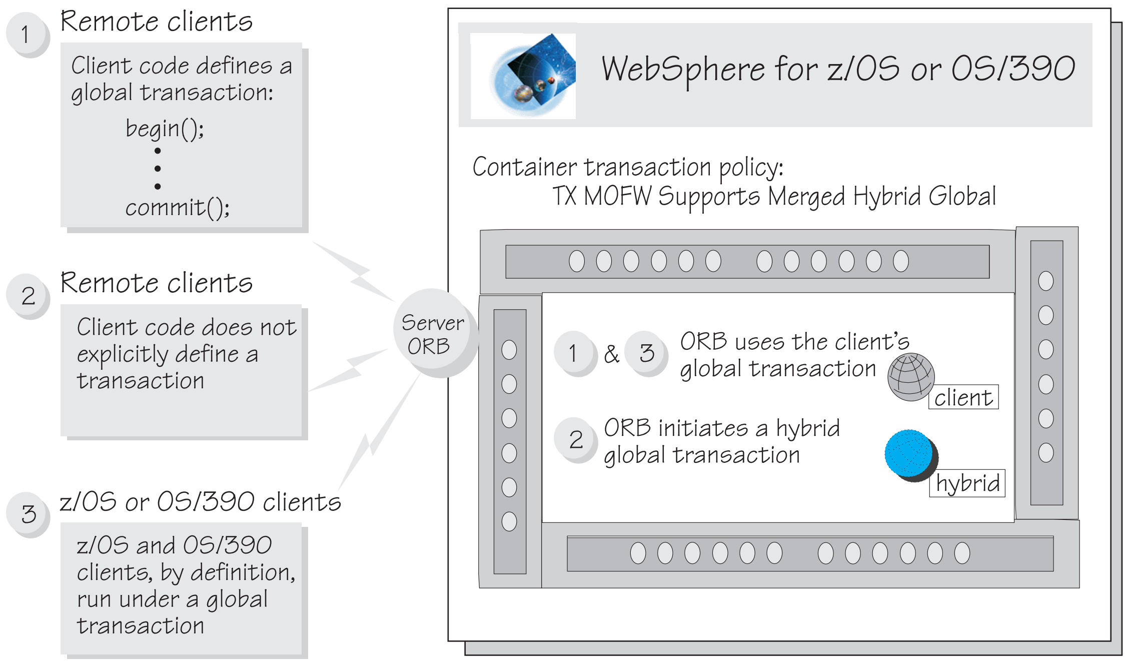
1
Remote clients
Client code defines a global transaction:
begin();
commit();
2
Remote clients
Client code does not explicitly define a transaction
3
z/OS or OS/390 clients
z/OS and OS/390 clients, by definition, run under a global transaction
Server ORB
WebSphere for z/OS or OS/390
Container transaction policy:
TX MOFW Supports Merged Hybrid Global
1 & 3 ORB uses the client's global transaction
client
2 ORB initiates a hybrid global transaction
hybrid

Figure 11. The OS/390 Component Broker MOFW server transactional environment for the TX MOFW Supports Merged Hybrid Global container policy

All objects that the server application uses must be configured to run in the same server instance. Additionally, the server application:

- Assumes that it is running under a global transaction, and
- Does not attempt to manage its transactional context.

Restrictions for CORBA applications and their clients

- Do not use z/OS or OS/390 Language Environment CEESETL callable service in your server application. This service, which is analogous to the C language function setlocale(), establishes a global locale operating environment. If your server application alters the global locale environment that WebSphere for z/OS sets for its application servers, results will be unpredictable.
- If you plan to run client applications on z/OS or OS/390, IBM strongly recommends that you design those clients to use the same code page that the OS/390 Component Broker MOFW servers use: EBCDIC IBM 1047. Other EBCDIC code pages might work, but results will be unpredictable.

  Notes:

  1. This restriction applies only to clients that use applications running in a MOFW server.

2. Client applications that run on non-z/OS or OS/390 platforms may use any code page.

Guidelines for getting the best performance from CORBA applications

- Install the server portion of your business applications on the same system in the network as the data store that the server application uses. Thus, for business applications requiring z/OS or OS/390 data stores, locate the server portions on WebSphere for z/OS servers.
- Because of the number of address spaces that may be required for your business application servers, place load libraries in the link pack area (LPA). IBM supplies parmlibs to help you do this. For further details, see the section about placing modules in the system's search order, in*z/OS MVS Initialization and Tuning Guide*, SA22-7591.

Background on setting workstation tools to generate application artifacts for z/OS or OS/390

Depending on the type of CORBA application you are developing, you will use one or more of the following workstation tools to develop component objects and generate source files for them. For components to be deployed in a WebSphere for z/OS server, make sure the workstation tools are set properly, as noted:

VisualAge for Java

Make sure that you:
- Select **Window** → **Options**. Select **Visual Composition** and clear **Inherit BeanInfo of bean superclass**.
- Add the appropriate features for the bean you are creating. Individual sample instructions list the correct features to add.

Object Builder

When you begin a new model, select **Platform** and set:
- **View** to 390
- **Generate** to 390
- **Constrain** to 390

Also, if you plan to use the IBM Distributed Debugger to provide trace information or debugging support, you need to set a default configuration option before you generate make files for your server application. To set this option:

1. Select **File** → **Preferences**
2. Select **Makefile Generation** under the Tasks and Objects folder
3. Click on the radio button for either **Trace build** or **Trace and debug build**
4. Click **OK**

Background on required component objects for CORBA applications

CORBA server applications consist of a collection of component objects, some of which are automatically generated for you, based on selections you make when using VisualAge for Java or Object Builder to develop your application. The number and type of component objects required depends on the type of server application you are developing, as you can see from Table 2.

Table 2. Summary of component objects for CORBA server applications

If your server application:	**Then the following component objects are required:**					
	Procedural adapter object	**Business object**	**Managed object**	**Data object**	**Persistent object**	**Schema**
Does not use any backing resource		✓	✓			
Uses CICS or IMS resources	✓	✓	✓	✓	✓	✓
Uses DB2		✓	✓	✓	✓	✓

Guidelines for developing CORBA business objects in C++ or Java

Because the WebSphere Application Server for z/OS and OS/390 Component Broker products define a common programming model, most of the information about designing and coding server applications appears in the following books:

- *Component Broker Programming Guide*
- *Component Broker Advanced Programming Guide*
- *Component Broker Programming Reference*
- *Component Broker Application Development Tools Guide*

These manuals define the concepts, coding practices, programming interfaces, and tools you need to understand to design and code WebSphere for z/OS business objects in C++ or Java. These manuals also note which concepts or interfaces do not apply for the z/OS or OS/390 platform

Note that your CORBA server application might also function as an Application Server client. In other words, the business object you code to run in a WebSphere for z/OS MOFW server can itself:

- Find or create, use, and delete other business objects, which may reside on any WebSphere for z/OS platform.
- If written in Java, find or create, use, and delete Enterprise beans running in WebSphere for z/OS J2EE servers.

In addition to using the common WebSphere Application Server programming books, you also need to be familiar with the information in the following topics, depending on the type of server application you are developing:

- "Guidelines for developing CORBA applications that use IMS resources"
- "Guidelines for developing CORBA applications that use CICS resources" on page 34

Guidelines for developing CORBA applications that use IMS resources

Any CORBA server applications that you develop will require these components: the business object and its associated classes, and a procedural adapter (PA) bean and its associated classes. Developing the PA bean is perhaps the most challenging part of creating a server application that accesses IMS resources, because creating the bean classes requires some knowledge of IMS and of the transaction that runs under IMS.

WebSphere for z/OS provides access to IMS resources through procedural application adapters: one that uses Open Transaction Manager Access (OTMA) to communicate with IMS, another that uses the advanced program-to-program communication component of MVS (APPC/MVS). These adapters enable a WebSphere for z/OS MOFW application server to communicate with an IMS subsystem; the PA bean provides the input and output those adapters use for communication with IMS. Your installation probably has either installed IMS with OTMA or established APPC connectivity to IMS. Check with your system programmer to determine which configuration your installation uses.

The following tables shows the subtasks and associated guidelines for developing a server application that accesses IMS resources:

Subtask	**Associated procedure (See . . .)**
Understanding how to design your application for a specific IMS procedural application adapter	• "Guidelines for designing for an IMS–OTMA adapter" on page 28 • "Guidelines for designing for an IMS–APPC adapter" on page 28
Understanding how IMS processes requests	"Background on IMS request processing" on page 29

Subtask	Associated procedure (See . . .)
Understanding how to code the PA bean and its associated classes	• "Background on coding the PAO and its associated classes using IBM VisualAge Java with EAB" on page 30 • "Steps for creating a COBOL file that defines input/output user data" on page 31 • "Background on coding PA bean mapper and information classes" on page 33 • "Background on coding PA bean command classes" on page 34

Guidelines for designing for an IMS–OTMA adapter

When communicating with a target transaction program in IMS, you may use only SendReceive requests. OS/390 Component Broker does not support requests to do Send-only or Receive-only processing with an IMS transaction program.

Guidelines for designing for an IMS–APPC adapter

For server applications to access IMS resources, your installation defines a OS/390 Component Broker MOFW server with an IMS-APPC logical resource mapping (LRM). With this configuration, the server uses APPC/MVS as its method of communicating with the IMS subsystem. In other words, the server allocates an APPC/MVS conversation with the IMS subsystem.

When defining an IMS-APPC LRM, your installation selects an APPC synchronization (sync) level, which determines the type of APPC/MVS conversation to use for communication; any conversation that the server allocates is either a protected or unprotected resource. The type of conversation, protected or unprotected, depends on two factors:

- The type of processing your application does, as outlined in Table 3 on page 29, and
- The transaction policy for the container (or containers) to which the IMS-APPC LRM is connected.

Recommendation: For better performance, design your CORBA application to require protected conversations only when necessary. Perhaps your application does not fit nicely into one of the two categories in Table 3 on page 29 because, for example, it both reads data and later reads data again, manipulates that data, and stores the result. In this case, you could design your server application to use two objects: one for read-only processing; another for the read-update-and-store processing. With this design, each object would have its own container connected to the same IMS-APPC LRM, and each container’s transaction policy would dictate whether the server uses a protected or unprotected conversation.

Table 3. The APPC/MVS conversation type and appropriate server application processing

This APPC/MVS conversation type:	Is best suited for this application processing:
Protected. For protected conversations, APPC/MVS, the OS/390 Component Broker MOFW server, IMS, and other system components work together to ensure that an application’s updates to distributed resources are coordinated, and treated as an atomic operation. In other words, the application’s updates are either all made (committed) or not made (rolled back).	Application processing for which data integrity is critical. For example, if your server application transfers money between checking and savings accounts, those transfers ought to be considered a single operation to ensure the integrity of the account balances.
Unprotected. Unprotected conversations have none of the overhead involved with coordinating an application’s processing, so application performance is faster. If conversation errors or failures occur, however, the resources an application uses might be in inconsistent states.	Application processing for which performance is more important than data integrity. For example, if a server application performs read-only requests to retrieve data, and does not rely on the value of that data remaining constant, that application is better suited for this environment because it has no dependency on data integrity.

Another design guideline concerns the type of request your server application sends to an IMS transaction program. When communicating with a target transaction program in IMS, you may use only send-receive requests. OS/390 Component Broker does not support requests to do send-only or receive-only processing with an IMS transaction program.

Background on IMS request processing

When it receives a request to run a transaction, IMS builds an input message for the request, using an 8-byte field to contain the name of the requested transaction **plus one blank**. This action causes no problems for the original customers of IMS: 3270 terminals and emulators. For you, however, this behavior might misalign user data that you want to pass to the requested IMS transaction. As you can see from Figure 12 on page 30, problems will occur if the name of the IMS transaction is anything other than 7 bytes in length:

- If the transaction name is less than 7 bytes, IMS will fill the remaining bytes of the 8-byte name field with the first bytes of your user data.
- If the transaction name is greater than 7 bytes, IMS places a blank at the offset where the IMS transaction expects to find the start of user data.

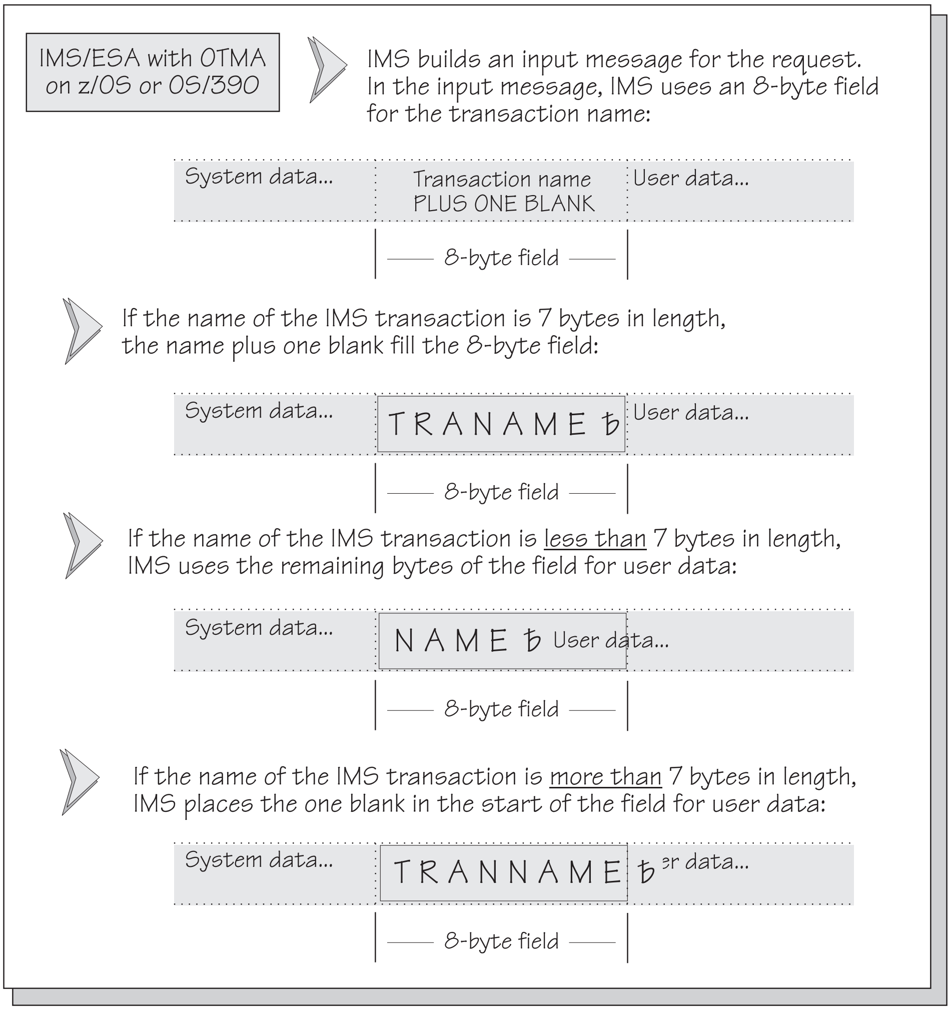
IMS/ESA with OTMA on z/OS or OS/390
IMS builds an input message for the request. In the input message, IMS uses an 8-byte field for the transaction name:
System data...
Transaction name PLUS ONE BLANK
User data...
8-byte field
If the name of the IMS transaction is 7 bytes in length, the name plus one blank fill the 8-byte field:
System data...
TRANAME b
User data...
8-byte field
If the name of the IMS transaction is less than 7 bytes in length, IMS uses the remaining bytes of the field for user data:
System data...
NAME b User data...
8-byte field
If the name of the IMS transaction is more than 7 bytes in length, IMS places the one blank in the start of the field for user data:
System data...
TRANNAME b
er data...
8-byte field

Figure 12. How IMS receives requests from an OS/390 Component Broker MOFW application server

In either case, results will not be what you expect. Fortunately, you can easily avoid the misalignment that occurs for transaction names that are less than 7 bytes long, by padding your user data with the correct amount of "fill." Instructions for calculating fill are covered in “Steps for creating a COBOL file that defines input/output user data” on page 31.

Background on coding the PAO and its associated classes using IBM VisualAge Java with EAB

When you create a CORBA application that uses the IMS or CICS procedural application adaptors, the first step is to create a procedural adapter (PA) bean using IBM VisualAge Java with EAB. You then import that PA bean into Object Builder to create the other component objects of your server application. A PA bean consists of several classes, which are illustrated in Figure 5 on page 7. As you can see from that illustration:

- The procedural adapter object (PAO) represents the essential state of a business object (that is, user data to be placed in permanent storage).
- The input or output information classes define the structure and content of data that the backing IMS or CICS transaction.
- The mapper classes of the PA bean enable the transfer of data between the PAO and the input or output information classes.

Steps for creating a COBOL file that defines input/output user data: When you are creating a PA bean for your own server application, you need to look at the existing IMS transaction's communication (or data) area to create a COBOL file that defines the area's structure and contents. VisualAge for Java uses this file to correctly generate the code for the input information and output information classes. These classes are critical elements of the PA bean— they provide the mechanism for presenting business object essential state (that is, user data to be placed in permanent storage) in a format that the IMS transaction can use.

Recommendation: Many IMS transactions use the same data area for both input and output, so the field names and characteristics are the same for both input and output. For your PA bean, however, you should define one set of fields for input, and a separate set of fields— with different names— for output. If you use the same field names for both input and output, VisualAge for Java automatically generates unique field names for the second set that you define. You may allow VisualAge for Java to generate unique names, but then you must remember those names, and remember to use them in the code you write for the insert(), retrieve(), update(), and del() methods of the PA bean.

Before you begin: You need to have access to the source code for the IMS transaction, specifically the code that defines the communication area that the transaction uses for input and output.

Perform the following steps to create a COBOL file to import into VisualAge for Java:

1. Create your own or copy a sample COBOL file into a working directory on your workstation.

---

2. If the name of the IMS transaction you want to use is less than 7 bytes long, calculate the amount of fill required for IMS to correctly align your user data. To calculate the fill:

- Look in the source code to find the size of the input field used for the transaction name. (Generally, this field is larger than necessary to fit the actual transaction name.)
- Use the following algorithm to calculate the amount of fill:

  ```
  Size_of_tran_name_field - (size_of_tran_name + 1) = size_of_fill
  ```

  For example, suppose the transaction you want to use is named PCTIA, and the size of the input field for the transaction name is 8 bytes. In this case, the calculation is:

  ```
  8 - (5 + 1) = 2
  ```

  For this example, you need to define 2 bytes of fill to correctly align the user data.

---

3. Edit the COBOL file in your working directory to add COBOL record definitions for the user data and, if necessary, the required fill. The following sample illustrates the COBOL record definitions you would need for the data area of an IMS transaction written in PL/1. Note that fill is required for the input data, but not for the output data:

For a PL/1 data structure like this:	**Use the following COBOL record definitions:**
Input data structure:	

`DCL 1 IN_MESSAGE STATIC,`
`    2 LL     BIN FIXED(31),`
`    2 ZZ     CHAR(2),`
`    2 TRANID CHAR(8),`
`    2 ITEMID CHAR(6),`
`    2 PRICE  CHAR(6);`
	`01 INPUT-MSG.`
	`   02 INFILL PICTURE X(2).`
	`   02 IN_ITEMID  PIC X(6).`
	`   02 IN_PRICE   PIC X(6).`

Output data structure:	

`DCL 1 OUT_MESSAGE BASED...,`
`    2 LL      BIN FIXED(31),`
`    2 ZZ      CHAR(2),`
`    2 DATE    CHAR(19),`
`    2 ITEMID  CHAR(6),`
`    2 PRICE   PICTURE'999V.99',`
`    2 ERRMSG  CHAR(78),`
`    2 ERRMSG2 CHAR(78);`
	`01 OUTPUT-MSG.`
	`   02  OUT_DATE PIC X(19).`
	`   02  OUT_ITEMID PIC X(6).`
	`   02  OUT_PRICE X(6).`
	`   02  OUT_ERRMSG PIC X(78).`
	`   02  OUT_ERRMSG2 PIC X(78).`

You know you are done when you have created a COBOL record definition for each user-data field in the transaction's data area. When you import this file, VisualAge for Java will correctly generate the input and output information classes for your PA bean to use.

Background on coding PA bean mapper and information classes: The mapper classes need to connect a PAO attribute to the corresponding field in the input or output information classes. So, when you use VisualAge for Java to create mapper classes, you are actually defining only the PAO attributes, and creating a link from each attribute to its corresponding field in the input and output information classes.

Before you can create the information and mapper classes, you need to look at the existing IMS or CICS transaction's communication (or data) area, and create a COBOL file that defines the area's structure and contents. VisualAge for Java uses this file to correctly generate the code for the input information and output information classes. How many classes you need to create depends on how the existing IMS or CICS transaction works:

If the IMS or CICS transaction works this way:	Then create the following:
The IMS or CICS transaction uses a single communication area to handle both input and output	• A COBOL file with one record that defines the structure and content of both input and output • A single information class that matches the COBOL file record • A single mapper class that matches the information class **Warning:** If the IMS or CICS transaction uses a single communication area, but you create separate input and output information classes and you use the same field names for both input and output, VisualAge for Java automatically generates unique field names for the second set that you define. You may allow VisualAge for Java to generate unique names, but then you must remember those names, and remember to use them in the code you write for the insert(), retrieve(), update(), and del() methods of the PAO.

If the IMS or CICS transaction works this way:	Then create the following:
The IMS or CICS transaction uses separate communication areas: one to handle input; another to handle output	• A COBOL file with two records: one that defines the structure and content of input; another that defines output • Two information classes: one that matches the COBOL file record for input; another that matches the record for output • Two mapper classes: one that matches the input information class; another that matches the output information class **Recommendation:** Whenever you create separate input and output information classes, make these selections to improve performance: • Select **custom records** when you specify the record style for the output information class • Deselect the **Generate with notification** option for the output information class

Background on coding PA bean command classes: The PA bean command classes wrap a single interaction with IMS; in other words, these classes package a request for the OS/390 Component Broker application server to send to IMS, or to format the result the server receives from IMS. When you use VisualAge for Java to create the command classes, you specify several key pieces of information:

The interaction spec
`IMSOTMAInteractionSpec` or `IMSAPPCInteractionSpec`

The connector
`com.ibm.connector.CB390.IMSOTMA.InteractionSpec` or
`com.ibm.connector.CB390.IMSAPPC.InteractionSpec`

Guidelines for developing CORBA applications that use CICS resources

When you create a CORBA application that uses the IMS or CICS procedural application adaptors, the first step is to create a procedural adapter (PA) bean using IBM VisualAge Java with EAB. You then import that PA bean into Object Builder to create the other component objects of your server application. A PA bean consists of several classes, which are illustrated in Figure 5 on page 7. As you can see from that illustration:

- The procedural adapter object (PAO) represents the essential state of a business object (that is, user data to be placed in permanent storage).
- The input or output information classes define the structure and content of data that the backing IMS or CICS transaction.

- The mapper classes of the PA bean enable the transfer of data between the PAO and the input or output information classes.

Background on coding PA bean mapper and information classes

The mapper classes need to connect a PAO attribute to the corresponding field in the input or output information classes. So, when you use VisualAge for Java to create mapper classes, you are actually defining only the PAO attributes, and creating a link from each attribute to its corresponding field in the input and output information classes.

Before you can create the information and mapper classes, you need to look at the existing IMS or CICS transaction's communication (or data) area, and create a COBOL file that defines the area's structure and contents. VisualAge for Java uses this file to correctly generate the code for the input information and output information classes. How many classes you need to create depends on how the existing IMS or CICS transaction works:

If the IMS or CICS transaction works this way:	**Then create the following:**
The IMS or CICS transaction uses a single communication area to handle both input and output	• A COBOL file with one record that defines the structure and content of both input and output • A single information class that matches the COBOL file record • A single mapper class that matches the information class **Warning:** If the IMS or CICS transaction uses a single communication area, but you create separate input and output information classes and you use the same field names for both input and output, VisualAge for Java automatically generates unique field names for the second set that you define. You may allow VisualAge for Java to generate unique names, but then you must remember those names, and remember to use them in the code you write for the insert(), retrieve(), update(), and del() methods of the PAO.

If the IMS or CICS transaction works this way:	Then create the following:
The IMS or CICS transaction uses separate communication areas: one to handle input; another to handle output	• A COBOL file with two records: one that defines the structure and content of input; another that defines output • Two information classes: one that matches the COBOL file record for input; another that matches the record for output • Two mapper classes: one that matches the input information class; another that matches the output information class **Recommendation:** Whenever you create separate input and output information classes, make these selections to improve performance: • Select **custom records** when you specify the record style for the output information class • Deselect the **Generate with notification** option for the output information class

Background on coding PA bean command classes

The PA bean command class wraps a single interaction with CICS; in other words, this class packages a request for the WebSphere for z/OS MOFW application server to send to CICS, or formats the result the server receives from CICS. When you use VisualAge for Java to create the command class, you specify several key pieces of information:

The interaction spec
: `CICSEXCIInteractionSpec`

The connector
: `com.ibm.connector.CB390.CICSEXCI.InteractionSpec`

The program name
: In the **pgmName** field, you enter the 8-character name of the program you want to run in the CICS region. This program name also must be specified on a predefined PROGRAM resource definition installed in the CICS region, or defined to a user-written autoinstall program.

The transaction ID
: In the CashAcct sample, you must make sure that the **transID** field is null, which specifies the default behavior for CICS processing. With this default behavior, the program you specified in the **pgmName** field runs under the CICS-supplied mirror transaction, CSMI. When you are developing a PA bean for your own server application, however, you might want the program to run under a different mirror transaction. If this is the case, see *CICS External Interfaces Guide*, SC34-5709 for requirements for specifying the transID.

Background on creating a container

When you use Object Builder to develop a server application to be deployed in a WebSphere for z/OS server, you need to associate a container with the application. Through Object Builder, provide only a container name. To define container properties, you must use the WebSphere for z/OS Administration application.

Note: This limitation applies only for WebSphere for z/OS MOFW servers. WebSphere for z/OS J2EE servers, in which only J2EE applications (Enterprise beans, servlets and JavaServer Pages) may be installed, do not require container definitions. The J2EE servers use the attributes set in the deployment descriptors for a given J2EE application or its individual components. For information about J2EE servers, see *WebSphere Application Server V4.0.1 for z/OS and OS/390: Assembling J2EE Applications*, SA22-7836.

Guideline: When you use Object Builder to configure an existing application for the z/OS or OS/390 platform, and that application is already associated with a default container, always replace that default by defining a new container. The default containers are designed for use only in WebSphere Application Server servers on workstation platforms. WebSphere for z/OS allows you to deploy applications associated with these default containers, but the resulting run-time environment might not match that on the workstation platforms.

For further details about using the WebSphere for z/OS Administration application to define containers for MOFW servers, see “Defining containers for MOFW servers” on page 75.

Chapter 3. Assembling CORBA applications on z/OS or OS/390

To assemble the portable and platform-specific components of a CORBA server application, programmers compile source files generated on the workstation into executable code that can run in a WebSphere for z/OS MOFW server.

To accomplish this work, programmers need some expertise with both workstation and z/OS or OS/390 application development. Specifically, they need:

- Some understanding of the process, tools, and output required for developing CORBA application components on the workstation
- Familiarity with software products for file transfer between the workstation and z/OS or OS/390
- Familiarity with working in the UNIX System Server (USS) environment, which includes using the hierarchical file system (HFS), setting environment variables, and using the `make` command to compile code, and so on.

The following table shows the subtasks and associated information for assembling a CORBA application.

Subtask	**Associated information (See . . .)**
Setting up the assembly environment on z/OS or OS/390	"Steps for setting up the application development environment" on page 40
Working with CORBA application files generated on the workstation	• "Steps for creating an HFS directory structure for CORBA application files from the workstation" on page 46 • "Steps for transferring files from the workstation to z/OS or OS/390" on page 47
Preparing to compile CORBA application files on z/OS or OS/390	• "Background on deciding where to place executable code for the server application" on page 49 • "Background on allocating data sets for the CORBA application's executable code" on page 53 • "Background on make processing" on page 55
Compiling CORBA application files into executable code	"Steps for compiling CORBA application source files on z/OS or OS/390" on page 64

Subtask	Associated information (See . . .)
Preparing the CORBA application for use	• “Steps for adding your CORBA application to the system search path” on page 67 • “Steps for binding data objects for your CORBA application” on page 68

Steps for setting up the application development environment

When system programmers at your site install WebSphere for z/OS, they also have the option of setting up the UNIX application development environment z/OS or OS/390. The instructions they receive in *WebSphere Application Server V4.0.1 for z/OS and OS/390: Installation and Customization*, GA22-7834 are listed here, so you may verify the correct environment yourself. If necessary, check *WebSphere Application Server V4.0.1 for z/OS and OS/390: Installation and Customization*, GA22-7834 for the software product versions and other requirements for both the z/OS or OS/390 UNIX and the workstation application environments.

Before you begin: You must have your Windows NT and OS/390 UNIX systems configured.

Follow these steps to set up the application development environment:

1. On Windows NT:
   a. For each application developer, install the WebSphere for Windows NT run-time and development environments. For installation instructions, see *Component Broker Quick Beginnings*, G04L-2375.
   b. IBM recommends you install an NFS client or equivalent on each application developer’s workstation. As an alternative, you may use the SAMBA client from MKS.
   c. Install Personal Communications/3270 (or equivalent host emulator software).

2. On z/OS or OS/390:
   a. Assure the Java JDK is configured.
   b. Assure that the C++ compiler is enabled.
   c. Assure that the Debug Tool data set is catalogued and added to the link list.
   d. Install the NFS server or equivalent. An alternative is the SAMBA server from MKS.

e. Allocate at least 100 cylinders of HFS space in the home directory for each application developer. You can maintain the application development storage the same way you maintain OS/390 or z/OS UNIX HFS storage.

f. Assure that each application developer has a TSO/E user ID that is authorized to use OS/390 or z/OS UNIX.

g. Check the number of file descriptor files defined (MAXFILEPROC statement in the BPXPRMxx parmlib member). You may need additional file descriptor files when you compile programs.

h. Check the maximum number of processes allowed (MAXPROCUSER statement in the BPXPRMxx parmlib member). You may need to add to the limit when you run makes.

i. Check each application developer's region size (MAXASSIZE in BPXPRMxx or ASSIZEMAX on the RACF ADDUSER or ALTUSER commands). The rule of thumb is to run with the largest region size possible, but start with a minimum size of 256 MB. The size can be limited by the IEFUSI exit, JES2 EXIT06, JES3 IATUX03, or TSO segment defaults. If the compiler runs out of memory, you may need to increase the application developer's region size.

For more information on BPXPRMxx, see *z/OS UNIX System Services Planning*, GA22-7800.

---

3. On z/OS or OS/390, check the CB390make.env file in `/usr/lpp/WebSphere390/CB390/samples`. Base what you do on the following table:

If . . .	**Then . . .**	**Notes**
You installed WebSphere for z/OS into the prescribed directories	Make no modifications to CB390make.env	

If . . .	Then . . .	Notes
You installed WebSphere for z/OS into directories other than those prescribed	Either: a. Copy CB390make.env and customize it b. Identify the new location through the CB390_ENVFILE variable in your .profile file. Example: `export CB390_ENVFILE=/etc/CB390make.env`	If IBM does maintenance on CB390make.env, you may need to make maintenance changes to your copied version.
	Or: Override the environment variables described in Table 4 through your .profile file.	If IBM does maintenance on CB390make.env, you may need to make changes to the export statements in your .profile file.

4. On z/OS or OS/390, set a system-wide default profile (/etc/profile) for the location of WebSphere for z/OS files.

   Table 4 describes the environment variables used for CORBA applications you develop for WebSphere for z/OS.

Table 4. General environment variables for MOFW component developers

Variable	Notes
CB390_ENVFILE=*path*	Location of the CB390make.env file. Default: `/usr/lpp/WebSphere390/CB390/samples/CB390make.env`
CB390_STDINC=*path*	Location of include files. Default: `/usr/include //'CBC.SCLBH.+'`
CLASSPATH=*path*	**For MOFW C++ clients,** you do not need to specify anything.
	For MOFW Java clients, specify `ws390crt.jar`
	For Java business objects in the server, specify `ws390srt.jar`
LIBPATH=*path*	Change the LIBPATH environment variable to include `/usr/lpp/WebSphere390/CB390/lib`.
IVB_DRIVER_PATH=*path*	Location of WebSphere for z/OS product files. Default: `/usr/lpp/WebSphere390/CB390/`
PATH=*path*	Change the PATH environment variable to include `/usr/lpp/WebSphere390/CB390/bin`. For tracing and debugging Java on z/OS or OS/390 include the path to the executable called irmtdbgj.

Table 4. General environment variables for MOFW component developers (continued)

Variable	Notes
SBBOEXEC_DSN= *DATA_SET_NAME*	Location of REXX EXECs in SBBOEXEC. Default: `BBO.SBBOEXEC`

Table 4 on page 42 lists the default values for environment variables that point to WebSphere for z/OS files. If the system programmers at your site installed the Application Server into directories other than those that IBM prescribes, they may have changed the /etc/profile to identify the location of WebSphere for z/OS files for all z/OS or OS/390 shell users. If they did not update this system-wide default profile to set the location of WebSphere for z/OS files, you need to find out where the product files are located, and use your $HOME/.profile file or a shell script to change the values of these environment variables accordingly.

Recommendation: If the environment variables in Table 4 on page 42 are not set in the /etc/profile, set them in your $HOME/.profile file. These environment variables should have the same settings for all CORBA applications you develop for WebSphere for z/OS.

Sample: The sample profile in Figure 13 on page 44 defines system-wide variables that may be copied into the /etc/profile. This sample not only sets the environment variables in Table 4 on page 42, but also includes settings that are required to correctly operate the C++ compiler. This sample /etc/profile is a variation of the IBM-supplied /samples/profile, which is described in more detail in *z/OS UNIX System Services Planning*, GA22-7800.

Notes for the C++ customization section:

1. The environment variables in the C++ customization section provide information to the c89/cc/c++ utilities, such as parts of names for dynamically allocated data sets.
2. If installation of the compiler, run-time library products, or both, use different values, then set the appropriate export lines to the correct value. Note that because the c89/cc/c++ utilities do not support a VOL=SER= parameter, you must catalog all named data sets that c89/cc/c++ use.
3. You might have to override the default esoteric unit for unnamed work data sets, if the c89/cc/c++ default (SYSDA) is not defined for the installed system. You may specify a null ("") value to allow c89/cc/c++ to use an installation-defined default.

4. Only the c89 command variables are explicitly shown. The cc and c++ variables are set by the command line beginning with `eval` at the end of this customization section.
5. This is not an exhaustive list of the environment variables that affect the behavior of c89/cc/c++. It is, however, all those that will normally might require customization by the system programmer.

   Recommendation: For easier migration, set only the variables for correct operation of the c89/cc/c++ utilities.

```
# To enable and disable lines in this profile you may remove or add  '#'
# to uncomment or comment the desired lines.
#
# Export the values so the system will have access to them.
#
# Improve the shell's performance for users from ISPF or with
# STEPLIB data sets allocated.  This performance improvement is not
# applicable to non-interactive shells, for example those started with
# the BPXBATCH and OSHELL utilities.
if [ -z "$STEPLIB" ] && tty -s;
then
    export STEPLIB=none
    exec sh -L
fi
#
# Set the time zone as appropriate.
TZ=EST5EDT
export TZ
#
# Set the language
LANG=C
export LANG
#
# Set a default command path, including your current working
# directory.
# PATH=/bin:.:/usr/lpp/java/J1.3/bin
PATH=/bin:.:/usr/lpp/java/J1.3/bin:/usr/lpp/WebSphere390/CB390/bin
export PATH
#
# Specify the directory to search for a DLL (Dynamic Link Library)
# filename. If not set, the working directory is searched. In the
# sample below, db2_install_path is the HFS where you installed DB2 for OS/390.
LIBPATH=/db2_install_path/lib:/usr/lpp/java/J1.3/bin:/usr/lpp/java/J1.3/bin/classic:/usr/lpp/WebSphere390
export LIBPATH
#
# Set the path for NLS files (message catalogs).
NLSPATH=/usr/lib/nls/msg/%L/%N
export NLSPATH
#
```

Figure 13. Sample /etc/profile

```
# Set the path for man pages (help files).
MANPATH=/usr/man/%L
export MANPATH
#
# Set the name of the system mail file and enable mail notification.
MAIL=/usr/mail/$LOGNAME
export MAIL
#
# Set the default file creation mask
umask 022
#
# Set the LOGNAME variable readonly so it is not accidentally modified.
readonly LOGNAME
#
# ========================================================================
# Start of c89/cc/c++ customization section
#
# High-Level Qualifier "prefixes" for data sets used by c89/cc/c++:
#
#   C/C++ Compiler:
#   ----------------------------------------
export _C89_CLIB_PREFIX="SYS1.CPP"
#
#
#   Prelinker and run-time library:
#   ----------------------------------------
export _C89_PLIB_PREFIX="SYS1.LEMVS"
#
#
#   OS/390 system data sets:
#   ----------------------------------------
export _C89_SLIB_PREFIX="SYS1"
#
#
# Esoteric unit for data sets:
#
#
#   Unit for (unnamed) work data sets:
#   ----------------------------------------
export _C89_WORK_UNIT="SYSALLDA"
#
#
# Commands to propogate c89 environment variables for cc and c++:
# =================================================================
#
eval "export $(typeset -x | grep "ˆ_C89_" | awk '{sub("_C89_","_CC_");printf
   "%s ",$0}')"
eval "export $(typeset -x | grep "ˆ_C89_" | awk '{sub("_C89_","_CXX_");printf
   "%s ",$0}')"
#
# End of c89/cc/c++ customization section
# ========================================================================
#
```

Figure 14. Sample /etc/profile, continued

```
# ======================================================================
# Start of WebSphere for z/OS customization section
#
# Provide the name of the WebSphere for z/OS environmental file
export CB390_ENVFILE=/usr/lpp/WebSphere390/CB390/samples/CB390make.env
#
# Provide system names for the include libraries, override taken.
export CB390_STDINC="/usr/include //'SYS1.CPP.SCLBH.+'"
#
# Provide the root structure for WebSphere for z/OS tree, default taken,
export CB390_ROOT=
#
# Provide the name for the REXX exec, override taken.
export SBBOEXEC_DSN=CB390.CB11002.SBBOEXEC
#
# Provide the path to the bin/obmdll20.mk, default taken
export IVB_DRIVER_PATH=/usr/lpp/WebSphere390/CB390/
#
#
# End of CB390 customization section
# ======================================================================
```

Figure 15. Sample /etc/profile, continued

Steps for creating an HFS directory structure for CORBA application files from the workstation

One of the products that IBM recommends for your application development environment is NFS, which allows you to access the HFS as a local drive on your workstation. If you are using NFS or an equivalent product, you may automatically place the files you create through workstation tools on OS/390, and the directory structure will exactly match the structure used on the workstation. If you are not using such a product, you must decide on a directory structure, and have to manually transfer the files after creating them on the workstation. Instructions for manual transfer appear in “Steps for transferring files from the workstation to z/OS or OS/390” on page 47.

Perhaps the easiest approach is to use a working directory structure that matches the structure of your working directories on the workstation. For example, if you are using Object Builder on Windows NT, your working directory on NT contains source files that Object Builder generates, with a subdirectory for the DDL files it generates. You could pattern your HFS directory to contain source files and the subdirectory for DDL. To create an HFS directory structure for your server application files, perform the following steps:

1. Choose an HFS directory structure for the files for the server application. These files include:

- Source files for the application (component) objects that you create through development tools on the workstation, and
- One or more data definition language (DDL) files that define the structure of the server application family, and define its home and container.

---

2. Regardless of the structure you choose, make sure you observe the following rules:
   - The directory structure must have a `PRODUCTION` subdirectory. When you compile code on z/OS or OS/390, the make process expects a `PRODUCTION` subdirectory in the directory in which you enter the make command; without that subdirectory, make does not know where to place output it generates.
   - The name of the directory for the DDL files must exactly match the name of the application family. For example, if you use the name `CashAcctAppFam` for a server application family, you may use a fully qualified directory name like the following examples:

     `/u/`*`userid`*`/CashAcctAppFam` or `/u/`*`userid`*`/Working390/CashAcctAppFam`
   - If you are developing Enterprise Java beans or server applications for use with procedural application adaptors, you have JAR files that need to be placed on z/OS or OS/390. You may place these JAR files in the same directory you use for other server application source files, or you may create a separate directory or subdirectory for them.

---

3. From the u/userid/ directory, enter `mkdir` commands as necesary to create the directory structure you want to use. For details about the `mkdir` command, see *z/OS UNIX System Services Command Reference*, SA22-7802.

When you have finished creating the directory structure, you are ready to transfer the files from the workstation to z/OS or OS/390. For instructions, see “Steps for transferring files from the workstation to z/OS or OS/390”.

Steps for transferring files from the workstation to z/OS or OS/390

If you are not using a product that allows you to access a z/OS or OS/390 hierarchical file system (HFS) as a local drive on your workstation, you need to manually transfer server application files from the workstation to z/OS or OS/390. Several software programs provide easier methods of transferring files from the workstation to z/OS or OS/390, and you may use any of them, as long as you are careful about ASCII to EBCDIC conversions. The instructions provided here assume you are using ftp to transfer files to the HFS on z/OS or OS/390.

Before you begin: Decide where you are going to place the files on z/OS or OS/390. See "Steps for creating an HFS directory structure for CORBA application files from the workstation" on page 46 for suggestions. If you need more background information about working with files, see the file system topics in *z/OS UNIX System Services User's Guide*, SA22-7801.

Complete the following instructions to use ftp, through MS-DOS, to manually transfer the files from Windows NT to the HFS on z/OS or OS/390. These instructions assume that you are going to transfer all files during one ftp session, but you are not required to do so. Once you establish the ftp session, enter all commands at the ftp prompt (`ftp>`).

1. From the base directory you used for Object Builder output (for example, `x:`), start an ftp session with the machine hosting the target HFS, using the command `ftp -i` *`target-machine`*

   Notes:

   a. The `-i` argument turns off ftp file prompting, so you can more easily move many files at once, which is useful for transferring the Object Builder source files.

   b. The default mode of transfer is ASCII, which is the correct mode to use for both the Object Builder source files and DDL files. Use binary mode to transfer any JAR files. To switch modes, enter either `ascii` or `bin` at the ftp prompt.

---

2. To transfer the Object Builder source files:

   a. Change to the HFS working directory where you intend to place the Object Builder source files, using the command `cd` *`directory-path`*

   b. Change to the NT subdirectory that contains the Object Builder source files, using the command `lcd` *`directory-path`*

   c. Transfer all of the files, using the command `mput *`

---

3. To transfer the DDL files:

   a. Change to the HFS directory to contain the DDL files, using the command `cd` *`directory-path`*

   b. Change to the NT subdirectory that contains the Object Builder-generated DDL files, using the command `lcd` *`directory-path`*

   c. Transfer all of the files in the NT subdirectory, using the command `mput *`

4. If you are developing CORBA applications for use with procedural application adaptors, you also have JAR files to transfer. You may need to repeat the following steps if you have more than one JAR file or more than one location (subdirectory) for JAR files. To transfer one JAR file:
   a. Set the correct mode of transfer for a JAR file, using the command `bin`
   b. Change to the HFS working directory where you intend to place the JAR file, using the command `cd` *`directory-path`*
   c. Change to the NT subdirectory that contains the JAR file, using the command `lcd` *`directory-path`*
   d. Transfer the JAR file, using the command `put` *`filename`*`.jar`
   e. After transferring the file, add the fully qualified file name (the directory path plus the file name; for example: `/u/`*`userid`*`/jarfiles/CICSEXCI/BeCash.jar`) to the CLASSPATH statement in the data set member you are using for environment variables. If the logical record size of that data set prevents you from adding the full name to the CLASSPATH statement, copy the JAR file into a directory that is already listed in the CLASSPATH statement, and expand the JAR file in place. To expand the JAR file, use the command `jar -xvf` *`filename`*`.jar`

5. End the ftp session, using the command `quit` or `bye`

Background on deciding where to place executable code for the server application

When you are assembling a server application to deploy in a WebSphere for z/OS MOFW server, you need to consider not only where to direct the make process to place executable code, but also where to place executables for the most efficient run-time performance:

- When you compile your CORBA application, you may direct the make process to place executable code in a partitioned data set (PDS), a partitioned data set extended (PDSE), or in the hierarchical file system (HFS). Because a PDS can contain load modules that are only 16 megabytes or less, your choice is more likely between only two options: PDSE or HFS.
- When you deploy your application in a server, you may use one of the following methods to identify the location of the executable code:
  - For code in the HFS, you may use either the LIBPATH environment variable or load the HFS files into the link pack area (LPA)

   - For code in a PDS or PDSE, you may use either the STEPLIB DD statement on the JCL procedure for the WebSphere for z/OS MOFW server, load the data set into LPA, or add the data set to the link list.

  Where you place code for run-time has an effect on both system and application performance, and on use of virtual storage. For example, before the server can run your application, the system must find and load your application's executable code into storage. The system uses the following search order for code; the closer your code is to the top of the search order, the better the system performs.
  1. LIBPATH variable
  2. STEPLIB DD statements
  3. LPA
  4. Link list

  To determine the best placement for your application's code, then, you need to know how frequently it will be used, and how your installation will use its production-level WebSphere for z/OS MOFW servers.

Although you do not necessarily need to consider run-time placement just to compile code, you should be aware of the run-time options so you can avoid additional work (such as copying HFS files to a PDSE, or the reverse). Use the following recommendations along with the information in Table 5 on page 52, which summarizes the advantages or disadvantages of each placement option. If necessary, see *z/OS MVS Initialization and Tuning Guide*, SA22-7591 for a complete discussion of placing modules in the system's search order, and the effects of placement on system and application performance.

Recommendations:

- If your installation uses the shared HFS function (available with z/OS Version 1 Release 1, and OS/390 Version 2 Release 9 or later), place executable code in the HFS. This choice is especially convenient if your server application is distributed among systems within a sysplex, or if your server application contains any Java code, because you have all your code in one place. Java class files must be in the HFS.

  If you decide to place executable code in the HFS, you may place the code in an HFS directory other than your working directory. If you choose to do so, you must specify output file names in the compiler options before using the `make` utility; otherwise, the compiler places output in your working directory.
- If your installation does not have the z/OS or OS/390 shared HFS function, you should still consider using the HFS for executable code. This choice is especially convenient if your server application contains any Java code, because you have all your code in one place. Java class files must be in the HFS.

Even without the z/OS or OS/390 shared HFS function, you can still place all of your server application code in the HFS, and use other methods to share the file system among systems on which the WebSphere for z/OS MOFW server is running. For example, you could take the following approach:

1. Create a file system that will hold all of the Java class files and executable code for the server application.
2. Compile and test the server application on one system.
3. Once the server application is ready for production, mount the file system on all the systems on which the WebSphere for z/OS MOFW server will run. Either mount all file systems in read-only mode, or mount one in write mode and all others in read-only mode.
4. To refresh the DLLs in the file system, do the following:
   a. Dismount the file system from all systems on which it was mounted
   b. Quiesce the WebSphere for z/OS MOFW servers that use the application
   c. Refresh the DLLs
   d. Remount the updated file system

Depending on the size of the server application, and how frequently you have to make updates to it, this approach might result in better manageability and performance than using the z/OS or OS/390 shared HFS function.

Table 5. Summary of compile and run-time placement options for a CORBA application's executable code

Placement option	For compile time:	For run-time:
HFS	Using the HFS is especially convenient because you have a single place for all of your server application files. Note that, if your application contains any Java code, the Java class files must be in the HFS.	**LIBPATH:** If you use the LIBPATH environment variable, the system creates one copy of your application's executable code for each server region, per server instance. Depending on the number of server regions created and of replicated server instances, this option might cause storage constraints. Also, to update copies of your application, you would have to quiesce all servers using your application before applying the updates.
		LPA: If you load your application into LPA, the system loads one copy of your application into shared virtual storage. This single copy is available to all WebSphere for z/OS MOFW servers that are running on the same system. Using LPA is an advantage when you know you will replicate server instances, or if you expect a heavy workload for your application.

Table 5. Summary of compile and run-time placement options for a CORBA application's executable code (continued)

Placement option	For compile time:	For run-time:
PDSE or **PDS**	Compared to a PDS, PDSEs offer several advantages: PDSEs are not only capable of containing load modules that are larger than 16 megabytes, but are also easy to manage because they do not require reorganization or compression as modules are added or deleted. If you decide to place executable code in PDSEs, you need to allocate the data sets before compiling your server application. See "Background on allocating data sets for the CORBA application's executable code" for more information.	**STEPLIB:** If you use a STEPLIB DD statement, the system creates one copy of your application's executable code for each server region, per server instance, which is the same behavior as using the HFS and the LIBPATH variable. So the considerations for using STEPLIB are the same as for using HFS and LIBPATH. Because PDSEs can be shared among systems in a sysplex, however, distribution among systems is easier than with alternative options (unless your installation has z/OS Version 1 Release 1, or OS/390 Version 2 Release 9 or later, for shared HFS support).
		LPA: In this case, the same considerations for using HFS and LPA apply.
		Link list: In this case, the same considerations for using the HFS and the LIBPATH variable apply.

Background on allocating data sets for the CORBA application's executable code

If you decide to place executable code for your server application in a data set, or if your server application uses DB2 resources, you have to allocate one or more partitioned data sets (PDS) or PDSEs before compiling your server application. Table 6 on page 54 lists the OS/390 Component Broker-specific requirements for these data sets, including compile-time environment variable settings, which are described in "Background on make processing" on page 55.

To allocate a data set, you may use any of several methods, which include the TSO/E allocate command, ISPF, or JCL statements in a batch job. If necessary, refer to one or more of the following resources for additional information:

- *z/OS TSO/E User's Guide*, SA22-7794 describes how to use the allocate command or ISPF, in the chapter on allocating data sets.

- *z/OS MVS JCL User's Guide*, SA22-7598 provides examples of allocating various types of data sets, in the chapter on allocating data set resources.
- *z/OS UNIX System Services User's Guide*, SA22-7801 explains how to use TSO/E commands or JCL in the USS environment.
- *DB2 Application Programming and SQL Guide*, SC26-9933 provides background information about programs that use DB2 resources. See the chapter on preparing an application program to run, but keep in mind that the make process that you will use to compile your program automatically runs the DB2 precompiler for you.

Table 6. Requirements for server application data sets

If . . .	Then you need to allocate. . .	Notes

You want to place your server application's executable code in a data set, rather than in the HFS	A load library data set with the following characteristics:
	LRECL=0
	BLKSIZE=6144
	RECFM=U
	DSORG=PO
		When you use the make utility to compile your application, make sure you set the following environment variables:
		NOHFSLNKOUT=1
		LOADLIB=*data_set_name*

Your server application will access DB2 data	One DBRMLIB data set, per DB2–backed object, with the following characteristics:
	LRECL=80
	BLKSIZE=6160
	RECFM=FB
	DSORG=PO
		For such server applications, the make process uses the DB2 precompiler to create a database request module (DBRM) for the SQL statements and host variable information extracted from the source program, along with information that identifies the program and ties the DBRM to the translated source statements. The DBRM becomes the input to the bind process.
		
		If you need help determining the amount of space your DBRMLIB data set requires, see the DBRM mapping macro DSNXDBRM, in library *prefix*.SDSNMACS, where *prefix* represents the name that your installation uses to identify DB2 library data sets.
		
		When you use the make utility to compile your application, set the following environment variables to identify the data sets you have allocated:
		DBRMHLQ
		DBRMQUAL

Background on make processing

To compile your CORBA server application, you use the `make` utility, which is available through the UNIX system services (USS) shell. When you develop your server application, Object Builder generates make files that contain some environment variable settings and options required for compiling and linking C++ or Java code on z/OS or OS/390. In most cases, you should be able to use the Object Builder-generated make file without alteration. Through various methods available in the USS shell environment, however, you may add or override environment variables or options, depending on the needs of the specific server application.

Before you use the make utility, you need to:

- Understand what settings are required for compiling a server application on z/OS or OS/390. See Table 7 on page 57 for a list of environment variables that you can set for compiling an application. Note that many variables have default settings provided through files shipped with the WebSphere for z/OS product, so you are not required to manually set all of the variables. You might, however, want to override some settings, depending on the application development environment at your installation, and on the server application you are developing.
- Understand how Object Builder sets some environment variables and options. You do not need to change these options, but you may do so; for example, you might want to change options that control the compiler listing.

  For your server application, Object Builder generates three make files: `all.mak`, *`xxx`*`C.mak` and *`xxx`*`S.mak` files. These three `.mak` files include the `prjdefs.mk` file, which contains build options specified through Object Builder. The default settings and options that Object Builder uses are listed in *Component Broker Application Development Tools Guide* under the configuration topic. The `prjdefs.mk` file might also contain compiler options set for other, non-z/OS or OS/390 platform compilers; you can ignore those settings.

  The *`xxx`*`C.mak` and *`xxx`*`S.mak` files also include the `obmdll30.mk` file, which is shipped with WebSphere for z/OS (the make process has access to it through the IVB_DRIVER_PATH variable). The `obmdll30.mk` file uses the `CB390make.rules` and `CB390make.env` files. The `CB390make.help` file contains more information about the contents of these files.
- Understand how the system determines which values to set for environment variables. Settings for the shell environment can be set in one or more of the following places, which the system searches in the order listed:
  1. The user profile in the security product that your installation uses
  2. The /etc/profile file, which is a system-wide file for all z/OS or OS/390 shell users

3. The $HOME/.profile file, which is a personalized file for an individual user
4. The file named in the ENV environment variable
5. A shell command or shell script

  If an environment variable appears in more than one of the following places, the system uses the last setting it found in this search order. For example, if one variable appears in the /etc/profile and in a shell script, the system uses the setting in the shell script.

- Know whether the system programmers at your site have changed the /etc/profile to identify the location of WebSphere for z/OS files for all z/OS or OS/390 shell users. If they did not update this system-wide default profile to set the location of WebSphere for z/OS files, you need to find out where the product files are located, and use your $HOME/.profile file or a shell script to change the values of these environment variables accordingly. See "Steps for setting up the application development environment" on page 40 for a sample profile that your system programmer might have used to update the /etc/profile with settings for all shell users.

When you run the utility, `make` does the following processing, using the resources, files, and environment settings listed:

1. Uses the `startup.mk` file, set up by your installation's system programmer, to find default rules for processing.
2. Uses the `all.mak` file to determine what processing to complete. As a result, make runs both the Object Builder-generated *xxx*`C.mak` and *xxx*`S.mak` files, to create client- and server-side bindings and headers, respectively.
3. Uses the IDL compiler on z/OS or OS/390 to compile interface definition language (IDL) files into C++ parts (.cpp files). These parts include client- and server-side bindings and headers.
4. Uses the DB2 pre-compiler to create database request modules (DBRMs), if the server application accesses DB2 data.
5. Uses the C++ compiler on z/OS or OS/390 to generate the object deck.
6. Uses the Java compiler on z/OS or OS/390 to generate Java class files, if the server application requires Java classes.
7. Uses the binder and linker to create export files, and client and server dynamic link libraries (DLLs).
8. Creates Java archive (JAR) files, if part of your server application contains code written in Java.

Table 7. Environment variables to set for compiling CORBA applications on z/OS or OS/390

Variable	Required/Optional for this type of CORBA application:		Notes
	C++	Java	
_CEE_PREFIX	Optional	Optional	Specifies the prefix of the data set that contains z/OS or OS/390 Language Environment files. Set this variable only if your system programmer did not install z/OS or OS/390 Language Environment in the prescribed data sets.
_CEE_CBC	Optional	Optional	Specifies the data set that contains the C++ compiler and class library files. Set this variable only if your system programmer did not install the z/OS or OS/390 C++ Compiler in the prescribed data sets.
CB390_ENVFILE	Optional	Optional	Specifies the location of the CB390make.env file. **Default:** /usr/lpp/WebSphere390/CB390/samples/CB390make.env The CB390make.env file contains default settings for many of the variables in this table. Change this variable only if you are using your own, edited copy of CB390make.env.
CB390_ROOT	Optional	Optional	Specifies the prefix path that the system uses to construct the value of IVB_DRIVER_PATH, when IVB_DRIVER_PATH is not specified. **Default:** No default value is set. If neither IVB_DRIVER_PATH nor CB390_ROOT are specified, the value of CB390_ROOT is null, and IVB_DRIVER_PATH is set to /usr/lpp/WebSphere390/CB390 Change this variable only if your system programmer did not install OS/390 Component Broker in the prescribed directories.
CB390_USR_CLASSPATH	Not applicable	Optional	Specifies the non-OS/390 Component Broker JAR files to be included for a server application. For example, if you are developing a server application for use with procedural application adaptors, you may prepend this variable to the CLASSPATH variable to include the server application JAR file that you created through VisualAge for Java. **Default:** OS/390 Component Broker does not set a default value for this variable.

Table 7. Environment variables to set for compiling CORBA applications on z/OS or OS/390 (continued)

Variable	Required/Optional for this type of CORBA application:		Notes
	C++	**Java**	
CB390_USR_CPPFLAGS	Optional	Optional	Specifies additional parameters for the C++ compiler. **Default:** OS/390 Component Broker does not set a default value for this variable.
CB390_USR_ CPPSHELLFLAGS	Optional	Optional	Specifies additional switches for the C++ shell command. **Default:** OS/390 Component Broker does not set a default value for this variable.
CB390_USR_CXX_ INCDIRS	Optional	Optional	Specifies additional directories to search for C++ headers. **Default:** OS/390 Component Broker does not set a default value for this variable.
CB390_USR_DLLFLAGS	Optional	Optional	Specifies additional parameters for linking DLLs. **Default:** OS/390 Component Broker does not set a default value for this variable.
CB390_USR_EXEFLAGS	Optional	Optional	Specifies additional parameters for linking executables (that is, main programs). **Default:** OS/390 Component Broker does not set a default value for this variable.
CB390_USR_IDLC_ INCLUDE	Optional	Optional	Specifies additional directories to search for IDL files. **Default:** OS/390 Component Broker does not set a default value for this variable.
CB390_USR_PATH	Optional	Optional	Specifies the search path for additional executable programs. **Default:** OS/390 Component Broker does not set a default value for this variable. If you specify a value for CB390_USR_PATH, OS/390 Component Broker prepends this value to the PATH variable setting.
CB390_USR_ PRLNKFLAGS	Optional	Optional	Specifies additional parameters for prelink (that is, the creation of export files). **Default:** OS/390 Component Broker does not set a default value for this variable.
CB390_STDINC	Optional	Optional	Specifies the location of the C++ run-time headers for OS/390 Component Broker. **Default:** /usr/include//'CBC.SCLBH.+' Change this variable only if your system programmer did not install OS/390 Component Broker in the prescribed directories.

Table 7. Environment variables to set for compiling CORBA applications on z/OS or OS/390 (continued)

Variable	**Required/Optional for this type of CORBA application:**		**Notes**
	C++	**Java**	
CLASSPATH	Optional	Required	Specifies Java class files for use by Java business objects. **Default:** The default value is set in CB390make.env file, and includes the following files that are required for all Java server applications: wszOSsrt.jar You also may either use this variable, or prepend the CB390_USR_CLASSPATH variable, to include the JAR files that you created for your server application.
DBRMHLQ	Optional	Optional	Specifies the high-level qualifier of the name for the MVS data set into which the DB2 pre-compiler will place the DBRMLIBs for your server application. **Default:** The default value is your user ID. Specify this variable only when your server application uses DB2 to store data.
DBRMQUAL	Optional	Optional	Specifies the middle-level qualifier of the name for the z/OS or OS/390 data set into which the DB2 pre-compiler will place the DBRMLIBs for your server application. **Default:** the name of the server DLL that contains the static SQL to be compiled. Specify this variable only when your server application uses DB2 to store data.
IVB_BATCH_ INCREMENTAL	Optional	Optional	**Recommendation:** Set to 1 to improve performance of an incremental build (that is, a rebuild after making small changes).

Table 7. Environment variables to set for compiling CORBA applications on z/OS or OS/390 (continued)

Variable	Required/Optional for this type of CORBA application:		Notes
	C++	**Java**	
IVB_BATCH_ PROCESS_ FACTOR	Optional	Optional	Specifies the maximum number of make targets to be built at the same time (in parallel). **Default:** The number of IDL files in your project. **Tip:** Large projects might demand too many parallel processes, so you might want to specify a constant value to control the number of processes used. (Your system programmer sets the maximum number of processes through the MAXPROCUSER statement in the BPXPRMxx parmlib member.) **Recommendation:** The best value to use depends on the average size of your applications. Start with a value of 10 and modify it as necessary.
IVB_BUILD_DEBUG	Optional	Optional	When set to 1, this variable specifies whether to compile C++ source with debug information. **Default:** OS/390 Component Broker does not set a default value for this variable.
IVB_BUILD_ UNOPTIMIZE	Optional	Optional	Specifies whether to compile C++ source without optimization. Setting this variable to 1 results in faster compilation. **Default:** OS/390 Component Broker does not set a default value for this variable.
IVB_BUILD_ VERBOSE	Optional	Optional	Specifies the amount of messages generated during compilation. **Default:** Object Builder sets the default value of 1, which results in the maximum amount of detailed compiler messages.
IVB_COMBINE_ SOURCE	Optional	Optional	Specifies whether header files are processed multiple times. **Default:** Object Builder sets the default value of 1, which results in header files being processed only once, thus reducing the time required for the build.
IVB_DRIVER_ PATH	Required	Required	Specifies the location of OS/390 Component Broker product files. **Default:** The default value, `/usr/lpp/WebSphere390/CB390`, is set in CB390make.env file.

Table 7. Environment variables to set for compiling CORBA applications on z/OS or OS/390 (continued)

Variable	Required/Optional for this type of CORBA application:		Notes
	C++	**Java**	
IVB_OPTIMIZE	Optional	Optional	Specifies whether DLLs are compiled with optimization, including inlining of code. **Default:** Object Builder sets the default value of 1 (equivalent to setting IVB_UNOPTIMIZE=0) **Recommendation:** Set to 0
IVB_PAX_LIST	Optional	Optional	Specifies whether C++ listings are compressed to save space. **Example:** For part `xyz.cpp`, the compressed listing is `xyz.clst.Z`. To decompress the `xyz.clst.Z` listing, use the following command: `pax -rf xyz.cpp.Z` The result of the decompression is listing `xyz.clst` **Recommendation:** Set to 1 to compress listings.
IVB_UNOPTIMIZE	Optional	Optional	Specifies whether DLLs are compiled with optimization, including inlining of code. **Default:** Object Builder sets the default value of 1 (equivalent to setting IVB_OPTIMIZE=0) **Recommendation:** Set to 0
JAVA_COMPILER	Not applicable	Optional	Specifies which Java compiler the make process should use. **Default:** OS/390 Component Broker does not set a default value for this variable.
JAVA_HOME	Not applicable	Optional	Specifies the directory in which the Java 2 Standard Edition (J2SE) Software Development Kit (SDK) is installed. **Default:** OS/390 Component Broker does not set a default value for this variable.

Table 7. Environment variables to set for compiling CORBA applications on z/OS or OS/390 (continued)

Variable	Required/Optional for this type of CORBA application:		Notes
	C++	**Java**	

LIBPATH	Required*	Required	Specifies the search path for Java DLLs in the HFS. **Default:** A default value might be set in your /etc/profile file. OS/390 Component Broker does not set a default value for this variable.
			
			* Required for C++ applications if you are using procedural application adaptors. In this case, as for Java applications, specify system, OS/390 Component Broker, and Java DLLs. For example:
			`LIBPATH=/`*`db2_install_path`*`/lib`
			`:/usr/lpp/java/J1.3/bin`
			`:/usr/lpp/java/J1.3/bin/classic`
			`:/usr/lpp/WebSphere390/CB390/lib`
			
			where *db2_install_path* is the HFS where you installed DB2 for OS/390.

LOADLIB	Required*	Required*	Specifies the name of the z/OS or OS/390 data set into which the make process will place the DLLs for your application. (If you are compiling a Java application, make places DLLs in the data set, but places the JAR file in the HFS.) **Default:** OS/390 Component Broker does not set a default value for this variable. * Required only if you set the NOHFSLNKOUT variable to 1, or you want to place DLLs in a data set instead of in the HFS.
NOHFSLNKOUT	Optional	Optional	Specifies whether the system is to place DLLs in the HFS or in a z/OS or OS/390 data set. You may set one of the following values: • 0 sends link-edit output to the HFS • 1 sends link-edit output to the data set specified through the LOADLIB variable. **Default:** 0

Table 7. Environment variables to set for compiling CORBA applications on z/OS or OS/390 (continued)

Variable	Required/Optional for this type of CORBA application:		Notes
	C++	Java	
PATH	Optional	Optional	Specifies the search path for files containing commands that you want to run. **Default:** A default value might be set in your /etc/profile file. OS/390 Component Broker alters the /etc/profile setting by first prepending IVB_DRIVER_PATH/bin to the PATH statement, and then prepending CB390_USR_PATH, if you specify that variable.
SBBOEXEC_DSN	Optional	Optional	Specifies the data set in which OS/390 Component Broker REXX EXECs reside. **Default:** 'BBO.SBBOEXEC' Change this variable only if your system programmer did not install OS/390 Component Broker in the prescribed directories.
STEPLIB	Required*	Required*	Specifies additional data sets that your application might need. **Default:** A default value might be specified in your /etc/profile file. OS/390 Component Broker does not set a default value for this variable. * Required only if both of the following conditions are true: • Your application uses DB2 • The DB2 pre-compiler data sets SDSNLOAD and SDSNEXIT are not in the system link list.

If necessary, see the following sources for further information:

For information about this topic:	See:
The USS shell environment	*z/OS UNIX System Services User's Guide*, SA22-7801 describes the USS shell environment and how to work with environment variables.

For information about this topic:	See:
The make utility	• *z/OS UNIX System Services Programming Tools*, SA22-7805, for a tutorial on using make. • *Component Broker Application Development Tools Guide*, for the default options that Object Builder uses. • *z/OS UNIX System Services Command Reference*, SA22-7802, for `make` command format and options, makefiles, usage notes, and other reference information.
IDL	• *Component Broker Programming Guide*, for a description of interface definition language and its syntax • "Appendix C. The Interface Definition Language (IDL) compiler" on page 171, for the `idlc` command, along with options and syntax for the OS/390 Component Broker IDL compiler.
The DB2 pre-compiler	*DB2 Application Programming and SQL Guide*, SC26-9933 lists pre-compiler options and default values that you might want to change for a particular server application.
C++	• *z/OS UNIX System Services Command Reference*, SA22-7802, for compiler options (under the c89 command description) that you might want to change for a particular server application. • *z/OS C/C++ User's Guide*, SC09-4767, for general information about the z/OS or OS/390 C++ compiler and its options.
Java on z/OS or OS/390	http://www.s390.ibm.com/java/

Steps for compiling CORBA application source files on z/OS or OS/390

Once you have the workstation files for your CORBA server application in a working directory on z/OS or OS/390, you can run the make file to produce the executable code and bindings required for running a server application in a WebSphere for z/OS MOFW application server.

When you generated server application artifacts on the workstation, Object Builder produced three makefiles: a client DLL makefile, a server DLL makefile, and an `all.mak` file. Use the `all.mak` file, not the individual DLL makefiles, to compile code for your server application. Using the `all.mak` file ensures that the DLLs are built in the correct order. The `all.mak` file also includes any IDL compile, Java compile, CPP compile, and link options that you specified in Object Builder for the client and server DLLs.

Before you begin:

- Decide where you want to place your server application's executable code. See "Background on deciding where to place executable code for the server application" on page 49 for recommendations.
- Make sure you have allocated any data sets that your server application requires. See "Background on allocating data sets for the CORBA application's executable code" on page 53 for instructions on allocating data sets.
- Make sure you understand how make processing works, and how environment variable settings influence make processing. See "Background on make processing" on page 55 for more information about make processing and environment variable settings.

Perform the following steps to compile the source files for your server application:

1. Make sure you have set the proper environment variables for compiling code. The environment variables you need to set include those that identify the location of WebSphere for z/OS product files, and identify where you want to place the executable code for your server application.

   To set the environment variables, do the following:

   a. Enter the `set` or `export` command to display the current settings for the shell environment.
   b. Compare the current shell settings to Table 7 on page 57, which describes the environment variables that you might need to set for make processing, depending on the type of server application you are going to compile. Some variable values that you should use depend on guidelines for your installation or your individual development environment.
   c. Either edit your $HOME/.profile file or create a shell script to add or override environment variable settings, as necessary, for each specific server application.

      Recommendations:

      - If the environment variables in Table 4 on page 42 are not set in the /etc/profile, set them in your $HOME/.profile file. These environment variables should have the same settings for all server applications you develop for WebSphere for z/OS. Make sure you export these variables, to pass them on to shell commands and scripts that run in your shell session.
      - Set the environment variables for make processing in a shell script. Consider using one shell script for each server application you are assembling, and use the following naming convention for each script:

        serverappname`_make_setup.sh`

With one script per application, you can change shell settings or compiler options as necessary for an individual application.

If you use a shell script to specify some options that also appear in the Object Builder-generated `prjdefs.mk` file for your server application, make sure you edit the `prjdefs.mk` file to comment out the duplicate options.

d. (Optional) If you want to change any of the default settings in the `CB390make.env` file shipped with WebSphere for z/OS, do the following:
   1) Make your own copy of the `CB390make.env` file. `CB390make.env` is in IVB_DRIVER_PATH/samples.
   2) Edit your copy of the `CB390make.env` and save your changes.
   3) Change the setting for the CB390_ENVFILE environment variable to point to your copy of `CB390make.env`.

---

2. If you use a shell script to set variables, you need to execute the script before using the make command to start make processing. To execute a shell script, enter the following:

```
. serverappname_make_setup.sh
```

---

3. From your working directory on z/OS or OS/390, enter `make -f all.mak`

Tips:

- Use the following make command to run the build process in the background, and to direct all messages in a file for later review, if necessary. For example, the file `./build_results.txt` will contain all messages generated during the make process.

```
make -f all.mak 1>build_results.txt 2>&1 &
```

- If you use a file to collect messages generated during the build process, you can view them before make processing completes by using the `head` and `tail` UNIX commands, which are described in *z/OS UNIX System Services Command Reference*, SA22-7802.
- If you are placing executable code in a load library, the make process might fail with a system completion (ABEND) code x37. These ABEND codes indicate errors such as data sets that are too small. Look up the specific ABEND code in *z/OS MVS System Codes*, SA22-7626 to determine and fix the problem. Then restart the build process.
- If the make process fails, fix the error and restart the build process. If you need to start from scratch again, issue the following command to clean off all files generated in the HFS during the first build attempt: `make –f all.mak clean`.

You know you are done when the server application executables are in either the data set you specified, or in your working directory on the HFS.

Steps for adding your CORBA application to the system search path

Before you deploy your application in a server, you may move the executable code into one of the following locations:

- For code in the HFS, you may load the HFS files into the link pack area (LPA)
- For code in a PDS or PDSE, you may load the data set into LPA or add the data set to the link list.

Where you place code for run-time has an effect on both system and application performance, and on use of virtual storage. To determine the best placement for your application’s code, see the recommendations and information in “Background on deciding where to place executable code for the server application” on page 49.

To load the executable code for your server application into the link pack area (LPA) or the link list, complete one of the following procedures:

- To move DLLs in the HFS into LPA, see the instructions in *z/OS UNIX System Services Planning*, GA22-7800.
- To move DLLs in a PDS or PDSE into LPA, do the following:
  1. Create a PROGxx parmlib member that uses LPA statements to add the client and server DLLs to LPA. For example, adding the WebSphere for z/OS installation verification program to LPA would require the following LPA statements:

     ```
     LPA ADD MODNAME=(JPOLICYS) DSNAME(high-level_qualifier.SBBOLOAD)
     LPA ADD MODNAME=(JPOLSQMO) DSNAME(high-level_qualifier.SBBOLOAD)
     LPA ADD MODNAME=(JPOLTRIR) DSNAME(high-level_qualifier.SBBOLOAD)
     LPA ADD MODNAME=(POLICYC) DSNAME(high-level_qualifier.SBBOLOAD)
     LPA ADD MODNAME=(POLICYS) DSNAME(high-level_qualifier.SBBOLOAD)
     LPA ADD MODNAME=(POLSQIR) DSNAME(high-level_qualifier.SBBOLOAD)
     LPA ADD MODNAME=(POLTRIR) DSNAME(high-level_qualifier.SBBOLOAD)
     LPA ADD MODNAME=(POLHMIR) DSNAME(high-level_qualifier.SBBOLOAD)
     ```

     If you need additional information about the PROGxx parmlib member, see *z/OS MVS Initialization and Tuning Reference*, SA22-7592.
  2. From an MVS console or TSO/E session, enter the SET PROG=xx command to load the application into LPA. If you need additional information about the SET PROG=xx command, see *z/OS MVS System Commands*, SA22-7627.
- To move DLLs in a PDS or PDSE into the link list, do the following:

1. Create a PROGxx parmlib member that uses a LNKLST ADD statement to specify the PDS or PDSE containing your server application’s executable code. If you need additional information about the LNKLST ADD statement or the PROGxx parmlib member, see *z/OS MVS Initialization and Tuning Reference*, SA22-7592.
2. From an MVS console or TSO/E session, enter the SET PROG=xx command to add the application to the link list. If you need additional information about the SET PROG=xx command, see *z/OS MVS System Commands*, SA22-7627.

Steps for binding data objects for your CORBA application

When your server application contains business objects and data objects that use DB2 to store persistent data, you need to set up the DB2 tables those objects use, and bind each data object into its own package in the DB2 Universal Database for z/OS and OS/390 plan. To set up the DB2 tables, see “Steps for preparing DB2” on page 81.

To bind data objects into packages, you need to use the database request modules (DBRMs) that the make process generated when you compiled your server application. The make process generates one DBRM library per data object.

Before you begin: You might want to either review or have available the following references:

- *DB2 Application Programming and SQL Guide*, SC26-9933, for background information about DB2 packages.
- *DB2 Command Reference*, SC26-9934, for details about the BIND PACKAGE subcommand syntax and options.
- *DB2 Administration Guide*, SC26-9931, for information about setting up an ID with DB2 Universal Database for z/OS and OS/390 SYSADM authority.

Perform the following steps to bind each data object into its own package:

1. Create a member in your working JCL data set, and copy the following JCL example into it.

```
//jobname JOB
//*
//BIND1    EXEC  PGM=IKJEFT01,REGION=0M
//SYSTSPRT DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//DBRMLIB  DD DISP=SHR,DSN=project_dbrmlib
//SYSTSIN  DD *
 DSN SYSTEM(DB2_subsystem_name)
 BIND PACKAGE(package_name) MEMBER(dbrm_name) ACTION(REPLACE) +
   bind_options
 END
/*
//BIND2    EXEC  PGM=IKJEFT01,REGION=0M
//SYSTSPRT DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//DBRMLIB  DD DISP=SHR,DSN=CB390_dbrmlib
//SYSTSIN  DD *
 DSN SYSTEM(DB2_subsystem_name)
   BIND PACKAGE(package_name) MEMBER(BBOSQDB ) ACTION(REPLACE) +
     ISOLATION(CS) CURRENTDATA(NO) RELEASE(COMMIT)              )
 END
/*
```

2. Edit the member to customize the JCL for your installation, as follows:
   - Update the JOB card with installation-specific parameters.
   - Delete or comment out the BIND2 step if the application and its clients will not be using the query service. You must use the BIND1 step for all DB2–backed server applications, but you need the BIND2 step only if the application uses the query service.
   - Replace the following variables with appropriate values, as follows:

Table 8. Variables to replace in your JCL to bind DB2 packages

Replace this variable:	**With this value:**
project_dbrmlib	Specify the make-generated DBRM library. If the make process generated more than one DBRM library because your application has more than one data object that uses DB2, either copy the make-generated DBRMLIB data sets into one partitioned data set, or add additional DBRMLIB DD statement to the BIND1 step, as necessary.
DB2_subsystem_name	Specify the DB2 subsystem that manages the tables your data objects will use.
package_name	Specify the name you want to use for this DB2 package.
dbrm_name	Specify the name of the load module that will be making database requests. In other words, use the name of the member in the library that you specified for *project_dbrmlib*. Each member corresponds to a data object in your server application. If you specified more than one *project_dbrmlib*, or have more than one member in the *project_dbrmlib*, make sure that you specify one BIND PACKAGE subcommand for each member.
bind_options	Specify the options that you want to use for binding the package.

- If you are using the BIND2 step, replace the *CB390_dbrmlib* on the DBRMLIB DD statement. Replace other variables as indicated in Table 8 on page 69.

3. From a user ID with DB2 Universal Database for z/OS and OS/390 SYSADM authority, submit the job.

Chapter 4. Deploying CORBA applications in WebSphere for z/OS MOFW servers

Once you have an executable CORBA server application, you need to set up its run-time environment, which is a WebSphere for z/OS application server. WebSphere for z/OS includes two types of application servers: One for Java 2 Enterprise Edition (J2EE) applications, the other for CORBA applications. The server for CORBA applications is known as the managed-object framework (MOFW) server. When you use the WebSphere for z/OS Administration application to define servers for your CORBA applications, you will notice that the Administration application uses two labels for servers: `J2EE server` and `Server` (for the MOFW server type). Make sure you use `Server` for your CORBA applications.

Rules:

- You cannot deploy CORBA applications in a J2EE server.
- You cannot deploy J2EE applications in a MOFW server.

To **deploy** a server application on z/OS or OS/390, one needs some knowledge of the application to be deployed and knowledge of z/OS or OS/390 and the subsystems that the CORBA application requires. z/OS or OS/390 system programmers and database administrators are the most likely personnel to have the skills required for deploying an application. In this book, personnel who assemble and deploy server applications are called **application assemblers**. In addition to some details about the server application to be deployed, these people need to know:

- How to work in the USS environment, and in TSO/E to access z/OS or OS/390 components and data sets.
- How to set up z/OS or OS/390 subsystem resources that the application requires. For example, these resources might include databases, transaction managers, and security products.

  The degree of expertise in each area depends on the type of server application to be deployed. For example, if your CORBA application accesses DB2 data directly instead of accessing DB2 through an IMS transaction, you do not need to know anything about IMS.
- How to define and activate a WebSphere for z/OS MOFW application server (that is, the run-time environment for the application to be deployed).

Background on naming rules for elements of the run-time environment

When you define a server configuration through the WebSphere for z/OS Administration application, you are creating a model of a run-time environment for a particular type of application. When you create a model, you must supply names for the following:

- A generic application environment, which is called a **server**.
- An entity that is responsible for a certain type of work. This entity is called a **server instance**, which consists of one control region, and one or more server regions.
- A logical resource mapping (LRM), which defines a particular **type** of resource manager (such as DB2, IMS, or CICS) that your application will use.
- An LRM instance, which identifies a particular resource manager subsystem (such as DB2SYSA, or IMS2).

Servers and LRMs are general definitions, so to speak; whereas server instances and LRM instances name specific entities that exist on z/OS or OS/390.

Rules: For your model to become an active run-time environment on z/OS or OS/390, you must make sure the names you use in the Administration application are the same as names you use for related definitions on z/OS or OS/390:

- Use the same generic name (for example, `PAAIMSV`) for the **server** you define in the administration application, and for the application environment you define in the workload manager IWMARIN0 dialog.
- You must use the same name (for example, `PAAIMSV1`) for the **server instance** not only in the administration application, but also in the following places:
  1. In the JCL procedure that you create to start the control region. This JCL proc is usually a copy of the BBOASR1 member of BBO.SBBOJCL. When you make a copy of this member for your own use, you must modify the PROC statement to identify the server instance; for example:
     ```
     //label    PROC  SRVNAME='PAAIMSV1'
     ```
  2. In the JCL procedure that you create to start the server region. This JCL proc is usually a copy of the BBOASR1S member of BBO.SBBOJCL. When you make a copy of this member for your own use, you must modify the IWMSSNM parameter to identify the server instance; for example:
     ```
     // label    PROC  IWMSSNM='PAAIMSV1', PARMS='-ORBsrvname ',
     //     CBCONFIG='/u/cb390'
     ```

For recommendations on naming conventions for the elements of a MOFW server, see *WebSphere Application Server V4.0.1 for z/OS and OS/390: Operations and Administration*, SA22-7835.

Background on setting environment variables for the WebSphere for z/OS MOFW server

Whenever you use the Administration application to create a new or modify an existing WebSphere for z/OS MOFW server, you have the chance to change environment variable settings.

"Appendix A. Environment files" on page 125 lists the environment variables you need to use for WebSphere for z/OS MOFW servers in which your server applications will run. Use that list to determine which variables you need to set. One variable that is particularly important is the CLASSPATH variable, to which you must manually add the fully qualified names of JAR files for your server application, if any.

Coding JCL procedures to start the WebSphere for z/OS MOFW server

In TSO, perform the following steps to create JCL procedures for the application control region and server region:

1. In your working PROCLIB data set, create a new member with a name that matches the generic server name. Copy the BBOASR1 sample member from BBO.SBBOJCL into this new member, and make appropriate updates according to comments in the file. Modify the PROC statement to identify the server instance name you will specify in the WebSphere for z/OS Administration application. This new member is now the JCL procedure you can use to start the application control region.

2. Also in your PROCLIB, create a new member to contain the JCL procedure for starting the application server region. Copy the BBOASR1S sample member from BBO.SBBOJCL into this new member, and make appropriate updates according to comments in the file, including::
   - Edit the IWMSSNM parameter to use the server instance name you will specify in the WebSphere for z/OSAdministration application.
   - Edit the member to identify the location (PDS, PDSE, or HFS) of your server application's executable code.

Defining the WebSphere for z/OS MOFW server

"Background on deploying server applications" on page 8 illustrates the steps you complete to define an application server, or run-time environment, for server applications. The steps in that process are the same whether you are defining an application server for testing purposes or for production-ready applications. Through the WebSphere for z/OS Administration application, you define a model configuration that you validate, commit, and activate. When you activate that configuration, WebSphere for z/OS system manager creates a run-time environment on z/OS or OS/390. Figure 8 on page 11 illustrates both a sample model and the run-time environment that results from activating that model.

This configuration is fairly simple: only one server application is defined to a container, which is connected to only one resource manager subsystem. More complex configurations are possible; for example, you may connect more than one resource manager to a particular container, which is an advantage that is unique to WebSphere for z/OS. The configuration you define depends on the qualities of service that your server applications require.

When you define a server configuration, the options or selections that you make have a direct impact on the behavior of the server application that is to run in that environment.

Note: WebSphere for z/OS includes two types of application servers: One for Java 2 Enterprise Edition (J2EE) applications, the other for CORBA applications. The server for CORBA applications is known as the managed-object framework (MOFW) server. When you use the WebSphere for z/OS Administration application to define servers for your CORBA applications, you will notice that the Administration application uses two labels for servers: `J2EE server` and `Server` (for the MOFW server type). Make sure you use `Server` for your CORBA applications.

Background on using the WebSphere for z/OS Administration application

When you first use the WebSphere for z/OS Administration application after installing the WebSphere Application Server product, you use the CBADMIN administrator user ID to log on. You may define additional administrator user IDs later, to allow access to the administration application from several workstations or sessions. When you have additional user IDs, however, keep the following rules in mind:

- You cannot use the same administrator ID to log on to multiple concurrent sessions of the application, from either a single workstation or from more than one workstation. For example, if you start the administration application on your workstation, using CBADMIN as the user ID, you cannot start another session using CBADMIN from either your own or a different workstation. You may, however, start another session using a different administrator user ID from any workstation.

- If you define several administrator user IDs, they all may be logged on simultaneously, but only one can update and activate a conversation at a time. While one administrator is activating a conversation, the others should use the administration application for only read or display functions.

Selecting server properties for a test system

For a complete list and explanation of MOFW server properties, use the help available through the Administration application, or see *WebSphere Application Server V4.0.1 for z/OS and OS/390: System Management User Interface*, SA22-7838.

Defining containers for MOFW servers

When you use the Administration application to add a container for your CORBA application, you need to know the following:

- The container name specified through Object Builder, during the packaging or configuration stage of developing the application.

  Rule: For a given server application, the container names specified in the Administration application and in Object Builder must match.
- The qualities of service that the server application requires, which determine the values you specify for the container properties.

Note: When you use Object Builder to configure an existing application for the z/OS or OS/390 platform, and that application is associated with a default container, WebSphere for z/OS allows you to successfully deploy that application. Because those default containers are designed for use only in WebSphere servers on workstation platforms, the resulting run-time environment might not match that on the workstation platforms. In a WebSphere for z/OS MOFW server, these default containers have pre-defined properties that match z/OS and OS/390-recommended properties for containers that manage transient objects.

When you define a container, you select various policies that determine how the container manages its objects, and how those objects behave. One of those policies is the container transaction policy, which defines the transactional scope for the object that the container manages. For most server application objects, you should select the Required transaction policy. Before you use any other transaction policy, make sure you read and understand "Background on the OS/390 Component Broker transactional environment" on page 17.

Restriction: When you define a container that contains a queryable home, you cannot define more than one logical resource mapping connection for the container.

Connecting the WebSphere for z/OS MOFW server to a back-end resource manager

Guidelines for supplying connection data for an IMS-OTMA LRM instance

For a WebSphere for z/OS MOFW server and an IMS subsystem to communicate through OTMA, you need to supply connection data related to the IMS and OTMA configuration.

The following list identifies the connection data you need to supply for an IMS-OTMA LRM instance, and how to determine appropriate values for that data:

XCF group name
Fill in the name specified on the GRNAME parameter in the DFSPBxxx proclib member used for IMS initialization.

XCF partner name
Fill in the name specified on the OTMANM parameter in the DFSPBxxx proclib member used for IMS initialization. Otherwise, use the name specified by the APPLID1 parameter in the DFSPBxxx member, which is the default XCF partner name if no OTMANM parameter is defined.

number of sessions
Specify 1.

TPIPE prefix
Specify a prefix, which must be four characters or less, for the system to use for all transaction pipes required for this LRM. When creating a transaction pipe for this LRM, the system generates a unique transaction pipe name by using this prefix and appending four characters of session-related information.

If you need more information about using OTMA for access to an IMS subsystem, see one or more of the following books:

- *WebSphere Application Server V4.0.1 for z/OS and OS/390: Installation and Customization*, GA22-7834, for instructions on setting up a WebSphere for z/OS MOFW server that uses IMS-OTMA procedural application adapter support.
- *IMS/ESA Open Transaction Manager Access Guide*, SC26-8743, for information about IMS with OTMA.

Guidelines for supplying connection data for an IMS-APPC LRM instance

For a WebSphere for z/OS MOFW server and an IMS subsystem to communicate through APPC/MVS, you need to supply connection data related to the specific APPC/MVS logical unit (LU) associated with the server, and the LU associated with IMS.

The following list identifies the connection data you need to supply for an IMS-APPC LRM instance, and how to determine appropriate values for that data:

Local LU name
Fill in the logical unit (LU) name associated with WebSphere for z/OS. This local LU name is defined in an LUADD statement in the APPCPMxx parmlib member for the system on which WebSphere for z/OS runs.

Look for the LUADD statement for the LU associated with WebSphere for z/OS. Use the value specified on the ACBNAME parameter as the **local** LU name.

Rule: Use only the value specified on the ACBNAME parameter, which is the network LU name. If you specify a network-qualified (or fully qualified) name for the local LU, you will receive error message BBOU0106E, which indicates that the local LU name is not valid.

Partner LU name
Fill in the name of the LU with which the WebSphere for z/OS server will initiate an APPC conversation. This partner LU is defined in an LUADD statement in the APPCPMxx parmlib member for the system on which IMS runs. The IMS subsystem may be, but does not have to be, on a system other than the one on which the WebSphere for z/OS server runs.

Look for the LUADD statement for the LU associated with IMS (an LU associated with IMS has the IMS subsystem name specified for the SCHED parameter on the LUADD statement). Use the value specified on the ACBNAME parameter as the **partner** LU name.

Tip: When you specify the partner LU name, you may use one of the following forms:

- Only the value specified on the ACBNAME parameter (in other words, the network LU name)
- A network-qualified name (in the form *networkID***.***networkLUname*)

  networkID is the value specified for the VTAM start option NETID and *networkLUname* is the value specified on the ACBNAME parameter.
- A VTAM generic resource name, if your installation is configured to use generic resources.

VTAM logmode name
Fill in the name of the VTAM logmode that designates the network properties to be associated with any APPC conversations between this local LU and its partner LU. Logmode names appear in the VTAM logon mode table, which reside in your installation's VTAMLIB data set.

APPC conversation time-out value
Specify the length of time, in minutes, for the WebSphere for z/OS server to wait for a response to the Allocate call and any subsequent calls the

server issues during its conversation with IMS. Valid time-out values range from 0 through 1440, which is 24 hours.

If you specify a value that is less than the value set for the `OTS_DEFAULT_TIMEOUT` environment variable, the APPC conversation time-out value will have no effect. Look for the `OTS_DEFAULT_TIMEOUT` environment variable setting that you use for the application server's control and server regions.

APPC sync level

Fill in one of the values listed in following table. This value controls the type of APPC/MVS conversation the WebSphere for z/OS server uses to communicate with IMS. Base your choice on the transaction policies you select for containers in this server configuration, and the characteristics of the applications to be deployed in this server.

Recommendation: Use a sync level value that corresponds with the transactional context of the request that the server is currently processing. The easiest way to match the sync level and context is to select **Autotran**, which lets the system determine the matching sync level.

If this LRM is connected to:	Then specify this sync level value:	Notes
One or more containers that **all** use the TX Required transaction policy	**Syncpt** (in certain cases, **None** is also acceptable)	Because this transaction policy enforces the use of a global transaction, the most logical value for the APPC sync level is **Syncpt**. With **Syncpt**, the server allocates a protected conversation, which preserves the global transactional context for the interaction between the server and the IMS subsystem, and allows the system to recover any resources if conversation errors or failures occur. In certain cases, however, you might consider using **None** when your application's processing does not depend on the ability to recover resources at this point in its processing. With **None**, APPC/MVS, WebSphere for z/OS, and IMS do not coordinate any processing done on behalf of a distributed application; without the overhead of coordination, your application's performance improves. **Recommendations:** • Use **Syncpt** if you cannot guarantee that your server application will always run on the same z/OS or OS/390 system on which the IMS subsystem runs. • Use **None** judiciously. In this case, resources that the application uses might be in inconsistent states if conversation errors or failures occur.
One or more containers that use a transaction policy other than TX Required	**Autotran**	Use **Autotran** with these policies, so the system can determine which conversation type, Syncpt or None, is appropriate for the transactional context associated with the current thread of execution. In other words, if the current thread has a local transactional context, the server uses a sync level of None; for a global transactional context, the server uses Syncpt.

If you need more information about using APPC/MVS for access to an IMS subsystem, see one or more of the following books:

- *WebSphere Application Server V4.0.1 for z/OS and OS/390: Installation and Customization*, GA22-7834, for instructions on configuring a WebSphere for z/OS MOFW server and an IMS subsystem as a local LU and partner LU, respectively. See the topic about using IMS-APPC procedural application adapter support.

- *z/OS MVS Planning: APPC/MVS Management*, SA22-7599, for general APPC/MVS configuration details.
- *OS/390 eNetwork Communications Server: SNA Resource Definition Reference*, SC31-8565, for VTAM definition statements.

Guidelines for importing CORBA application DDL

To correctly run and manage your server application, the WebSphere for z/OS MOFW server needs to know the contents of your CORBA server application. The server gets this information when you import the DDL files that Object Builder generated when you packaged your server application.

Recommendation: Consider using HFS files, rather than z/OS or OS/390 data sets, for both input and output files for the import process. The input file is already in the HFS, in your working directory. You can define an output file in the same directory; just make sure you use a name that is not exactly the same as the input file.

Before you begin:

- If you want to use a z/OS or OS/390 data set for the import process, you must copy the contents of the DDL file into a data set. For instructions on copying HFS files into data sets, see *z/OS UNIX System Services User's Guide*, SA22-7801.
- Make sure that you have set up the appropriate security definitions for the import DDL process. Because the input and output files for this process are associated with the BBOSMSS address space user ID, that user ID must have access to the files:
  - If you use a data set for the DDL to be imported, the user ID must have read access to the input data set, and alter access to the output data set.
  - If you use an HFS file for the DDL, the user ID must have the ability to search the directories to find the input file, the ability to read the input file, and the ability to write to the output file.
- If you use a data set for the DDL to be imported, make sure that these data sets are not in use by another process. For example, you cannot use ISPF to edit or browse the data set or data set member at the same time you start the import.

Rules:

- You cannot delete and re-import the same application family in one conversation. To re-import an application family, you must:
  1. Delete the application family in one conversation.
  2. Activate that conversation.
  3. Create another conversation, and re-import the DDL.

- You must define a container in the WebSphere for z/OS Administration application before you can import any CORBA application DDL that references the container.
- When you import specific DDL, you may associate only one logical resource mapping with the container referenced in the DDL.
- You cannot import DDL containing objects that are already in an active conversation.
- If input and output files used for DDL import reside in the hierarchical file system (HFS), those files must be available (through NFS mount or replication, or shared HFS) to every system in the sysplex that has Systems Management running on it.
- If input and output files used for DDL import are z/OS or OS/390 data sets (either sequential or partitioned), the characteristics must be RECFM=VB, LRECL=1020, BLKSIZE=1024.
- Another process cannot be using the data set to be imported during the import process. For example, you cannot use ISPF to edit or browse the data set or data set member at the same time you start the import.
- If you use a partitioned data set to store the DDL to be imported, you must specify the member name when you supply the input file name for the import process.
- If you specify a pre-allocated data set as the output file for the import process, you must supply a member name.
- When you have to import both base and specific DDL, import the base DDL first, then the specific DDL.

Preparing resource managers for processing your application

To ensure that your installation has correctly set up a resource manager for use with WebSphere for z/OS, see *WebSphere Application Server V4.0.1 for z/OS and OS/390: Installation and Customization*, GA22-7834 for specific requirements. The procedures in this section assume that your installation has correctly installed and configured any resource managers that your applications will need to use.

Steps for preparing DB2

If your server application uses DB2 to store data, you need to complete the following tasks before running your application:

- Bind each server application data object into its own package in the DB2 Universal Database for z/OS and OS/390 plan. For instructions, see “Steps for binding data objects for your CORBA application” on page 68.
- Create new or check existing DB2 tables that your server application will use.

  Recommendations:

  - Although Object Builder can generate database schema files, create new database tables through the SQL processor using file input (SPUFI) facility. For details about creating database tables, see *DB2 Application Programming and SQL Guide*, SC26-9933.
  - Use row locks instead of page locks. For information about the advantages of using row locks, see *DB2 Administration Guide*, SC26-9931.

Steps for preparing IMS

Before running any client applications that drive your CORBA application on z/OS or OS/390, complete the following steps to prepare the IMS subsystem:

1. Set the IMS parallel scheduling limit to 0 (zero), which means that any number of transactions can be scheduled.
   - To determine the current setting, you may issue the IMS `/DISPLAY TRANSACTION` command, and look for the value under the `PARLM` heading in the display results.
   - To set a new value for the parallel scheduling limit, you may issue the IMS `/ASSIGN PARLIM` command.

   For more information about using IMS commands, see *IMS/ESA Operations Guide*, SC26-8741 or *IMS/ESA Operator's Reference*, SC26-8742.

   ---

2. Start enough IMS message processing regions (MPRs) to handle the number of client application requests. The number of MPRs required depends on your application's processing and the IMS environment in which it runs.
   a. Determine the number of MPRs you need, using your knowledge of your server application and its clients and guidelines in the following chart:

For this type of logical resource mapping (LRM):	Use this information to determine the number of MPRs:
IMS-OTMA or IMS-APPC with a sync level of Syncpt	In these environments, the WebSphere for z/OSMOFW server, IMS, and other system components work together to coordinate all processing within a single client transaction, so that all updates are either completed successfully or rolled back. Because one single client transaction may generate several IMS transactions, and thus use several MPRs, delays might occur because each MPR must wait until the client transaction completes before they are free to handle other work. To avoid such delays in processing, IMS needs at least the same number of started MPRs as the number of IMS transactions that the client application will generate. For example, consider a CORBA application that uses `begin` and `commit/rollback` instructions to define the scope of one transaction. Within that one transaction, the client application generates as many as three separate IMS transactions. An IMS transaction consists of the processing required to receive a request for work, to invoke a program to do the work, and to transmit the response to the requestor. In this particular case, you need at least three active MPRs to handle the requests from a single client application.
IMS-APPC with a sync level of None	In this environment, the WebSphere for z/OS MOFW server, IMS, and other system components do not coordinate all processing within a single client transaction, so MPRs are ready for new work as soon as a single IMS transaction completes. In this case, you might be able to achieve acceptable application performance with fewer MPRs than the number of IMS transactions that the client application will generate.
IMS-APPC with a sync level of Autotran	In this environment, the system decides which sync level, Syncpt or None, to use for any APPC/MVS conversation it has to start to communicate with IMS. To avoid the delays that might occur when the system uses a sync level of Syncpt, use the same guidelines listed above for IMS-OTMA or IMS-APPC with a sync level of Syncpt.

b. Determine how many MPRs are currently active. To do so, you may issue the IMS `/DISPLAY ACTIVE REGION` command.

c. If you need more MPRs than the number of currently active regions, use the IMS sample job named IMSMSG to start new regions. This sample job invokes the IMS DFSMPR procedure to start new regions. Make sure you specify a unique name for each MPR.

Depending on your installation's practice, you will find the IMSMSG job in either the IMS.JOBS or IMS.PROCLIB data set. If you need more information about this job, see *IMS/ESA Installation Volume 2: System Definition and Tailoring*, GC26-8737.

Adding CORBA application interfaces to the WebSphere for z/OS interface repository

Client applications can use either static bindings or dynamic interface invocation to use a CORBA application. To enable clients to use dynamic interface invocation, you must populate the WebSphere for z/OS interface repository for each CORBA application that you deploy on the z/OS or OS/390 platform.

Object Builder creates the source file for an interface-repository loader program as part of the build process for each server application. Running the `all.mak` file compiles not only the server application code, but also compiles the loader program. The resulting executable program, *`application-name`*`IR.exe`, is placed in the same directory as the server application DLLs.

To run the loader program to add server application interfaces to the interface repository, perform the following steps:

1. Copy the BBOIRC3 member from BBO.SBBOEXEC into your working JCL data set.

---

2. Edit your copy of the BBOIRC3 member, as follows:
   - Change PGM name to *`application-name`*`IR`
   - Specify one of the following options, in single quotes, on `SET ARGV`:

     IRdelete
     Removes only the current interface information from the IR database

     IRforce
     Forces the deletion of inheriting interfaces as well as current interface and repopulate all of them

     (default)
     Forces deletion of only the current interface information, and repopulate this interface information

---

3. Run the loader program from either the z/OS or OS/390 UNIX environment or from TSO/E:

- From the z/OS or OS/390 UNIX environment, enter the executable name (and arguments, if you do not want the default); for example:

  ```
  executable-name,argv=IRforce
  ```
- From TSO/E: specify the JCL procedure name, and executable name (and arguments, if you do not want the default); for example:

  ```
  s procname,exe=executable-name,argv=IRdelete
  ```

Chapter 5. Developing, assembling, and deploying client applications on z/OS or OS/390

Perhaps the most common approach to developing z/OS or OS/390 client applications will be to:

1. Code and test both client and CORBA applications on the workstation, using Visual Age Component Development tools
2. Assemble, deploy, and test server applications in a WebSphere for z/OS(MOFW) server
3. Port client applications to z/OS or OS/390:
    a. Editing source code as appropriate, on either the workstation or z/OS or OS/390
    b. Compiling and link-editing code on z/OS or OS/390

You may, however, code and test client applications directly on z/OS or OS/390, without first creating a version of the client application that runs on the workstation.

Regardless of the approach, however, client application programmers require the following skills to develop client applications that run on z/OS or OS/390:

- The C++ or Java programming languages
- The WebSphere for z/OS common client programming model, which is described in detail in *WebSphere Application Server for OS/390 Component Broker: Programming Guide*.
- Design considerations and z/OS or OS/390-specific guidelines for both client and server applications in “Chapter 2. Developing CORBA applications for WebSphere for z/OS” on page 15.
- z/OS or OS/390 UNIX system services, including working with shells, setting environment variables, using the `make` command to compile code, and other common programming tasks
- Security services available through the installation’s security product

The following table shows the subtasks and associated information for preparing and running client applications on z/OS or OS/390.

Subtask	Associated information (See . . .)
Learning which environments WebSphere for z/OS supports for client applications	“Background on supported client run-time environments” on page 88

Subtask	Associated information (See . . .)
Learning about the application development environment on z/OS or OS/390	"Background on setting up the application development environment on z/OS or OS/390" on page 92
Developing client applications	• "Background on designing and coding clients for your server applications" on page 91 • "Background on the OS/390 Component Broker transactional environment" on page 17 • "Restrictions for CORBA applications and their clients" on page 24
Assembling client applications on z/OS or OS/390	• "Background on deciding where to place executable code for the client application" on page 92 • "Background on allocating data sets for the client application's executable code" on page 92 • "Steps for setting environment variables for make processing" on page 92 • "Steps for compiling client applications on z/OS or OS/390" on page 93
Deploying client applications on z/OS or OS/390	• "Background on setting up security for servers and z/OS or OS/390 clients" on page 95 • "Steps for running a client application on z/OS or OS/390" on page 95

Background on supported client run-time environments

For CORBA applications that run in its MOFW servers, OS/390 Component Broker supports client applications that run on z/OS or OS/390, and on other platforms, as shown in Figure 16 on page 90. For client applications that run on z/OS or OS/390, the run-time environment includes z/OS or OS/390 Language Environment and UNIX System Services (USS):

- z/OS or OS/390 Language Environment provides common services and language-specific routines, such as message handling and storage management, in a single run-time environment for C++, Java, and other programs.

  Because various elements of z/OS or OS/390 require Language Environment, system programmers at your site have already set up this run-time environment as part of installing z/OS or OS/390. The common run-time library and run-time libraries for supported languages are already available to your client application, through the link list, STEPLIB, or run-time library services (RTLS).

- UNIX System Services provides a shell, the standard UNIX command-line interface, that allows users to interact with z/OS or OS/390, plus all of the utilities that UNIX users expect in their work environment. Through UNIX System Services, users can develop or port application programs, manage files, control processes, and so on.

  As with Language Environment, system programmers at your site have already set up the USS environment for all users. As part of preparing and running OS/390 Component Broker client applications on z/OS or OS/390, you have to modify or tailor the environment as instructed in this chapter.

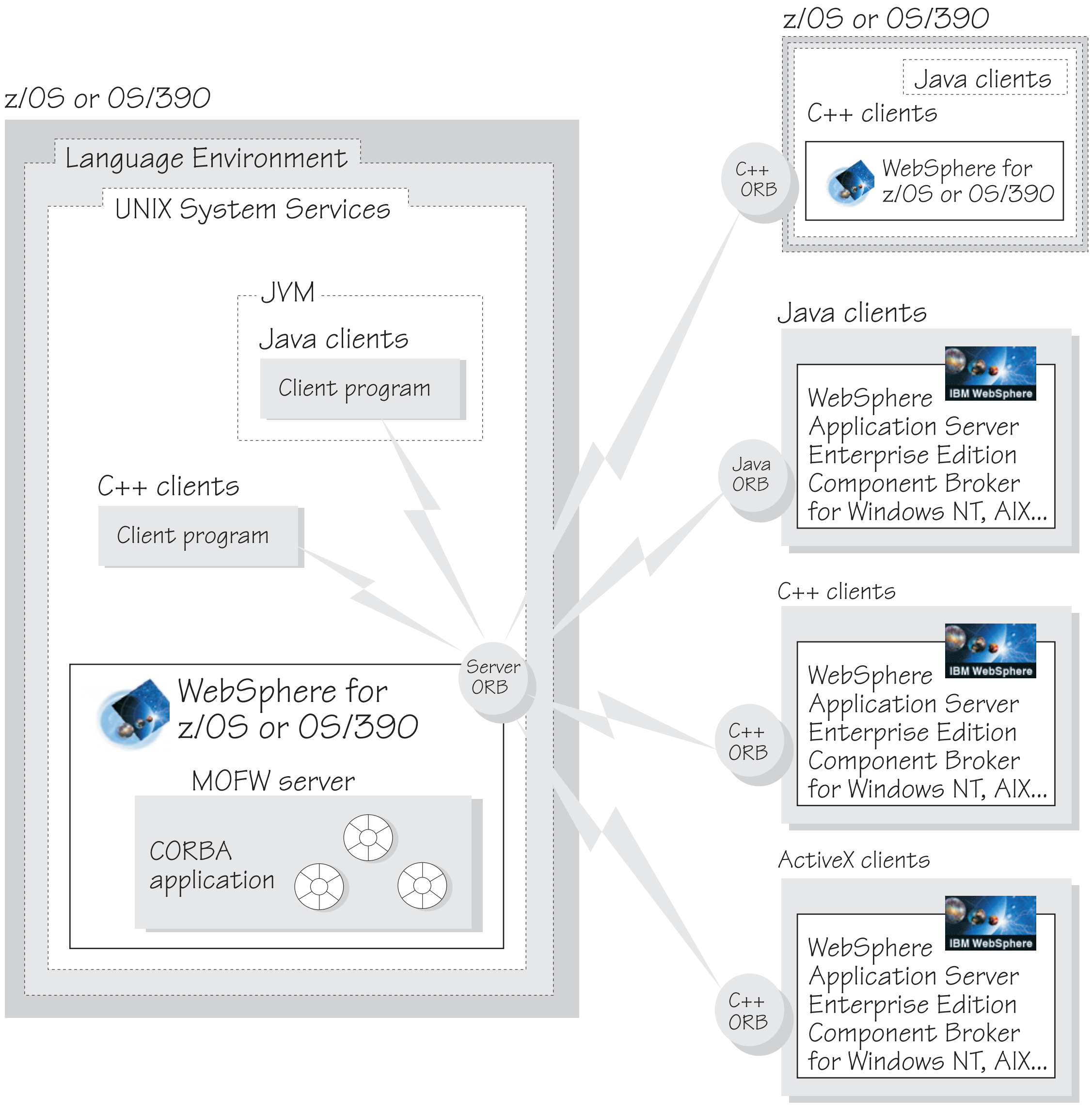
z/OS or OS/390
Language Environment
UNIX System Services
JVM
Java clients
Client program
C++ clients
Client program
WebSphere for z/OS or OS/390
MOFW server
CORBA application
Server ORB
z/OS or OS/390
Java clients
C++ clients
C++ ORB
WebSphere for z/OS or OS/390
Java clients
Java ORB
WebSphere Application Server Enterprise Edition Component Broker for Windows NT, AIX...
IBM WebSphere
C++ clients
C++ ORB
WebSphere Application Server Enterprise Edition Component Broker for Windows NT, AIX...
IBM WebSphere
ActiveX clients
C++ ORB
WebSphere Application Server Enterprise Edition Component Broker for Windows NT, AIX...
IBM WebSphere

Figure 16. Clients that OS/390 Component Broker supports for CORBA applications in MOFW servers

The topics addressed in this chapter apply to only the clients that run on z/OS or OS/390. These clients can run either:

1. On the same system as the OS/390 Component Broker MOFW server that manages the objects that the clients use; or
2. On another system on which OS/390 Component Broker is running.

In the first case, illustrated on the left side of Figure 16 on page 90, the client application is local to the OS/390 Component Broker MOFW server. In the second case, illustrated on the upper right side of Figure 16 on page 90, the client application is remote to the OS/390 Component Broker MOFW server.

If necessary, see the following sources for further information:

For information about this topic:	See:
Configuring and running client applications on platforms other than z/OS or OS/390	• *WebSphere Application Server for OS/390 Component Broker Programming Guide* • *WebSphere Application Server for OS/390 Component Broker System Administration Guide*
Configuring and running IMS applications as OS/390 Component Broker clients	“Appendix B. An IMS application as an WebSphere for z/OS client” on page 159
The z/OS or OS/390 Language Environment	• *z/OS Language Environment Concepts Guide*, SA22-7567, for a conceptual introduction to Language Environment, including descriptions of the program model, callable services, and glossary of terms. • *z/OS Language Environment Programming Guide*, SA22-7561, for instructions on creating and running application programs under Language Environment.
The USS shell environment	*z/OS UNIX System Services User’s Guide*, SA22-7801, for descriptions of the USS shell environment and instructions for using the various system services.

Background on designing and coding clients for your server applications

Because the WebSphere Application Server for z/OS and OS/390 Component Broker products define a common programming model, most of the information about designing and coding client applications appears in the following books:

- *Component Broker Programming Guide*
- *Component Broker Advanced Programming Guide*
- *Component Broker Programming Reference*
- *Component Broker Application Development Tools Guide*

These manuals define the concepts, coding practices, programming interfaces, and tools you need to understand to design and code Component Broker client applications. These manuals also note which concepts or interfaces do not apply for the z/OS or OS/390 platform, but these restrictions apply only to CORBA applications running in a WebSphere for z/OS MOFW server. The Component Broker client programming model is identical for all platforms on which the Component Broker products run.

Background on setting up the application development environment on z/OS or OS/390

When system programmers at your site install OS/390 Component Broker, they also have the option of tailoring the UNIX application development environment on z/OS or OS/390. The instructions they receive in *WebSphere Application Server V4.0.1 for z/OS and OS/390: Installation and Customization*, GA22-7834 are listed in "Steps for setting up the application development environment" on page 40. Generally speaking, the tailoring they do for application assemblers, such as allocating HFS space, is adequate for programmers who develop and run client applications on z/OS or OS/390.

If necessary, see *WebSphere Application Server V4.0.1 for z/OS and OS/390: Installation and Customization*, GA22-7834 for specific software products and release levels for the client environment.

Background on deciding where to place executable code for the client application

Executable code for your client application can reside in a partitioned data set (PDS), a partitioned data set extended (PDSE), or in the hierarchical file system (HFS). Because a PDS can contain load modules that are only 16 megabytes or less, your choice is more likely between only two options: PDSE or HFS. See "Background on deciding where to place executable code for the server application" on page 49 for recommendations, which apply to client application code as well as server application code.

Background on allocating data sets for the client application's executable code

If you decide to place executable code for your client application in a data set, or if your client application uses DB2 resources directly, see "Background on allocating data sets for the CORBA application's executable code" on page 53 for information about allocating one or more partitioned data sets (PDS) or PDSEs before compiling your client application.

Steps for setting environment variables for make processing

Perhaps the easiest way to compile your client application is to use the make utility, which is available through the UNIX system services (USS) shell. "Steps for compiling client applications on z/OS or OS/390" on page 93 contains a sample make file that you can use; the sample make file is based on the make files that Object Builder generates for CORBA applications to be deployed in OS/390 Component Broker MOFW servers. In most cases, the requirements for make processing, the environment variable settings, and the instructions for compiling code are the same for both server applications and their z/OS or OS/390 clients. Use the information presented in "Background on make processing" on page 55, for a description of make processing and environment variable settings.

Steps for compiling client applications on z/OS or OS/390

Perhaps the easiest way to compile your client application is to use the make utility, which is available through the UNIX system services (USS) shell. These instructions contain a sample make file that you can use; the sample make file is based on the make files that Object Builder generates for CORBA applications to be deployed in OS/390 Component Broker MOFW servers. In most cases, the requirements for make processing, the environment variable settings, and the instructions for compiling code are the same for both server applications and their z/OS or OS/390 clients.

Perform the following steps to compile your client application:

1. Make sure you have set the proper environment variables for compiling code. The environment variables you need to set include those that:
   - Identify the location of OS/390 Component Broker product files
   - Identify where you want to place the executable code for your server application

   See "Background on make processing" on page 55 for instructions on setting variables for make processing.

2. Make sure you have allocated any data sets that your client application requires. See "Background on allocating data sets for the CORBA application's executable code" on page 53 for instructions on allocating data sets.

3. Copy the following sample make file into your working HFS directory, using the file naming convention: *filename*.mak

```
all: pass3 pass7

OBJ_FILES=client.o
EXE_FILES=client.exe

.INCLUDE: obmdll30.mk
```

```
client.o.cppflags=NOEXPO
client.o : client.cpp
client.exe: client.o
```

---

4. From your working directory on z/OS or OS/390, enter `make -f` *`filename`*`.mak`

   Tips:

   - Use the following make command to run the build process in the background, and to direct all messages in a file for later review, if necessary. For example, the file `./build_results.txt` will contain all messages generated during the make process.

     ```
     make -f filename.mak 1>build_results.txt 2>&1 &
     ```

   - If you use a file to collect messages generated during the build process, you can view them before make processing completes by using the `head` and `tail` UNIX commands, which are described in *z/OS UNIX System Services Command Reference*, SA22-7802.
   - If you are placing executable code in a load library, and the data set you allocated is too small, the make process fails with a system completion (ABEND) code x37. To fix the problem, allocate a larger data set and restart the build process.
   - If the make process fails, fix the error and restart the build process. If you need to start from scratch again, issue the following command to clean off all files generated in the HFS during the first build attempt: `make –f` *`filename`*`.mak clean`.

If necessary, see the following sources for further information:

For information about this topic:	**See:**
The USS shell environment	*z/OS UNIX System Services User's Guide*, SA22-7801 describes the USS shell environment and how to work with environment variables.
The make utility	• *z/OS UNIX System Services Programming Tools*, SA22-7805, for a tutorial on using make. • *z/OS UNIX System Services Command Reference*, SA22-7802 , for `make` command format and options, makefiles, usage notes, and other reference information.

For information about this topic:	See:
IDL	• *Component Broker Programming Guide*, for a description of interface definition language and its syntax • “Appendix C. The Interface Definition Language (IDL) compiler” on page 171, for the `idlc` command, along with options and syntax for the OS/390 Component Broker IDL compiler.
The DB2 pre-compiler	*DB2 Application Programming and SQL Guide*, SC26-9933 lists pre-compiler options and default values that you might want to change for a particular server application.
C++	• *z/OS UNIX System Services Command Reference*, SA22-7802 , for compiler options (under the c89 command description) that you might want to change for a particular server application. • *z/OS C/C++ User's Guide*, SC09-4767, for general information about the z/OS or OS/390 C++ compiler and its options.
Java on OS/390	http://www.s390.ibm.com/java/

Background on setting up security for servers and z/OS or OS/390 clients

Several security issues, including user identification and authentication, apply for client programs running on z/OS or OS/390. How you set up security for your client applications depends primarily on which WebSphere for z/OS MOFW servers the client application will use, whether the client is local or remote to those servers, and how secure their communication needs to be.

For further information about setting up security for the client run-time environment, see *WebSphere Application Server V4.0.1 for z/OS and OS/390: Installation and Customization*, GA22-7834.

Steps for running a client application on z/OS or OS/390

Once you have executable code for your client application, you can run the client application to use CORBA applications (that is, objects) deployed in WebSphere for z/OS MOFW application servers.

Perform the following steps to run your client application:

1. Set up the appropriate security mechanisms for your client and the WebSphere for z/OS MOFW servers that it will use. For further information about setting up security for the client run-time environment, see *WebSphere Application Server V4.0.1 for z/OS and OS/390: Installation and Customization*, GA22-7834.

2. Set the proper environment variables for the client run-time environment. You may use the same mechanisms and recommendations that you used to set compile-time variables. If necessary, see "Steps for compiling CORBA application source files on z/OS or OS/390" on page 64. For the actual variable values to set, however, use the tables and definitions in "Appendix A. Environment files" on page 125.

---

3. If you use a shell script to set variables, you need to execute the script before running your client application. To execute a shell script, enter the following:

   ```
   . clientappname_setup.sh
   ```

---

4. Start your client application. You may use any of the methods available through the z/OS or OS/390 UNIX environment. If necessary, see *z/OS UNIX System Services User's Guide*, SA22-7801 or *z/OS Language Environment Programming Guide*, SA22-7561 for details about running programs.

Chapter 6. Working with CORBA applications in a production system

In addition to step-by-step installation through the WebSphere for z/OS Administration application, you may use the following alternative methods of installing applications in a WebSphere for z/OS:

For information about:	See . . .
Using the export/import function of the Administration application	"Steps for using the export/import process through the Administration application"
Using the System Management Scripting APIs	"Installing applications using scripts" on page 99

Steps for using the export/import process through the Administration application

After you have finished testing your CORBA applications, you can use the WebSphere for z/OS Administration application to export the (MOFW) server configuration you have been using on your test system, and import that model configuration on a production system. Through this export/import process, you create an HFS file that contains the server definition, which you transfer to a production system. This process can be quicker and less error-prone than defining a server configuration from scratch.

Perform the following steps to use the export/import process:

1. In the Administration application, export the server model of the MOFW server in which your application is deployed:
   a. Select the server in the active image.
   b. Select the **export server...** action of the Selected menu bar choice. The **Export server** dialog box appears.
   c. Enter the fully qualified name of an HFS file to contain the output of the export process.
   d. Click **OK**.

   Result: WebSphere for z/OS uses the specified output HFS file to store all of the information associated with the server, including the following:
   - All of the server properties
   - Containers
   - Application families

- Logical resource mappings
- Applications
- Client interfaces
- Classes
- DLLs
- Homes
- Query meta-data objects

---

2. Copy or move the output HFS file to the z/OS or OS/390 production system on which you want the server to run. See *z/OS UNIX System Services User's Guide*, SA22-7801 for methods of and instructions for moving or copying files.

   Warning: Do not edit the output HFS file.

---

3. In the Administration application, import the MOFW server model by completing the following steps:
   a. Add a conversation, if necessary.
   b. Select the **Servers** folder.
   c. Select the **import server...** action of the Selected menu bar choice. The **Import server** dialog box appears.
   d. For **Server name**, enter a name that is unique to this WebSphere for z/OS configuration.
   e. For **Input file**, enter the fully qualified name of the HFS file that you moved or copied to the production system.
   f. Click **OK**.
   g. Modify the properties of the server, including **Control region proc name** and **Debugger allowed**.
   h. Add server instances for the production system, as appropriate.
   i. Add logical resources for the production system, as appropriate.

---

4. Also in the Administration application:
   a. Validate the imported model by selecting the conversation, then selecting **Validate**. When message BBON0442I appears in the status bar, the new conversation is valid.
   b. Commit the conversation by selecting the conversation, then selecting **Commit**. Answer Yes to the question: "Do you still want to commit?" When message BBON0444I appears in the status bar, the new conversation was committed.
   c. Complete OS/390 tasks, as appropriate.

d. Activate the conversation by selecting the conversation, then selecting **Activate**. Answer Yes to the question: ″Do you want to activate conversation... ?″ At the bottom of the dialog, a message indicates when the server definition has been activated.

Installing applications using scripts

To install applications in a MOFW server without using the WebSphere for z/OS Administration application, you may use the System Management Scripting APIs, which provide exactly the same capabilities as the Administration application. Using the scripts might provide a quicker, less error-prone method of installing applications into a production server, for example. For more information about using the System Management Scripting APIs, see *WebSphere Application Server V4.0.1 for z/OS and OS/390: System Management Scripting API*, SA22-7839.

Chapter 7. Collecting data about CORBA application activity

WebSphere for z/OS offers several different methods of collecting information about CORBA applications running in a MOFW server:

For information about:	See . . .
Using SMF records to collect accounting information	"Collecting CORBA application information through SMF records"
Using the IBM Distributed Debugger to collect diagnostic data for distributed applications	"Debugging and tracing distributed applications"
Using JRas support to enable applications to issue messages and trace entries	"Logging messages and trace data for Java applications" on page 106

Collecting CORBA application information through SMF records

If you want to collect and record statistics related to your server applications, you may define a MOFW server to use the z/OS or OS/390 systems management facility (SMF). Through SMF activity and interval records, the MOFW server records application details that you may use for application profiling. To enable SMF recording, you must define the MOFW server to create SMF records, and perform other administration tasks; for further details, start with the SMF topic in *WebSphere Application Server V4.0.1 for z/OS and OS/390: Operations and Administration*, SA22-7835.

Debugging and tracing distributed applications

The IBM Distributed Debugger and Object Level Trace tools enable you to monitor and debug distributed applications, including the components that run in an WebSphere for z/OS server. The IBM Distributed Debugger and Object Level Trace provide debugging and tracing capabilities for Java or C++ application components and their Java or C++ clients, which may reside on platforms other than z/OS or OS/390.

The following table shows the subtasks and associated information for using the IBM Distributed Debugger and Object Level Trace tools, which are hereafter called Debugger and OLT, respectively.

Subtask	Associated information (See . . .)
Learning concepts related to the Debugger and OLT	The **InfoCenter** for WebSphere V3.5 or V4.0 for distributed platforms. The InfoCenter is available at: `http://www.ibm.com/software/webservers/appserv/`
Installing the Debugger and OLT on your workstation	The **InfoCenter** as listed above
Setting up the workstation and z/OS or OS/390 environments and applications for using the tools	• "Steps for starting the Debugger and OLT on your workstation" • "Steps for preparing the Debugger and OLT for Windows Java clients" on page 103 • "Steps for preparing the Debugger and OLT for Windows C++ clients" on page 104 • "Step for preparing z/OS or OS/390 Java clients" on page 104 • "Steps for preparing z/OS or OS/390 C++ clients" on page 105 • "Steps for preparing server applications in a WebSphere for z/OS MOFW server" on page 105
Using the Debugger and OLT interfaces and output	The **InfoCenter** as listed above

Steps for starting the Debugger and OLT on your workstation

Perform the following steps to start the Debugger and OLT on a Windows NT or 2000 workstation:

1. Start the OLT Viewer from either the Windows Taskbar Start Menu, or type `olt` from an MS-DOS command prompt.

2. Start the Browser Preferences window by selecting **File → Preferences...**

3. From the Browser Preferences window, click on **OLT**. Write down the OLT Server TCP/IP port value (the default is 2102), which is the value you will later specify for the client environment variable.

4. From the Client Controller, in the Execution Mode list box, set the application execution mode to one of the following:
   - `Trace only`
   - `Debug only`
   - `Trace and debug`
   - `No trace and debug`

   Make sure you hit the Apply button to save any changes.

Notes:

a. Remember whether you select the trace-only option or a trace-and-debug option, because you must specify the same option in Object Builder to correctly prepare your server application for use with the Debugger and OLT.

b. If the execution mode is set to `Debug only` or `Trace and debug`, the debugger host name and debugger TCP/IP port field is enabled. You can change the debugger host name and port to values that reflect the location of the Debugger. This host name should be the same as the value you specify for the OLT Server host name if OLT and the debugger interface run on the same machine. Otherwise, these host name values will be different. If your installation does not have DNS configured for the WebSphere for z/OS environment, make sure you use an IP address as the Remote Debugger host name.

   The default setting for the Debugger host name is the local host name, and the default for the Debugger TCP/IP port is 8001.

Before continuing to the next procedure, make sure that you remember the following:
- The monitoring mode you selected for debugging.
- The IP addresses and port numbers for the machines on which the OLT server and OLT client controller are running.

Steps for preparing the Debugger and OLT for Windows Java clients

To prepare the Debugger and OLT for Java clients that run on Windows and use Java business objects that are installed in a MOFW server, perform the following steps:

1. Create the client startup command based on the following default startup command:

```
java -Xdebug -Xnoagent -Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=8888
-Djava.compiler=NONE -Dcom.ibm.debug.jdwpport=8888
-Xbootclasspath/a:%JAVA_HOME%\lib\tools.jar -classpath %SOMCBASE%\lib\somojor.zip;
%SOMCBASE%\samples\InstallVerification\ProgrammingModel\BusinessObjects\Policy\Working\NT\TRACE_DEBUG\JCB\jcbPolicyC.jar;
%SOMCBASE%\lib\dertrjrt.jar;%CLASSPATH%
-Dcom.ibm.CORBA.EnableApplicationOLT=true
-Dcom.ibm.CORBA.OLTApplicationHost=lei01.torolab.ibm.com
-Dcom.ibm.CORBA.OLTApplicationPort=2102 -DOLTClient=true
-Dcom.ibm.CORBA.BootstrapHost=boss0012.12.ibm.com PolicyApp [options]
```

---

2. Change `address` and `jdwpport` to the same value that you will use for the JVM_DEBUG_PORT variable for the MOFW server in which the Java business objects are installed..

---

3. Change the `OLTApplicationHost` to the hostname where the OLT runs.

---

4. Change the `OLTApplicationPort` to the OLT server's TCP/IP port.

5. Change the `BootstrapHost` to the host name where the WebSphere for z/OS MOFW server runs.

Steps for preparing the Debugger and OLT for Windows C++ clients

Before you begin:

- Make sure DCE is installed and running.
- Make sure CBConnector is running.
- Bring up the WebSphere for z/OS Administration application with Expert mode.

To prepare the Debugger and OLT for C++ clients that run on Windows and use business objects that are installed in a MOFW server, perform the following steps using the WebSphere for z/OS Administration application:

1. Go to `host images`, open your IP name image, then open **client style image → preference**, and edit the default client setting.

2. Under the `orb` panel, change the name server to where the WebSphere for z/OS MOFW server is running.

   Example: `boss0109.12.ibm.com`

3. Go to the `transaction` panel and change the `deferred transaction begin option` to `never`.

4. Go to the main panel and set the `OLThostname` and `OLTport` to values that point to the OLT tools.

5. Click the Apply button.

Step for preparing z/OS or OS/390 Java clients

To prepare Java clients that run on z/OS or OS/390 and use Java business objects that are installed in a MOFW server, perform the following step:

- Add the following environment variables to the Java client's shell script:

```
export HOME=/tmp
export IVB_DEBUG_ENABLED=1 # enable the OLT tools
export IVB_TRACE_PORT=2102 # OLT Server TCP/IP port
export IVB_TRACE_HOST=address # the IP address where the OLT is running, or
                              # if WebSphere for z/OS has DNS set up, you may
                              # use the host name for IVB_TRACE_HOST
```

Steps for preparing z/OS or OS/390 C++ clients

Before you begin: Find out whether the IBM Distributed Debugger and Object Level Trace are installed and available at your installation. You will need to know whether the systems on which your WebSphere for z/OS server and client applications will run have access to the cataloged data set for the Debug Tool, which works with components of the IBM Distributed Debugger and Object Level Trace. The Debug Tool is part of the C/C++ with Debug Tool feature of z/OS or OS/390, and of VisualAge for Java, Enterprise Edition for OS/390.

To prepare C++ clients that run on z/OS or OS/390 and use business objects that are installed in a MOFW server, perform the following steps:

1. Add the following environment variables to the client's environment file:

   ```
   HOME=/tmp                     // this can be any existing writable directory
   IVB_DEBUG_ENABLED=1           // this is to enable the client for debug/trace
                                 //   tool
   IVB_TRACE_PORT=2102           // default port for the OLT Server TCP/IP port
   IVB_TRACE_HOST=address       // ip address or hostname of the work station
                                 //   where the OLT is running
   ```

2. Make sure the client and the server are able to access the OLT dlls or the Debug Tool datasets.

Steps for preparing server applications in a WebSphere for z/OS MOFW server

Before you begin: Find out whether the IBM Distributed Debugger and Object Level Trace are installed and available at your installation. You will need to know whether the systems on which your WebSphere for z/OS server and client applications will run have access to the cataloged data set for the Debug Tool, which works with components of the IBM Distributed Debugger and Object Level Trace. The Debug Tool is part of the C/C++ with Debug Tool feature of z/OS or OS/390.

Follow these instructions, from the workstation where you installed the Debugger and OLT, to enable the Java or C++ business objects for OLT:

1. Compile the business object with the compiler debug option.

2. Start the WebSphere for z/OS Administration application and create a new conversation for a new MOFW server.

3. Expand the Servers folder and highlight the MOFW server. Then, in the server properties form:

- Check the `Debugger allowed` check box.
- Set the `Object Level Trace Hostname` to the name of the machine where OLT is running, and set the `Object Level Trace Port` to the value you set in "Steps for starting the Debugger and OLT on your workstation" on page 102. (The default port is `2102`).
- Add the following environment variable only for Java business objects:

  ```
  JVM_DEBUG_PORT=xxxx
  ```

  where *xxxx* is the port number that the Debugger will use to connect to the running JVM.

Logging messages and trace data for Java applications

The WebSphere for z/OS run-time supports the Ras Toolkit for Java, which enables you to issue messages from and collect trace data for your Java server applications that run in WebSphere for z/OS J2EE or MOFW servers. Through WebSphere for z/OS extensions to the toolkit, known as JRas support, your Java application's messages can appear on the z/OS or OS/390 master console or in the error log stream, depending on the message type. All messages are logged in the component trace (CTRACE) data set for WebSphere for z/OS. Also, your application's trace entries can appear in the same CTRACE data set.

You might want to issue messages to the master console to report serious error conditions for mission-critical applications. Through the master console, an operator can receive and, if necessary, take action in response to a message that indicates the status of your application. In addition, by directing messages to the master console, you can trigger automation packages to take action for specific conditions or events related to your application's processing.

With JRas support, you may direct error messages to the error log stream. Any messages that your application issues also appear in the CTRACE data set for WebSphere for z/OS. Logging the messages in these system resources can help you more easily diagnose errors related to your application's processing.

Similarly, issuing requests to log trace data in the CTRACE data set is another method of recording error conditions, or collecting application data for diagnostic purposes. You can select the amount and types of trace data to be collected, so you have the ability to run your application with minimal tracing, when performance is a priority, or to run your application with detailed tracing, when you need to recreate a problem and collect additional diagnostic information.

Recommendation: The error log stream, the CTRACE data set for WebSphere for z/OS, and the master console are primarily intended for recording diagnostic data for or monitoring system components and critical applications. Depending on your installation's configuration, directing application messages and data to these resources might have an adverse affect on system performance. For example, if you send application data to the CTRACE data set, trace entries in that data set might wrap more quickly, which means you might lose some critical diagnostic data because the system writes new entries over existing ones when wrapping occurs. Use this logging support judiciously.

Notes:

1. You can use the WebSphere for z/OS support for logging messages and trace data only for Java applications (not for Java applets).
2. The WebSphere for z/OS support for the Ras Toolkit is not the same as the JRas support supplied in Enterprise Edition V3.02. The new JRas support:
   - Always logs messages that your application issues. This change means that, once you code an application to issue messages and run that application, its messages will always be collected and logged. With Enterprise Edition V3.02, you had the ability turn off message collection.
   - Requires a different mechanism for enabling the collection of trace data. With Enterprise Edition V3.02, environment variables for the MOFW application server controlled the collection of trace data; with WebSphere for z/OS V4.0, a customer-supplied trace settings file enables or disables the collection of trace data.
   - Uses different classes for obtaining message or trace loggers, but the same methods: the createRASTraceLogger and createRASMessageLogger methods. The WebSphere for z/OS V4.0 methods, however, have slightly different signatures than those for Enterprise Edition V3.02.

     Although the Enterprise Edition V3.02 createRASTraceLogger and createRASMessageLogger methods are deprecated, you do not have to change any of the programs you coded to use them, unless those programs must run on another platform as well as on z/OS or OS/390. With WebSphere for z/OS V4.0, calls to createRASTraceLogger or createRASMessageLogger are delegated to the same methods in the new WebSphere for z/OS V4.0 class. To run your application on additional platforms, such as Windows NT, you must recode your program to use the new methods.

     For descriptions of the methods you can issue from your server application to issue messages or log trace entries, refer to the InfoCenter at `http://www.ibm.com/software/webservers/appserv/library.html`. The

InfoCenter describes the JRas Facility methods in the `com.ibm.ras` package, as it applies to all supported platforms, including z/OS or OS/390.

The following table shows the subtasks and associated procedures for logging messages and trace data for your Java application:

Subtask	Associated procedure (See . . .)
Determining which types of messages and trace data to issue or collect	• "Background on issuing application messages to the z/OS or OS/390 master console" • "Background on issuing trace requests for your application" on page 110
Preparing your Java server application to issue messages and trace requests	"Steps for coding your Java application to issue messages and trace requests" on page 112
Preparing the z/OS or OS/390 run-time environment for logging messages and collecting trace data	"Steps for preparing the z/OS or OS/390 environment for logging Java application messages and trace requests" on page 118
Viewing messages or trace data collected for your Java server application	• "Background on viewing messages and trace data" on page 121 • "Steps for using IPCS in batch mode to format application trace data" on page 122

Background on issuing application messages to the z/OS or OS/390 master console

With the WebSphere for z/OS run-time support for the Ras Toolkit (JRas support), you can issue messages from your Java application to the master console. You might want to issue messages to the master console to report serious error conditions for mission-critical applications, or to trigger automation packages. The messages your application issues also appear in the component trace (CTRACE) data set that WebSphere for z/OS uses, and in its error log stream if the messages are classified as error messages. Logging the messages is another method of recording error conditions, or collecting application data for diagnostic purposes.

WebSphere for z/OS provides code that creates and manages a message logger, which processes your application's messages. The message logger runs in the Java virtual machine (JVM) for the WebSphere for z/OS J2EE or MOFW server in which your Java application will run. To use a message logger, all you need to do in your Java application is:

1. Define the message logger,

2. Drive the method to instruct WebSphere for z/OS to create the message logger, and
3. Code messages at appropriate points in your application. To direct specific messages to the master console, your code must include the appropriate classification for each message.

Specific instructions for updating your application to use JRas support appear in “Steps for coding your Java application to issue messages and trace requests” on page 112. Before you can use those instructions to properly code messages, however, you need to understand the concepts in the following topics:
- “Defining messages through inline method calls or a message properties file”
- “Understanding how the message type affects message destinations” on page 110

Defining messages through inline method calls or a message properties file

If you want to issue messages from your Java application, you may either define the messages inline, or use a separate file to contain the messages. Generally speaking, defining messages inline is faster and requires fewer steps to complete; using a separate message properties file is a better approach for both usability and for text translation, if you plan to provide message text in a variety of languages. Regardless of whether you use the file or inline approach for defining messages, you must code methods in your Java application to issue messages at appropriate points in its processing. At those points, you use methods defined in the `RASIMessageLogger` interface to issue messages.

If you define messages inline, use `textMessage` methods to issue messages from your application. The string that you specify on the method call is what the message logger sends to the master console, error log stream, or CTRACE data set.

If you plan to use a message properties file, you need to:
1. Create the message properties file.
2. Define all messages using a key/text pair.

   The key enables the message logger to locate the appropriate message in the message file; the text is what the message logger sends to the master console, error log stream, or CTRACE data set.
3. Use the appropriate methods to tell the message logger where to find message text for your application’s messages.

   You can identify the message file to the message logger through two mechanisms:

- The `setMessageFile` method, which registers one message properties file to serve as the default file for retrieving message text.
- The `message` or `msg` methods, on which you may specify the name of the message properties file.

See "Steps for coding your Java application to issue messages and trace requests" on page 112 for specific instructions for creating a message file, rules for defining the messages in it, and examples.

Understanding how the message type affects message destinations

When you code the method to issue a message, you assign a message type to characterize the message as an error, warning, or informational message. The `RASIMessageEvent` interface defines the message types. These types define the destination of each message:

- Only informational messages (`TYPE_INFORMATION` or `TYPE_INFO`) are sent to the master console.
- Only error messages (`TYPE_ERROR` or `TYPE_ERR`) are sent to the error log stream.
- All three types of messages are sent to the CTRACE data set.

Note that messages are always logged; once you code an application to issue messages, and run that application on z/OS or OS/390, its messages will always be collected and logged.

Background on issuing trace requests for your application

The purpose of collecting trace data is to provide sufficient information to diagnose a problem with your application. With the WebSphere for z/OS run-time support for the Ras Toolkit (JRas support), you can issue trace requests from your Java application, and have the resulting trace data recorded in the component trace (CTRACE) data set that WebSphere for z/OS uses. Your application's trace data appears in the CTRACE data set for the WebSphere for z/OS J2EE or MOFW server in which your application runs.

WebSphere for z/OS provides code that creates and manages a trace logger, which processes your application's trace requests. The trace logger runs in the Java virtual machine (JVM) for the WebSphere for z/OS J2EE or MOFW server in which your Java application will run. To use a trace logger, all you need to do in your Java application is:

1. Define the trace logger,
2. Drive the method to instruct WebSphere for z/OS to create the trace logger, and
3. Code trace requests at appropriate trace points in your application.

Specific instructions for updating your application to use JRas support appear in "Steps for coding your Java application to issue messages and trace

requests” on page 112. Before you can use those instructions to properly code trace requests, however, you need to understand the concepts in the following topics:
- “Determining where to place trace points and what data to request”
- “Assigning trace types to trace points”

Determining where to place trace points and what data to request

To collect trace data for a Java application running in a WebSphere for z/OS J2EE or MOFW server, you must decide where to locate trace points in your application’s code. At those trace points, you can use `RASTraceLogger` class interfaces to request a trace entry. Typical trace points include:
- Method entry
- Method exit
- Start of a functional request
- Major checkpoints in the process of completing a request
- Completion of a functional request
- Interface to another system function
- Any unusual event, such as a detected I/O error or an unexpected exception

You must also decide what information to record in the trace entries, which can hold a variable amount of data. WebSphere for z/OS automatically collects the address space identifier (ASID) and task control block (TCB) for the unit of work or transaction, and Java name for the thread. The following are suggestions on the additional types of data you might place in the trace entries for a Java application running in a WebSphere for z/OS J2EE or MOFW server:
- Identification of the unit of work or transaction that is being serviced by the application. This can be the JOBNAME, USERID, or transaction identifier.
- For entries that trace the start of a functional request, the input parameters.
- For internal checkpoints, an identification that ties this trace entry to the original request, and information on the current status of the process.
- For unusual events, the cause of the problem and any additional data. For example, you could record any exceptions and stack traces.
- On return from a service, the return code and reason code.
- For trace entries being used for analysis rather than as a debugging aid, whatever information the user of the application needs.

Assigning trace types to trace points

For each trace point you define in your Java application, you use methods defined in the `RASITraceLogger` interface to request trace entries. As part of each trace request, you should assign a trace type for this specific request. The `RASITraceEvent` interface defines the types that you may use.

Note: The Enterprise Edition V3.02 JRas support required you to assign a trace level to trace points in your application. These assignments are still supported, so you do not have to recode any applications that use trace levels.

After you code trace requests, your Java application is capable of issuing trace requests while it runs. To actually record the trace data requested, however, the WebSphere for z/OS J2EE or MOFW server in which your application runs must be enabled for tracing. "Steps for preparing the z/OS or OS/390 environment for logging Java application messages and trace requests" on page 118 provides more detail about enabling tracing for specific trace types.

Steps for coding your Java application to issue messages and trace requests

By coding instructions for issuing messages and logging trace entries, you can improve the reliability, availability, and serviceability (Ras) of your Java server application. When your Java application runs in a WebSphere for z/OS J2EE or MOFW server, its messages appear in one or more of the following destinations, depending on the message type:

- The z/OS or OS/390 master console
- The error log stream that WebSphere for z/OS uses
- The component trace (CTRACE) data set that WebSphere for z/OS uses.

The application's trace entries appear in the same CTRACE data set.

Before you begin:

- If you want to issue messages from your Java application, you may either define the messages inline, or use a separate file to contain the messages. Decide which approach you want to use before you start coding. If necessary, see "Defining messages through inline method calls or a message properties file" on page 109 for more information about these two approaches.
- For descriptions of the JRas interfaces and methods you can use to issue messages or log trace entries, refer to the InfoCenter at `http://www.ibm.com/software/webservers/appserv/library.html`. The InfoCenter describes the JRas Facility methods in the `com.ibm.ras` package, as it applies to all supported platforms, including z/OS or OS/390.

Perform the following steps to add code to your Java server application to direct messages and trace entry requests to z/OS or OS/390 message and trace data logging facilities.

1. (Optional) Create a message properties file if you want to log messages from your application, and have not defined messages inline. For each message that the Java application issues, define the message in a key/text pair:

- Use the text portion to indicate what is to appear on the master console or in the error log stream
- Use the key, in both the message properties file and in your Java application code, to enable the run-time code to find the correct message text.

Rules:

- Always use an equals sign to separate the key from the text. For example:

  ```
  BBOJ0001=BBOJ0001 Java BO created.
  BBOJ0002=BBOJ0002 Policy number {0} obtained.
  ```

- Message text that contains variable data requires special coding to indicate the placement and content. To correctly define messages with variable text, use braces {} to indicate that a variable is to appear at a particular place in the text. Within the braces, use a digit to indicate which variable belongs at this place.

  For example, suppose your code contains the following instructions:

  ```
  String day = "Monday";
  Integer temp = new Integer(75);
  msgLogger.message(RASIMessageEvent.TYPE_INFO,
                    this,
                    "methodName",
                    "APPL061I",
                    day,
                    temp);
  ```

  To correctly define this message, you would code the following in your message properties file:

  ```
  APPL061I=APPL061I On {0}, it is {1} degrees.
  ```

---

2. Using an appropriate application development tool for your application, edit the source code for your Java application as follows:

   - Add import statements for the `com.ibm.ras` and `com.ibm.WebSphere` packages. For example, type the following:

     ```
     import com.ibm.ras.*;
     import com.ibm.websphere.ras.*;
     ```

   - Add definition statements for the message and trace loggers. For example, type the following:

     ```
     private RASIMessageLogger msgLogger = null;
     private RASITraceLogger trcLogger = null;
     ```

---

3. Edit the constructor for your Java application to create the message logger, trace logger, or both:

For this type of logger:	Complete the following steps:
Message	• Use the createRASMessageLogger method to request a message logger • (Optional) Define the message properties file, if you are using the file, rather than inline, approach for issuing messages from your application.
Trace	Use the createRASTraceLogger method to request a trace logger

Rules:

- Applications must refer to the object returned by the `createRASMessageLogger` method as a type RASIMessageLogger object.
- Applications must refer to the object returned by the `createRASTraceLogger` method as a type RASITraceLogger object.

Tip: Avoid using logger names that begin with the `com.ibm.` prefix, which is reserved for use by WebSphere for z/OS.

4. If you want to issue messages from your Java application, add messages at appropriate points in the application's source code.

   Rules:

   - If you are defining messages inline, use the `textMessage` methods in the `RASIMessageLogger` interface, specifying the complete message in a string on the method call.
   - If you are using a message properties file, use the `message` or `msg` methods in the `RASIMessageLogger` interface, specifying the message key on the method call. For example:

     ```
     msgLogger.message(RASIMessageEvent.TYPE_INFO,
                           "com.myCompany.JRasSample",
                           "doSomething",
                           "BBOJ0001");
     ```

   - For each message, assign an appropriate type, as defined in the `RASIMessageEvent` interface. These types define the destination of each message:

Message type	Destination
`TYPE_INFORMATION` or `TYPE_INFO`	Master console and CTRACE data set
`TYPE_ERROR` or `TYPE_ERR`	Error log and CTRACE data set
`TYPE_WARNING` or `TYPE_WARN`	CTRACE data set only

Notes:

a. Assign only one message type to each message.
b. If you do not assign a type to a message, or specify "null" for the type, the Java compiler issues an error message.
c. If you assign a type that is not valid, the message logger processes the message as a `TYPE_INFORMATION` (or `TYPE_INFO`) message.

- Each character used in a message must map to an EBCDIC character.
- When routing a message to the master console, WebSphere for z/OS sends only the first 700 characters of message text.

Limitation: When writing an error message to the error log stream, WebSphere for z/OS uses only 512 characters of data, including the information it adds to the message text for identification. (This additional information consists of the date, time, organization name, and so on.) See *WebSphere Application Server V4.0.1 for z/OS and OS/390: Messages and Diagnosis*, GA22-7837 for the format and content of error log stream entries for application messages.

---

5. If you want to collect trace data for your Java application, add trace requests at appropriate points in the application's source code.

Rules:

- For each trace request, assign an appropriate type as defined in the RASITraceEvent interface.

  Note: If you do not assign a type to a trace request, the trace logger ignores that trace request.
- Each character used in trace data must map to an EBCDIC character.

Limitation: When processing trace data, WebSphere for z/OS uses only a limited amount of hexadecimal or character data:

- For hexadecimal trace data (from tracing Java byte arrays), WebSphere for z/OS truncates the data after 1024 bytes.
- For character trace data, WebSphere for z/OS substitutes the literal `***BUFFER OVERFLOW***` when that trace data exceeds 16384 characters. This cumulative limit includes 1-byte string terminators for each character string.

Tip: To improve your application's performance, you may use one of the following alternatives:

- Wrap trace calls in a test of the `RASTraceLogger.isLogging` variable, which is set to `false` when trace logging is not active.

- Use the `isLogging` method in an `if` statement to test whether trace logging is active for any level of tracing.
- Use the `isLoggable` method to determine whether logging is active for the designated trace type.

With the first two approaches, the overhead of creating a trace entry does not take place if trace logging is not active. In contrast, the `isLoggable` method requires more overhead, but might be the better option, especially if some level of tracing is always active.

6. Using the appropriate application development tools for your Java application, generate and compile the code for your application.

When you have executable code for your Java application, you are ready to complete the steps listed in “Steps for preparing the z/OS or OS/390 environment for logging Java application messages and trace requests” on page 118.

Example: The following example illustrates the coding requirements described in the instructions above. The example assumes the use of a message properties file, named `com/myCompany/JRasSample.properties`, which contains the following message definitions:

```
BBOJ0001=BBOJ0001 Java BO created.
BBOJ0002=BBOJ0002 Policy number {0} obtained.
BBOJ0003=BBOJ0003 Java BO destroyed.
```

```
package com.myCompany;

// Import JRas and Websphere packages
import com.ibm.ras.*;
import com.ibm.websphere.ras.*;

public class JRasSample
{
  // Loggers
  private RASIMessageLogger msgLogger = null;
  private RASITraceLogger trcLogger = null;
  // Message file
  private static final String MSG_FILE = "com.myCompany.JRasSample";
  // Array of trace objects
  Object[] objs = new Object[3];

  // Constructor
  public JRasSample()
  {
    // Get logger manager object
    Manager manager = Manager.getManager();
    // Get logger
    trcLogger = manager.createRASTraceLogger("com.myCompany","myProduct",
                                             "myComponent","myLogger.COM");
    msgLogger = manager.createRASMessageLogger("com.myCompany","myProduct",
                                               "myComponent","myLogger.COM");
    msgLogger.setMessageFile(MSG_FILE);
  }

  // Example of JRas trace events and messages
  public int doSomething(String parm1,String parm2,String parm3)
  {
    int returnValue = 0;
    byte[] byteArray = {1,2,3,4,5};

    // Trace input parameters
    objs[0] = parm1;
    objs[1] = parm2;
    objs[2] = parm3;
    if (trcLogger.isLoggable(RASITraceEvent.TYPE_ENTRY_EXIT))
      trcLogger.entry(RASITraceEvent.TYPE_ENTRY_EXIT,
                      "com.myCompany.JRasSample",
                      "doSomething",
                      objs);
    if (trcLogger.isLoggable(RASITraceEvent.TYPE_MISC_DATA))
    {
      // Trace a text string
      trcLogger.trace(RASITraceEvent.TYPE_MISC_DATA,
                      "com.myCompany.JRasSample",
                      "doSomething",
                      "Text data to be traced");
      // Trace binary data
      trcLogger.trace(RASITraceEvent.TYPE_MISC_DATA,
                      "com.myCompany.JRasSample",
                      "doSomething",
                      byteArray);
      // Trace the current stack
      trcLogger.stackTrace(RASITraceEvent.TYPE_MISC_DATA,
                           "com.myCompany.JRasSample",
                           "doSomething");
    }
    // Issue informational message to WTO and CTRACE
    msgLogger.message(RASIMessageEvent.TYPE_INFO,
                      "com.myCompany.JRasSample",
                      "doSomething",
                      "BBOJ0001");
    // Issue warning message to CTRACE
    msgLogger.message(RASIMessageEvent.TYPE_WARN,
                      "com.myCompany.JRasSample",
                      "doSomething",
                      "BBOJ0002",
                      "123");
    // Issue error message to error log and CTRACE
    msgLogger.message(RASIMessageEvent.TYPE_ERR,
                      "com.myCompany.JRasSample",
                      "doSomething",
                      "BBOJ0003");
    // Trace return value
    if (trcLogger.isLoggable(RASITraceEvent.TYPE_ENTRY_EXIT))
      trcLogger.exit(RASITraceEvent.TYPE_ENTRY_EXIT,
                     "com.myCompany.JRasSample",
                     "doSomething",
                     returnValue);
    return returnValue;
  }

  // This method is invoked when a JRasSample object is traced
  public String toString()
  {
    String traceString = "This is the JRasSample object trace data";
    return traceString;
  }

  public static void main(String[] args)
  {
    JRasSample sample = new JRasSample();
    sample.doSomething("parm1","parm2","parm3");
  }
}
```

Steps for preparing the z/OS or OS/390 environment for logging Java application messages and trace requests

Before you begin:

- Check with the appropriate installation personnel to determine whether error log streams and component trace data sets were set up during the installation process for WebSphere for z/OS. While error logs and CTRACE data sets might be available already, your installation personnel might determine that changes are necessary to handle your application data, as well as current data from other WebSphere for z/OS servers and applications. For example, your installation may set up either a common error log stream for all WebSphere for z/OS servers, or a separate log stream for each individual server. Your installation might want to switch from using a common log to separate logs, to accomodate additional diagnostic data from your Java applications.
- To turn on tracing for an application in a J2EE or MOFW server, you need to edit or create a JVM properties file. This task might require special authorization to edit or store this file in the appropriate directory. Check with the system programmer who installed WebSphere for z/OS on your system.

Notes:

1. Instructions for setting up error log streams appear in *WebSphere Application Server V4.0.1 for z/OS and OS/390: Installation and Customization*, GA22-7834.

2. Instructions for setting up and running CTRACE appear in *WebSphere Application Server V4.0.1 for z/OS and OS/390: Messages and Diagnosis*, GA22-7837.

Perform the following steps to set up the z/OS or OS/390 environment for JRas support:

1. On z/OS or OS/390, create a trace settings file in the hierarchical file system (HFS), if you want to enable the WebSphere for z/OS J2EE or MOFW server to collect and log your application's trace data. In this file, type the trace settings that you want, in the following format:
   `logger_name=type=[enabled|disabled]`

   Example:

   ```
   myLogger.COM=all=enabled
   ```

   logger_name corresponds to the logger name that you specified in the source code for your application, when you coded the create method to obtain a trace logger. To enable logging support for more than one logger name, you may specify a common prefix with an asterisk (for example, `a.b.c.*`), rather than spelling out each logger name in its entirety. Specifying something like `a.b.c.*` enables logging for loggers named `a.b.c.d` and `a.b.c.e`

   Tip: Avoid using logger names that begin with the `com.ibm.` prefix, which is reserved for use by WebSphere for z/OS.

   type corresponds to one of the property values in the following table. Property types are case-sensitive.

Table 9. Trace setting property types and their corresponding JRas trace types

Specifying this property type:	Enables tracing for the following JRas trace types:
all	All supported RASITraceEvent types
event	• RASITraceEvent.TYPE_ERROR_EXC • RASITraceEvent.TYPE_SVC • RASITraceEvent.TYPE_OBJ_CREATE • RASITraceEvent.TYPE_OBJ_DELETE • RASITraceEvent.TYPE_LEVEL1
entryExit	• RASITraceEvent.TYPE_ENTRY_EXIT • RASITraceEvent.TYPE_API • RASITraceEvent.TYPE_CALLBACK • RASITraceEvent.TYPE_PRIVATE • RASITraceEvent.TYPE_PUBLIC • RASITraceEvent.TYPE_STATIC • RASITraceEvent.TYPE_LEVEL1 • RASITraceEvent.TYPE_LEVEL2

Table 9. Trace setting property types and their corresponding JRas trace types (continued)

Specifying this property type:	Enables tracing for the following JRas trace types:
debug	• RASITraceEvent.TYPE_MISC_DATA • RASITraceEvent.TYPE_LEVEL1 • RASITraceEvent.TYPE_LEVEL2 • RASITraceEvent.TYPE_LEVEL3

Rules:

- You may use the same trace properties file to enable different trace types for given loggers. If you do not use a separate line to define each logger's trace types, you must use a single colon (:) to distinguish each logger's trace settings.

  Example (separate line for each logger):

  ```
  com.aCompany.*=all=enabled
  com.anotherCompany.*=event=enabled
  ```

  Example (same line for each logger):

  ```
  com.aCompany.*=all=enabled:com.anotherCompany.*=event=enabled
  ```
- To specify more than one trace type for a logger, separate each trace type with a comma (,)

  Example:

  ```
  com.aCompany.aComponent=debug=enabled,event=enabled
  ```

---

2. Create a new or edit an existing Java virtual machine (JVM) properties file to point to the trace settings file you just created. This properties file, named `jvm.properties`, changes the default settings for the JVM that runs in a WebSphere for z/OS J2EE or MOFW server.

   Rules:

   - You must set the `com.ibm.ws390.trace.settings` system property to the fully qualified directory path and file name for your trace settings file. If you do not specify this system property, or specify the path and file name incorrectly, all trace types are disabled (the default setting).
   - You must make the `jvm.properties` file accessible to WebSphere for z/OS, so it can find and use your property settings when activating the server. Place the `jvm.properties` file in the same HFS directory in which WebSphere for z/OS places the `current.env` file containing environment variable settings for the server in which your Java application will run. See "Appendix A. Environment files" on page 125 for more information about this directory.

- Trace logging cannot be dynamically started or stopped.

---

3. Check the environment variable settings related to the J2EE or MOFW server's use of component trace. You might want to modify some of the values to accomodate additional trace entries in the CTRACE data set. Specifically, check the following environment variable settings:
   - TRACEBUFFCOUNT
   - TRACEBUFFSIZE

---

4. Start the WebSphere for z/OS J2EE or MOFW server in which your application will run:
   - If you have set up JRas support for an existing application that you already installed in a server, you need to:
     a. Make sure your newly compiled code replaces the existing code.
     b. Make sure the WebSphere for z/OS server picks up any modifications you made to the `jvm.properties` file or the environment variables. You need to stop and restart the server to pick up these changes.
   - If you have set up JRas support for a brand-new application, follow the appropriate process to assemble and install your Java application in a WebSphere for z/OS server. For Java applications to be installed in a MOFW server, see "Chapter 4. Deploying CORBA applications in WebSphere for z/OS MOFW servers" on page 71.

Background on viewing messages and trace data

Once your Java application starts running, you can view its messages and trace data, as follows:

If you want to view this type of output:	**Use the following instructions:**
Messages on the z/OS or OS/390 master console	The message logger automatically routes messages to the master console in a readable format. Their appearance and duration depend on how your installation has set up its console configuration. If necessary, see *z/OS MVS Planning: Operations*, SA22-7601 for an explanation of ways to configure consoles, including controlling message display, scrolling, and deletion.
Messages in the error log stream	To view messages in the error log stream, use the log browse utility (BBORBLOG). See *WebSphere Application Server V4.0.1 for z/OS and OS/390: Messages and Diagnosis*, GA22-7837 for instructions for using the log browse utility, and for examples of message output.

If you want to view this type of output:	Use the following instructions:
Messages or trace data in Component Trace	To view messages or application trace data in Component Trace, you must use the interactive problem control system (IPCS) in one of the following ways: • Line mode on a terminal (IPCS CTRACE command), • Full-screen mode on a terminal (IPCS dialog), or • Batch mode, using the terminal monitor program. **Recommendation:** If you are not familiar with IPCS, TSO/E and ISPF, use IPCS in batch mode to format and view trace data, as described in "Steps for using IPCS in batch mode to format application trace data". See *WebSphere Application Server V4.0.1 for z/OS and OS/390: Messages and Diagnosis*, GA22-7837 for instructions for using the IPCS dialog, and for examples of message and trace data output.

Note: When you view the trace data for your Java application, messages and CTRACE records might not appear in the order in which your application issued the message or trace requests. All message requests appear in sequential order, relative to each other. Similarly, all CTRACE records appear in order, relative to each other. Different types of trace data, however, might not be in sequence; for example, messages issued after trace requests might show up in trace output before the trace requests.

Steps for using IPCS in batch mode to format application trace data

To view messages or application trace data from Component Trace, you must use the interactive problem control system (IPCS) to format the data. Using IPCS in batch mode is the easiest method of formatting data, especially if you do not have much experience with using IPCS, TSO/E and ISPF. Through batch mode, you can use IPCS to format trace data and write it to an MVS data set. Optionally, you may copy the contents of that data set into an HFS file for viewing.

Before you begin: You must create an IPCS dump directory before you can use IPCS in batch mode. When setting up IPCS, your installation may customize IPCS for its users. This customization can include modifying the IBM-supplied BLSCDDIR CLIST with default values for creating an IPCS dump directory.

If your installation has modified the BLSCDDIR CLIST, perform the following steps to create an IPCS dump directory:

1. Decide on a fully-qualified data set name for the directory.

2. From the TSO/E command prompt, enter the BLSCDDIR command, specifying the data set name. For example, to create a dump directory named IBMUSER.DDIR, enter:

   ```
   %blscddir dsn('ibmuser.ddir')
   ```

If your installation has not customized IPCS, you might need to alter other BLSCDDIR CLIST parameters. See *z/OS MVS IPCS User's Guide*, SA22-7596 and *z/OS MVS IPCS Commands*, SA22-7594 for more details about using the BLSCDDIR CLIST to create a dump directory.

Perform the following steps to use IPCS in batch mode to format application trace data:

1. Create a file and copy the following sample JCL into it. This JCL invokes IPCS to extract and format JRAS trace data and write it into an MVS data set, and then uses the TSO/E OPUT command to copy the formatted data from the MVS data set into an HFS file.

   ```
   //IBMUSERX   JOB ,
   // CLASS=J,NOTIFY=&SYSUID,MSGCLASS=H
   //IPCS       EXEC PGM=IKJEFT01,REGION=4096K,DYNAMNBR=50
   //IPCSDDIR   DD DSN=IBMUSER.DDIR,DISP=SHR
   //IPCSDOC    DD SYSOUT=H
   //JRASTRC    DD DSN=IBMUSER.CB390.CTRACE,DISP=SHR
   //IPCSPRNT   DD DSN=IBMUSER.IPCS.OUT,DISP=OLD
   //SYSTSPRT   DD SYSOUT=*
   //SYSTSIN    DD *
   IPCS
   DROPDUMP DDNAME(JRASTRC)
   PROFILE LINESIZE(80)PAGESIZE(99999999)
   SETDEF NOCONFIRM
   CTRACE COMP(SYSBBOSS) DDNAME(JRASTRC) FULL PRINT +
          NOTERMINAL
   DROPDUMP DDNAME(JRASTRC)
   END
   /*
   //OPUT       EXEC PGM=IKJEFT01,REGION=4096K,DYNAMNBR=50
   //SYSTSPRT   DD SYSOUT=*
   //SYSTSIN    DD *
   oput 'ibmuser.ipcs.out' '/u/ibmuser/ipcs/jrastrace.txt' TEXT
   /*
   ```

2. Edit the sample JCL to replace IBMUSER.DDIR with the data set name that you used for the IPCS dump directory you created.

   Notes:

   a. Use the PAGESIZE parameter on the PROFILE statement only if you do not want to print the output data set.
   b. You may replace the HFS file name with the name of an existing HFS file, but you do not have to do so. The OPUT command processing will

create a new HFS file, if the one specified does not exist, and grants read and write access to that file for your user ID only.

If you do specify an existing HFS file, the `OPUT` command processing will write over any data that is already in that file. If you want to know more about the `OPUT` command, see *z/OS UNIX System Services Command Reference*, SA22-7802.

c. Change the data set name specified on the `JRASTRC DD` in the example to the name of the data set containing the CTRACE data.

d. Change the name of the MVS data set on both the `JRASTRC DD` statement and the `OPUT` command in the SYSTSIN stream, as necessary. The formatted output of the JRAS CTRACE data is first written to the MVS data set specified by the `IPCSPRNT DD` statement and then (optionally) copied to the HFS data set. You must either pre-allocate this data set, or change the sample JCL to allocate the data set. This data set should have a record format of VBA and a record length of 133.

---

3. Submit the JCL to start the IPCS batch job.

Once you are done you can use a UNIX editor, such as vi, to view your trace data in the HFS file. If you want to know more about the UNIX editors, see *z/OS UNIX System Services User's Guide*, SA22-7801.

Appendix A. Environment files

This appendix provides reference information for environment files and environment variables.

Environment files and environment variables

This section describes:

- How WebSphere for z/OS manages environment variables and environment files.
- How run-time server start procedures point to their environment files.
- Environment variables for z/OS or OS/390 clients.
- The syntax and meaning of the run-time environment variables.

Note: You may require additional environment variables to be set in your z/OS or OS/390 application development environment. See "Steps for setting up the application development environment" on page 40.

How WebSphere for z/OS manages server environment variables and environment files

After the bootstrap process during installation and customization, WebSphere for z/OS manages environment data through the Administration application and writes the environmental data into the system management database. To add or change environment variable data, you must enter environment data pairs (an environment variable name and its value) on the sysplex, server, or server instance properties form. When you activate a conversation or prepare for a cold start, the environment variable data is written to HFS files. WebSphere for z/OS determines which values are the most specific for an environment file. For instance, a setting for a server instance takes precedence over the setting for the same variable for its server, and a setting for a server takes precedence over the setting for the same variable for its sysplex.

If you modify an environment file directly and not through the Administration application, any changes are overwritten when you activate a conversation or prepare for a cold start.

When you activate a conversation or prepare for a cold start, WebSphere for z/OS writes the environment data to an HFS file for each server instance. The path and name for each environment file is:

CBCONFIG`/controlinfo/envfile/`*SYSPLEX*`/`*SRVNAME*`/current.env`

where

CBCONFIG
Is a read/write directory that you specify at installation time as the directory into which WebSphere for z/OS is to write configuration data and environment files. At installation time, we call this directory TARGETDIR. The default is `/WebSphere390/CB390`.

Rule: The System Management group (default CBCFG1) and user ID (default CBSYMSR1) must own each directory and subdirectory in CBCONFIG. If the System Management group and user ID do not own CBCONFIG, use the chown command to make them the owner of each directory and subdirectory in CBCONFIG. Thus, if you use the default CBCONFIG, you must use the chown command to give the System Management group and user ID ownership of /WebSphere390 and /WebSphere390/CB390.

Example:

```
chown -R CBSYMSR1:CBCFG1 /WebSphere390
```

SYSPLEX
Is the name of your sysplex. WebSphere for z/OS derives this name from the predefined &SYSPLEX JCL variable.

SRVNAME
Is the server instance name.

| Except for the initial installation of WebSphere for z/OS, you must manage
| the environment variables through the Administration application. At initial
| installation, the customization dialog modifies an initial environment file,
| which the bootstrap job uses.

There are, therefore, two distinct situations in which you define environmental data for your servers. Matching those situations are two distinct ways you create the environment data:

| 1. Prior to the bootstrap process, the customization dialog creates the
| environment file for you. The bootstrap job reads the file and places the
| environmental data into the system management database.
2. Defining and managing environmental data through the Administration application. In this situation, you enter environment data pairs (an environment name and its value—no "=") through a panel in the Administration application.

How run-time server start procedures point to their environment files

WebSphere for z/OS run-time server start procedures must point to an environment file for configuration information. The start procedures use a BBOENV DD statement with a PATH parameter that points to an HFS file. The BBOENV DD statement is:

```
//BBOENV    DD   PATH='&CBCONFIG/&RELPATH/&SYSPLEX/&SRVNAME/current.env'
```

where

&CBCONFIG
: Is a variable you set in the start procedure. It must match the read/write directory that you specify at installation time as the directory into which WebSphere for z/OS is to write configuration data and environment files. The default is `WebSphere390/CB390`.

&RELPATH
: Is a subdirectory (`controlinfo/envfile`). Its value must not change.

&SYSPLEX
: Is the name of your sysplex. Because it is a predefined JCL variable, you do not need to set it in your start procedure.

&SRVNAME
: Is the server instance name. By specifying the server instance name when you start the procedure, you can use the same start procedure for other server instances.

  Example: To pass the server instance name BBOASR1A to its start procedure, specify:

  ```
  s bboasr1.bboasr1a,srvname='BBOASR1A'
  ```

  To use the same start procedure for server instance BBOASR1B, specify:

  ```
  s bboasr1.bboasr1b,srvname='BBOASR1B'
  ```

Environment variables for z/OS or OS/390 clients

The Administration application does not manage environment variables for z/OS or OS/390 clients. You must create and manage z/OS or OS/390 client environment files and point to them from client programs. Table 10 on page 130 tells you which environment variables are required or optional for z/OS or OS/390 clients.

Note on using substitution variables

You cannot use variable substitution ($ variables) in environment statements. The variable substitution that is used in UNIX shell environments is not implemented in the Language Environment (LE). Because WebSphere for z/OS processes environment variables in the Language Environment, use of variables such as $PATH in a path environment variable will fail.

Example:

UNIX shell environments often set up paths by appending the new path to the existing path, like this:

```
PATH=yourdir
PATH=$PATH/mydir
```

The resulting path is PATH=yourdir/mydir after substitution for the $PATH variable. However, because WebSphere for z/OS processes the environment variables in the Language Environment, where no variable assignment is made, the resulting path would be PATH=$PATH/mydir.

Environment variable syntax

You must follow this syntax only when defining your initial environment file before the bootstrap process.

Rules: The following are the syntax rules:

- The syntax of the environment variables follows this pattern:

  ```
  VARIABLE=VALUE
  ```

  Where:

  VARIABLE
    is the environment variable.

  VALUE
    is the setting for the variable. The descriptions define possible values for each variable.
- Leading and trailing white space (blanks or tabs) for both variables and values is ignored.

  Example: The two following lines yield the same result:

  ```
  VARIABLE1=VALUE1
  ```

  and

  ```
        VARIABLE1      =      VALUE1
  ```
- "=" is required.
- Blank lines are ignored.
- Code upper and lowercase characters as documented in this topic.
- To comment out an environment variable, simply add a character, such as '#', to the variable. For example, you could change `TRACEALL=0` to `#TRACEALL=0`. The system ignores such coding because the variable does not begin with an alphabetic character.
- | Language Environment limits the size of environment variables to 2K.

Environment variable use

Not all environment variables need to be used for each server or client. Table 10 on page 130 tells you where to use a given environment variable. Here are the meanings for what appears in each column:

- "R" means required.
- "O" means optional.
- "F" means required in a future release.

- A blank in the Default column means the variable is not set.
- A blank in other columns means the variable is not used.

Footnotes appear at the end of the table.

Note: The default settings and examples use the standard _CEE_ENVFILE syntax. You do not use this syntax when defining environmental data in the Administration application.

Table 10. Where to use environment variables

Environment variable=<default>	**Daemon server instance**	**System Management server instance**	**Naming server instance**	**Interface Repository instance**	**Business application server instance**	**z/OS or OS/390 client**
BBOC_HTTP_IDENTITY=*USER_ID*					R[1]	
BBOC_HTTP_INPUT_TIMEOUT =*n*					R[1]	
BBOC_HTTP_LISTEN_IP_ADDRESS=*IP_ADDRESS*					R[1]	
BBOC_HTTP_OUTPUT_TIMEOUT=*n*					R[1]	
BBOC_HTTP_PERSISTENT_SESSION_TIMEOUT =*n*					R[1]	
BBOC_HTTP_PORT=*n*					R[1]	
BBOC_HTTP_SESSION_GC=*n*					R[1]	
BBOC_HTTP_TRANSACTION_CLASS= *TRANSACTION_CLASS*					R[1]	
BBODUMP=3	O	O	O	O	O	
BBODUMP_CEE3DMP_OPTIONS=	O	O	O	O	O	
BBOLANG=ENUS	O	O	O	O	O	O
BEAN_DELETE_SLEEP_TIME=4200		R[2]				
CBCONFIG=/WebSphere390/CB390	R	R	R	R	R	
CLASSPATH=		O	O	O	O[3]	
CLIENT_DCE_QOP=NO_PROTECTION						O
CLIENT_HOSTNAME=						O
CLIENTLOGSTREAMNAME=						O
CLIENT_RESOLVE_IPNAME=<value for RESOLVE_ IPNAME>		O	O	O	O	O
CLIENT_TIMEOUT=						
com.ibm.ws.naming.ldap.containerdn= <ibm-wsnTree=t1,o=<org>, c=<country>>			O			

Table 10. Where to use environment variables (continued)

Environment variable=<default>	**Daemon server instance**	**System Management server instance**	**Naming server instance**	**Interface Repository instance**	**Business application server instance**	**z/OS or OS/390 client**
com.ibm.ws.naming.ldap.domainname= *domain name*			O			
com.ibm.ws.naming.ldap.masterurl= ldap://<ip name>:<port>			O			
CONFIGURED_SYSTEM=	R[4]	R[4]	R[4]	R[4]	R[4]	
DAEMON_IPNAME=	R	O				
DAEMON_PORT=5555	O[5]	O[5]				
DEFAULT_CLIENT_XML_PATH=						O[6]
DEFAULT_UNAUTH_CLIENT_ID=CBGUEST		O				
DM_GENERIC_SERVER_NAME=CBDAEMON	O[5]	O[5]				
DM_SPECIFIC_SERVER_NAME=DAEMON01	O[7]	O[7]	O[7]	O[7]	O[7]	
HOME=						O
IBM_OMGSSL=0					O	
ICU_DATA=/usr/lpp/WebSphere/bin/		R				
IR_GENERIC_SERVER_NAME=CBINTFRP		O				
IR_SPECIFIC_SERVER_NAME=INTFRP01	O[7]	O[7]	O[7]	O[7]	O[7]	
IRPROC=BBOIR	O	O				
IVB_DEBUG_ENABLED=					O[8]	O[8]
IVB_DRIVER_PATH= /usr/lpp/WebSphere		R				
IVB_TRACE_HOST=						O[8]
IVB_TRACE_PORT=2102						O[8]
java.naming.security.credentials=<password>		O				
java.naming.security.principal=<userid>		O				

Table 10. Where to use environment variables (continued)

Environment variable=<default>	**Daemon server instance**	**System Management server instance**	**Naming server instance**	**Interface Repository instance**	**Business application server instance**	**z/OS or OS/390 client**
JAVA_COMPILER=					O	O
JAVA_IEEE754=						O[9]
JVM_BOOTCLASSPATH=		O			O	
JVM_BOOTLIBRARYPATH=		O			O	
JVM_DEBUG=		O			O	
JVM_DEBUG_PORT=					O	O[8]
JVM_ENABLE_CLASS_GC=		O			O	
JVM_ENABLE_VERBOSE_GC=		O			O	
JVM_EXTRA_OPTIONS=					O	
JVM_HEAPSIZE=256					O	
JVM_LOCALREFS=		O			O	
JVM_LOGFILE=					O	O
JVM_MINHEAPSIZE=		O			O	
LDAPBINDPW=		F	R[10]			
LDAPCONF=		F	R[10]			
LDAPHOSTNAME=		F	R[10]			
LDAPIRBINDPW=		F		R[11]		
LDAPIRCONF=		F		R[11]		
LDAPIRHOSTNAME=		F		R[11]		
LDAPIRNAME=		F		R[11]		
LDAPIRROOT=		F		R		
LDAPNAME=		F	R[10]			

Table 10. Where to use environment variables (continued)

Environment variable=<default>	**Daemon server instance**	**System Management server instance**	**Naming server instance**	**Interface Repository instance**	**Business application server instance**	**z/OS or OS/390 client**
LDAPROOT=		F	R			
LIBPATH=		O	O	O	O[3]	
LOGSTREAMNAME=	O	O				
MIN_SRS=[0 for MOFW, 1 for J2EE]					O	
NM_GENERIC_SERVER_NAME=CBNAMING		O				
NM_SPECIFIC_SERVER_NAME=NAMING01	O[7]	O[7]	O[7]	O[7]	O[7]	
NMPROC=BBONM	O	O				
OTS_DEFAULT_TIMEOUT=30	O	O	O	O	O	
OTS_MAXIMUM_TIMEOUT=60	O	O	O	O	O	
PATH=					O	O
RAS_MINORCODEDEFAULT= NODIAGNOSTICDATA						
REM_DCEPASSWORD=						O
REM_DCEPRINCIPAL=						O
REM_PASSWORD=		O[12]	O[12]	O[12]	O[12]	O
REM_USERID=		O[12]	O[12]	O[12]	O[12]	O
RESOLVE_IPNAME=		O[13]	O[14]	O[14]	O[14]	R[15]
RESOLVE_PORT=900		O	O	O	O	O
SM_DEFAULT_ADMIN= CBADMIN		O				
SM_GENERIC_SERVER_NAME=CBSYSMGT		O				
SM_SPECIFIC_SERVER_NAME=SYSMGT01	O[7]	O[7]	O[7]	O[7]	O[7]	
SMPROC=BBOSMS	O	O				
SOMOOSQL=					O	

Table 10. Where to use environment variables (continued)

Environment variable=<default>	Daemon server instance	System Management server instance	Naming server instance	Interface Repository instance	Business application server instance	z/OS or OS/390 client
SRVIPADDR=	O	O	O	O	O	
SSL_KEYRING=						O
SYS_DB2_SUB_SYSTEM_NAME=DB2	R	R	R	R	R	
TRACEALL=1	O	O	O	O	O	O
TRACEBASIC=	O	O	O	O	O	O
TRACEBUFFCOUNT=4	O	O	O	O	O	
TRACEBUFFLOC=(Server: BUFFER, Client: SYSPRINT)	O	O	O	O	O	O
TRACEBUFFSIZE=1M	O	O	O	O	O	
TRACEDETAIL=	O	O	O	O	O	O
TRACEMINORCODE=						
TRACEPARM=00	O					

Table 10. Where to use environment variables (continued)

Environment variable=<default>	Daemon server instance	System Management server instance	Naming server instance	Interface Repository instance	Business application server instance	z/OS or OS/390 client

Notes:

1. Required if using the HTTP Transport Handler function of the J2EE server.
2. Required when stateful session beans in J2EE servers are activated based on a transaction, rather than activated only once.
3. Required for server regions that use Java, including the IMS PAA and CICS PAA.
4. This environment variable is automatically added to each server instance's environment file and should not be edited.
5. If you specify a value for the Daemon Server, you must provide the same value for the System Management Server control region.
6. Required when the client uses the System Management Scripting API.
7. You must specify this for the second and subsequent systems in a sysplex.
8. Required only when you are using the IBM Object Level Trace and Distributed Debugger Tools to trace and/or debug client and server application components.
9. Required for Java clients that run on z/OS or OS/390.
10. LDAPCONF is mutually exclusive with LDAPBINDPW, LDAPHOSTNAME, and LDAPNAME. Either LDAPCONF is required, or LDAPBINDPW, LDAPHOSTNAME, and LDAPNAME are required.
11. LDAPIRCONF is mutually exclusive with LDAPIRBINDPW, LDAPIRHOSTNAME, and LDAPIRNAME. Either LDAPIRCONF is required, or LDAPIRBINDPW, LDAPIRHOSTNAME, and LDAPIRNAME are required.
12. Used when a server becomes a remote client of another server.
13. For the control region, the default is the value of DAEMON_IPNAME during bootstrap.
14. For the server region, the default is the local system IP name. Generally, do not code.
15. Optional if a Daemon Server is on the same system as the client, in which case the default is the local system IP name.

Environment variable descriptions

BBOC_HTTP_IDENTITY=*USER_ID*
Specifies a valid SAF user ID which will be used as the current security principal for this HTTP request. The user ID will be treated as an authenticated user by the Web container. If this variable is not specified, the request will be executed under the SCO's "local Identity" (Local_Identity()).

Example:

```
BBOC_HTTP_IDENTITY=SECURITY1
```

BBOC_HTTP_INPUT_TIMEOUT=*n*
The time in seconds that the J2EE server will allow for the complete HTTP request to be received before cancelling the connection. The default value is 10 seconds.

Example:

```
BBOC_HTTP_INPUT_TIMEOUT=10
```

BBOC_HTTP_LISTEN_IP_ADDRESS=*IP_ADDRESS*
Specifies the IP address, in dotted decimal format, that WebSphere for z/OS J2EE servers use to listen for HTTP client connection requests. This IP address is used by the server to bind to TCP/IP. Normally, the server will listen on all IP addresses configured to the local TCP/IP stack. However, if you want to fence the work or allow multiple heterogeneous servers to listen on the same port, you can use BBOC_HTTP_LISTEN_IP_ADDRESS. The specified IP address becomes the only IP address over which this control region receives inbound HTTP requests.

Example:

```
BBOC_HTTP_LISTEN_IP_ADDRESS=9.117.43.16
```

BBOC_HTTP_OUTPUT_TIMEOUT=*n*
The time, in seconds, that the J2EE server will wait from the time the complete HTTP request is received until output is available to be sent to the client. The default value is 120 seconds.

Example:

```
BBOC_HTTP_OUTPUT_TIMEOUT=120
```

BBOC_HTTP_PERSISTENT_SESSION_TIMEOUT =*n*
Specifies the time, in seconds, that the J2EE server will wait between requests issued over a persistent connection from an HTTP client. After the server sends a response, it uses the persistent timeout to determine how long it should wait for a subsequent request before cancelling the persistent connection. The default value is 30 seconds.

Example:

```
BBOC_PERSISTENT_SESSION_TIMEOUT=10
```

BBOC_HTTP_PORT=*n*

Specifies the port at which the J2EE server listens for HTTP requests. Any requests received over the HTTP port will be directed to the Web container for processing.

If this variable is not specified, the J2EE server will not listen for HTTP requests directly.

The use of this HTTP port does not preclude the use of the WebSphere for z/OS plug-in with this J2EE server instance. The Web container is capable of simultaneously processing requests received directly through the HTTP port as well as from the WebSphere for z/OS plug-in.

Note: Currently, HTTP requests received over this HTTP port are not able to be authenticated using the mechanisms described in the J2EE Specification.

Example:

```
BBOC_HTTP_PORT=8080
```

BBOC_HTTP_SESSION_GC=*n*

An integer value indicating the maximum number of HTTP requests that will be processed over a single connection from an HTTP client. When the maximum number of requests have been processed, the client connection will be closed. Set this value to 0 or 1 to turn off persistent connection processing. The default value is 50.

Example:

```
BBOC_HTTP_SESSION_GC=50
```

BBOC_HTTP_TRANSACTION_CLASS=*TRANSACTION_CLASS*

A valid WLM transaction class, which will be used in the creation of the WLM enclave for all HTTP requests. If a valid WLM transaction class is not specified, no transaction class will be set for the enclave.

Example:

```
BBOC_HTTP_TRANSACTION_CLASS=TCLASSA
```

BBODUMP=*n*

Specifies the default dump used by the signal handler. Valid values and their meanings are:

0 No dump is generated.

1 A ctrace dump is taken.

2 A cdump dump is taken.

3 A csnap dump is taken.

4 A CEE3DMP dump is taken. CEE3DMP generates a dump of Language Environment and the member language libraries. Sections of the dump are selectively included, depending on dump options specified, either by default or through the BBODUMP_CEE3DMP_OPTIONS environment variable. By default, this value passes THREAD(ALL) BLOCKS to CEE3DMP. You can override the default options for CEE3DMP through the BBODUMP_CEE3DMP_OPTIONS environment variable.

For more information about CEE3DMP and its options, see *z/OS Language Environment Programming Reference*, SA22-7562.

If you do not specify BBODUMP, the default value is 3 (a csnap dump is taken).

Example:

```
BBODUMP=3
```

BBODUMP_CEE3DMP_OPTIONS=*options*

Specifies dump options to be used with a CEE3DMP. This environment variable is used when you specify BBODUMP=4. For an explanation of CEE3DMP and valid dump options, see *z/OS Language Environment Programming Reference*, SA22-7562.

Rule: The maximum length of the option string on this environment variable is 255. If the option string is longer than 255, you receive message BBOU0514W and the CEE3DMP dump options are set to THREAD(ALL) BLOCKS.

Example:

```
BBODUMP_CEE3DMP_OPTIONS=NOTRACEBACK NOFILES
```

BBOLANG=*LANGUAGE*

The name of the WebSphere for z/OS message catalog used. The default is ENUS.

BEAN_DELETE_SLEEP_TIME=n

The time in seconds allowed before an expired stateful session bean's state is deleted from its backing datastore (DB2). The default time is 4200 seconds (70 minutes). You can increase the time to 2147483 seconds (24.85 days). **Recommendation:** Do not set this variable less than 300 seconds (5 minutes).

Example: `BEAN_DELETE_SLEEP_TIME=1000000`

CBCONFIG=*path*

Specifies a read/write directory in the HFS into which WebSphere for z/OS writes configuration and environment files when a conversation is activated. The &CBCONFIG variable in control and server region start procedures must match this value. In this way, WebSphere for z/OS can find the appropriate environment file for a server when those start procedures are executed. The default is /WebSphere390/CB390.

Example: `CBCONFIG=/WebSphere390/CB390`

| **Rule:** The System Management group (default CBCFG1) and user ID (default CBSYMSR1) must own each directory and subdirectory in CBCONFIG. If the System Management group and user ID do not own CBCONFIG, use the chown command to make them the owner of each directory and subdirectory in CBCONFIG. Thus, if you use the default CBCONFIG, you must use the chown command to give the System Management group and user ID ownership of /WebSphere390 and /WebSphere390/CB390.

| **Example:**

```
chown -R CBSYMSR1:CBCFG1 /WebSphere390
```

| **CLASSPATH=***path1***:***[path2]***:...**

Specifies Java class files—.jar files and classes.zip files—for use by Java business objects in server regions. Specify your Java business object's .jar files when you use Java business objects. The entire CLASSPATH statement must be on one line only.

Example:

```
CLASSPATH=/usr/lpp/db2/db2710/classes/db2j2classes.zip: . . .
```

CLIENT_DCE_QOP=*value*

The level of DCE message protection used by a local z/OS or OS/390 client to apply to the current transaction flows. Normally, you would set DCE security for an z/OS or OS/390 client that accesses servers on remote systems. Note that the DCE level for a server is set through the Administration application.

When enabled on client and server, DCE authentication offers each proof of the other's legitimacy with a handshake message exchange using DCE's third-party authentication scheme. Once this exchange has taken place, messages can be assigned one of three levels of protection, which are the values of this environment variable:

NO_PROTECTION

DCE assures only that the messages and their replies are from the legitimate sender. This is the default.

INTEGRITY

DCE assures that the message is from the legitimate sender and it has not been modified in any way since the sender sent it.

CONFIDENTIALITY

DCE encrypts the message so that none but the legitimate receiver can read it.

CLIENT_HOSTNAME=

Allows an z/OS or OS/390 client to determine its host IP name when no Daemon is running on the same system. When a client program issues the CBSeriesGlobal::hostName() method, the system checks the CLIENT_HOSTNAME environment variable first and returns this value, if it is set. If the value is not set, the system returns the IP name of the Daemon running on that system, if the Daemon is running. The default value is null.

Example: `CLIENT_HOSTNAME=MYSYS.SYS.COM`

CLIENTLOGSTREAMNAME=*LOG_STREAM_NAME*

The WebSphere for z/OS error log stream to which an z/OS or OS/390 client ORB writes error information.

Example: `CLIENTLOGSTREAMNAME=MY.CLIENT.ERROR.LOG`

CLIENT_RESOLVE_IPNAME=*IP_NAME*

The Internet Protocol name that an z/OS or OS/390 client, or server region acting as a client, uses to access the bootstrap server (that is, when the client or server region invokes the resolve_initial_references method). The default is the value specified by the RESOLVE_IPNAME environment variable, which is the Internet Protocol name associated with the System Management Server (the default bootstrap server). If RESOLVE_IPNAME is not set, the value is the system on which the client or server region is running.

The CLIENT_RESOLVE_IPNAME environment variable allows you to specify a bootstrap server running on a remote system, while other clients use a local bootstrap server defined by the RESOLVE_IPNAME environment variable.

Note: The TCP/IP port number for the CLIENT_RESOLVE_IPNAME is defined by the RESOLVE_PORT environment variable.

The value of CLIENT_RESOLVE_IPNAME can be up to 255 characters.

Example: `CLIENT_RESOLVE_IPNAME=REMHOST`

CLIENT_TIMEOUT=*n*

Sets the time-out value for response from a client method call. The values are in integers and signify the time in tenths of seconds (thus, a value of 10 is 1 second). The default value is 0, which means no time-out value is set.

Example: `CLIENT_TIMEOUT=20`

com.ibm.ws.naming.ldap.containerdn=*ibm-wsnTree=t1,o=org,c=country*

The starting point of WsnName tree. Only the Naming server uses this environment variable. By default, the system expects the value to be `ibm-wsnTree=t1,o=WASNaming,c=us`. If you take the default, delete this environment variable from your environment file.

This value must match the value specified in LDAP initialization file (our sample is bboldif.cb). If you've modified the organization or country in your bboldif.cb file, use the same value on this environment variable. Note that case does not matter in LDAP, though it does matter for the environment variables. The ″o=,c=″ portion must also be specified as a suffix in bboslapd.conf.

Example:

```
suffix   "o=WASNaming,c=us"
```

Tip: The suffix statement appears as:

```
suffix         "<ws_rdn>"
```

in the sample bboslapd.conf we ship.

Example:

```
com.ibm.ws.naming.ldap.containerdn=ibm-wsnTree=t1,o=WASNaming,c=us
```

com.ibm.ws.naming.ldap.domainname=*domain name*
Uniquely identifies the host root and is the basis for partitioning the JNDI global name space. Only the Naming server uses this environment variable. By default, the system expects the value to be the domain name of the sysplex on which Naming Server is running. If you want the default, delete this environment variable from the environment file. If you want a different domain name, specify it.

Example:

```
com.ibm.ws.naming.ldap.domainname=plex1
```

com.ibm.ws.naming.ldap.masterurl=ldap://*IP_name***:***port*
The LDAP Server IP Name and port number. Only the Naming server uses this environment variable. By default, the system expects the IP name to be the same as the system on which the Naming Server runs and the port to be 1389. If your LDAP server is running on a system other then the one the Naming Server runs on or uses a port other than 1389, update this environment variable. Otherwise, delete this environment variable.

Example:

```
com.ibm.ws.naming.ldap.masterurl=ldap://wsldap:1389
```

| **CONFIGURED_SYSTEM=***system*
| Specifies the name of the system to which the server instance was
| originally configured. During prepare for cold start, cold start, and server
| activation, the run time adds this environment variable to each server
| instance's environment file automatically.

Rule: Do not manually add or change this environment variable at any time, such as:

- In the initial environment file before bootstrap
- Through the Administration application (SM EUI)
- In an existing server environment file.

DAEMON_IPNAME=*IP_NAME*
The Internet Protocol name that the Daemon Server registers with the Domain Name Service (DNS). Any CORBA client communication with WebSphere for z/OS requires this IP name.

You must define the DAEMON_IPNAME environment variable at installation time, before you start the Daemon bootstrap process. Otherwise, WebSphere for z/OS issues an error message and terminates the Daemon.

The bootstrap process sets, among other things, the Daemon IP name in the system management database. After bootstrap, WebSphere for z/OS uses the value in the system management database. It is possible that, after bootstrap, the value of the DAEMON_IPNAME environment variable could change to a value other than what is in the system management database. If this happens, an error message is issued, but the Daemon initializes with the Daemon IP name from the system management database.

To place Daemon server instances in the same host cluster, you must code the same DAEMON_IPNAME value for each server instance.

Rules:

- The value for DAEMON_IPNAME must be a fully-qualified long name.
- The first-level qualifier can be from 1 to 18 characters.
- Once chosen, the port and IP name for the Daemon should not change, since every object reference includes the port and IP name—if you change them, existing objects will no longer be accessible.

Example: `DAEMON_IPNAME=CBQ091.PDL.POK.IBM.COM`

DAEMON_PORT=*n*
The port number at which the Daemon Server listens for requests. The default is 5555. If you specify a value, you must provide the same value for the System Management Server control region.

Example: `DAEMON_PORT=5555`

DEFAULT_CLIENT_XML_PATH=*path*
Specifies the location of a set of XML files that hold default parameter

lists used by the System Management Scripting API. You must set this environment variable for clients that use the System Management Scripting API.

IBM provides a set of sample XML files that contain default parameter lists. After installation, these samples reside in `/usr/lpp/WebSphere/samples/smapi`. For information about the XML files and the parameter lists, see *WebSphere Application Server V4.0.1 for z/OS and OS/390: System Management Scripting API*, SA22-7839.

You can override the default behavior of the System Management Scripting API in two ways:

1. Specifying the parameters explicitly in the REXX script that calls the System Management Scripting API. By specifying parameters explicitly, you do not have to modify the XML samples IBM provides. You simply need to code

   ```
   DEFAULT_CLIENT_XML_PATH=/usr/lpp/WebSphere/samples/smapi
   ```

   in your client environment file.
2. Copying the XML files to another directory (the samples IBM provides are read-only), making modifications to the parameter lists, then changing the DEFAULT_CLIENT_XML_PATH to point to the new directory. Making these changes is required only if you want to override permanently the default behavior of the System Management Scripting API.

Example: `DEFAULT_CLIENT_XML_PATH=/usr/lpp/WebSphere/samples/smapi`

| **DEFAULT_UNAUTH_CLIENT_ID=***user_id*
| The default local and remote user ID that the System Management server
| associates with servers. If you allow unauthenticated client requests on a
| server, and do not explicitly specify your own local and remote user ID
| for that server, those requests run under the authority of this user ID.

| If you do not define this environment variable, the default local and
| remote user ID is CBGUEST.

| You must define this user ID to z/OS or OS/390 and give it appropriate
| security authorizations (for example, RACF permissions and LDAP
| permissions).

| This environment variable is used only by the System Management server.
| Using this environment variable in the environment file for other servers
| takes no effect. That is, you cannot use this environment variable for other
| servers to define the default local and remote ID that is used by those
| servers. Rather, you must define the default through the server properties
| panel in the Administration application. To do this

- Select the “Allow non-authenticated clients” checkbox. The Administration application supplies the value for the local and remote identity from the value on the DEFAULT_UNAUTH_CLIENT_ID variable (or, if not specified, it supplies CBGUEST).
- Type over the supplied values with your value.

The System Management server uses this environment variable during bootstrap. After bootstrap, you can modify the value only at the sysplex level through the Administration application.

Example: `DEFAULT_UNAUTH_CLIENT_ID=DUDE`

DM_GENERIC_SERVER_NAME=*SERVER_NAME*
The server name for the Daemon Server. The default is CBDAEMON. If you specify a value, you must provide the same value for the System Management Server control region.

Example: `DM_GENERIC_SERVER_NAME=CBDAEMON`

DM_SPECIFIC_SERVER_NAME=*SERVER_INSTANCE_NAME*
A server instance name of the Daemon Server. The default is DAEMON01. You must specify this environment variable for all server instances in the second and subsequent systems in a sysplex.

Example: `DM_SPECIFIC_SERVER_NAME=DAEMON01`

HOME=*path*
Specifies the home directory. This variable is set automatically from the security product user profile when the user logs in to the UNIX shell.

IBM_OMGSSL=[0 | 1]
Specifies whether only CORBA-compliant security tags will be exported by the server. The value 1 means only CORBA-compliant tags are exported. The value 0 (the default) means CORBA-compliant and non-compliant tags are exported.

Use value 1 when the server uses only SSL basic authentication for its security and clients (such as CICS or other OEM ORBs) use CORBA-compliant tags. This is only in the case when the server uses SSL basic authentication. If your server supports SSL client certificates as well, you do not have to set this variable.

Use value 0 (or take the default) when your server uses SSL basic authentication and interoperates with WebSphere clients on distributed platforms or WebSphere Application Server Enterprise Edition for OS/390 V3.02.

Example: `IBM_OMGSSL=1`

ICU_DATA=*path*
The path to binary files required by the XML Parser used by the System

Management server during bootstrap and import server processing. If you installed the WebSphere for z/OS code in the default directory, you do not need to change this path. The default path is `/usr/lpp/WebSphere/bin/`.

Example: `ICU_DATA=/usr/lpp/WebSphere/bin/`

IR_GENERIC_SERVER_NAME=*SERVER_NAME*
The server name of the Interface Repository Server. The default is CBINTFRP. You must define a workload management (WLM) application environment using this name for the Interface Repository Server server regions to work.

IR_SPECIFIC_SERVER_NAME=*SERVER_INSTANCE_NAME*
A server instance name of the Interface Repository Server. The default is INTFRP01. You must specify this environment variable for all server instances in the second and subsequent systems in a sysplex.

IRPROC=*PROC_NAME*
The start procedure used by the Daemon Server to start the Interface Repository Server. The default is BBOIR. You can supply the name of your own start procedure. If you do so, copy the information from the default start procedure to your new start procedure.

Example: `IRPROC=BBOIR`

| **IVB_DEBUG_ENABLED=1**
| Enables the z/OS or OS/390 client and the application server to load the
| object level trace run time, and to use object level trace for tracing and/or
| debugging client and server application components. The value 1 is
| required for the application server, and for both C++ or Java clients
| running on z/OS or OS/390, when debugging C++ or Java business
| objects, servlets, JSPs, or Enterprise beans.

IVB_DRIVER_PATH=*path*
The name of the directory where WebSphere for z/OS files reside after SMP/E installation. The default is `/usr/lpp/WebSphere`.

Example: `IVB_DRIVER_PATH=/usr/lpp/WebSphere`

| **IVB_TRACE_HOST=***IP_ADDRESS (or HOSTNAME)*
| Specifies the workstation IP address (or host name if you have the DNS
| server setup correctly) where the object level trace viewer runs. Use this
| when you are tracing and/or debugging your client and server
| components with the IBM Object Level Trace and Distributed Debugger
| Tools.

| **Example:** `IVB_TRACE_HOST=MYHOST.IBM.COM`

| **IVB_TRACE_PORT=***port*
| Specifies the same port as the TCP/IP port specified for the object level

trace server. Use this when you are tracing and/or debugging your client and server components with the IBM Object Level Trace and Distributed Debugger Tools. The default is 2102.

Example: `IVB_TRACE_PORT=2102`

java.naming.security.credentials=*password*
The password used by the distinguished name specified by java.naming.security.principal. The password must match the password defined for the administrator access ID (default is WASAdmin) by the LDAP initialization file during initial system customization. IBM provides the WASAdmin access ID in a sample LDIF file called bboldif.cb. The default value is `secret`.

Example: `java.naming.security.credentials=secret`

Recommendation: You should change the IBM-supplied password.

java.naming.security.principal=*distinguished_name*
Distinguished name (user ID) defined to have write access to WsnName directory. Specify this only if you want to provide read/write access to all JNDI users. The distinguished name must match the one defined for the administrator access ID (default is WASAdmin) by the LDAP LDIF file during initial system customization. IBM provides the WASAdmin access ID in a sample LDAP initialization file called bboldif.cb. The default value is `cn=WASAdmin,o=WASNaming,c=us`.

Example:

```
java.naming.security.principal=cn=WASAdmin,o=WASNaming,c=us
```

Recommendation: We suggest you keep the WASAdmin access ID.

JAVA_COMPILER=
Specifies the use of the just-in-time (JIT) compiler.

If you use the environment variable, a null value (`JAVA_COMPILER=`) turns the JIT compiler on. Any other value turns the JIT compiler off.

By default, a Java virtual machine (JVM) running on z/OS or OS/390 uses the JIT compiler, so you do not have to explicitly set this environment variable. If you are debugging Java business objects or J2EE application components, however, turn off the JIT compiler by specifying a non-null value.

Example: `JAVA_COMPILER=NONE`

JAVA_IEEE754=EMULATION
Specifies the correct executable code for the system to load for the Java virtual machine (JVM) in which Java clients on z/OS or OS/390 run. This environment variable setting is required only for Java clients that run on z/OS or OS/390.

JVM_BOOTCLASSPATH=*path1***:***[path2]*
: Enables the use of bootclasspath. This option is equivalent to the `-Xbootclasspath/p:` Java invocation option.

JVM_BOOTLIBRARYPATH=*path1***:***[path2]*
: Enables the use of bootlibrarypath. This option is equivalent to the `-Dsun.boot.library.path=` Java invocation option.

| **JVM_DEBUG=1**
| This option is equivalent to the `–verbose:class,jni` Java invocation
| option. It reroutes JVM messages to SYSOUT for debugging purposes. Set
| JVM_DEBUG=1 to invoke JVM messaging.

| **JVM_DEBUG_PORT=***port*
| Specifies a TCP/IP port that the distributed debugger uses to connect to
| the JVM.

JVM_ENABLE_CLASS_GC=1
: Enables class objects to be garbage collected. The value 1 is required for enabling class object garbage collection. This option is equivalent to the `-Xnoclassgc` Java invocation option.

JVM_ENABLE_VERBOSE_GC=1
: Sets verbose garbage collection on or off. The value 1 is required for enabling garbage collection messages. This option is equivalent to the `-verbose:gc` Java invocation option.

JVM_EXTRA_OPTIONS=*string*
: Allows you to specify one new Java environment variable that is not already predefined by IBM (those predefined variables start with `JVM_`). With `JVM_EXTRA_OPTIONS`, *string* is the new Java option or property that you want to specify.

JVM_HEAPSIZE=*n*
: Sets the maximum size (in megabytes) of the JVM heap. The default is 256 MB. This option is equivalent to the `-Xmx=xxxM` Java invocation option.

  Example: `JVM_HEAPSIZE=256 # specifies a 256 MB heap`

JVM_LOCALREFS=
: Should only be used under the direction of IBM support. The default is 128.

JVM_LOGFILE=*filename*
: Specifies the HFS file in which messages from the JVM will be logged.

  Recommendation: Use this variable only in a single-server environment. If you use `JVM_LOGFILE` in a multiple-server environment, all the servers write to the same file, so you might have difficulty using the file for diagnostic purposes. In a multiple-server environment, use `JVM_DEBUG=1` to direct JVM messages to the SYSOUT for a specific server.

JVM_MINHEAPSIZE=*n*
: Sets the mimimum size (in megabytes) of the JVM heap. The default is 256 MB. This option is equivalent to the `-Xms=xxxM` Java invocation option. For optimal performance, specify the same value for `JVM_HEAPSIZE` and `JVM_MINHEAPSIZE`.

LDAPBINDPW=*password*
: The password the Naming Server uses to bind to the LDAP server. Used in conjunction with LDAPNAME.

LDAPCONF=*filename*
: The LDAP configuration file used by WebSphere for z/OS. If you designate a file in the HFS, do not use quotes. If you designate an MVS data set, enclose the data set in single quotes.

  Example: `LDAPCONF='bbo.s21slapd.conf'`

LDAPHOSTNAME=*name:port*
: The host name of the LDAP server that the Interface Repository Server uses as its data store.

LDAPIRBINDPW=*password*
: The password the Interface Repository Server uses to bind to the LDAP server. Used in conjunction with LDAPIRNAME.

LDAPIRCONF=*filename*
: The LDAP configuration file used by the LDAP server that the Interface Repository Server uses as its data store. If you designate a file in the HFS, do not use quotes. If you designate an MVS data set, enclose the data set in single quotes.

LDAPIRHOSTNAME=*name:port*
: The host name of the LDAP server that the Interface Repository Server uses as its data store.

LDAPIRNAME
: The LDAP entry name that the Interface Repository Server uses to authenticate itself to the LDAP server that it uses as its data store.

LDAPIRROOT=*root*
: The LDAP entry name at which the Interface Repository Server anchors its data.

  Example: `LDAPIRROOT=o=BOSS,c=U`

LDAPNAME
: The LDAP entry name that the Naming Server uses to authenticate itself to the LDAP server that it uses as its data store.

LDAPROOT=*root*
: The LDAP entry name at which the Naming Server anchors its data.

Example: `LDAPROOT=o=BOSS,c=US`

LIBPATH=*path1***:***[path2]***:...**

Specifies the DLL search paths for Java in the hierarchical file system (HFS). Specify system, WebSphere for z/OS, and Java DLLs.

Example:

```
LIBPATH=/db2_install_path/lib:/usr/lpp/java/J1.3/bin:/usr/lpp/java/J1.3/bin/classic:/usr/lpp/WebSphere/lib
```

where *db2_install_path* is the HFS where you installed DB2.

LOGSTREAMNAME=*LOG_STREAM_NAME*

The WebSphere for z/OS error log stream name the Daemon and System Management servers use during bootstrap. If not specified in the environment file for the Daemon and System Management servers during bootstrap, the system uses the following algorithm to form an error log stream name. WebSphere for z/OS:

1. Takes the first qualifier in the Daemon Server's IP name.
2. If the first qualifier is more than 8 characters, divides the qualifier into 8-character strings and separates them with periods.
3. Adds a high-level qualifier "BBO".

For example, if the Daemon IP name is MYDAEMONSERVER.IBM.COM, the algorithm would produce an error log stream name BBO.MYDAEMON.SERVER.

After bootstrap, you can create or change an error log stream name for the entire sysplex, a server, or a server instance through the Administration application. A server error log stream setting overrides the general WebSphere for z/OS setting, and a server instance setting overrides a server setting. Thus, you can set up general error logging, but direct error logging for servers or server instances to specific log streams.

During processing, if the specified log stream is not found or not accessible, a message is issued and errors are written to the server's joblog.

Example: `LOGSTREAMNAME=MY.CB.ERROR.LOG`

Tip: Do not put the log stream name in quotes. Log stream names are not data set names.

MIN_SRS=*nn*

The number of server regions to be kept running once those server regions have initialized. That is, workload management will not direct the server region to shut down even though it becomes inactive. Use this environment variable when the response time for the workload requires that several server regions are always ready to process work.

The default for J2EE servers is 1. For MOFW servers, the default is 0. The maximum value is 20. If you specify more than 20, the variable is set to 20.

WebSphere for z/OS garbage collection may cause a server region to refresh, but the minimum number of server regions will not fall below the value specified on this environment variable.

Example: `MIN_SRS=2`

NM_GENERIC_SERVER_NAME=*SERVER_NAME*
The server name of the Naming Server. The default is CBNAMING. You must define a workload management (WLM) application environment using this name for the Naming Server server regions to work.

Example: `NM_GENERIC_SERVER_NAME=CBNAMING`

NM_SPECIFIC_SERVER_NAME=*SERVER_INSTANCE_NAME*
The server instance name of the Naming Server. The default is NAMING01. You must specify this environment variable for all server instances in the second and subsequent systems in a sysplex.

Example: `NM_SPECIFIC_SERVER_NAME=NAMING01`

NMPROC=*PROC_NAME*
The start procedure used by the Daemon Server to start the Naming Server. The default is BBONM. You can supply the name of your own start procedure. If you do so, copy the information from the default start procedure to your new start procedure.

Example: `NMPROC=BBONM`

OTS_DEFAULT_TIMEOUT=*n*
The amount of time (in seconds) given by default to an application transaction to complete. This amount of time is given to the application transaction if it does not set its own time-out value through the `current –> set_timeout` method.

The default is 30 seconds and the maximum value is 2147483 seconds (24.85 days). You should not use a null or 0 value.

Note: When a conversation is activated, the system performs special processing for the System Management server instances **only**.

- If the OTS_DEFAULT_TIMEOUT variable is not set, it is added.
- If the value for OTS_DEFAULT_TIMEOUT is less than 3600 (seconds), it is set to 3600.

This special processing is performed for the System Management server instances because the server instances sometimes perform long-running transactions. Other server instances do not require such lengthy transaction defaults.

Example: `OTS_DEFAULT_TIMEOUT=30`

OTS_MAXIMUM_TIMEOUT=*n*
The maximum allowable amount of time (in seconds) given to an application transaction to complete. If an application assigns a greater amount of time, the system limits the time to the OTS_MAXIMUM_TIMEOUT value.

The default is 60 seconds and the maximum value is 2147483 seconds (24.85 days). You should not use a null or 0 value.

Note: When a conversation is activated, the system performs special processing for the System Management server instances **only**.
- If the OTS_MAXIMUM_TIMEOUT variable is not set, it is added.
- If the value for OTS_MAXIMUM_TIMEOUT is less than 3600 (seconds), it is set to 3600.

This special processing is performed for the System Management server instances because the server instances sometimes perform long-running transactions. Other server instances do not require such lengthy transaction defaults.

Example: `OTS_MAXIMUM_TIMEOUT=60`

PATH=*path*
Specifies the path.

RAS_MINORCODEDEFAULT=*value*
Determines the default behavior for gathering documentation about system exception minor codes. Use only under the guidance of IBM Service.

CEEDUMP
Captures callback and offsets.

Tip: It takes time for the system to take CEEDUMPs and this may cause transaction timeouts. For instance, your OTS_DEFAULT_TIMEOUT may be set to 30 seconds, but, since taking a CEEDUMP can take longer than 30 seconds, your application transaction may time out. To prevent this from happening, either:
- Increase the transaction timeout value.

  or

- Code RAS_MINORCODEDEFAULT=NODIAGNOSTICDATA. Be sure TRACEMINORCODE is **not** in the environment file.

TRACEBACK
Captures Language Environment and z/OS UNIX traceback data.

SVCDUMP
Captures an MVS dump (but will not produce a dump in the client).

NODIAGNOSTICDATA
The default. This setting will not cause the gathering of a CEEDUMP, TRACEBACK, or SVCDUMP.

Note: Sometimes results depend on the setting of another environment variable, TRACEMINORCODE. If you code TRACEMINORCODE=(null value) and RAS_MINORCODEFAULT=TRACEBACK you get a traceback. But, if you code RAS_MINORCODEDEFAULT=NODIAGNOSTICDATA and TRACEMINORCODE=ALL, you also get a traceback. So, specifying RAS_MINORCODEDEFAULT=NODIAGNOSTICDATA does not cancel TRACEBACK; it simply does not cause a TRACEBACK to be gathered.

REM_DCEPASSWORD=*password*
The password of the remote DCE principal passed in the security context when an z/OS or OS/390 client makes a request to a system outside the sysplex and SSL Type 1 authentication is being used. The password must conform to DCE requirements for passwords.

Example: `REM_DCEPASSWORD=mydcePW`

REM_DCEPRINCIPAL=*principal*
The principal passed in the security context when a client makes a request to a system outside the sysplex and SSL Type 1 authentication is being used. This principal must be defined on the target server. The value must conform to DCE requirements for principals.

Example: `REM_DCEPRINCIPAL=myDCEprin`

REM_PASSWORD=*password*
The password used in the security context when a client makes a request to a remote z/OS or OS/390 system and user ID/password security or SSL security is being used.

Example: `REM_PASSWORD=MYPASSW`

REM_USERID=*USER_ID*
The user ID used in the security context when a client makes a request to a remote z/OS or OS/390 system and user ID/password security or SSL security is being used.

Example: `REM_USERID=MCOX`

RESOLVE_IPNAME=*IP_NAME*
The Internet Protocol name that the System Management Server registers with the Domain Name Service (DNS). Any CORBA client communication with WebSphere for z/OS requires this IP Name. If not set, the Resolve IP Name is the system on which the program is running.

Rule: The value for RESOLVE_IPNAME should be a fully-qualified name, but it cannot exceed 255 characters.

Example: `RESOLVE_IPNAME=CBQ091.COMPANY.NY.COM`

RESOLVE_PORT=*n*
The port number at which the System Management Server listens for requests. The default is 900. This is a well-known port for Object Request Brokers, so IBM advises that you do not change this variable. If you already have an application that uses this port, consider using TCP/IP bind-specific support and the SRVIPADDR environment variable.

Example: `RESOLVE_PORT=900`

SM_DEFAULT_ADMIN=*USER_ID*
The user ID for the administrator who uses the Administration and Operations applications. This environment variable is used by the System Management bootstrap during installation—setting this environment variable after the System Management bootstrap runs has no effect. If you do not define this environment variable, the default user ID is CBADMIN. You must define this user ID to z/OS or OS/390 and give it appropriate security authorizations (for example, RACF permissions and LDAP permissions).

Note: After the System Management bootstrap runs, you can define additional administrator user IDs only through the Administration application. Those user IDs do not replace the user ID defined by SM_DEFAULT_ADMIN.

Example: `SM_DEFAULT_ADMIN=DUDE`

SM_GENERIC_SERVER_NAME=*SERVER_NAME*
The server name of the Systems Management Server. The default is CBSYSMGT. You must define a workload management (WLM) application environment using this name for the Systems Management Server server regions to work.

Example: `SM_GENERIC_SERVER_NAME=CBSYSMGT`

SM_SPECIFIC_SERVER_NAME=*SERVER_INSTANCE_NAME*
The server instance name of the Systems Management Server. The default

is SYSMGT01. You must specify this environment variable for all server instances in the second and subsequent systems in a sysplex.

Example: `SM_SPECIFIC_SERVER_NAME=SYSMGT01`

SMPROC=*PROC_NAME*
The start procedure used by the Daemon Server to start the Systems Management Server. The default is BBOSMS. You can supply the name of your own start procedure. If you do so, copy the information from the default start procedure to your new start procedure.

Example: `SMPROC=BBOSMS`

SOMOOSQL=*value*
Improves performance for client applications that use object-oriented SQL queries on string attributes. By using SOMOOSQL=1, string comparisons are pushed down to the database.

The default value is null (SOMOOSQL=).

Rule: You can use SOMOOSQL=1 only when the database and server region address spaces have been declared to run in the same locale.

SRVIPADDR=*IP_ADDRESS*
The IP address in dotted decimal format that WebSphere for z/OS servers use to listen for client connection requests.

This IP address is used by the server to bind to TCP/IP. Normally, the server will listen on all IP addresses configured to the local TCP/IP stack. However if you want to fence the work or allow multiple heterogeneous servers to listen on the same port, you can use SRVIPADDR. The specified IP address becomes the only IP address over which WebSphere for z/OS receives inbound requests. Normally, you also have to map the Daemon IP name, resolve IP name, or host name of the server that you are on to this particular SRVIPADDR.

SSL_KEYRING=*keyring*
The name of the z/OS or OS/390 client's key ring used in SSL processing. This key ring must reside in RACF.

Example: `SSL_KEYRING=IVPRING`

SYS_DB2_SUB_SYSTEM_NAME=*NAME*
The DB2 name used by Daemon and System Management servers to connect to the database. Use either the DB2 subsystem name or group attachment name. The default is DB2. If the default is not correct for your installation, change the environment variable to match the correct value.

Example: `SYS_DB2_SUB_SYSTEM_NAME=DB21`

TRACEALL=*n*
Specifies the default tracing level for WebSphere for z/OS. Valid values and their meanings are:

0	No tracing
1	Exception tracing, the default
2	Basic and exception tracing
3	Detailed tracing, including basic and exception tracing

Use this variable in conjunction with the TRACEBASIC and TRACEDETAIL environment variables to set tracing levels for WebSphere for z/OS subcomponents. Do not change this variable unless directed by IBM service personnel.

Example: `TRACEALL=1`

TRACEBASIC=*n* | (*n*,...)
Specifies tracing overrides for particular WebSphere for z/OS subcomponents. Subcomponents, specified by numbers, receive basic and exception traces. If you specify more than one subcomponent, use parentheses and separate the numbers with commas. Contact IBM service for the subcomponent numbers and their meanings. Other parts of WebSphere for z/OS receive tracing as specified on the TRACEALL environment variable. Do not change TRACEBASIC unless directed by IBM service personnel.

Example: `TRACEBASIC=3`

TRACEBUFFCOUNT=*n*
Specifies the number of trace buffers to allocate. Valid values are 4 through 8. The default is 4.

TRACEBUFFLOC=SYSPRINT | BUFFER
Specifies where you want trace records to go: either to sysprint (SYSPRINT) or to a memory buffer (BUFFER), then to a CTRACE data set. The default is to direct trace records to sysprint for the client and to a buffer for all other WebSphere for z/OS processes. For servers, you may specify one or both values, separated by a space. For clients, you may specify TRACEBUFFLOC=SYSPRINT only.

Example: `TRACEBUFFLOC=SYSPRINT BUFFER`

TRACEBUFFSIZE=*n*
Specifies the size of a single trace buffer in bytes. You can use the letters "K" (for kilobytes) or "M" (for megabytes). Valid values are 128K through 4M. The default is 1M.

TRACEDETAIL=*n* **|** **(***n***,...)**

Specifies tracing overrides for particular WebSphere for z/OS subcomponents. Subcomponents, specified by numbers, receive detailed traces. If you specify more than one subcomponent, use parentheses and separate the numbers with commas. Contact IBM service for the subcomponent numbers and their meanings. Other parts of WebSphere for z/OS receive tracing as specified on the TRACEALL environment variable. Do not change TRACEDETAIL unless directed by IBM service personnel.

Examples:

```
TRACEDETAIL=3
```

```
TRACEDETAIL=(3,4)
```

TRACEMINORCODE=*value*

Enables traceback of system exception minor codes. Use only when instructed by IBM Service. Values are:

ALL|all
Enables traceback for all system exception minor codes.

minor_code
Enables traceback for a specific minor code. Specify the code in hex, such as X'C9C21234'.

(null value)
The default. This setting will not cause gathering of a traceback.

Note: Sometimes results depend on the setting of another environment variable, RAS_MINORCODEDEFAULT. If you code TRACEMINORCODE=ALL and RAS_MINORCODEDEFAULT=NODIAGNOSTICDATA, you get a traceback. But, if you code TRACEMINORCODE=(null value) and RAS_MINORCODEFAULT=TRACEBACK you also get a traceback. So, specifying TRACEMINORCODE=(null value) does not cancel TRACEBACK; it simply does not cause a TRACEBACK to be gathered.

TRACEPARM=*SUFFIX* **|** *MEMBER_NAME*

Identifies the CTRACE PARMLIB member. The value can be either a two-character suffix, which is added to the string CTIBBO to form the name of the PARMLIB member, or the fully-specified name of the PARMLIB member. For example, you could use the suffix "01", which the system resolves to "CTIBBO01". A fully-specified name must conform to the naming requirements for a CTRACE PARMLIB member. For details, see *z/OS MVS Diagnosis: Tools and Service Aids*, GA22-7589.

The default value is 00.

If this environment variable is specified and the PARMLIB member is not found, the default PARMLIB member, CTIBBO00, is used. If neither the specified nor the default PARMLIB member is found, tracing is defined to CTRACE, but there is no connection to a CTRACE external writer. For details on the PARMLIB member and the use of the CTRACE external writer, see *WebSphere Application Server V4.0.1 for z/OS and OS/390: Messages and Diagnosis*, GA22-7837.

Note that the Daemon Server is the only server that recognizes this environment variable.

Example: `TRACEPARM=01`

Appendix B. An IMS application as an WebSphere for z/OS client

WebSphere for z/OS allows IMS transaction-processing applications to act as client applications. In other words, an IMS application, running in an IMS message processing region, may create or find, use, and delete CORBA business objects that run in a WebSphere for z/OS MOFW server. Figure 17 illustrates the environment and processing through which an IMS application becomes a WebSphere for z/OS client.

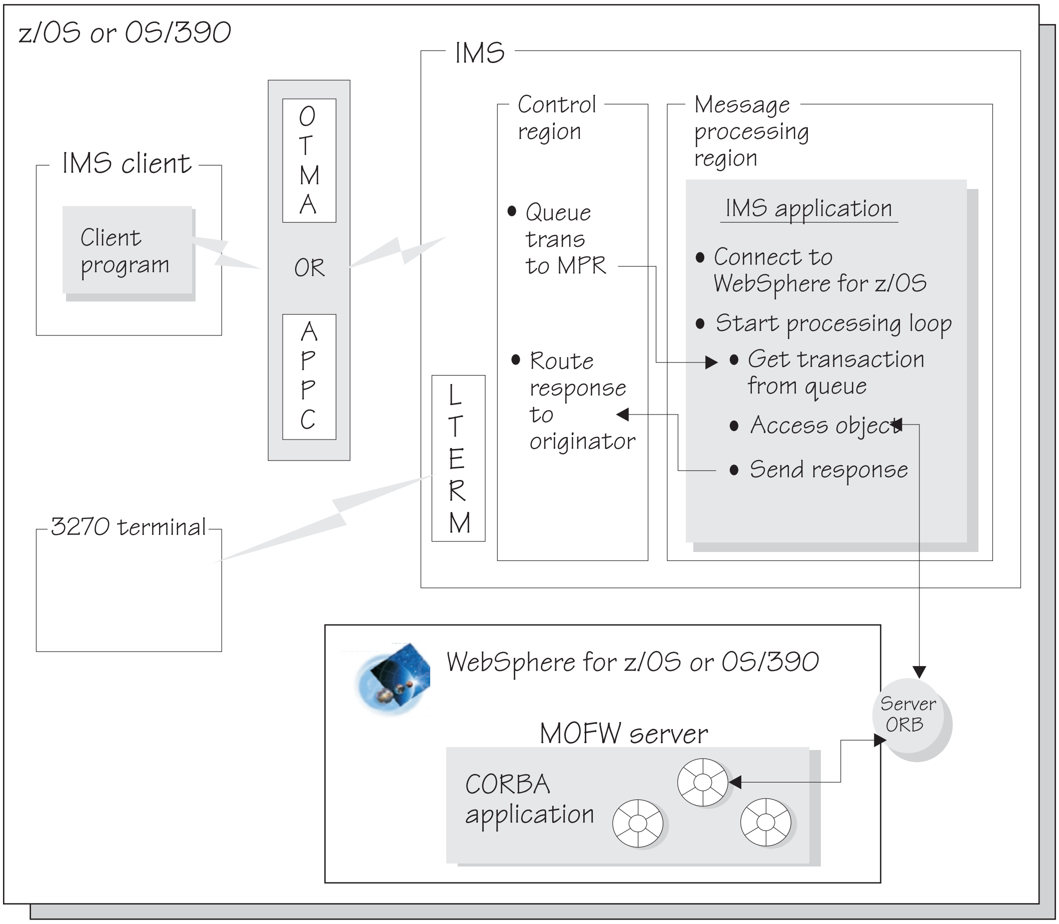

Figure 17. An IMS application running as a client of a WebSphere for z/OS MOFW server

In this illustration, the following processing takes place:

- The IMS client submits a transaction to IMS, which can be configured to receive requests and send responses through one of the following:
  - Open Transaction Manager Access (OTMA), or
  - The advanced program-to-program communication component of MVS (APPC/MVS)

  Alternatively, a transaction request can come from a 3270 terminal that is an IMS-defined logical terminal (LTERM).
- The IMS control region queues the transaction request for the IMS application that runs as a WebSphere for z/OS client.
- The IMS application follows the WebSphere Application Server for z/OS and OS/390 Component Broker client programming model to connect to a WebSphere for z/OS MOFW server, and drives methods against objects in that server to satisfy the original IMS client transaction request. The IMS application then sends a response to the IMS client.

The following table shows the subtasks and associated procedures for creating and running an IMS application as a WebSphere for z/OS client:

Subtask	Associated procedure (See . . .)
Developing and assembling an IMS application to run as a WebSphere for z/OS client	• "Background on designing the IMS application" • "Steps for developing and compiling the IMS application" on page 164
Preparing the run-time environment for the IMS application	• "Background on security for the IMS application" on page 163 • "Steps for setting up the run-time environment for the IMS application" on page 166

Background on designing the IMS application

For an IMS transaction-processing application as a client of WebSphere for z/OS, you need to consider two programming characteristics that make your application different from other IMS applications:

- Use of the Component Broker client programming model and instructions, and
- The transactional scope of IMS processing.

For your IMS application to become a client of a WebSphere for z/OS MOFW server, you must code it according to the Component Broker client programming model, which prescribes the following processing sequence:

1. Using the `CBSeriesGlobal::Initialize` method to establish a connection with WebSphere for z/OS
2. Using the `CBSeriesGlobal::orb` method to obtain a pointer to WebSphere for z/OS
3. Using the `resolve_initial_references` method to obtain a reference to the WebSphere for z/OS naming server
4. Locating a factory finder to find the homes for the WebSphere for z/OS objects that the client wants to use
5. Finding or creating, using, and deleting objects in WebSphere for z/OS MOFW servers, as part of handling transactions from IMS clients.

Recommendation: To achieve the best performance as a client of WebSphere for z/OS, design your IMS application to initialize with WebSphere for z/OS only once, when the application is first loaded. Then the application can continuously process transactions queued to it without reloading and reinitializing. In other words, design your application as a wait for input (WFI) application, which does the following processing:
1. Issue the `CBSeriesGlobal::Initialize` method
2. Issue the `CBSeriesGlobal::orb` method
3. Issue the `resolve_initial_references` method
4. Locate the object factories and homes needed for transaction processing
5. Begin a loop, in which the IMS application does the following processing:
   a. Issues a Get Unique request to obtain the next transaction
   b. Processes the transaction by issuing methods against objects in the WebSphere for z/OS MOFW servers
   c. Responding to the originator of the IMS transaction.

This design yields the best performance for an IMS application as a client of WebSphere for z/OS, but the level of security you select for the application might override its processing such that reloading is required for processing each transaction request. "Background on security for the IMS application" on page 163 describes security options and their effect on application processing.

The transactional scope, as well as the Component Broker client programming model, also makes your IMS application different from other IMS applications. When your IMS application requests a transaction, for example, the system sets a specific transaction context under which your application runs while it processes that single transaction request. When your IMS application issues a method against a CORBA object in an WebSphere for z/OS MOFW server, however, that server might not use the transaction context under which your IMS application is running. The WebSphere for z/OS MOFW server decides what transactional context to use, depending on the transaction policy in effect for the container in which the object resides. For some container transaction policies, the WebSphere for z/OS MOFW server initiates a separate transaction context to process the method request against an object, and commits all changes before returning to the caller (that is, the IMS application). Because of this behavior, you need to consider the following recovery situations:

- If a termination error occurs under IMS during the processing of an IMS transaction, and the termination error occurs before any update-object request is sent to WebSphere for z/OS, recovery for the IMS transaction is the same as for IMS applications that are not WebSphere for z/OS clients.
- If a termination error occurs under WebSphere for z/OS during the processing of an object request for an IMS transaction, and the object container policy propagated the IMS transactional context, WebSphere for z/OS rolls back any processing completed for the object request. In this case, the IMS application may, if it is designed to do so, catch the error notification from WebSphere for z/OS, and choose to either continue processing or terminate the IMS transaction.
- If a termination error occurs under IMS during the processing of an IMS transaction, *after* WebSphere for z/OS has successfully processed and committed object updates on behalf of the IMS application, your application cannot roll back the updates that WebSphere for z/OS completed. Your IMS application may, however, roll back its own processing. In this case, you should design your application to properly communicate not only the termination error, but also the fact that some updates have been completed, and that they cannot be rolled back.

If necessary, see one or more of the following references for additional information:

For more information about this topic:	See:
Details about the Component Broker client programming model and the methods clients may use	• *Component Broker Programming Guide* • *Component Broker Programming Reference*
Details about defining an IMS application as a wait-for-input (WFI) application, using the TRANSACT macro	*IMS/ESA Installation Volume 2: System Definition and Tailoring*, GC26-8737
Details about WebSphere for z/OS MOFW server container transaction policies and their implications for application processing	"Background on the OS/390 Component Broker transactional environment" on page 17
Specific rules and restrictions for coding an IMS application as a client of WebSphere for z/OS	"Steps for developing and compiling the IMS application" on page 164

Background on security for the IMS application

Before you install your IMS application, you need to determine which level of security is required for it, based on your knowledge of the application itself and of your installation's security requirements. You can set security controls at the level of either the user ID of the message processing region (MPR), or the user ID of the originator of the IMS transaction. Setting security at the level of the MPR user ID means that all transactions that your IMS application processes will run under the user ID of the MPR in which your application runs. This security setting is the default security setting for transactions received from IMS-defined logical terminals (LTERMs). In this case, the system checks to see whether the transaction originator is authorized to issue the transaction, but then uses the MPR user ID to check for authorization to:
- Update databases,
- Use UNIX System Services, and
- Use WebSphere for z/OS MOFW servers and the CORBA objects in those servers.

With security at the level of MPR user ID, all authorization checks occur when the IMS application is initially loaded. The application may then process transactions without any further authorization checks, because the user ID under which transaction processing occurs will not change.

Recommendation: If possible, use this level of security, along with the recommended design in "Background on designing the IMS application" on page 160, to achieve the best possible performance for an IMS application as a WebSphere for z/OS client. This security setting, with a wait-for-input (WFI) design, allows the application to continuously loop to process additional transactions without repeating the initialization process.

Tip: For transactions received from OTMA or APPC, one possible way to set the level of security to MPR user ID is to use the IMS `/SECURE OTMA PROFILE` or `/SECURE APPC` command, respectively.

Setting security at the level of the user ID of the transaction originator means that all transactions that your IMS application processes will run under the user ID of the transaction originator. This security setting is the default security setting for transactions received from IMS clients through OTMA or APPC. In this case, the system uses the transaction originator's user ID not only to see whether the originator is authorized to issue the transaction, but also uses the originator's ID for all other authorization checks. The system also reloads the application before the IMS application may process a transaction. This level of security is known as "full security."

With full security, the IMS application must be reloaded for each transaction that it processes. This reloading is required to establish the proper security environment, based on the user ID that issued the IMS transaction, for authorization to use UNIX System Services. This reinitialization occurs even if the IMS application is designed as recommended in "Background on designing the IMS application" on page 160.

See one or more of the following references for additional information:

For more information about this topic:	See:
Details for establishing IMS security, especially for changing the default security for transactions received from LTERMs, OTMA, or APPC	• *IMS/ESA Administration Guide: System*, SC26-8730 • *IMS/ESA Open Transaction Manager Access Guide*, SC26-8743
Specific instructions for setting up security for an IMS application as a client of WebSphere for z/OS	"Steps for setting up the run-time environment for the IMS application" on page 166

Steps for developing and compiling the IMS application

To design and code an IMS application that is a WebSphere for z/OS client, you need to complete several steps in addition to those you usually complete to design and code applications that run in IMS message processing regions.

Before you begin: Make sure you understand the information presented in "Background on designing the IMS application" on page 160, which recommends a design for optimal performance.

Perform the following steps to create and compile an IMS application as a WebSphere for z/OS client:

1. Write new or edit existing source code for an IMS application.

   Rules:

   - Use the C++ programming language for this application.
   - Use the Component Broker programming model to design and code your application as a client of aWebSphere for z/OS MOFW server. Briefly, to follow the Component Broker programming client model, the IMS application must:
     – Use the `CBSeriesGlobal::Initialize` method to establish a connection with WebSphere for z/OS
     – Use the `CBSeriesGlobal::orb` method to obtain a pointer to WebSphere for z/OS
     – Use the `resolve_initial_references` method to obtain a reference to the WebSphere for z/OS naming server
     – Locate a factory finder to find the homes for the WebSphere for z/OS CORBA objects that the client wants to use

– Find or create, use, and delete objects in WebSphere for z/OS MOFW servers, as part of handling transactions from IMS clients.

For further details about the Component Broker client programming model and the methods clients may use, see *WebSphere Application Server for OS/390 Component Broker: Programming Guide* and *WebSphere Application Server for OS/390 Component Broker: Programming Reference*.

- Use one of the following z/OS or OS/390 Language Environment interfaces to pass requests to IMS. Depending on the interface you use, include the appropriate header file:

For this interface:	**Include this header file:**
CTDLI	ims.h
CEETDLI	leawi.h

For additional information about these interfaces, see one or more of the following:

– *z/OS Language Environment Programming Guide*, SA22-7561 and *IMS/ESA Application Programming: Transaction Manager*, SC26-8729 for usage details, and

– *z/OS Language Environment Programming Reference*, SA22-7562 for interface syntax.

- Use a `#pragma runopts(...)` statement in the source code, to specify run-time options for the IMS application. The `runopts` statement must include the following options:

– `POSIX(ON)`, to indicate that the UNIX Systems Services environment is required

– `ENVAR("_CEE_ENVFILE=DD:BBOENV")`, to link to a file containing WebSphere for z/OS-specific environment variable settings. For information about creating this file, see "Steps for setting up the run-time environment for the IMS application" on page 166.

If necessary, see *z/OS Language Environment Programming Guide*, SA22-7561 for general information about run-time options.

Restriction: Do not issue any `begin` or `commit` requests to define a transaction that includes methods driven against CORBA objects in a WebSphere for z/OS MOFW server. WebSphere for z/OS does not support this capability for client applications running in the IMS environment.

---

2. Compile the IMS application source code, using the following compilation parameters: `TARGET(IMS)` and `PLIST(OS)`

If necessary, see *z/OS Language Environment Programming Guide*, SA22-7561 for further details about running applications under IMS.

Once you have completed these steps to code and compile the IMS application, you are ready to complete WebSphere for z/OS-specific installation set-up tasks that are required for IMS transactions that run as WebSphere for z/OS clients. These tasks are listed in "Steps for setting up the run-time environment for the IMS application".

Steps for setting up the run-time environment for the IMS application

To set up the environment for an IMS application that is a WebSphere for z/OS client, you need to complete several steps in addition to those you usually complete to install and run applications in IMS message processing regions.

Before you begin: Make sure you understand the information presented in "Background on security for the IMS application" on page 163.

Perform the following steps to set up the run-time environment for an IMS application as a WebSphere for z/OS client:

1. Create an HFS file to contain WebSphere for z/OS-specific environment variable settings for the IMS application. Use the table and environment variable descriptions in "Appendix A. Environment files" on page 125 to determine which run-time environment variables you need to set for this client.

   Note: You are setting variables for the **client** run-time environment, not for the WebSphere for z/OS MOFW server environment.

2. Write a new or edit an existing JCL procedure to start the IMS message processing region in which the IMS application will run:
   - Use the IMS sample job named IMSMSG to start message processing regions. This sample job invokes the IMS DFSMPR procedure to start a new region. Make sure you specify a unique name for each MPR.

     Depending on your installation's practice, you will find the IMSMSG job in either the IMS.JOBS or IMS.PROCLIB data set.
   - Add a BBOENV DD statement to the JCL procedure, to identify the HFS file you created to define WebSphere for z/OS-specific environment variable settings for the client run-time environment. Use the same syntax for this DD statement as shown in "How run-time server start procedures point to their environment files" on page 126.

If you need more information about the IMS sample job named IMSMSG or DFSMPR procedure, see *IMS/ESA Installation Volume 2: System Definition and Tailoring*, GC26-8737.

---

3. Determine which level of security is required for the IMS application, based on the information in "Background on security for the IMS application" on page 163 and your knowledge of your installation's security requirements. You can set security controls at the level of either the user ID of the message processing region (MPR), or the user ID of the originator of the IMS transaction.

   Then complete the appropriate security tasks:

   - Define an OMVS segment for either the user ID of the MPR, or the user ID of the originator of the IMS transaction, depending on the security level you have selected. The user ID needs this authorization because UNIX System Services are required to extract information to process IMS service requests.

     If necessary, see *z/OS UNIX System Services Planning*, GA22-7800 for further details about defining UNIX users to RACF, or verifying user OMVS segments.
   - Use the CBIND class in RACF to establish authorization to access WebSphere for z/OS MOFW servers, and to pass requests to those servers (that is, use managed objects in those servers). WebSphere for z/OS uses two profiles in the CBIND class:
     - `CB.BIND.`**`server_name`** controls client access to WebSphere for z/OS servers
     - `CB.`**`server_name`** controls client use of managed objects in those WebSphere for z/OS servers.

     Depending on the security level you have selected, establish authorization as follows:

For this level of security:	**Use the following CBIND profiles:**

User ID of MPR	Authorize the user ID of the MPR to have read access to each server that the IMS application needs to access, and to the managed objects in each of those servers. For example:
	`PERMIT CB.server1       CLASS(CBIND) ID(mpr_user_id) ACCESS(READ)`
	`PERMIT CB.BIND.server1  CLASS(CBIND) ID(mpr_user_id) ACCESS(READ)`

For this level of security:	Use the following CBIND profiles:
User ID of IMS transaction originator	Authorize the user ID of the MPR and the user IDs of transaction originators as follows: – The user ID of the MPR must have control access to each server that the IMS application needs to access (`CB.BIND.`**`server_name`**). – The user IDs of all originators must have read access to managed objects in each server (`CB.`**`server_name`**).

If necessary, see the information about setting up system security in *WebSphere Application Server V4.0.1 for z/OS and OS/390: Installation and Customization*, GA22-7834 for further details.

- Specify the appropriate parameters on the TRANSACT macro statement for the IMS application:

For this level of security:	Specify the following parameters:
User ID of MPR	– WFI parameter The WFI parameter specifies that the IMS application will wait for input if the IMS message queue is empty when the application requests another transaction. **Tip:** As an alternative, you may specify PWFI=Y as an input parameter on the DFSMPR procedure for the MPR. – PROCLIM=65535 This PROCLIM setting allows the IMS application to process the maximum number of transactions before the IMS application is re-loaded. This parameter combination allows the IMS application to continuously process transactions without re-initializing, which yields the best possible performance for an IMS application running as a WebSphere for z/OS client.
User ID of IMS transaction originator	PROCLIM=0 This PROCLIM setting indicates that the IMS application must be reloaded after it processes each transaction. This setting is necessary for properly reestablishing the UNIX Systems Services authorization environment, as the user ID changes from transaction to transaction.

If necessary, see the information about defining IMS transactions in *IMS/ESA Installation Volume 2: System Definition and Tailoring*, GC26-8737 for further details about the TRANSACT macro.

Once you have completed these steps to set up the IMS application as a client of WebSphere for z/OS, you are ready to complete any further IMS-specific installation tasks that are usually required for IMS transactions. If necessary, see *IMS/ESA Installation Volume 2: System Definition and Tailoring*, GC26-8737 for further details.

Appendix C. The Interface Definition Language (IDL) compiler

Use the IDL compiler to create usage and implementation bindings for interfaces described in one or more files containing CORBA 2.0-compliant IDL statements. To use the compiler, enter the idlc command with valid option values and the names of the IDL files to be compiled.

idlc command syntax

idlc *[-options] <file_name>*

[options]
One or more option values that determine how the IDL compiler processes your request. Note that a dash (-) precedes the first option value. Additional syntax rules apply when you specify more than one option, or specify an option with an argument. See "idlc command option syntax and values" on page 172 for description of all option values, associated arguments, and the remaining syntax rules.

<file_name>
The name of one or more IDL files to be compiled. The IDL compiler processes files with the ".idl" extension. You may use a wildcard character (*) only once in the nonpath portion of the file name. If you do not precede the file name with the path, the IDL compiler looks only in the current directory for the file you specified.

The following examples are acceptable methods of specifying IDL files in directory "E:\idl\src" (the current directory, for these examples):

```
E:\idl\src\xyz.idl
E:\idl\src\xyz
E:\idl\src\*.idl
E:\idl\src\x*.idl
xyz.idl
xyz
x*
```

If you need to use the same options, or a similar set of options, each time you run the compiler, you can use the IDLC_OPTIONS environment variable instead of retyping a lengthy command. You can add any of the idlc command options to the IDLC_OPTIONS environment variable; then when you next enter an idlc command, the IDL compiler processes these options before any options that you specify on the idlc command itself. For example, suppose you added the following options to the IDLC_OPTIONS environment variable:

```
-m cpponly -mdllname=mydll
```

If you then type the command `idlc -ehh idlfile`, the results of your request are the same as if you had typed the following:

```
idlc -m cpponly -mdllname=mydll -ehh idlfile
```

You can use the same technique to specify emitters as well. Instead of specifying an emit list on the idlc command itself, you can add a list of emitter names to the IDLC_EMIT environment variable. As with options in the IDLC_OPTIONS environment variable, the IDL compiler processes the list in the IDLC_EMIT environment variable as well as options that you specify directly on the idlc command.

idlc command option syntax and values

If you specify more than one option value or an option with an argument, the following syntax rules apply:

- You may separate each value with a blank, or run the option values together. If you use blanks, begin each option value with a dash:

  ```
  idlc [-p -V -v]
  ```

  If you run the option values together, omit all but the first dash:

  ```
  idlc [-pVv]
  ```

- You may specify arguments for some options. If you do specify an argument, it must follow the option to which it applies. For example, you may specify any one of the following lines:

  ```
  idlc [-p -m tie]
  idlc [-p -mtie]
  idlc [-pm tie]
  idlc [-pmtie]
  ```

  If you want to specify more than one option with an associated argument, you cannot run the options together, as in the last two lines in the example. (When you run option values together, you may specify an argument only after the last option.) With multiple options and arguments, you must separate the option-argument combinations.

- All option values are case-sensitive, even on platforms that do not require case-sensitive file names. Make sure you enter each option value exactly as shown in Table 11 on page 173.

Table 11 on page 173 lists and describes all option values and arguments.

Table 11. idlc command options

Command Option	Description
-?	Writes a brief description of the idlc command syntax to standard output.
-D<*define-expression*>	Predefines a preprocessor variable for the IDL compiler.
-d<*directory-name*>	Specifies a directory where the compiler should place emitted output files. The default is the current directory.
-e<*emit-list*>	Specifies a list of emitters to run. The emit list consists of short emitter names. Separate each emitter name in the list with a colon or a semicolon. See "Supported emitters" for a list of emitters that you can specify through this option.
-h	Writes a brief description of the idlc command syntax to standard output.
-I<*include-directory*>	Adds a directory to the list of directories that the IDL compiler uses to find #include files. In addition to the -I option, you can use the IDLC_INCLUDE environment variable to specify a list, with *include-directory* names separated by the path separator character (\).
-i<*file-name*>	Specifies the name of a file to be compiled. Use this option only if you want to compile a file that contains IDL, but has an extension other than ".idl" in its file name.
-J	Passes options to the Java interpreter that the IDL compiler uses. For example, you can set the heap size for the interpreter to 32M with the following: `-J"-mx32m"`
-m<*name[=value]*>	Specifies an output modifier that affects the bindings that the emitter produces. See "Supported output modifiers" on page 175 for a list of output modifiers that you can specify through this option.
-p	Specifies the -D_PRIVATE_ preprocessor variable for the IDL compiler.
-s<*emit-list*>	Same as -e<*emit-list*>
-V	Shows the version number of the idlc command.
-v	Specifies verbose mode, which means that you can see all the internal commands (and their arguments) that the IDL compiler issues.

Supported emitters

Table 12 lists the emitters that you can specify through the `e` option on the `idlc` command, which invokes the IDL compiler.

Table 12. Supported emitters to specify on the command to invoke the IDL compiler

Type of emitter	Emitter names/descriptions
Repository emitter	**ir** Updates the CORBA interface repository with the interfaces in this compilation unit.

Table 12. Supported emitters to specify on the command to invoke the IDL compiler (continued)

Type of emitter	Emitter names/descriptions
C++ file emitters	**hh** Produces C++ usage bindings. If you do not specify modifiers by using the -m option on the idlc command, the emitter produces bindings with support for remote, cross-language operation. If you specify the **cpponly**, **localonly**, or **somthis** modifier, the emitter produces specialized bindings. See the description of the -m option for more information. **ic** Produces a Component Broker C++ managed object implementation template. If you do not specify modifiers by using the -m option on the idlc command, the emitter produces pure CORBA C++ bindings that are suitable for use with a standalone ORB. If you specify the **mo** modifier, the emitter produces bindings that support Component Broker managed objects. See the description of the -m option for more information. **ih** Produces a C++ implementation header. If you do not specify modifiers by using the -m option on the idlc command, the emitter produces pure CORBA C++ bindings that are suitable for use with a standalone ORB. If you specify the **mo** modifier, the emitter produces bindings that support Component Broker managed objects. See the description of the -m option for more information. **sc** Produces a C++ server implementation binding. Typically, this file is the <class_name>_S.cpp file. If you do not specify modifiers by using the -m option on the idlc command, the emitter produces bindings with support for remote, cross-language operation. If you specify the **cpponly**, **localonly**, or **somthis** modifier, the emitter produces specialized bindings. See the description of the -m option for more information. **uc** Produces the client-side implementation binding. This is the <class_name>_C.cpp file. If you do not specify modifiers by using the -m option on the idlc command, the emitter produces bindings with support for remote, cross-language operation. If you specify the **cpponly**, **localonly**, or **somthis** modifier, the emitter produces specialized bindings. See the description of the -m option for more information.

Table 12. Supported emitters to specify on the command to invoke the IDL compiler (continued)

Type of emitter	Emitter names/descriptions
Java file emitters	**bj** Creates files to support business objects written in Java. Those files are: • _<interface_name>Wrapper.java that replaces _<interface_name>Skeleton.java • _<interface_name>Impl.java that is the implementation-side proxy for the C++ managed object associated with the Java business object. **sj** Produces a Java implementation skeleton. **uj** Produces the cross-language Java usage bindings.

Supported output modifiers

Table 13 lists the output modifiers that you can specify through the `m` option on the `idlc` command, which invokes the IDL compiler. These output modifiers affect the bindings that the emitters produce.

Table 13. Supported output modifiers to specify on the command to invoke the IDL compiler

Name/Value	Description
cpponly	Suppresses the production of cross-language bindings and produces standard CORBA C++ bindings that are suitable for use with a standalone ORB. This modifier affects the bindings that the hh, sc, and uc emitters produce.
dllname=<value>	Specifies the name of the DLL that contains classes into which the emitter is to place Windows NT import/export specifications.
IRforce	Forces the ir emitter to replace existing objects when you update the interface repository with objects that have the same names as the existing objects.
LINKAGE=<value>	Inserts customized C++ linkage modifiers into the generated bindings.
localonly	Generates bindings that can be used only to access a local object for all of the most-derived interfaces in the IDL file. This modifier is an alternative to using the **localonly** keyword on a #pragma in the IDL file itself.
mo	Generates bindings that support Component Broker managed objects. If you do not specify this modifier, the emitter produces pure CORBA C++ bindings that are suitable for use with a standalone ORB. This modifier affects only the ih and ic emitters.
nohelper	Prevents the uj emitter from generating the <interface_name>Helper.java file.

Table 13. Supported output modifiers to specify on the command to invoke the IDL compiler (continued)

noholder	Prevents the uj emitter from generating the <interface_name>Holder.java file.
noimpl	Prevents the bj emitter from generating the <interface_name>Impl.java file.
noimplbase	Prevents the sj emitter from generating the <interface_name>ImplBase.java file.
nointerface	Prevents the uj emitter from generating the <interface_name>.java file.
noskeleton	Prevents the sj emitter from generating the <interface_name>Skeleton.java file.
nostub	Prevents the uj emitter from generating the <interface_name>Stub.java file.
notcconsts	Eliminates the generation of C++ TypeCode constants and overloaded Any operators.
nowrapper	Prevents the bj emitter from generating the <interface_name>Wrapper.java file.
orbadapter	Generates C++ bindings that allow the C++ ORB to dispatch Java implementations.
postInclude=<file-name>	Adds the following line to the usage binding (.hh) file, just before the end of the file: `#include <file-name>`
preInclude=<file-name>	Adds the following line to the usage binding (.hh) file, just before the include statement for corba.h: `#include <file-name>`
tie	Generates bindings that assume delegation rather than inheritance.

idlc command results

When you enter an idlc command but do not specify any emitters, either through an emit list on the command itself or through the IDLC_EMIT environment variable, the IDL compiler only checks for syntax errors in the input files. Otherwise, the IDL compiler produces output files based on the emitters you specified. These output files contain language-specific usage and implementation bindings for the IDL interfaces. The output file names vary, depending on the language that you select:

- When you specify emitters for C++, each emitter produces one type of output file, whose name begins with the same name as the IDL source file. For the IDL source file named Policy.idl, for example, each emitter produces an output file as follows:

Emitter	Output file name
ic	Policy_I.cpp
ih	Policy.ih
hh	Policy.hh
sc	Policy_S.cpp
uc	Policy_C.cpp

The IDL compiler places these output files in the directory that you specified on the idlc command, or in the current directory, if you did not specify a directory.

- When you specify emitters for Java, each emitter can produce one or more types of output file. For the IDL source file named Policy.idl, for example, each emitter produces an output file as follows:

Emitter

Output file name

bj
- _PolicyWrapper.java
- _PolicyImpl.java

sj
- _PolicySkeleton.java
- _PolicyImplBase.java

uj
- Policy.java
- PolicyHelper.java
- PolicyHolder.java
- _PolicyStub.java
- Additional .java files for elements of the Policy interface. For example, if the IDL for Policy defines an exception named InvalidAmount, the uj emitter produces an output file named InvalidAmount.java.

To maintain consistency between the names of Java packages and the directory structure where Java source files reside, the IDL compiler creates a subdirectory where it places the output files. The subdirectory is a subdirectory of the directory that you specified on the idlc command, or of the current directory, if you did not specify a directory.

Appendix D. Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
USA

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

Examples in this book

The examples in this book are samples only, created by IBM Corporation. These examples are not part of any standard or IBM product and are provided to you solely for the purpose of assisting you in the development of your applications. The examples are provided "as is." IBM makes no warranties express or implied, including but not limited to the implied warranties of merchantability and fitness for a particular purpose, regarding the function or performance of these examples. IBM shall not be liable for any damages arising out of your use of the examples, even if they have been advised of the possibility of such damages.

These examples can be freely distributed, copied, altered, and incorporated into other software, provided that it bears the above disclaimer intact.

Programming interface information

This publication documents intended Programming Interfaces that allow the customer to write programs to obtain services of WebSphere for z/OS.

Trademarks

The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both:

AIX
CICS
DB2
IBM
IMS
IMS/ESA
Language Environment
Open Class
OS/390
RACF
VisualAge
VTAM
WebSphere
z/OS

Glossary

For more information on terms used in this book, refer to one of the following sources:

- Sun Microsystems Glossary of Java Technology-Related Terms, located on the Internet at:

  `http://java.sun.com/docs/glossary.html`
- *IBM Glossary of Computing Terms*, located on the Internet at:

  `http://www.ibm.com/ibm/terminology/`
- The Sun Web site, located on the Internet at:

  `http://www.sun.com/`

Index

Program Number: 5655-F31

Printed in the United States of America

SA22-7848-02