CICS Transaction Server for z/OS
Version 4 Release 1

Java Applications in CICS

SC34-7025-02

CICS Transaction Server for z/OS
Version 4 Release 1

Java Applications in CICS

SC34-7025-02

Note
Before using this information and the product it supports, read the information in "Notices" on page 395.

This edition applies to Version 4 Release 1 of CICS Transaction Server for z/OS (product number 5655-S97) and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Preface

This manual documents intended Programming Interfaces that allow the customer to write programs to obtain the services of Version 4 Release 1.

What this information is about

This information tells you how to develop and use Java applications and enterprise beans in CICS®.

Who should read this information

This information is intended for:

- Experienced Java application programmers who may have little experience of CICS, and no great need to know more about CICS than is necessary to develop and run Java programs.
- Experienced CICS users and system programmers, who need to know about CICS requirements for Java support.

Changes in CICS Transaction Server for z/OS, Version 4 Release 1

For information about changes that have been made in this release, please refer to *What's New* in the information center, or the following publications:

- *CICS Transaction Server for z/OS What's New*
- *CICS Transaction Server for z/OS Upgrading from CICS TS Version 3.2*
- *CICS Transaction Server for z/OS Upgrading from CICS TS Version 3.1*
- *CICS Transaction Server for z/OS Upgrading from CICS TS Version 2.3*

Any technical changes that are made to the text after release are indicated by a vertical bar (|) to the left of each new or changed line of information.

Part 1. Java development roadmaps

Each of these roadmaps outlines the steps required to develop a particular type of Java application in CICS.

Chapter 1. JCICS application roadmap

These are the steps you must perform to develop a JCICS application.

About this task

Procedure

1. Write a Java application, using the JCICS classes to access CICS services and resources. See Chapter 6, “Java programming using JCICS,” on page 17.
2. Use the Java Virtual Machine in CICS to execute your application. See Chapter 12, “Understanding JVMs,” on page 73 and Chapter 11, “Setting up Java support,” on page 65.

Results

Chapter 2. CICS IIOP application roadmap

These are the steps you must perform to develop a CICS IIOP application.

About this task

Procedure

1. Set up CICS as an IIOP server. See Chapter 16, “Configuring CICS for IIOP,” on page 181.
2. Write your IIOP server application, also known as a “stateless CORBA object”. See “Developing stateless CORBA objects” on page 371, “Creating the Interface Definition Language (IDL)” on page 374, and “Developing an IIOP server program” on page 375.
3. Write your client program. See “Developing the IIOP client program” on page 378.

Results

Chapter 3. CICS enterprise beans roadmap

These are the steps you must perform to develop a CICS enterprise bean application.

About this task

Procedure

1. Familiarize yourself with CICS support for enterprise beans by reading Chapter 18, “What are enterprise beans?,” on page 213.
2. Read the overview of the steps involved in setting up a CICS EJB server in “Configuring CICS as an EJB server—overview” on page 228.
3. Set up a basic, single-region EJB server and name server—see “Setting up a single-region EJB server” on page 239.
4. Test your single-region EJB server by running the EJB installation verification program (IVP)—see Chapter 20, “Using the EJB IVP,” on page 257.
5. Further test your EJB server by running the EJB sample applications—see Chapter 21, “Running the sample EJB applications,” on page 263.
6. Optionally, expand your single-region EJB server into a multi-region server capable of load balancing—see “Setting up a multi-region EJB server” on page 246.
7. Implement any security controls required by your system—see Chapter 27, “Managing security for enterprise beans,” on page 345.
8. Code your session bean. If you are not using an Integrated Development Environment (IDE), see “Coding a session bean” on page 290.
9. Follow the deployment process described in Chapter 23, “Deploying enterprise beans,” on page 303, using the tools as described in “Using CICS deployment tools for enterprise beans” on page 304.
10. Write the client program. See “Writing the client program” on page 293.

Results

Part 2. Developing Java applications for CICS

What you need to know to develop and use CICS applications written in Java.

Chapter 4. Java applications in CICS

You can write Java application programs that use CICS services and execute under CICS control, but these programs are handled differently from procedural programs written in the traditional CICS languages, such as COBOL and C.

The Java language is designed to be portable and architecture-neutral. The bytecode generated by compilation is portable, but requires a machine-specific interpreter for execution on different platforms. CICS provides this execution environment using a Java Virtual Machine (JVM) that is executing under CICS control.

Types of Java application in CICS

You can write these types of Java application in CICS.

JCICS applications
You can write Java programs that use the JCICS class library. JCICS allows you to access CICS resources such as VSAM files, CICS transient data queues and temporary storage. It also allows you to link to CICS applications written in other languages. Most of the functions of the **EXEC CICS** programming interface are supported. JCICS is supplied in the `dfjcics.jar` JAR file and can be downloaded to your workstation. It is also available with some releases of VisualAge® for Java.

JCICS applications are run in the CICS JVM. You can read more about JCICS in "The JCICS class library" on page 17.

Stateless CORBA objects
Stateless CORBA objects are Java server applications that communicate with a client application using the IIOP protocol. No state is maintained in object attributes between successive invocations of methods; state is initialized at the start of each method call and referenced by explicit parameters.

Stateless CORBA objects can receive inbound requests from a client and can also make outbound IIOP requests.

Method invocations may participate in **Object Transaction Service (OTS) distributed transactions**. If a client calls an IIOP application within the scope of an OTS transaction, information about the transaction flows as an extra parameter on the IIOP call. If a target stateless CORBA object implements the `CosTransactions::TransactionalObject` interface, the object is treated as transactional.

Note: An *OTS transaction* is a distributed unit of work, not a CICS transaction instance or resource definition.

Stateless CORBA objects can use the JCICS API to interact with CICS.

CICS stateless CORBA objects execute in the CICS JVM.

You can read more about CICS stateless CORBA objects in Chapter 29, "Stateless CORBA objects," on page 371.

Enterprise beans
Enterprise beans are portable Java components that comply with Sun Microsystems' *Enterprise JavaBeans Specification, Version 1.1*. CICS has

implemented these interfaces by mapping them to underlying CICS services. Enterprise beans can link to other CICS applications using **connectors**. You can also develop enterprise beans that use the JCICS class library to access CICS services or programs directly, but these applications will not be portable to a non-CICS EJB-compliant server.

The Enterprise JavaBeans (EJB) specification defines transactional distributed objects that communicate using the Java Remote Method Invocation (RMI) interface. CICS supports RMI over IIOP, mediated using a CORBA Object Request Broker (ORB).

Enterprise beans execute in the CICS JVM.

You can read more about Enterprise beans in Chapter 18, "What are enterprise beans?," on page 213.

Table 1 shows the features that can be used in the different types of Java application in CICS:

Table 1. Java application features

Feature	Non-IIOP CICS appl.	CICS stateless CORBA object	CICS session bean
Outbound IIOP	YES	YES	YES
Inbound IIOP	NO	YES	YES
APPC/MRO outbound UOW	YES	YES	YES
APPC/MRO inbound UOW	YES	NO	NO
EXEC CICS SYNCPOINT UOW	YES	NO	NO
Outbound OTS transaction	NO	YES	YES
Inbound OTS transaction	NO	YES	YES
Container managed OTS transaction	NO	NO	YES
Bean managed OTS transaction	NO	NO	YES
Factory publication to JNDI	NO	YES	YES
Application Metadata	NO	NO	YES
State managed	NO	NO	YES
Outbound Secure Sockets Layer (SSL)	YES	YES	YES
Inbound Secure Sockets Layer (SSL)	NO	YES	YES
Assertions	YES	YES	YES

Chapter 5. What you need to know about CICS

CICS is a transaction processing subsystem.

This means that it provides services for a user to run applications online, by request, at the same time as many other users are submitting requests to run the same applications, using the same files and programs. CICS manages the sharing of resources, integrity of data, and prioritization of execution, while maintaining fast response times.

A CICS application is a collection of related programs that together perform a business operation, such as processing a product order or preparing a company payroll. CICS applications execute under CICS control, using CICS services and interfaces to access programs and files.

CICS applications are run by submitting a **transaction** request. The term transaction has a special meaning in CICS; "CICS transactions" explains the difference from the more common industry usage. Execution of the transaction consists of running one or more **application programs** that implement the required function. In CICS documentation you may find CICS application programs sometimes called **programs**, and sometimes the term transaction is used to imply the processing done by the application programs.

To develop and run CICS applications, you need to understand the relationship between CICS programs, transactions, and tasks. These terms are used throughout CICS documentation and appear in many programming commands.

CICS transactions

A transaction is a piece of processing initiated by a single request.

The request is typically made by an end-user at a terminal. However, it could be made from a Web page, from a remote workstation program, or from an application in another CICS region; or it might be triggered automatically at a predefined time. The CICS Internet Guide and the CICS External Interfaces Guide describe different ways of running CICS transactions.

A single transaction consists of one or more **application programs** that, when run, carry out the processing needed.

However, the term **transaction** is used in CICS to mean both a single event and all other transactions of the same type. You describe each transaction-type to CICS with a TRANSACTION resource definition. This definition gives the transaction type a name (the transaction identifier, or TRANSID) and tells CICS several things about the work to be done, such as which program to invoke first, and what kind of authentication is required throughout the execution of the transaction.

You run a transaction by submitting its TRANSID to CICS. CICS uses the information recorded in the TRANSACTION definition to establish the correct execution environment, and starts the first program.

The term **transaction** is now used extensively in the IT industry to describe a **unit of recovery** or what CICS calls a **unit of work**. This is typically a complete logical

operation that is recoverable; it can be committed or backed out as an entirety as a result of a programmed command or of system failure. In many cases, the scope of a CICS transaction is also a single unit of work, but you should be aware of the difference in meaning when reading CICS documentation.

CICS tasks

You will also see the word **task** used extensively in CICS documentation.

This word has a specific meaning in CICS. When CICS receives a request to run a transaction, it starts a new task that is associated with this *one instance* of the execution of the transaction type. That is, a CICS task is one execution of a transaction, with its own private set of data, usually on behalf of a specific user. You can also consider a task as a *thread*. Tasks are *dispatched* by CICS according to their priority and readiness. When the transaction completes, the task is terminated.

CICS application programs

You write a CICS program in much the same way as you write any other program.

You can use COBOL, C, C++ , Java, PL/I, or assembler language to write CICS application programs. Most of the processing logic is expressed in standard language statements, but to request CICS services you must use one of the following:

- "**EXEC CICS**" commands provided by the CICS application programming interface (API)
- The Java class library for CICS (JCICS)
- The C++ class library for CICS

The use of the "**EXEC CICS**" API is described in the *CICS Application Programming Reference* and the *CICS System Programming Reference*. It can be used in COBOL, C, C++, PL/I, or assembler programs. *It cannot be used in Java programs*.

In Java programs, you can use the JCICS classes to access CICS services and link to CICS application programs written in other languages. JCICS is described in "The JCICS class library" on page 17. (The types of Java program that you can write are listed in "Types of Java application in CICS" on page 11.)

You can write enterprise beans that use the interfaces defined in Sun Microsystem's *Enterprise JavaBeans Specification, Version 1.1*. CICS implements this specification by mapping program requests transparently to underlying CICS services. (You can also write enterprise beans that use the JCICS classes to call CICS services directly, but if you do so your beans will not be portable to non-CICS servers.)

CICS services

CICS provides the following services, which Java programs can access through the JCICS programming interface.

CICS services managers traditionally have the word "control" in their titles—for example, "terminal control" and "program control". You will find these terms used extensively in CICS publications:

Data management services
CICS provides:

- Record-level sharing, with integrity, in accessing Virtual Storage Access Method (VSAM) datasets. CICS logs activity to support:
  - Data backout (in the case of transaction or system failure)
  - Forward recovery (in the case of media failure)

  Management of VSAM data is provided by CICS **File Control**.

  CICS also implements two proprietary file structures, and provides commands to manipulate them:

  Temporary Storage
  Temporary storage (TS) is a means of making data readily available to multiple transactions. Data is kept in **queues**, which are created as required by programs. Queues can be accessed sequentially or by item number.

  Temporary storage queues can reside in main memory, or be written to a storage device.

  A temporary storage queue can be thought of as a named scratch-pad.

  Transient Data
  Transient data (TD) is also available to multiple transactions, and is kept in queues. However, unlike TS queues, TD queues must be predefined and can only be read sequentially. Each item is removed from the queue when it is read.

  Transient data queues are always written to a dataset. You can define a transient data queue so that writing a specific number of items to it acts as a trigger to start a specific transaction. (The triggered transaction might, for example, process the queue.)
- Access to data in other databases (including DB2®), through interfaces with database products.

Communications services

CICS provides commands that give access to a wide range of terminals—displays, printers, and workstations—using SNA and TCP/IP protocols. Management of SNA and TCP/IP networks is provided by CICS **terminal control**.

You can write programs that use Advanced Program-to-Program Communication (APPC) commands to start and communicate with other programs in remote systems, using SNA protocols. CICS APPC implements the peer-to-peer distributed application model.

CICS also provides an Object Request Broker (ORB) to implement the inbound and outbound IIOP protocols defined by the Common Object Request Broker Architecture (CORBA). The ORB supports requests to execute Java stateless objects and enterprise beans.

The following CICS proprietary communications services are provided:

Function shipping
Program requests to access resources (files, queues, and programs) that are defined as existing on remote CICS regions are automatically routed by CICS to the owning region.

Distributed program link (DPL)
Program-link requests for a program defined as existing on a remote

CICS region are automatically routed to the owning region. CICS provides commands to maintain the integrity of the distributed application.

Asynchronous processing
CICS provides commands to allow a program to start another transaction in the same, or in a remote, CICS region and optionally pass data to it. The new transaction is scheduled independently, in a new task. This function is similar to the **fork** operation provided by other software products.

Transaction routing
Requests to run transactions that are defined as existing on remote CICS regions are automatically routed to the owning region. Responses to the end-user are routed back to the region that received the request.

Unit of work services
When CICS creates a new task to run a transaction, a new unit of work (UOW) is started automatically. (Thus CICS does not provide a **BEGIN** command, because one is not required.) CICS transactions are always executed *in-transaction*.

CICS provides a SYNCPOINT command to commit or roll back recoverable work done. When the syncpoint completes, CICS automatically starts another unit of work. If you terminate your program without issuing a SYNCPOINT command, CICS takes an implicit syncpoint and attempts to commit the transaction.

The scope of the commit includes all CICS resources that have been defined as recoverable, and any other resource managers that have registered an interest through interfaces provided by CICS.

If you write enterprise beans using transaction services provided by commands defined by the Java Transaction Service (JTS), these commands (including BEGIN) are mapped by CICS to its unit of work services.

Program services
CICS provides commands that enable a program to link or transfer control to another program, and return.

Diagnostic services
CICS provides commands that enable you to trace programs and produce dumps.

Other services
CICS provides other services, such as journaling, timer, and storage management, that are not available through the JCICS interface. These are described in the *CICS Application Programming Guide*.

Chapter 6. Java programming using JCICS

You can write Java application programs that use CICS services and execute under CICS control.

You can write Java programs on a workstation, or in the z/OS® UNIX System Services shell. You can use any editor of your choice, or a visual composition environment such as WebSphere Studio Application Developer.

CICS provides a Java class library, known as JCICS, supplied in the `dfjcics.jar` JAR file. JCICS is the Java equivalent of the **EXEC CICS** application programming interface (API) that you would use with other CICS supported languages, such as COBOL. It allows you to access CICS resources and integrate your Java programs with programs written in other languages. Most of the functions of the **EXEC CICS** API are supported. For a description of the JCICS API, see "The JCICS class library."

The Java language is designed to be portable and architecture-neutral. The bytecode generated by compilation is portable, but requires a machine-specific interpreter for execution on different platforms. CICS provides this execution environment by means of a Java Virtual Machine (JVM) that executes under CICS control. You can read about the CICS JVM in Chapter 12, "Understanding JVMs," on page 73.

The JCICS class library

The Java class library for CICS, JCICS, supports most of the functions of the **EXEC CICS** API commands.

These are described in "JCICS command reference" on page 21.

The JCICS classes are fully documented in JAVADOC that is generated from the class definitions. This is available in the CICS Information Center, and can be found in the JCICS Class Reference.

Translation

There is no need for a CICS translator for Java programs.

JavaBeans

Some of the classes in JCICS may be used as JavaBeans, which means that they can be customized in an application development tool such as WebSphere Studio Application Developer, serialized, and manipulated using the JavaBeans API.

The JavaBeans in JCICS are currently:

- Program
- ESDS
- KSDS
- RRDS
- TDQ
- TSQ

- AttachInitiator
- EnterRequest

These beans do not define any events; they consist of properties and methods. They can be instantiated at run-time in one of three ways:

1. By calling the new method for the class itself. (This is the recommended way.)
2. By calling Beans.instantiate() for the name of the class, with property values set manually.
3. By calling Beans.instantiate() of a `.ser` file, with property values set at design time.

If either of the first two options are chosen, then the property values, including the name of the CICS resource, must be set by invoking the appropriate set methods at run-time.

Library structure

Each JCICS library component falls into one of four categories, Interfaces, Classes, Exceptions, or Errors.

`Interfaces`
Some interfaces are provided to define sets of constants. For example, the `TerminalSendBits` interface provides a set of constants that can be used to construct a `java.util.BitSet`.

`Classes`
The supplied classes provide most of the JCICS function. The `API` class is an abstract class that provides common initialization for every class that corresponds to a part of the CICS API, except for ABENDs and exceptions. For example, the Task class provides a set of methods and variables that correspond to a CICS task.

`Errors and Exceptions`
The Java language defines both exceptions and errors as subclasses of the class `Throwable`. JCICS defines `CicsError` as a subclass of `Error`. `CicsError` is the superclass for all the other CICS error classes, which are used for severe errors.

JCICS defines `CicsException` as a subclass of `Exception`. `CicsException` is the superclass for all the CICS exception classes (including the `CicsConditionException` classes such as `InvalidQueueIdException`, which represents the CICS QIDERR condition).

See "Error handling and abnormal termination" on page 23 for further information.

CICS resources

CICS resources, such as programs or temporary storage queues, are represented by instances of the appropriate Java class, identified by the values of various properties such as name and, for some classes, a SYSID (the identifier of the CICS system that owns the resource).

Resources must be defined to CICS using the CICS Explorer™, CEDA transaction, or CICSPlex® SM BAS. See the *CICS Resource Definition Guide* or the *CICSPlex System Manager Concepts and Planning* manual for information about defining CICS resources. It is possible to use implicit remote access by defining a resource locally to point to a remote resource.

CICS storage requirements

Memory requirements to run Java programs are higher than for conventional programs.

Therefore:

1. You should ask your CICS system programmer to set the value of the **EDSALIM** system initialization parameter to a minimum of 200MB, otherwise a short-on-storage condition may occur.

   Note that you cannot change the value of **EDSALIM** during CICS execution by means of CEMT SET commands. Furthermore, dynamic changes to **EDSALIM** are cataloged in the local catalog, and the value in the local catalog overrides the **EDSALIM** parameter specified in the system initialization table during all forms of restart: initial, cold, and warm. Therefore, to change **EDSALIM**, you must specify it as a system initialization table override or re-initialize the CICS catalog data sets.
2. Your CICS job should set a minimum REGION value of 400MB.

Command arguments

Many CICS programming commands pass data in a structure known as a "communications area" (**COMMAREA**).

An alternative, and more flexible, method of passing data between programs, is to use a channel: channels are described in "Channels and containers" on page 24. The COMMAREA or channel, and any other parameters, are passed as arguments to the appropriate methods.

Many of the methods are overloaded—that is, they have different versions that take either a different number of arguments or arguments of a different type. There may be one method that has no arguments, or the minimum mandatory arguments, and another that has all of the arguments. For example, there are the following different link() methods in the Program class:

link()
: This version does a simple LINK without using a COMMAREA to pass data, nor any other options.

link(com.ibm.cics.server.CommAreaHolder)
: This version does a simple LINK, using a COMMAREA to pass data but without any other options.

link(com.ibm.cics.server.CommAreaHolder, int)
: This version does a distributed LINK, using a COMMAREA to pass data and a DATALENGTH value to specify the length of the data within the COMMAREA.

link(com.ibm.record.IByteBuffer)
: This version does a LINK using an object that implements the IByteBuffer interface of the Java Record Framework supplied with VisualAge for Java.

link(com.ibm.cics.server.Channel)
: This version does a LINK using a channel to pass data in one or more containers.

Serializable classes

These JCICS classes are serializable and so can survive a Passivate/Activate cycle.

- AddressResource

- AttachInitiator
- CommAreaHolder
- EnterRequest
- ESDS
- File
- KeyedFile
- KSDS
- NameResource
- Program
- RemotableResource
- Resource
- RRDS
- StartRequest
- SynchronizationResource
- SyncLevel
- TDQ
- TSQ
- TSQType

System.out and System.err

For each Java-related CICS task, CICS automatically creates two Java `PrintWriters` that can be used as standard out and standard error streams. The standard out and standard error streams are public fields in the `Task` called `out` and `err`.

If a CICS task is being driven from a terminal (the terminal is called a **principal facility** in this case), CICS maps the standard out and standard error streams to the task's terminal.

If the task does not have a terminal as its principal facility, the standard out and standard error streams are sent to System.out and System.err. System.out and System.err are mapped to the CICS transient data queues CESO and CESE, respectively. Your CICS system programmer creates these queues, and others used for CICS messages, during CICS installation. You can access and print or display these message queues using utility programs such as the DFH$TDWT sample program described in the *CICS Customization Guide*. DFH$TDWT is supplied with the CICS pregenerated system in CICSTS41.CICS.CICS.SDFHLOAD.

Threads

Only the initial thread in the JVM can access the JCICS API. You can create other threads, but you must route all requests to the JCICS API through the initial thread. In a JVM server environment, multiple initial threads can access the JCICS API using the same JVM.

Additionally, you must ensure that all threads other than the initial thread have terminated before doing any of the following:

- link()
- xctl()
- setNextTransaction(), setNextCOMMAREA()
- commit(), rollback()

- returning an `AbendException`

JCICS command reference

Many of the options and services available to non-Java programs through the **EXEC CICS** API are available to Java programs through JCICS. This section shows the relationship between **EXEC CICS** commands and the equivalent JCICS function.

For a full description of the **EXEC CICS** commands, see the *CICS Application Programming Reference*.

JCICS support is described under the following headings:

CICS exception handling in Java programs

CICS ABENDs and exceptions are integrated into the Java exception-handling architecture. All regular CICS ABENDs are mapped to a single Java exception, `AbendException`, whereas each CICS condition is mapped to a separate Java exception.

This leads to an ABEND-handling model in Java that is similar to the other programming languages; a single handler is given control for every ABEND, and the handler has to query the particular ABEND and then decide what to do.

If the exception representing a condition is caught by CICS itself, it is turned into an ABEND.

Java exception-handling is fully integrated with the ABEND and condition-handling in other languages, so that ABENDs can propagate between Java and non-Java programs, in the standard language-independent way. A condition is mapped to an ABEND before it leaves the program that caused or detected the condition.

However, there are several differences to the abend-handling model for other programming languages, resulting from the nature of the Java exception-handling architecture and the implementation of some of the technology underlying the Java API:

- ABENDs that are considered unhandleable in other programming languages can be caught in Java programs. These ABENDs typically occur during SYNCPOINT processing. To avoid these ABENDs interrupting Java applications, they are mapped to an extension of an unchecked exception; therefore they do not have to be declared or caught.
- Several internal CICS events, such as program termination, are also mapped to Java exceptions and can therefore be caught by a Java application. Again, to avoid interrupting the normal case, these are mapped to extensions of an unchecked exception and so do not have to be caught or declared.

Note: CICS requires the Language Environment® product to be installed and active on your OS/390® system in order to run Java applications. You should not specify the Language Environment run-time option TRAP=OFF, because this will disable abend handling in JCICS.

There are three CICS-related class hierarchies of exceptions:

1. CicsError, which extends java.lang.Error and is the base for AbendError and UnknownCicsError.
2. CicsRuntimeException, which extends java.lang.RuntimeException and is in turn extended by:

   `AbendException`
   Represents a normal CICS ABEND.

   `EndOfProgramException`
   Indicates that a linked-to program has terminated normally.

   `TransferOfControlException`
   Indicates that a program has used an xctl() method, the equivalent of the CICS XCTL command.
3. CicsException, which extends java.lang.Exception and has the subclass:

   `CicsConditionException.`
   The base class for all CICS conditions.

CICS error-handling commands

CICS condition handling is integrated into the Java exception architecture as described above. The way that the equivalent "**`EXEC CICS`**" command is supported in Java is described below:

`HANDLE ABEND`
To handle an ABEND generated by a program in any CICS-supported language, use a Java try-catch statement, with AbendException appearing in a catch clause.

`HANDLE CONDITION`
To handle a specific condition, such as PGMIDERR, use a catch clause that names the appropriate exception—in this case InvalidProgramException. Alternatively, use a catch clause naming CicsConditionException, if all CICS conditions are to be caught.

`IGNORE CONDITION`
This command is not relevant in Java applications.

`POP HANDLE and PUSH HANDLE`
These commands are not relevant in Java applications. The Java exceptions used to represent CICS ABENDs and conditions are caught by any catch block in scope.

CICS conditions

The condition-handling model in Java is different from other CICS programming languages.

In COBOL, you can define an exception-handling label for each condition. If that condition occurs during the processing of a CICS command, control transfers to the label.

In C and C++, you cannot define an exception-handling label for a condition; to detect a condition, the RESP field in the EIB must be checked after each CICS command.

In Java, any condition returned by a CICS command is mapped into a Java exception. You can include all CICS commands in a try-catch block and do specific processing for each condition, or have a single null catch clause if the particular exception is not relevant. Alternatively, you can let the condition propagate, to be handled by a catch clause at a larger scope.

See "JCICS exception mapping" on page 39 for a description of the relationship between CICS conditions and Java exceptions.

Error handling and abnormal termination

To initiate an ABEND from a Java program, you must invoke one of the Task.abend(), or Task.forceAbend() methods.

Methods	JCICS class	EXEC CICS commands
abend(), forceAbend()	Task	ABEND

ABEND

To initiate an ABEND from a Java program, invoke one of the Task.abend() methods. This causes an abend condition to be set in CICS and an AbendException to be thrown. If the AbendException is not caught within a higher level of the application object, or handled by an ABEND-handler registered in the calling program (if any), CICS terminates and rolls back the transaction.

The different abend() methods are:

- abend(String *abcode*), which causes an ABEND with the ABEND code *abcode*.
- abend(String *abcode*, boolean *dump*), which causes an ABEND with the ABEND code *abcode*. If the **dump** parameter is false, no dump is taken.
- abend(), which causes an ABEND with no ABEND code and no dump.

ABEND CANCEL

To initiate an ABEND that cannot be handled, invoke one of the Task.forceAbend() methods. As described above, this causes an AbendCancelException to be thrown which can be caught in Java programs. If you do so, you must re-throw the exception to complete **ABEND_CANCEL** processing, so that, when control returns to CICS, CICS will terminate and roll back the transaction. Only catch the AbendCancelException for notification purposes and then re-throw it.

The different forceAbend() methods are:

- forceAbend(String *abcode*), which causes an **ABEND CANCEL** with the ABEND code *abcode*.

- forceAbend(String *abcode*, boolean *dump*), which causes an **ABEND CANCEL** with the ABEND code *abcode*. If the **dump** parameter is false, no dump is taken.
- forceAbend(), which causes an **ABEND CANCEL** with no ABEND code and no dump.

APPC mapped conversations

APPC unmapped conversation support is not available from the JCICS API.

APPC mapped conversations:

Methods	JCICS class	EXEC CICS Commands
initiate()	AttachInitiator	ALLOCATE, CONNECT PROCESS
converse()	Conversation	CONVERSE
get*() methods	Conversation	EXTRACT ATTRIBUTES
get*() methods	Conversation	EXTRACT PROCESS
free()	Conversation	FREE
issueAbend()	Conversation	ISSUE ABEND
issueConfirmation()	Conversation	ISSUE CONFIRMATION
issueError()	Conversation	ISSUE ERROR
issuePrepare()	Conversation	ISSUE PREPARE
issueSignal()	Conversation	ISSUE SIGNAL
receive()	Conversation	RECEIVE
send()	Conversation	SEND
flush()	Conversation	WAIT CONVID

Basic Mapping Support (BMS)

Basic mapping support (BMS) is an application programming interface between CICS programs and terminal devices. JCICS provides support for some of the BMS application programming interface.

Methods	JCICS class	EXEC CICS Commands
sendControl()	TerminalPrincipalFacility	SEND CONTROL
sendText()	TerminalPrincipalFacility	SEND Text
	Not supported	SEND MAP, RECEIVE MAP

Channels and containers

Containers are named blocks of data designed for passing information between programs. Containers are grouped together in sets called *channels*. You can use channel and container-related JCICS commands when writing CICS enterprise beans. However, CICS does not support the transmission of channels over IIOP request streams.

For introductory information about channels and containers, and guidance about using channels in non-Java applications, see Enhanced inter-program data transfer: channels as modern-day COMMAREAs, in the *CICS Application Programming Guide*.

CICS provides the following JCICS classes that CICS Java programs can use to pass and receive channels:

- com.ibm.cics.server.CCSIDErrorException
- com.ibm.cics.server.Channel
- com.ibm.cics.server.ChannelErrorException
- com.ibm.cics.server.Container
- com.ibm.cics.server.ContainerErrorException
- com.ibm.cics.server.ContainerIterator

Note: CICS does not support the transmission of channels over IIOP request streams. This means that you cannot, for example, pass a channel to an enterprise bean on a remote region.

Table 2 lists the classes and methods that implement JCICS support for channels and containers.

Table 2. JCICS support for channels and containers

Methods	**JCICS class**	**EXEC CICS Commands**
containerIterator()	Channel	STARTBROWSE CONTAINER
createContainer()	Channel	
deleteContainer()	Channel	DELETE CONTAINER CHANNEL
getContainer()	Channel	
getName()	Channel	
delete()	Container	DELETE CONTAINER CHANNEL
get(), getLength()	Container	GET CONTAINER CHANNEL [NODATA]
getName()	Container	
put()	Container	PUT CONTAINER CHANNEL
getOwner()	ContainerIterator	
hasNext()	ContainerIterator	
next()	ContainerIterator	GETNEXT CONTAINER BROWSETOKEN
remove()	ContainerIterator	
link()	Program	LINK
xctl()	Program	XCTL
setNextChannel()	TerminalPrincipalFacility	RETURN CHANNEL
issue()	StartRequest	START CHANNEL
createChannel()	Task	
getCurrentChannel()	Task	ASSIGN CHANNEL
containerIterator()	Task	STARTBROWSE CONTAINER

The CICS condition CHANNELERR results in a ChannelErrorException being thrown; the CONTAINERERR CICS condition results in a ContainerErrorException; the CCSIDERR CICS condition results in a CCSIDErrorException.

Creating channels and containers in JCICS

To create a channel, use the createChannel() method of the Task class.

For example:

```
Task t=Task.getTask();
Channel custData = t.createChannel("Customer_Data");
```

The string supplied to the createChannel method is the name by which the Channel object is known to CICS. (The name is padded with spaces to 16 characters, to conform to CICS naming conventions.)

To create a new container in the channel, use the Channel's createContainer() method. For example:

```
Container custRec = custData.createContainer("Customer_Record");
```

The string supplied to the createContainer() method is the name by which the Container object is known to CICS. (The name is padded with spaces to 16 characters, if necessary, to conform to CICS naming conventions.) If a container of the same name already exists in this channel, a ContainerErrorException is thrown.

Putting data into a container

To put data into a Container object, use the Container.put() method.

To put data into a Container object, use the Container.put() method. Data can be added to a container as a byte array or a string. For example:

```
String custNo = "00054321";
byte[] custRecIn = custNo.getBytes();
custRec.put(custRecIn);
```

Or :

```
custRec.put("00054321");
```

Passing a channel to another program or task

To pass a channel on a program-link or transfer program control (XCTL) call, use the link() and xctl() methods of the Program class, respectively.

```
programX.link(custData);

programY.xctl(custData);
```

To set the next channel on a program-return call, use the setNextChannel() method of the TerminalPrincipalFacility class:

```
terminalPF.setNextChannel(custData);
```

To pass a channel on a START request, use the issue method of the StartRequest class:

```
startrequest.issue(custData);
```

Receiving the current channel

It is not necessary for a program to receive its current channel explicitly—see “Browsing the current channel” on page 27. However, a program can get its current channel from the current task; this enables it to extract containers by name:

```
Task t = Task.getTask();
Channel custData = t.getCurrentChannel();
if (custData != null) {
    Container custRec = custData.getContainer("Customer_Record");
} else {
    System.out.println("There is no Current Channel");
}
```

Getting data from a container

Use the Container.get() method to read the data in a container into a byte array.

```
byte[] custInfo = custRec.get();
```

Browsing the current channel

A JCICS program that is passed a channel can access all of the `Container` objects without receiving the channel explicitly.

To do this, it uses a `ContainerIterator` object. (The `ContainerIterator` class implements the java.util.Iterator interface.) When a Task object is instantiated from the current task, its containerIterator() method returns an Iterator for the current channel, or null if there is no current channel. For example:

```
Task t = Task.getTask();
ContainerIterator ci = t.containerIterator();
While (ci.hasNext()) {
    Container custData = ci.next();
    // Process the container...
}
```

A JCICS example

This example shows an excerpt of a Java class called Payroll that calls a COBOL server program named `PAYR`. The Payroll class uses the JCICS com.ibm.cics.server.Channel and com.ibm.cics.server.Container classes to do the same things that a non-Java client program would use **`EXEC CICS`** commands to do.

```
import com.ibm.cics.server.*;
public class Payroll {
     ...
     Task t=Task.getTask();

     // create the payroll_2004 channel
     Channel payroll_2004 = t.createChannel("payroll-2004");

     // create the employee container
     Container employee = payroll_2004.createContainer("employee");

     // put the employee name into the container
     employee.put("John Doe");

     // create the wage container
     Container wage = payroll_2004.createContainer("wage");

     // put the wage into the container
     wage.put("2000");

     // Link to the PAYROLL program, passing the payroll_2004 channel
     Program p = new Program();
     p.setName("PAYR");
     p.link(payroll_2004);

     // Get the status container which has been returned
     Container status = payroll_2004.getContainer("status");

     // Get the status information
     byte[] payrollStatus = status.get();
     ...
}
```

Figure 1. Java class that uses the JCICS com.ibm.cics.server.Channel and com.ibm.cics.server.Container classes to pass a channel to a COBOL server program

Diagnostic services

The JCICS application programming interface has support for these CICS trace and dump commands.

Methods	JCICS class	EXEC CICS Commands
	Not supported	DUMP
enterTrace()	EnterRequest	ENTER
enableTrace(), disableTrace()	Region, Task	TRACE

Document services

This section describes JCICS support for the commands in the DOCUMENT application programming interface.

You cannot use document support with the VisualAge for Java, Enterprise Edition for OS/390, bytecode binder.

Class `Document` maps to the **EXEC CICS** DOCUMENT API. Constructors for class `DocumentLocation` map to the AT and TO keywords of the **EXEC CICS** DOCUMENT API. Setters and getters for class SymbolList map to the SYMBOLLIST, LENGTH, DELIMITER, and UNESCAPE keywords of the **EXEC CICS** DOCUMENT API.

Methods	JCICS class	EXEC CICS Commands
create*()	Document	DOCUMENT CREATE
append*()	Document	DOCUMENT INSERT
insert*()	Document	DOCUMENT INSERT
addSymbol()	Document	DOCUMENT SET
setSymbolList()	Document	DOCUMENT SET
retrieve*()	Document	DOCUMENT RETRIEVE
get*()	Document	DOCUMENT

Environment services

CICS environment services provide access to CICS data areas, parameters, and resource attributes that are relevant to an application program.

The **EXEC CICS** commands and options that have equivalent JCICS support are:

- ADDRESS
- ASSIGN
- INQUIRE SYSTEM
- INQUIRE TASK
- INQUIRE TERMINAL/NETNAME

ADDRESS

The following support is provided for the ADDRESS options.

For complete information about the **EXEC CICS ADDRESS** command, see ADDRESS , in the *CICS Application Programming Reference* .

ACEE The Access Control Environment Element (ACEE) is created by an external security manager when a CICS user signs on. This option not supported in JCICS.

COMMAREA
A COMMAREA contains user data that is passed with a command. The COMMAREA pointer is passed automatically to the linked program by the **CommAreaHolder** argument . See "Command arguments" on page 19 for more information.

CWA The Common Work Area (CWA) contains global user data, sharable between tasks.

EIB contains information about the CICS command last executed. Access to EIB values is provided by methods on the appropriate objects. For example,

eibtrnid
is returned by the getTransactionName() method of the Task class.

eibaid is returned by the getAIDbyte() method of the TerminalPrincipalFacility class.

eibcposn
is returned by the getRow() and getColumn() methods of the Cursor class.

TCTUA
The Terminal Control Table User Area (TCTUA) contains user data associated with the terminal that is driving the CICS transaction (the principal facility). This area is used to pass information between application programs, but only if the same terminal is associated with the application programs involved. The contents of the TCTUA can be obtained using the getTCTUA() method of the TerminalPrincipalFacility class.

TWA The Transaction Work Area (TWA) contains user data that is associated with the CICS task. This area is used to pass information between application programs, but only if they are in the same task. A copy of the TWA can be obtained using the getTWA() method of the Task class.

ASSIGN

The following support is provided for the ASSIGN options.

For detailed information about the **EXEC CICS** ASSIGN command, see ASSIGN, in the *CICS Application Programming Reference* .

Methods	JCICS class	EXEC CICS Commands
getABCODE()	AbendException	ASSIGN ABCODE
getAPPLID()	Region	ASSIGN APPLID
getCurrentChannel()	Task	ASSIGN CHANNEL
getCWA()	Region	ASSIGN CWALENG
getName()	TerminalPrincipalFacility or ConversationPrincipalFacility	ASSIGN FACILITY
getFCI()	Task	ASSIGN FCI
getNetName()	TerminalPrincipalFacility or ConversationPrincipalFacility	ASSIGN NETNAME

Methods	JCICS class	EXEC CICS Commands
getPrinSysid()	TerminalPrincipalFacility or ConversationPrincipalFacility	ASSIGN PRINSYSID
getProgramName()	Task	ASSIGN PROGRAM
getQNAME()	Task	ASSIGN QNAME
getSTARTCODE()	Task	ASSIGN STARTCODE
getSysid()	Region	ASSIGN SYSID
getTCTUA()	TerminalPrincipalFacility	ASSIGN TCTUALENG
getTERMCODE()	TerminalPrincipalFacility	ASSIGN TERMCODE
getTWA()	Task	ASSIGN TWALENG
getUserid(), Task.getUSERID()	Task, TerminalPrincipalFacility or ConversationPrincipalFacility	ASSIGN USERID

No other ASSIGN options are supported.

INQUIRE SYSTEM

Support is provided for the INQUIRE SYSTEM options.

Methods	JCICS class	EXEC CICS Commands
getAPPLID(), getSYSID()	Region	INQUIRE SYSTEM

No other INQUIRE SYSTEM options are supported.

INQUIRE TASK

The following support is provided for the INQUIRE TASK options.

Methods	JCICS class	EXEC CICS Commands
getAPPLID(), getSYSID()	Task	INQUIRE TASK FACILITY
getSTARTCODE()	Task	INQUIRE TASK STARTCODE
get TransactionName()	Task	INQUIRE TASK TRANSACTION
getUserid()	Task	INQUIRE TASK USERID

Notes:

FACILITY

You can find the name of the task's principal facility by calling the getName() method on the task's principal facility, which can in turn be found by calling the getPrincipalFacility() method on the current Task object.

FACILITYTYPE

You can determine the type of facility by using the Java instanceof operator to check the class of the returned object reference.

No other INQUIRE TASK options are supported.

INQUIRE TERMINAL and INQUIRE NETNAME

The following support is provided for INQUIRE TERMINAL and INQUIRE NETNAME options.

Methods	JCICS class	EXEC CICS Commands
Terminal.getUser(), getUserid()	Terminal, ConversationalPrincipalFacility	INQUIRE TERMINAL USERID INQUIRE NETNAME USERID

Note: You can also find the USERID value by calling the getUSERID() method on the current Task object, or on the object representing the task's principal facility

No other INQUIRE TERMINAL or NETNAME options are supported.

File services

JCICS provides classes and methods which map to the **EXEC CICS** commands for each type of CICS file (and index).

CICS supports the following types of files:

- Key Sequenced Data Sets (KSDS)
- Entry Sequenced Data Sets (ESDS)
- Relative Record Data Sets (RRDS)

KSDS and ESDS files can have alternate (or secondary) indexes. (CICS does not support access to an RRDS file through a secondary index.) Secondary indexes are treated by CICS as though they were separate KSDS files in their own right, which means they have separate FD entries.

There are a few differences between accessing KSDS, ESDS (primary index), and ESDS (secondary index) files, which means that you cannot always use a common interface.

Records can be read, updated, deleted, and browsed in all types of file, with the exception that records cannot be deleted from an ESDS file.

See VSAM data sets: KSDS, ESDS, RRDS, in the *CICS Application Programming Guide*, for more information about data sets.

Java commands that read data support only the equivalent of the SET option on **EXEC CICS** commands. The data returned is automatically copied from CICS storage to a Java object.

The Java interfaces relating to File Control are in five categories:

File The superclass for the other file classes; contains methods common to all file classes.

KeyedFile
Contains the interfaces common to a KSDS file accessed through the primary index, a KSDS file accessed through a secondary index, and an ESDS file accessed through a secondary index.

KSDS Contains the interface specific to KSDS files.

ESDS Contains the interface specific to ESDS files accessed through Relative Byte Address (RBA, its primary index) or Extended Relative Byte Address (XRBA). To use XRBA instead of RBA, issue the setXRBA(true) method.

RRDS Contains the interface specific to RRDS files accessed through Relative Record Number (RRN, its primary index).

For each file, there are two objects that can be operated on—the File object and the FileBrowse object. The `File` object represents the file itself and can be used with methods to perform the following operations:

- DELETE
- READ
- REWRITE
- UNLOCK
- WRITE
- STARTBR

A File object is created by the user application explicitly instantiating the desired file class. The FileBrowse object represents a browse operation on a file. (There can be more than one active browse against a specific file at any time, each browse being distinguished by a REQID.) Methods can be invoked against a file browse object to perform the following operations:

- ENDBR
- READNEXT
- READPREV
- RESETBR

A FileBrowse object is not instantiated explicitly by the user application; it is created and returned to the user class by the methods that perform the STARTBR operation.

The following tables show how the JCICS classes and methods map to the **EXEC CICS** commands for each type of CICS file (and index). In these tables, the JCICS classes and methods are shown in the form `class.method()`. For example, KeyedFile.read() refers to the read() method in the KeyedFile class.

This table shows the classes and methods for keyed files:

KSDS primary or secondary index	ESDS secondary index	CICS File command
KeyedFile.read()	KeyedFile.read()	READ
KeyedFile.readForUpdate()	KeyedFile.readForUpdate()	READ UPDATE
KeyedFile.readGeneric()	KeyedFile.readGeneric()	READ GENERIC
KeyedFile.rewrite()	KeyedFile.rewrite()	REWRITE
KSDS.write()	KSDS.write()	WRITE
KSDS.delete()		DELETE
KSDS.deleteGeneric()		DELETE GENERIC
File.unlock()	File.unlock()	UNLOCK
KeyedFile.startBrowse()	KeyedFile.startBrowse()	START BROWSE
KeyedFile.startGenericBrowse()	KeyedFile.startGenericBrowse()	START BROWSE GENERIC
KeyedFileBrowse.next()	KeyedFileBrowse.next()	READNEXT
KeyedFileBrowse.previous()	KeyedFileBrowse.previous()	READPREV

KSDS primary or secondary index	ESDS secondary index	CICS File command
KeyedFileBrowse.reset()	KeyedFileBrowse.reset()	RESET BROWSE
FileBrowse.end()	FileBrowse.end()	END BROWSE

This table shows the classes and methods for non-keyed files. ESDS and RRDS are accessed by their primary indexes:

ESDS primary index	RRDS primary index	CICS File command
ESDS.read()	RRDS.read()	READ
ESDS.readForUpdate()	RRDS.readForUpdate()	READ UPDATE
ESDS.rewrite()	RRDS.rewrite()	REWRITE
ESDS.write()	RRDS.write()	WRITE
	RRDS.delete()	DELETE
File.unlock()	File.unlock()	UNLOCK
ESDS.startBrowse()	RRDS.startBrowse()	START BROWSE
ESDS_Browse.next()	RRDS_Browse.next()	READNEXT
ESDS_Browse.previous()	RRDS_Browse.previous()	READPREV
ESDS_Browse.reset()	RRDS_Browse.reset()	RESET BROWSE
FileBrowse.end()	FileBrowse.end()	END BROWSE
ESDS.setXRBA()		

Data to be written to a file must be in a Java byte array.

Data is read from a file into a RecordHolder object; the storage is provided by CICS and will be automatically released at the end of the program.

The **KEYLENGTH** does not need to be explicitly specified on any File method; the length used will be the actual length of the key passed. When a FileBrowse object is created, it contains the keylength of the key specified on the startBrowse method, and this length is passed to CICS on subsequent browse requests against that object.

It is not necessary for the user to provide a **REQID** for a browse operation; each browse object will contain a unique REQID which is automatically used for all subsequent browse requests against that browse object.

HTTP and TCP/IP services

Getters in classes `HttpHeader`, `NameValueData`, and `FormField` return HTTP headers, name and value pairs, and form field values for the appropriate API commands.

Methods	JCICS class	`EXEC CICS` Commands
get*()	CertificateInfo	EXTRACT CERTIFICATE / EXTRACT TCPIP
get*()	HttpRequest	EXTRACT WEB
getHeader()	HttpRequest	WEB READ HTTPHEADER
getFormField()	HttpRequest	WEB READ FORMFIELD
getContent()	HttpRequest	WEB RECEIVE

Methods	JCICS class	EXEC CICS Commands
getQueryParm()	HttpRequest	WEB READ QUERYPARM
startBrowseHeader()	HttpRequest	WEB STARTBROWSE HTTPHEADER
getNextHeader()	HttpRequest	WEB READNEXT HTTPHEADER
endBrowseHeader()	HttpRequest	WEB ENDBROWSE HTTPHEADER
startBrowseFormField()	HttpRequest	WEB STARTBROWSE FORMFIELD
getNextFormField()	HttpRequest	WEB READNEXT FORMFIELD
endBrowseFormField()	HttpRequest	WEB ENDBROWSE FORMFIELD
startBrowseQueryParm()	HttpRequest	WEB STARTBROWSE QUERYPARM
getNextQueryParm()	HttpRequest	WEB READNEXT QUERYPARM
endBrowseQueryParm()	HttpRequest	WEB ENDBROWSE QUERYPARM
writeHeader()	HttpResponse	WEB WRITE
getDocument()	HttpResponse	WEB RETRIEVE
getCurrentDocument()	HttpResponse	WEB RETRIEVE
sendDocument()	HttpResponse	WEB SEND

Note: Use the method get HttpRequestInstance() to obtain the HttpRequest object.

Each incoming HTTP request processed by CICS Web support includes an HTTP header. If the request uses the POST HTTP verb it also includes document data. Each response HTTP request generated by CICS Web support includes an HTTP header and document data.

To process this JCICS provides the following Web and TCP/IP services:

HTTP Header
You can examine the HTTP header using the HttpRequest class. With HTTP in GET mode, if a client has filled in an HTTP form and selected the submit button, the query string is submitted.

SSL CICS Web support provides the TcpipRequest class, which is extended by HttpRequest to obtain more information about which client submitted the request as well as basic information on the SSL support. If an SSL certificate is provided, you can use the CertificateInfo class to examine it in detail.

Documents
If a document is published to the server (HTTP POST), it is provided as a CICS document. You can access it by calling the getDocument() method on the HttpRequest class. See “Document services” on page 28 for more information about processing existing documents.

To serve the HTTP client web content resulting from a request, the server programmer needs to create a CICS document using the Document Services API and call the sendDocument() method.

For more information on CICS Web support see CICS Web support concepts and structure, in the *CICS Internet Guide*. For more information on the JCICS Web classes see the *JCICS Class Reference*.

Program services

JCICS supports the CICS program control commands as described.

Methods	JCICS class	EXEC CICS Commands
link()	Program	LINK
SetNextTransaction(), setNextCOMMAREA(), setNextChannel()	TerminalPrincipalFacility	RETURN
xctl()	Program	XCTL
	Not supported	SUSPEND

LINK and XCTL

You can transfer control to another program that is defined to CICS using the link() and xctl() methods. The target program can be in any language supported by CICS.

If you use the xctl() method, a TransferOfControlException is thrown to the issuing program, even if it completes successfully.

RETURN

Only the pseudoconversational aspects of this command are supported. It is not necessary to make a CICS call to return; the application can terminate as normal. The pseudoconversational functions are supported by methods in the TerminalPrincipalFacility class: setNextTransaction() is equivalent to using the TRANSID option of RETURN; setNextCOMMAREA() is equivalent to using the COMMAREA option; while setNextChannel() is equivalent to using the CHANNEL option. These methods can be invoked at any time during the running of the program, and take effect when the program terminates.

Note: The length of the COMMAREA provided is used as the LENGTH value for CICS. This value should not exceed 32 500 bytes if the COMMAREA is to be passed between any two CICS servers (for any combination of product/version/release). This limit allows for the 32 500 byte COMMAREA and space for headers.

Scheduling services

JCICS provides support for the CICS scheduling services, which let you retrieve data stored for a task, cancel interval control requests, and start a task at a specified time.

Methods	JCICS class	EXEC CICS Commands
cancel()	StartRequest	CANCEL
retrieve()	Task	RETRIEVE
issue()	StartRequest	START

To define what is to be retrieved by the Task.retrieve() method, use a java.util.BitSet object. The com.ibm.cics.server.RetrieveBits class defines the bits which can be set in the BitSet object; they are:

- RetrieveBits.DATA
- RetrieveBits.RTRANSID
- RetrieveBits.RTERMID
- RetrieveBits.QUEUE

These correspond to the options on the **EXEC CICS** RETRIEVE command.

The Task.retrieve() method retrieves up to four different pieces of information in a single invocation, depending on the settings of the RetrieveBits. The DATA, RTRANSID, RTERMID and QUEUE data are placed in a RetrievedData object, which is held in a RetrievedDataHolder object. The following example retrieves the data and transid:

```
BitSet bs = new BitSet();
bs.set(RetrieveBits.DATA, true);
bs.set(RetrieveBits.RTRANSID, true);
RetrievedDataHolder rdh = new RetrievedDataHolder();
t.retrieve(bs, rdh);
byte[] inData = rdh.value.data;
String transid = rdh.value.transId;
```

Serialization services

JCICS provides support for the CICS serialization services, which let you schedule the use of a resource by a task.

Methods	JCICS class	EXEC CICS Commands
dequeue()	SynchronizationResource	DEQ
enqueue(), tryEnqueue()	SynchronizationResource	ENQ

Storage services

No support is provided for explicit storage management using CICS services (such as **EXEC CICS** GETMAIN). You should find that the standard Java storage management facilities are sufficient to meet the needs for task-private storage.

Sharing of data between tasks must be accomplished using CICS resources.

Names are generally represented as Java strings or byte arrays; you must ensure that these are of the necessary length.

Temporary storage queue services

JCICS support for the temporary storage commands.

Methods	JCICS class	EXEC CICS Commands
delete()	TSQ	DELETEQ TS
readItem(), readNextItem()	TSQ	READQ TS
writeItem(), rewriteItem() writeItemConditional() rewriteItemConditional()	TSQ	WRITEQ TS

DELETEQ TS
You can delete a temporary storage queue (TSQ) using the delete() method in the TSQ class.

READQ TS
The CICS INTO option is not supported in Java programs. You can read a specific item from a TSQ using the readItem() and readNextItem methods in the TSQ class. These methods take an ItemHolder object as one of their

arguments, which will contain the data read in a byte array. The storage for this byte array is created by CICS and is garbage-collected at the end of the program.

WRITEQ TS
You must provide data to be written to a temporary storage queue in a Java byte array. The writeItem() and rewriteItem() methods suspend if a NOSPACE condition is detected, and wait until space is available to write the data to the queue. The writeItemConditional() and rewriteItemConditional() methods do not suspend in the case of a NOSPACE condition, but return the condition immediately to the application as a NoSpaceException.

Terminal services

JCICS provides support for these CICS terminal services commands.

Methods	JCICS class	EXEC CICS Commands
converse()	TerminalPrincipalFacility	CONVERSE
	Not supported	HANDLE AID
receive()	TerminalPrincipalFacility	RECEIVE
send()	TerminalPrincipalFacility	SEND
	Not supported	WAIT TERMINAL

If a task has a terminal as a principal facility, CICS automatically creates two Java PrintWriters that can be used as standard output and standard error streams. They are mapped to the task's terminal. The two streams, called `out` and `err`, are public files in the Task object and can be used just like System.out and System.err.

Data to be sent to a terminal must be provided in a Java byte array. Data is read from the terminal into a DataHolder object. CICS provides the storage for the returned data and it will be deallocated when the program ends.

Transient data queue services

JCICS support for the transient data commands. All options are supported except INTO.

Methods	JCICS class	EXEC CICS Commands
delete()	TDQ	DELETEQ TD
readData(), readDataConditional()	TDQ	READQ TD
writeData()	TDQ	WRITEQ TD

DELETEQ TD
You can delete a transient data queue (TDQ) using the delete() method in the TDQ class.

READQ TD
The CICS INTO option is not supported in Java programs. You can read from a TDQ using the readData() or the readDataConditional() method in the TDQ class. These methods take as a parameter an instance of a DataHolder object that will contain the data read in a byte array. The storage for this byte array is created by CICS and is garbage-collected at the end of the program.

The readDataConditional() method drives the CICS NOSUSPEND logic. If a QBUSY condition is detected, it is returned to the application immediately as a QueueBusyException.

The readData() method suspends if it attempts to access a record in use by another task and there are no more committed records.

WRITEQ TD
You must provide data to be written to a TDQ in a Java byte array.

Unit of work (UOW) services

JCICS provides support for the CICS SYNCPOINT service.

Methods	JCICS class	EXEC CICS Commands
commit(), rollback()	Task	SYNCPOINT

Web services

JCICS supports all the API commands that are available for working with web services in an application.

Methods	JCICS class	EXEC CICS commands
invoke()	WebService	INVOKE WEBSERVICE
create()	SoapFault	SOAPFAULT CREATE
addFaultString()	SoapFault	SOAPFAULT ADD FAULTSTRING
addSubCode()	SoapFault	SOAPFAULT ADD SUBCODESTR
delete()	SoapFault	SOAPFAULT DELETE
create()	WSAEpr	WSAEPR CREATE
delete()	WSAContext	WSACONTEXT DELETE
set*()	WSAContext	WSACONTEXT BUILD
get*()	WSAContext	WSACONTEXT GET

The following example shows how you might use JCICS to create a web service request:

```
Channel requesterChannel = Task.getTask().createChannel("TestRequester");
                    Container appData = requesterChannel.createContainer("DFHWS-DATA");
                    byte[] exampleData = "ExampleData".getBytes();
                    appData.put(exampleData);

                    WebService requester = new WebService();
                    requester.setName("MyWebservice");
                    requester.invoke(requesterChannel, "myOperationName");

                    byte[] response = appData.get();
```

To handle the application data that is sent and received in a web service request, you can use a tool such as JZOS to generate classes for you if you are working with structured data. For more information, see the IBM® Redbooks® publication, Java Application Development for CICS. You can also use Java to generate and consume XML directly.

JCICS exception mapping

In Java, a condition returned by a CICS command is mapped into a Java exception.

Table 3. Java exception mapping

CICS condition	Java Exception	CICS condition	Java Exception
ALLOCERR	AllocationErrorException	CBIDERR	InvalidControlBlockIdException
CCSIDERR	CCSIDErrorException	CHANNELERR	ChannelErrorException
CONTAINERERR	ContainerErrorException	DISABLED	FileDisabledException
DSIDERR	FileNotFoundException	DSSTAT	DestinationStatusChangeException
DUPKEY	DuplicateKeyException	DUPREC	DuplicateRecordException
END	EndException	ENDDATA	EndOfDataException
ENDFILE	EndOfFileException	ENDINPT	EndOfInputIndicatorException
ENQBUSY	ResourceUnavailableException	ENVDEFERR	InvalidRetrieveOptionException
EOC	EndOfChainIndicatorException	EODS	EndOfDataSetIndicatorException
EOF	EndOfFileIndicatorException	ERROR	ErrorException
EXPIRED	TimeExpiredException	FILENOTFOUND	FileNotFoundException
FUNCERR	FunctionErrorException	IGREQID	InvalidREQIDPrefixException
IGREQCD	InvalidDirectionException	ILLOGIC	LogicException
INBFMH	InboundFMHException	INVERRTERM	InvalidErrorTerminalException
INVEXITREQ	InvalidExitRequestException	INVLDC	InvalidLDCException
INVMPSZ	InvalidMapSizeException	INVPARTNSET	InvalidPartitionSetException
INVPARTN	InvalidPartitionException	INVREQ	InvalidRequestException
INVTSREQ	InvalidTSRequestException	IOERR	IOErrorException
ISCINVREQ	ISCInvalidRequestException	ITEMERR	ItemErrorException
JIDERR	InvalidJournalIdException	LENGERR	LengthErrorException
MAPERROR	MapErrorException	MAPFAIL	MapFailureException
NAMEERROR	NameErrorException	NODEIDERR	InvalidNodeIdException
NOJBUFSP	NoJournalBufferSpaceException	NONVAL	NotValidException
NOPASSBKRD	NoPassbookReadException	NOPASSBKWR	NoPassbookWriteException
NOSPACE	NoSpaceException	NOSPOOL	NoSpoolException
NOSTART	StartFailedException	NOSTG	NoStorageException
NOTALLOC	NotAllocatedException	NOTAUTH	NotAuthorisedException
NOTFND	RecordNotFoundException	NOTOPEN	NotOpenException
OPENERR	DumpOpenErrorException	OVERFLOW	MapPageOverflowException
PARTNFAIL	PartitionFailureException	PGMIDERR	InvalidProgramIdException
QBUSY	QueueBusyException	QIDERR	InvalidQueueIdException
QZERO	QueueZeroException	RDATT	ReadAttentionException
RETPAGE	ReturnedPageException	ROLLEDBACK	RolledBackException
RTEFAIL	RouteFailedException	RTESOME	RoutePartiallyFailedException
SELNERR	DestinationSelectionErrorException	SESSBUSY	SessionBusyException
SESSIONERR	SessionErrorException	SIGNAL	InboundSignalException
SPOLBUSY	SpoolBusyException	SPOLERR	SpoolErrorException

Table 3. Java exception mapping (continued)

CICS condition	Java Exception	CICS condition	Java Exception
STRELERR	STRELERRException	SUPPRESSED	SuppressedException
SYMBOLERR	SymbolErrorException	SYSBUSY	SystemBusyException
SYSIDERR	InvalidSystemIdException	TASKIDERR	InvalidTaskIdException
TCIDERR	TCIDERRException	TEMPLATERR	TemplateErrorException
TERMERR	TerminalException	TERMIDERR	InvalidTerminalIdException
TOKENERR	TokenErrorException		
TRANSIDERR	InvalidTransactionIdException	TSIOERR	TSIOErrorException
UNEXPIN	UnexpectedInformationException	USERIDERR	InvalidUserIdException
WRBRK	WriteBreakException	WRONGSTAT	WrongStatusException

Note: `NonHttpDataException` is thrown by `getContent()` if the CICS command WEB RECEIVE indicates that the data received is a non-HTTP message (by setting TYPE=HTTPNO).

Using JCICS

You use the classes from the JCICS library like normal Java classes. Your applications declare a reference of the required type and a new instance of a class is created using the `new` operator.

You name CICS resources using the setName method to supply the name of the underlying CICS resource.

Once created, you can manipulate objects using standard Java constructs. Methods of the declared objects may be invoked in the usual way. Full details of the methods supported for each class are available on-line in the supplied HTML JAVADOC files; a summary is provided in "JCICS command reference" on page 21.

Writing the main method

For Java programs, CICS attempts to pass control to method main(CommAreaHolder) in the class specified by the JVMCLASS option of the PROGRAM resource definition. If this method is not found, CICS tries to invoke method main(String[]).

Creating objects

To create an object you need to perform these steps.

About this task

1. Declare a reference. For example:
   ```
   TSQ tsq;
   ```
2. Use the new operator to create an object:
   ```
   tsq = new TSQ()
   ```
3. Use the setName method to give the object a name:
   ```
   tsq.setName("JCICSTSQ");
   ```

Using objects

You can use this example to create a TSQ object and invoke the delete method on the temporary storage queue object you have just created, catching the exception thrown if the queue is empty.

```
// Define a package name for the program
package unit_test;

// Import the JCICS package
import com.ibm.cics.server.*;

// Declare a class for a CICS application
public class JCICSTSQ {

    // The main method is called when the application runs
    public static void main(CommAreaHolder cah) {

         try {
             // Create and name a Temporary Storage queue object
             TSQ tsq = new TSQ();
             tsq.setName("JCICSTSQ");

             // Delete the queue if it exists
             try {
                   tsq.delete();
             } catch(InvalidQueueIdException e) {
                  // Absorb QIDERR
                  System.out.println("QIDERR ignored!");
             }

             // Write an item to the queue
             String transaction = Task.getTask().getTransactionName();
             String message = "Transaction name is - " + transaction;
             tsq.writeItem(message.getBytes());

         } catch(Throwable t) {
             System.out.println("Unexpected Throwable: " + t.toString());
         }

         // Return from the application
         return;
    }
}
```

Important:

- **You are strongly recommended not to use finalizers in CICS Java programs**. For an explanation of why finalizers are not recommended, see the *IBM Developer Kit and Runtime Environment, Java Technology Edition Diagnostics Guide*, which is available to download from www.ibm.com/developerworks/java/jdk/diagnosis/.
- **You are strongly recommended not to end a CICS Java program by issuing a System.exit() call.**

  When Java applications are run in CICS, the public static void main() method is called through the use of another Java program called the **Java wrapper**. The use of the wrapper allows CICS to initialize the environment for Java applications and, more importantly, to clean up any processes that are used during the life of the application. Killing the JVM, even with a clean return code of 0, does not allow this cleanup process to run, and may lead to data inconsistency. Also, a System.exit() call makes the continuous JVM mode unusable, because it terminates the JVM instance. The recommended approach is to allow the program to run to the end of the public static void main() method and the JVM to terminate cleanly.

Chapter 7. Accessing data from CICS applications written in Java

CICS applications written in Java can use a variety of methods to access data. The methods available depend on the type of data to be accessed.

Accessing relational data

To access relational data, a CICS application written in Java can use any of the following methods:

- A JCICS LINK command, or the **CCI Connector for CICS TS**, to link to a program that uses Structured Query Language (SQL) commands to access the data. For information about using the CCI Connector for CICS TS, see Chapter 25, "The CCI Connector for CICS TS," on page 319.
- Where a suitable driver is available, use Java Data Base Connectivity (JDBC) or Structured Query Language for Java (SQLJ) calls to access the data directly. Suitable JDBC drivers are available for DB2. Using JDBC and SQLJ to access DB2 data from Java programs and enterprise beans written for CICS, in the *CICS DB2 Guide*, tells you how to use the JDBC and SQLJ application programming interfaces and the DB2-supplied JDBC drivers to access data held in a DB2 database.

  Note: To use JDBC or SQLJ from a Java program or enterprise bean with a Java 2 security policy mechanism active, you must use the JDBC 2.0 driver provided by DB2 Version 7. The JDBC 1.2 driver provided by DB2 does not support Java 2 security, and will fail with a security exception. Requirements to support Java programs in the CICS DB2 environment, in the *CICS DB2 Guide*, tells you how to grant permissions to the JDBC driver in your Java 2 security policy.
- **Data Access beans** developed using Visual Age for Java. Data Access beans give you a fast, easy, non-programming way of building SQL queries. They might have a higher overhead than plain JDBC or SQLJ calls, as you cannot tailor them so precisely for your application. However, if you are not experienced in JDBC or SQLJ programming, Data Access beans reduce application development time and are more convenient to use. Data Access beans are described in "Using Data Access beans" on page 44.
- JavaBeans that use JDBC or SQLJ as the underlying access mechanism. You can use any suitable Java integrated development environment (IDE) to develop such JavaBeans.
- Entity beans. CICS does not support entity beans running under CICS but does support access to entity beans running on other EJB servers. A CICS enterprise bean could, for example, use an entity bean running on WebSphere® Application Server to access DB2 on z/OS.

Accessing DL/I data

To access DLI data, a CICS application written in Java can use a JCICS LINK command, or the CCI Connector for CICS TS, to link to a program that issues EXEC DLI commands to access the data. For information about using the CCI Connector for CICS TS, see Chapter 25, "The CCI Connector for CICS TS," on page 319.

Accessing VSAM data

To access VSAM data, a CICS application written in Java can use either of the following methods:

- Use a JCICS LINK command, or the CCI Connector for CICS TS, to link to a program that issues CICS File Control commands to access the data. For information about using the CCI Connector for CICS TS, see Chapter 25, "The CCI Connector for CICS TS," on page 319.
- Use the JCICS File Control classes to access VSAM directly.

Note:

1. All the above techniques can be used by both CICS enterprise beans and CICS Java programs.
2. The same data can be accessed by CICS enterprise beans, CICS Java programs, and (excluding CICS VSAM data) by non-CICS entity beans.
3. For all the above techniques except the use of entity beans, data integrity is maintained by the CICS recovery manager. When entity beans are used, you can use CICS and, for example, WebSphere Application Server's global transactional support, to maintain data integrity.
4. You can encapsulate JCICS commands in a JavaBean. This makes it easier to program the enterprise beans that use JCICS to access data.

Using Data Access beans

To access relational databases, CICS applications written in Java can use JDBC or SQLJ calls together with a suitable JDBC driver.

However, if you are not experienced in JDBC or SQLJ programming, you might find it more convenient to use Data Access beans, which package the native JDBC calls with extra function. Data Access beans are JavaBeans, not enterprise beans. They are a feature of VisualAge for Java.

Three Data Access beans provide core function for accessing databases:

- Select bean
- Modify bean
- ProcedureCall bean

Additional beans provide user interfaces to invoke methods on the core beans and to help display output from the database:

- CellSellector bean
- RowSelector bean
- ColumnSelector bean
- CellRangeSellector bean

All the beans mentioned are non-visual.

The Select, Modify, and ProcedureCall beans have properties that contain connection aliases and SQL specifications. These properties allow you to connect to relational databases and access data. You can also use parameterized SQL statements with the Select, Modify, and ProcedureCall beans.

For detailed programming information about Data Access beans, see the softcopy document *Data Access*, supplied with VisualAge for Java Enterprise Edition, Version 4.

Chapter 8. Connectivity from Java applications in CICS

Java programs in the CICS environment can open TCP/IP sockets and communicate with external processes. This means you can use Java programs as a gateway to connect to other enterprise applications that might not be available to CICS programs in other languages. For example, you could write a Java program to communicate with a remote LDAP server, servlet, database, CORBA program, or enterprise bean.

In some cases, this connectivity is integrated with CICS to provide enterprise qualities of service, such as distributed transactions and identity propagation. In other cases, connectivity can be used but without distributed transactions and other services provided by CICS. Depending on the type of connectivity you require, third party vendor products might be available which enable connectivity with enterprise applications that are not natively supported by CICS.

Generally, JVMs in the CICS environment are similar in capability to batch mode JVMs. A batch mode JVM is one that runs as a standalone process outside the CICS environment, and is typically started from a UNIX System Services command line or with a JCL job. Most applications that can be made to work in a batch mode JVM should also run in a JVM in CICS to the same extent. For example, if you can write a batch mode Java application to communicate with a non-IBM database using a third-party JDBC driver, then the same application should work in a JVM in CICS. If you want to use vendor supplied code such as non-IBM JDBC drivers in a JVM in CICS, you should consult with your vendor to determine whether they support their code executing in a JVM in CICS.

Some batch mode applications might behave in a different way when hosted in a JVM in CICS. This might occur because of the way in which CICS reuses JVMs. Any data stored in static variables persists across uses of the JVM. “Programming for JVMs in CICS” on page 133 has more information about application behavior in JVMs in CICS.

Differences in behavior can also occur with applications that communicate using IIOP. These applications use the CICS Object Request Broker (ORB) and are subject to the benefits and limitations that this confers. “The Object Request Broker (ORB)” on page 169 has more information about the CICS ORB.

Batch mode applications that run in a JVM in the CICS environment do not normally exploit the capabilities of CICS. For example, if a Java program in CICS updates records in a non-IBM database using a third-party JDBC driver, CICS is not aware of this activity, and does not attempt to enrol the updates into the current CICS transaction.

Chapter 9. Using CICS Transaction Gateway resource adapters to call CICS server programs

These topics describe the CICS resource adapters supplied by the CICS Transaction Gateway (CICS TG) product. The resource adapters enable non-CICS Java client programs to call CICS TS server programs.

Note: These topics do not describe the CICS Transaction Gateway resource adapters, classes, or API in detail. It is intended to be read in conjunction with the *CICS Transaction Gateway Programming Guide* and the *CICS Transaction Gateway Programming Reference*.

The CICS TG resource adapters allow you to reuse your CICS programs in new Java applications. Frequently, new Java applications can be developed more quickly and reliably by harnessing the power of existing (Java or non-Java) CICS programs. Typically, the Java client application is network-based and the CICS program is written in a language such as COBOL. The following sections review the several ways in which existing CICS TS for z/OS programs can be accessed from Java code, and shows how the CICS resource adapters fit into this pattern.

CICS support for the J2EE Connector Architecture (JCA)

The Java 2 Platform Enterprise Edition (J2EE) Connector Architecture defines a standard means of connecting a J2EE-compliant platform to a heterogeneous Enterprise Information System (EIS) such as CICS. Java applications interact with resource adapters using the Common Client Interface (CCI), which is an open standard defined by the JCA.

The J2EE connector architecture enables an EIS vendor to provide a standard resource adapter for its EIS. A resource adapter is the middle-tier between a Java application and an EIS, and permits the Java application to connect to the EIS.

The CICS Transaction Gateway implements the JCA by providing J2EE CICS resource adapters that support the Common Client Interface.

Accessing CICS programs from Java programs *outside* CICS

From the network, a Java client application can use any of the following methods to call a CICS TS program:

The CICS Transaction Gateway API
The CICS Transaction Gateway API provides, among other things, the following facilities:

The External Call Interface
The External Call Interface (ECI) enables a non-CICS application to call a CICS server program. To be eligible, the server program must be capable of being called from another CICS program by an **EXEC CICS LINK** command. It can be COMMAREA-based or (when the IPIC protocol is used) can use containers to transfer data.

CICS programs that are invoked by an ECI request must follow the rules for distributed program link (DPL) requests. For information

about DPL requests see the *CICS Application Programming Guide*. For information about the API restriction for DPL requests see the *CICS Application Programming Reference*.

The External Presentation Interface
The External Presentation Interface (EPI) enables a non-CICS user application to call a 3270-based CICS server program and use its output. It does this by allowing the client application to install and delete virtual IBM 3270 terminals in the CICS server. The definitions used by the EPI are treated by the CICS server as remote 3270 terminal definitions and therefore support automatic transaction initiation requests (ATI).

The External Security Interface
The External Security Interface (ESI) enables a non-CICS user application to perform certain security functions. This includes accessing the information about user IDs held in the CICS external security manager (ESM), and setting the default security credentials to be used for a server connection.

For introductory and explanatory information about the CICS Transaction Gateway API, see the *CICS Transaction Gateway Programming Guide*. For reference information about the API, see the *CICS Transaction Gateway Programming Reference*.

The ECI resource adapters
The ECI resource adapters provide a high-level CCI interface to the External Call Interface that can be used to link to CICS TS server programs and pass data in COMMAREAs or containers. The resource adapters can be deployed into a J2EE application server to allow J2EE enterprise applications to access CICS. When the JCA is used, connection pooling, security, and transaction context are managed by the J2EE application server instead of the application.

For z/OS, two ECI resource adapters are supplied:

CICS Transaction Gateway on z/OS
Adapter `cicseciXA.rar`, which supports 2-phase commit.

CICS Transaction Gateway on z/OS and distributed platforms
Adapter `cicseci.rar`, which supports single-phase commit only.

As well as the basic call-program function, in CICS TS for z/OS, Version 3.2 the ECI resource adapters support the following additional features:

- Containers. For information about using containers instead of COMMAREAs with your CICS server programs, see the *CICS Application Programming Guide*.
- IPIC connections (also known as *IPCONNs*) to CICS. For information about the attributes of IPIC connections, see the *CICS Resource Definition Guide*.

  You cannot install static IPCONN connections to non-CICS Java clients: these connections are always autoinstalled. See the *CICS Customization Guide*.
- Secure Sockets Layer (SSL) authentication. SSL is supported on IPIC connections between the CICS TG and CICS, and as on client-to-CICS TG connections. For information about using SSL authentication, see the *CICS RACF Security Guide*.

The EPI resource adapter
The EPI resource adapter provides a high level CCI interface to the External Presentation Interface that can be used to install terminals and run 3270-based transactions on a CICS TS server. There is no support for Automatic

Transaction Initiation (ATI). The resource adapter can be deployed into a J2EE application server to allow J2EE enterprise applications to access CICS. When the JCA is used, connection pooling, security, and transaction context are managed by the J2EE application server instead of the application.

Method requests for enterprise beans and CORBA stateless objects
These are used to invoke the functions of Java objects (enterprise beans and CORBA stateless objects) in a CICS EJB server. For more information, see Chapter 18, “What are enterprise beans?,” on page 213.

Accessing CICS programs from Java programs *inside* CICS

A Java client program running within CICS can access non-Java CICS programs by using:

JCICS
A CICS Java program or enterprise bean can use the JCICS classes to link to a CICS server program. The server program can be written in any of the CICS-supported languages and be either local or remote. It must use a suitable communications area and must not do any 3270-based terminal input/output.

The Java programmer requires a detailed knowledge of CICS.

For information about JCICS, see Chapter 6, “Java programming using JCICS,” on page 17.

The CCI Connector for CICS TS
The CCI Connector for CICS TS enables a CICS Java program or enterprise bean to link to a CICS server program. As with JCICS, the server program can be written in any of the CICS-supported languages and be either local or remote. It must use a suitable communications area and not do any 3270-based terminal input/output. Note that, unlike the ECI resource adapter used by external Java client programs, the CCI Connector for CICS TS does not support:
- Containers
- SSL security

The Java programmer does not require a detailed knowledge of CICS.

For more information, see Chapter 25, “The CCI Connector for CICS TS,” on page 319.

Using the CICS resource adapters: examples

In the example scenario shown in Figure 2 on page 50, a Java application uses the ECI resource adapter to link to a CICS TS z/OS server program. This is an example of a **3-tier configuration**: that is, the connection between the client application and the CICS TS server is via an intermediate system. Because the client application is not running on the same host as the CICS TG, the CICS Transaction Gateway daemon is used to listen for and communicate with the client. On z/OS, the CICS TG uses the External CICS Interface (EXCI) or the IPIC driver to pass requests to CICS. (To CICS, these appear to be ECI calls.)

The picture also shows a Java servlet. It too uses the ECI resource adapter to connect to a CICS TS z/OS server program. This is an example of a **2-tier configuration**: that is, there is a direct connection, via the ECI Adapter, between the client application and the CICS TS server. Because the servlet is running on the same host as the CICS TG, it uses the “local” protocol to communicate with the latter.

The CICS Transaction Gateway on z/OS supports the external call interface but not the external presentation interface. Thus, the ECI and the ECI resource adapter are supported but not the EPI or the EPI resource adapter. Only Java client programs are supported. ECI calls can be made over EXCI or IPIC connections to the CICS server.

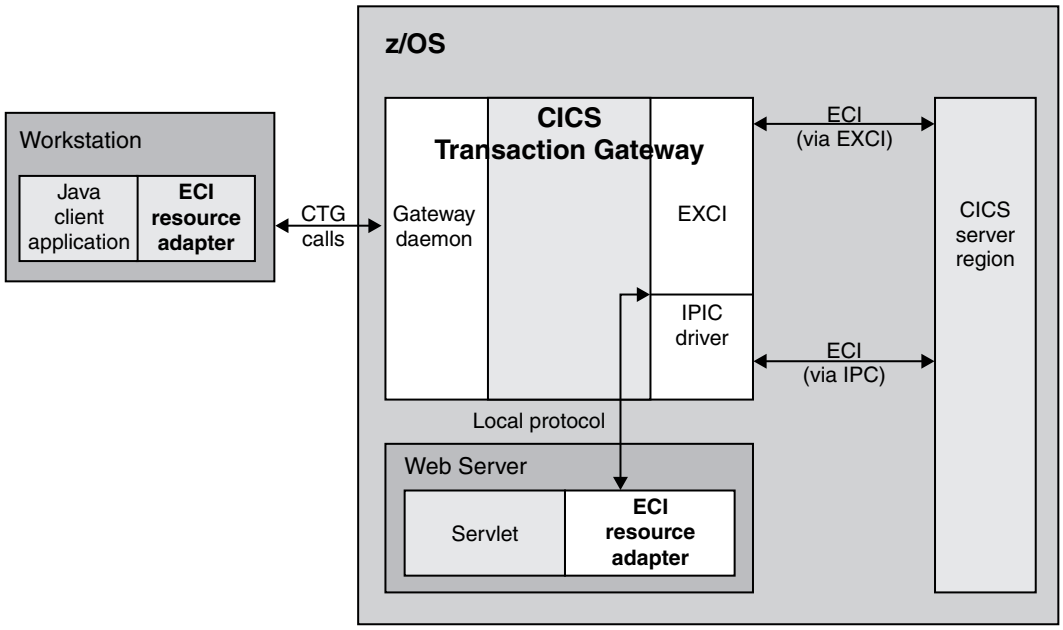

Figure 2. Java clients connect to a CICS server program from outside CICS. A Java application uses the ECI resource adapter to link to a CICS TS z/OS server program. This is an example of a 3-tier configuration. Because the client application is not running on the same host as the CICS TG, the CICS Transaction Gateway daemon is used to listen for and communicate with the client. The CICS TG uses EXCI or the IPIC driver to pass requests to CICS. (To CICS, these appear to be ECI calls. If EXCI is used, the CICS server region must be on the same z/OS operating system, or Parallel Sysplex, as the CICS TG.)

A variation is shown in the next figure. In this example, the CICS Transaction Gateway runs on a Windows machine. The CICS Transaction Gateways on Windows and Linux support the full CICS Transaction Gateway API (the ECI and ECI resource adapter, and the EPI and EPI resource adapter) is supported. In other words, the Java client can access 3270-based CICS programs, as well as CICS programs that use a suitable communications area or containers.

ECI calls can be made over APPC, TCP62, ECI over TCP/IP, or IPIC connections to the CICS server. EPI calls are supported only on APPC connections.

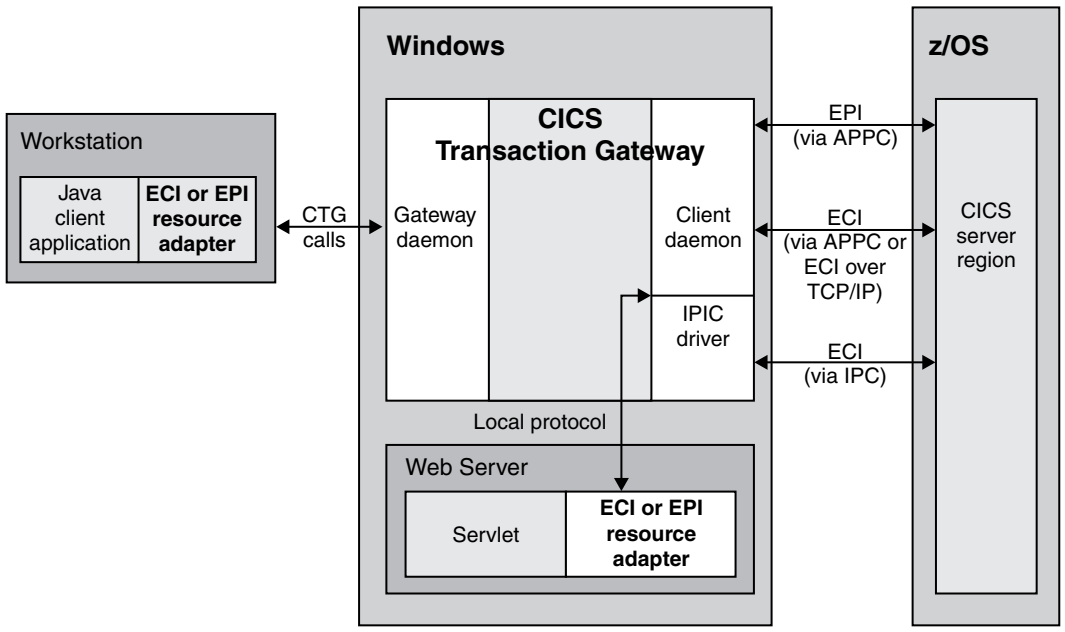

Figure 3. Java clients connect to a CICS TS z/OS server program from outside CICS. In this example, the CICS Transaction Gateway runs on a Windows machine. Both ECI and EPI calls are supported.

To use CICS programs as servers in this way, the Java programmer requires some knowledge of CICS programming.

Setting up a Java client to use the ECI resource adapter

For guidance about how to program the CICS resource adapters, see the *CICS Transaction Gateway Programming Guide*. For definitive reference information about the resource adapter classes, see the *CICS Transaction Gateway Programming Reference*. This topic covers only those aspects of the Adapter that are specific to CICS TS for z/OS, Version 4.1.

`Connection protocol`
When using the ECI resource adapter to connect to a CICS TS for z/OS, Version 3.2 or later region, you are recommended to use the IPIC protocol because, unlike EXCI, APPC, and ECI over TCP/IP, it supports containers and SSL authentication. The value to specify is `ISC`.

`ECI resource adapter properties`

`ServerName (ECIConnectionFactory:ServerName)`
Specifies the target CICS server.

For IPIC (IP) connections, the format is: `tcp://cicserver.ibm.com:portnumber` or `ssl://cicserver.ibm.com:portnumber`

`ConnectionURL`
Specifies the URL of the target CICS server.

In **3-tier mode** (that is, when the connection between the client application and the CICS TS server is via an intermediate system) use the format: `tcp://cicserver.ibm.com` or `ssl://cicserver.ibm.com`

Specify the port number on PortNumber.

In **2-tier mode** (that is, when the client and resource adapter are in the same host operating system as the CICS TG, so that there is a direct IPCONN, via the ECI Adapter, between the client application and the CICS TS server) use the format: `local://`.

System initialization parameters

IP interconnectivity requires CICS TCP/IP services to be activated. To activate them at CICS startup, specify TCPIP=YES as a system initialization parameter. (The default value of the TCPIP parameter is NO.)

For reference information about the TCPIP system initialization parameter, see the *CICS System Definition Guide*.

Defining CICS connections for Java clients

To allow a Java client to connect to a CICS server, you should use IPIC connections with the ECI resource adapter.

To do this, install a TCPIPSERVICE with PROTOCOL(IPIC) in the CICS server region. The client uses this TCPIPSERVICE to call into the CICS server. CICS then uses the autoinstall process to create an IPCONN and complete the connection to the client.

For information about the attributes of IPIC connections, see the *CICS Resource Definition Guide*.

See too the *CICS Customization Guide*.

Chapter 10. Using the JCICS sample programs

CICS provides sample programs that show you how to use the JCICS classes, and how to combine Java programs with CICS programs written in other languages.

The Java source files, together with makefiles to build the sample programs, are installed in z/OS UNIX System Services.

The web sample is run using a Web browser. The other sample programs are run by entering a transaction name at a 3270 CICS screen. The following samples are provided:

"Hello World" samples
Two simple "Hello World" programs are supplied:

- The JHE1 transaction runs a sample that uses only Java services
- The JHE2 transaction runs a sample that uses JCICS. The JCICS sample demonstrates the use of the JCICS `TerminalPrincipalFacility` class.

Program Control samples
There are two Program Control samples: the first demonstrates how to use a COMMAREA and the second how to use a channel.

COMMAREA sample
This sample demonstrates the use of the JCICS `Program` class to pass a communications area (COMMAREA) to another program:

1. A transaction, JPC1, invokes a Java class that constructs a COMMAREA and links to a C program (DFH$LCCA).
2. DFH$LCCA processes the COMMAREA, updates it, and returns.
3. The Java program checks the data in the COMMAREA and schedules a pseudoconversational transaction to be started, passing the started transaction the changed data in its COMMAREA.
4. The started transaction executes another Java class that reads the COMMAREA and validates it again.

This sample also shows you how to convert ASCII characters in the Java code to and from the equivalent EBCDIC used by the native CICS program.

Channel sample
This sample demonstrates the use of the JCICS `Program` class to pass a channel to another program:

1. A transaction, JPC3, invokes a Java class that constructs a `Channel` object with two `Containers`, and links to a C program (DFH$LCCC).
2. DFH$LCCC processes the containers, creates a new response container, and returns.
3. The Java program checks the data in the response container and schedules a pseudoconversational transaction to be started, passing the `Channel` object to the started transaction.
4. The started transaction executes another Java class that browses the `Channel` using a `ContainerIterator` object, and displays the name of each container it finds.

TDQ transient data sample
This sample shows you how to use the JCICS TDQ class. It consists of a single transaction, JTD1, that invokes a single Java class, TDQ.ClassOne. TDQ.ClassOne writes some data to a transient data queue, reads it, and then deletes the queue.

TSQ temporary storage sample
This sample shows you how to use the JCICS TSQ class. It consists of a single transaction, JTS1, that invokes a single Java class, TSQ.ClassOne, and uses an auxiliary temporary storage queue.

This sample also shows you how to build a class as a dynamic link library (DLL) which can be shared with other Java programs.

Web sample
This sample shows you how to use the JCICS Web and Document classes. You invoke this sample application from a suitable Web browser. It obtains information about the inbound client request, the HTTP headers and the TCP/IP characteristics of the transaction. This information is written to the standard output stream System.out and inserted into a response document. Information about the document is also obtained and written to System.out and inserted into the response document. The response document is then sent to the client.

Building the JCICS sample programs

The Java source and makefiles are stored in the z/OS UNIX System Services file system during CICS installation. To build the samples in the z/OS UNIX System Services environment, you must define three environment variables and install a group.

About this task

You can define the environment variables in the profile for z/OS UNIX System Services, using the **export** command, or you can enter the export command manually when z/OS UNIX System Services is running.

Procedure

1. *PATH* is the z/OS UNIX System Services search path. Define the *PATH* environment variable by adding:
   ```
   /usr/lpp/java/J6.0/bin
   ```
   where *java/J6.0* is your install location for the IBM SDK for z/OS, Java Technology Edition on z/OS UNIX. This is the path for the Java executables. You can use the export command to add the path as follows:
   ```
   export PATH=/usr/lpp/java/J6.0/bin:$PATH
   ```
2. *CICS_HOME* is the install directory for CICS Transaction Server for z/OS files in z/OS UNIX System Services. Define the *CICS_HOME* environment variable as follows:
   ```
   /usr/lpp/cicsts/cicsts41
   ```
   where *cicsts41* is defined by the USSDIR installation parameter when you installed CICS TS (cicsts41 is the default). You can use the export command to set the directory prefix as follows:
   ```
   export CICS_HOME=/usr/lpp/cicsts/cicsts41
   ```
   The $CICS_HOME/samples/dfjcics directory contains the makefiles.

The `$CICS_HOME/samples/dfjcics/examples` directory contains the Java source.

3. JAVA_HOME specifies the path to the IBM SDK for z/OS, Java Technology Edition subdirectories. Define the JAVA_HOME environment variable as follows:

   `/usr/lpp/java/J6.0/`

   where *java/J6.0/* is your install location for the IBM SDK for z/OS, Java Technology Edition on z/OS UNIX.
4. Install the group DFH$JVM in order to run the samples. CICS resource definitions for all the sample programs and transactions are supplied in this group.
5. If you want to run the Web sample program, which is invoked via a browser, you need to follow the instructions in Configuring CICS Web support base components, in the *CICS Internet Guide*. Use the sample programs DFH$WB1A (Assembler) or DFH$WB1C (C) to confirm that CICS Web support is configured correctly.
6. Follow the instructions in "Building the Java samples."

Related concepts

Chapter 6, "Java programming using JCICS," on page 17
You can write Java application programs that use CICS services and execute under CICS control.

"The JCICS class library" on page 17
The Java class library for CICS, JCICS, supports most of the functions of the **EXEC CICS** API commands.

Related tasks

Chapter 10, "Using the JCICS sample programs," on page 53
CICS provides sample programs that show you how to use the JCICS classes, and how to combine Java programs with CICS programs written in other languages.

"Building the Java samples"
To build the Java samples, you need write permission for the z/OS UNIX directory in which the samples are stored and for its subdirectories.

"Running the JCICS samples" on page 56
You must perform these steps to run the JCICS samples.

Related reference

"JCICS command reference" on page 21
Many of the options and services available to non-Java programs through the **EXEC CICS** API are available to Java programs through JCICS. This section shows the relationship between **EXEC CICS** commands and the equivalent JCICS function.

Building the Java samples

To build the Java samples, you need write permission for the z/OS UNIX directory in which the samples are stored and for its subdirectories.

About this task

These directories are part of the directory structure that includes the other CICS files which have been installed on z/OS UNIX. If you do not want users to have write permission for these directories, you should copy the samples directory and its subdirectories to another location on z/OS UNIX before building the samples.

If you use OMVS to perform this task, note that you might need to increase the size of your TSO region when you are using the IBM SDK for z/OS, Java Technology Edition.

Build the samples as follows:

1. Change directory to `samples/dfjcics`.
2. Type `make jvm` to build all the samples, or alternatively:

   ```
   make -f <sample_name>.mak jvm
   ```

   where `sample_name` is the name of the specific sample you want to build.

   The makefiles invoke `javac` and store the output files in the `$CICS_HOME/samples/dfjcics/examples/sample_name` z/OS UNIX directory, where `sample_name` is the name of the sample program.

The following CICS C language programs are stored in SDFHSAMP during CICS installation. They are linked by the Program Control and one of the "Hello World" Java sample programs. You need to compile and translate these supplied C programs, link them into DFHRPL or a dynamic LIBRARY concatenation, and define them to CICS as described in "Defining CICS resources."

- DFH$LCCA
- DFH$JSAM
- DFH$LCCC

Note:

1. In the names of sample programs and files described in this book, the dollar symbol ($) is used as a national currency symbol and is assumed to be assigned the EBCDIC code point X'5B'. In some countries a different currency symbol, for example the pound symbol (£), or the yen symbol (¥), is assigned the same EBCDIC code point. In these countries, the appropriate currency symbol should be used instead of the dollar symbol.
2. DFH$LCCA and DFH$JSAM are standard CICS programs that could be written in any of the CICS-supported languages. If, for example, you do not have a C compiler, you could write COBOL versions of the supplied programs and use them in place of the supplied C versions.

Defining CICS resources

Install the group DFH$JVM in order to run the samples. CICS resource definitions for all the sample programs and transactions are supplied in this group.

Running the JCICS samples

You must perform these steps to run the JCICS samples.

Before you begin

Before you run the JCICS samples, you must build them. See "Building the JCICS sample programs" on page 54.

Procedure

1. Add the string `/usr/lpp/cicsts/cicsts41/samples/dfjcics` to the standard class path in the default JVM profile DFHJVMPR, using the CLASSPATH_SUFFIX option. Where `/usr/lpp/cicsts/cicsts41` is the value of CICS_HOME.
2. Follow the appropriate procedure to run each sample:
   - "Running the Hello World samples" on page 57

Running the Hello World samples

There are two "Hello World" samples, HelloWorld, and HelloCICSWorld.

HelloWorld

This is the standard Java application that uses only Java services. It uses the following Java class:

- `HelloWorld` (PROGRAM name DFJ$JHE1)

and the following C language CICS program:

- DFH$JSAM

Note: DFH$JSAM is a standard CICS program that could be written in any of the CICS-supported languages. If, for example, you do not have a C compiler, you could write a COBOL version of DFH$JSAM and use it in place of the supplied C version. Alternatively, you could bypass DFH$JSAM altogether by changing the JHE1 TRANSACTION definition to run program DFJ$JHE1. However, if you do this bear in mind that the Java program does not write anything to the terminal; so your only indication that the application has run successfully is the message in the `stdout` file.

Run the JHE1 CICS transaction to execute the Java standard application. You should receive the following messages from JHE1:

- `"SAMPLE *COMPLETED*, SEE STOUT"`, on your terminal
- `"Hello from a regular Java application"`, in your `stdout` file

HelloCICSWorld

This is the JCICS application. It uses the following Java class:

- `HelloCICSWorld` (PROGRAM name DFJ$JHE2)

Run the JHE2 transaction to execute the JCICS application. You should receive the following message from JHE2 on your terminal:

```
Hello from a Java CICS application
```

Running the Program Control samples

You must perform these steps to run the COMMAREA sample, and the channel sample.

About this task

The COMMAREA sample

This sample uses the following Java classes:

- `ProgramControl.ClassOne` (PROGRAM name DFJ$JPC1)
- `ProgramControl.ClassTwo` (PROGRAM name DFJ$JPC2)

and the following C language program:

- DFH$LCCA

Run the JPC1 CICS transaction to execute the sample. You should receive the following messages on `Task.out` (normally your terminal):

```
Entering ProgramControlClassOne.main()
About to link to C program
Leaving ProgramControlClassOne.main()
```

If you now clear the screen, you should see:

```
Entering ProgramControlClassTwo.main()
data received correctly
Leaving ProgramControlClassTwo.main()
```

The channel sample

This sample uses the following Java classes:

- `ProgramControl.ClassThree` (PROGRAM name DFJ$JPC3)
- `ProgramControl.ClassFour` (PROGRAM name DFJ$JPC4)

and the following C language program:

- DFH$LCCC

Run the JPC3 CICS transaction to execute the sample. You should receive the following messages on `Task.out` ICS transaction to execute the sample. You should receive the following messages on `Task.out` (normally your terminal):

```
Entering ProgramControlClassThree.main()
About to link to C program
Leaving ProgramControlClassThree.main()
```

If you now clear the screen, you should see:

```
Entering ProgramControlClassFour.main()
ProgramControlClassFour invoked with Container "IntData         "
ProgramControlClassFour invoked with Container "StringData      "
ProgramControlClassFour invoked with Container "Response        "
Leaving ProgramControlClassFour.main()
```

Note that the messages that list the containers may appear in a different order from that shown above.

Note: DFH$LCCA and DFH$LCCC are standard CICS programs that could be written in any of the CICS-supported languages. If, for example, you do not have a C compiler, you could write COBOL versions of DFH$LCCA and DFH$LCCC and use them in place of the supplied C versions.

Running the TDQ sample

You must run the JTD1 CICS transaction to execute the sample.

About this task

This sample uses the following Java class:

- `TDQ.ClassOne` (PROGRAM name DFJ$JTD1)

You should receive the following messages on `Task.out`:

```
Entering examples.TDQ.ClassOne.main()
Entering writeFixedData()
Leaving writeFixedData()
Entering writeFixedData()
Leaving writeFixedData()
Entering readFixedData()
Leaving readFixedData()
Entering readFixedDataConditional()
Leaving readFixedDataConditional()
Leaving examples.TDQ.ClassOne.main()
```

Running the TSQ sample

You must run the JTS1 CICS transaction to execute the sample.

About this task

This sample uses the following Java classes:

- `TSQ.ClassOne` (PROGRAM name DFJ$JTS1)
- `TSQ.Common` (PROGRAM name DFJ$JTSC)

You should receive the following messages on `Task.out`:

```
Entering TSQ.ClassOne.main()
Entering TSQ_Common.writeFixedData()
Leaving TSQ_Common.writeFixedData()
Entering TSQ_Common.serializeObject()
Leaving TSQ_Common.serializeObject()
Entering TSQ_Common.updateFixedData()
Leaving TSQ_Common.updateFixedData()
Entering TSQ_Common.writeConditionalFixedData()
Leaving TSQ_Common.writeConditionalFixedData()
Entering TSQ_Common.updateConditionalFixedData()
Leaving TSQ_Common.updateConditionalFixedData()
Entering TSQ_Common.readFixedData()
Leaving TSQ_Common.readFixedData()
Entering TSQ_Common.deserializeObject()
Leaving TSQ_Common.deserializeObject()
Entering TSQ_Common.readFixedConditionalData()
Number of items returned is 3
Leaving TSQ_Common.readFixedConditionalData()
Entering TSQ_Common.deleteQueue()
Leaving TSQ_Common.deleteQueue()
Leaving TSQ.ClassOne.main()
```

Running the web sample

This sample uses the Java class: `Web.Sample1 (PROGRAM name DFJ$JWB1)`

About this task

To invoke this sample, start your Web browser and enter a URL that connects to CICS Web support with the absolute path `/CICS/CWBA/DFJ$JWB1`

The browser should display the following response document::

```
Web Sample1

Inbound Client Request Information:

Method: GET

Version: HTTP/1.1

Path: /cics/cwba/jcicxsa1

Request Type: HTTPYES

Query String: null

HTTP headers:

Value for HTTP header User-Agent is 'Mozilla/4.75 €en€ (WinNT; U)'

Browse of HTTP Headers started
```

```
Name: Host Value: winmvs2d.hursley.ibm.com:27361

Name: Connection Value: Keep-Alive, TE

Name: Accept Value: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png,
      */*

Name: Accept-Encoding Value: gzip

Name: Accept-Language Value: en

Name: Accept-Charset Value: iso-8859-1,*,utf-8

Name: Cookie Value: PBC_NLSP=en_US

Name: TE Value: chunked

Name: Via Value: HTTP/1.0 sp15ce18.hursley.ibm.com (IBM-PROXY-WTE-US)

Name: User-Agent Value: Mozilla/4.75 €en€ (WinNT; U)

Browse of HTTP Headers completed

TCPIP Information:

Client Name: sp15ce18.hursley.ibm.com

Server Name: winmvs2d.hursley.ibm.com

Client Address: 9.20.136.28

ClientAddrNu: 9.20.136.28

Server Address: 9.20.101.8

ServerAddrNu: 9.20.101.8

Clientauth: NO

SSL: NO

TcpipService: HTTPNSSL

PortNumber: 27361

Document Information:

Doctoken: 33 92 112 0 0 0 0 1 64 64 64 64 64 64 64 64

Docsize: 2762
```

The sample also writes information messages to standard output stream `System.out` and error messages to the standard output stream `System.err`.

Here is an example of the output written to the `System.out` output stream:

```
Sample1 started
Method: GET (3)
Version: HTTP/1.1 (8)
Path: /cics/cwba/jcicxsa1 (19)
Request Type: HTTPYES
Value for HTTP header User-Agent is 'Mozilla/4.75  en  (WinNT; U)'
HTTP headers:
Name: Host (4)
Value: winmvs2d.hursley.ibm.com:27361 (30)
Name: Connection (10)
Value: Keep-Alive, TE (14)
```

```
Name: Accept (6)
Value: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png, */* (67)
Name: Accept-Encoding (15)
Value: gzip (4)
Name: Accept-Language (15)
Value: en (2)
Name: Accept-Charset (14)
Value: iso-8859-1,*,utf-8 (18)
Name: Cookie (6)
Value: PBC_NLSP=en_US (14)
Name: TE (2)
Value: chunked (7)
Name: Via (3)
Value: HTTP/1.0 sp15ce18.hursley.ibm.com (IBM-PROXY-WTE-US) (52)
Name: User-Agent (10)
Value: Mozilla/4.75  en  (WinNT; U) (28)
Client Name: sp15ce18.hursley.ibm.com (24)
Server Name: winmvs2d.hursley.ibm.com (24)
Client Address: 9.20.136.28 (11)
ClientAddrNu: 9.20.136.28
Server Address: 9.20.101.8 (10)
ServerAddrNu: 9.20.101.8
Clientauth: NO
SSL: NO
TcpipService: HTTPNSSL
PortNumber: 27361
Doctoken: Doctoken: 33 92 112 0 0 0 0 1 64 64 64 64 64 64 64 64
Docsize: 2762
Sample1 complete
```

Part 3. Setting up Java support and JVMs

This section tells you what you need to know to set up Java support and Java Virtual Machines (JVMs) in CICS.

Chapter 11. Setting up Java support

Perform the basic setup tasks, and verify that Java support works in your CICS region, using the CICS-supplied sample JVM profiles, JVM properties files, and Java sample programs.

Procedure

1. Verify that your Java components are installed correctly using the supplied checklist in Verifying your Java components installation, in the *CICS Transaction Server for z/OS Installation Guide*.
2. Set the **JVMPROFILEDIR** system initialization parameter to a new location in z/OS UNIX where you want to store the JVM profiles used by the CICS region, and copy the supplied sample JVM profiles to this location. "Setting the location for the JVM profiles" tells you how to do this. If you intend to use JVM profiles which you set up in a previous CICS release, you can copy them into this location later on.
3. Give your CICS region permission to access the resources held in z/OS UNIX, including your JVM profiles and JVM properties files, and other directories and files needed to create JVMs. "Giving CICS regions permission to access z/OS UNIX directories and files" on page 66 tells you how to do this.
4. Run the Java sample programs to verify that Java support works in your region. "Checking your Java support setup using the sample programs" on page 70 contains a task list that describes how to set up and run the supplied sample programs.

What to do next

When you have run the supplied Java sample programs, read through the Chapter 12, "Understanding JVMs," on page 73 section for conceptual information on how to use JVMs in CICS. Then read the section Chapter 13, "Setting up Java support and using JVMs," on page 93 to find out how to create and customize your JVM profiles and properties files, manage the shared class cache and perform tasks such as monitoring and debugging your Java applications.

Setting the location for the JVM profiles

CICS looks for JVM profiles in the z/OS UNIX directory that is specified by the **JVMPROFILEDIR** system initialization parameter, and loads the JVM profiles from this directory. You need to set **JVMPROFILEDIR** to a new location where you want to store the JVM profiles used by the CICS region, and copy the supplied sample JVM profiles into this directory so that you can use them to verify your installation.

About this task

The CICS-supplied sample JVM profiles are tailored for your system during the CICS installation process, so you can use them right away to verify your installation. If you are setting up Java support in CICS for the first time, you can customize your copies of these files to use for your JVMs when you start to run your own Java applications.

If you already have JVM profiles which you set up in a previous CICS release, you can copy these to use for your applications later on instead of the CICS-supplied samples. The settings that are suitable for use in JVM profiles can change from one CICS release to another, so for ease of problem determination, use the CICS-supplied samples to verify your installation, and then switch to using copies of your existing JVM profiles.

Procedure

1. Set the **JVMPROFILEDIR** system initialization parameter to the location on z/OS UNIX where you want to store the JVM profiles used by the CICS region. The value that you specify can be up to 240 characters long.

   The supplied setting for the **JVMPROFILEDIR** system initialization parameter is `/usr/lpp/cicsts/cicsts41/JVMProfiles`, which is the installation location for the sample JVM profiles. This directory is not a safe place to store your customized JVM profiles, because you risk losing your changes if the sample JVM profiles are overwritten when program maintenance is applied. So you must always change **JVMPROFILEDIR** to specify a different z/OS UNIX directory where you can store your JVM profiles. Choose a directory where you can give appropriate permissions to the users who must customize the JVM profiles.
2. Copy the four CICS-supplied sample JVM profiles, DFHJVMPR and, DFHJVMCD. from their install location, into the z/OS UNIX directory that you specified for the **JVMPROFILEDIR** system initialization parameter. The DFHJVMCD profile, although not strictly a sample JVM profile, is required for internal CICS Java transactions, and for managing the Shared Class Cache.

   When you install CICS, the CICS-supplied sample JVM profiles are placed in the `/usr/lpp/cicsts/cicsts41/JVMProfiles directory`. The `/usr/lpp/cicsts/cicsts41` directory is the installation directory for CICS files on z/OS UNIX. This directory is specified by the **USSDIR** parameter in the DFHISTAR installation job, which is passed to the *uss_path* variable used by the DFHIJVMJ job, which creates the sample profiles.

Giving CICS regions permission to access z/OS UNIX directories and files

CICS requires access to directories and files in z/OS UNIX. During installation, each of your CICS regions was assigned a z/OS UNIX user identifier (UID), and they were connected to a RACF group which was assigned a z/OS UNIX group identifier (GID). The UID and GID are used to grant permission for the CICS region to access the directories and files in z/OS UNIX.

About this task

Because your CICS regions have a UID, and their connect group (the RACF group) has a GID, z/OS UNIX System Services treats each CICS region as a UNIX user. There are four ways to grant a user permissions to access z/OS UNIX directories and files.

- You could set the “other” permissions for the directory or file so that every user has access. This would give access to all the CICS regions, but it would also give access to every other z/OS UNIX user, so this option might not be suitable for use in your production environment.
- You could make the user the owner of the directory or file, with the appropriate owner permissions. This option can only be used for one user (so one CICS region) at a time. This is a good solution to use for the home directory for each CICS region, but it is not such a good solution to use for directories and files

that are needed by more than one CICS region. If you chose during installation to assign the same UID to all your CICS regions, you can make that UID the owner of the directories and files. However, there are a number of disadvantages associated with the sharing of UIDs, so it is not normally recommended.

- You could give the appropriate group permissions for the directory or file, to the RACF group which was assigned a GID during installation, to which your CICS regions connect. This might often be the safest option for a production environment, so this topic explains how to do it. If this method is not the most suitable for your environment, then you might prefer to give CICS access to the files using owner permissions or "other" permissions, or perhaps a combination of the three types of permission, depending on the level of security that you require for each type of directory or file.
- You could use access control lists (ACLs) to control access to files and directories by individual UIDs and GIDs. With ACLs, you can give more than one group permissions for directories or files on z/OS UNIX, so you do not need to ensure that all your CICS regions connect to the same RACF group. ACLs are created and checked by RACF, so if you are using a different security product, check its documentation to see if ACLs are supported. For more information about using ACLs, see *z/OS UNIX System Services Planning*, GA22–7800.

Procedure

1. Identify the directories and files in z/OS UNIX to which your CICS regions require access in connection with the CICS facility that you are setting up. The listing at the end of this topic describes the resources to which CICS needs access, and the permissions that you need to give in each case.
2. Ensure that you are either a superuser on z/OS UNIX, or the owner of the directories and files. For directories and files supplied by CICS or by the IBM JVM, the owner is initially set as the UID of the system programmer who installs the product. Also, when you are giving CICS access using group permissions, the owner of the directories and files must be connected to the RACF group that you chose for your CICS regions to access z/OS UNIX, which was assigned a GID during installation. The owner could have that RACF group as their default group (DLFTGRP) or be connected to it as one of their supplementary groups.
3. Display each of the directories and files. Go to the directory where you want to start, and issue the following UNIX command:

   ```
   ls -la
   ```

   For example, if this command is issued in the z/OS UNIX System Services shell environment when the current directory is the home directory of CICSHT##, a list such as the following is displayed:

   ```
   /u/cicsht##:>ls -la
   total 256
   drwxr-xr-x   2 CICSHT## CICSTS41     8192 Mar 15  2008 .
   drwx------   4 CICSHT## CICSTS41     8192 Jul  4 16:14 ..
   -rw-------   1 CICSHT## CICSTS41     2976 Dec  5  2008 Snap0001.trc
   -rw-r--r--   1 CICSHT## CICSTS41     1626 Jul 16 11:15 dfhjvmerr
   -rw-r--r--   1 CICSHT## CICSTS41        0 Mar 15  2008 dfhjvmin
   -rw-r--r--   1 CICSHT## CICSTS41      458 Oct  9 14:28 dfhjvmout
   /u/cicsht##:>
   ```
4. Assuming that you are using the group permissions to give access, check that the group permissions for each of the directories and files give the level of access that CICS requires for the resource. Permissions are indicated, in three sets, by the letters `r`, `w`, `x` and `-`. These represent **read**, **write**, **execute**, and **none** respectively, and are shown in the left-hand column of the display,

starting with the second character. The first set are the owner permissions, the second the group permissions, and the third "other" permissions. In the example above, for the last file `event.log`, the owner has **read** and **write** permissions, but the group and all others have only **read** permissions.

5. If you need to change the group permissions for a resource, use the UNIX command `chmod`. *z/OS UNIX System Services Command Reference*, SA22-7802, and *z/OS UNIX System Services User's Guide*, SA22-7801, has information about using this command. The following examples should help.

   ```
   chmod -R g=rwx directory
   ```

   Sets the group permissions for the named directory and its subdirectories and files to **read**, **write** and **execute** (`-R` applies permissions recursively to all subdirectories and files).

   ```
   chmod g+rx filename
   ```

   Sets the group permissions for the named file to **read** and **execute**.

   ```
   chmod g-w filename filename
   ```

   Sets the group **write** permission off for the two named files. In all these examples, **g** is for group permissions. If you need to correct other permissions, **u** is for user (owner) permissions, and **o** is for other permissions.

6. Assign the group permissions for each resource to the RACF group that you chose for your CICS regions to access z/OS UNIX, which was assigned a GID during installation. You need to do this for each directory and its subdirectories, and for the files in them. To do this, issue the UNIX command

   ```
   chgrp -R GID directory
   ```

   where *GID* is the numeric GID of the RACF group, and *directory* is the full path of a directory where you want to give the CICS regions permissions. For example, to assign the group permissions for the `/usr/lpp/cicsts/cicsts41` directory, use the command

   ```
   chgrp -R GID /usr/lpp/cicsts/cicsts41
   ```

   The `-R` in the command means that the group is changed for not only the named directory, but also all the subdirectories, and all the files in the directory and subdirectories. Because your CICS region user IDs are connected to the RACF group, the CICS regions now have the appropriate permissions for all these directories and files.

7. When you make changes to the CICS facility that you are setting up, such as moving files or creating new files, remember to repeat this procedure to ensure that your CICS regions have permission to access the new or moved files.

What to do next

If you need more general information about the UNIX facilities that you can use to control access to z/OS UNIX files and directories, see *z/OS UNIX System Services Planning*, GA22-7800.

Java resources in z/OS UNIX

CICS requires access to these directories and files in z/OS UNIX for Java support.

Resources needed to create JVMs

The directories and files that every CICS region needs to create JVMs are set up when you install CICS, and when you install the IBM SDK for z/OS, Java Technology Edition. These directories and files are:

- Most of the files in the `/usr/lpp/cicsts/cicsts41` directory and its subdirectories. The `/usr/lpp/cicsts/cicsts41` directory is the install directory for CICS files on z/OS UNIX. This directory is specified by the USSDIR parameter in the DFHISTAR install job, which is passed to the `uss_path` variable used by the DFHIJVMJ job which creates the sample JVM profiles. The files in this directory and its subdirectories include the supplied sample JVM profiles and the CICS-supplied JAR files such as `dfjcics.jar` and `dfjcsi.jar`.
- Some of the files in the directories that contain the IBM JVM code.
  - For The default paths for the directories are `/usr/lpp/java/J6.0/bin` and `/usr/lpp/java/J6.0/bin/j9vm`. The *java/J6.0* directory names are the defaults when you install the SDK.

Each CICS region requires **read and execute** access to these directories and files.

Working directory for each CICS region

The working directories that you have specified for input, output and messages from the JVMs in each individual CICS region are specified on the WORK_DIR option in the JVM profiles used in the CICS region, and also in any Java class that you have specified on the USEROUTPUTCLASS option to redirect stdout and stderr output from JVMs.

The default working directories are as follows:

- For the WORK_DIR option, the default working directory as specified in the supplied sample JVM profiles is the home directory of the CICS region user ID (that is, the directory `/u/`*`CICS region userid`*), which you should have created during installation. If the CICS region user ID does not have this home directory, `/tmp` is used by default as the working directory.
- For the USEROUTPUTCLASS option, if you are using the CICS-supplied sample class com.ibm.cics.server.SJMergedStream, the default working directory is the directory specified on the WORK_DIR option in the JVM profile. If you are using the alternative CICS-supplied sample class com.ibm.cics.server.SJTaskStream, the default working directories are `/work_dir/applid/stdout` and `/work_dir/applid/stderr`, where `work_dir` is the directory specified on the WORK_DIR option in the JVM profile, and `applid` is the applid of the CICS region. The USEROUTPUTCLASS option is **not** active in the supplied sample JVM profiles.

If you have specified a different directory on the WORK_DIR option, or used the USEROUTPUTCLASS option to specify a Java class, in any of the JVM profiles in your CICS region, find out the names of the z/OS UNIX directories that are used by the WORK_DIR option or the Java class.

Each CICS region requires **read**, **write** and **execute** access to the z/OS UNIX directories that you have identified as being used as a working directory or for output from JVMs in that region. If a directory is unique to a CICS region (for example, if it is based on a unique home directory that you created for the region, or if it was created using the special symbol &APPLID; and so includes the CICS region's unique applid), then you can make the CICS region's UID the owner of the directory and its subdirectories, and use the owner permissions to give the

appropriate permissions to the CICS region. However, if more than one CICS region uses a particular directory, then you need to use group permissions so that all the CICS regions have access to the directory.

Your chosen directory for the JVM profiles

CICS requires access to the directory where you chose to store the JVM profiles used by each CICS region, as described in "Setting the location for the JVM profiles" on page 65, and the files that you have placed there. This directory is specified by the **JVMPROFILEDIR** system initialization parameter. The CICS region requires **read** and **execute** access to the directory and files.

Further directories and files

Other directories and files that you have told a CICS region to use in the process of creating JVMs, or in support of CORBA applications and enterprise beans, need the correct permissions applied too.

If you are starting to set up JVMs in a CICS region for the first time, you probably do not have any other directories and files at this stage. You will have other directories and files if:

- You add directory paths to the CLASSPATH_PREFIX or CLASSPATH_SUFFIX option in a JVM profile, so that the JVM will search those directories for application classes.
- You add directory paths to the LIBPATH_PREFIX or LIBPATH_SUFFIX, option in a JVM profile, which specify directories for native C dynamic link library (DLL) files that are used by the JVM.
- You create your own JVM profiles or JVM properties files. (You can use the **EXEC CICS INQUIRE JVMPROFILE** command to find the z/OS UNIX directory that contains a JVM profile, provided that the JVM profile has been used during the lifetime of the CICS region. The z/OS UNIX directory for a JVM properties file is specified by the JVMPROPS option on the JVM profiles that reference it.)
- You move any of the files that every CICS region needs to create JVMs, that is, the files in the `/usr/lpp/cicsts/cicsts41` directory, or the directories installed with the IBM SDK for z/OS that contain the IBM JVM code.
- You set up a shelf directory or a deployed JAR file directory (also known as a pickup directory) to support CORBA applications or enterprise beans.

Each CICS region requires **read** and **execute** access to all the z/OS UNIX directories and files that you have identified in this category.

Checking your Java support setup using the sample programs

Run the "Hello World" and "Hello CICS World" CICS-supplied sample programs to verify that Java support has been successfully installed and set up in your CICS region.

Before you begin

Before running the sample programs, make sure you have completed the other setup tasks described in Chapter 11, "Setting up Java support," on page 65.

About this task

To set up and run the supplied sample programs:

Procedure

1. Build the sample programs.
   a. Define the environment variables *PATH*, *CICS_HOME* and *JAVA_HOME* in the profile for z/OS UNIX System Services. "Building the JCICS sample programs" on page 54 tells you what to define for each variable.
   b. Install the group DFH$JVM in order to run the samples.
   c. Build the Java samples, as described in "Building the Java samples" on page 55.
2. Add the string `/usr/lpp/cicsts/cicsts41/samples/dfjcics` to the standard class path in the default JVM profile DFHJVMPR, using the CLASSPATH_SUFFIX option. Make sure you use the copy of DFHJVMPR in the z/OS UNIX directory that you specified for the **JVMPROFILEDIR** system initialization parameter. You will need to uncomment the CLASSPATH_SUFFIX option.
3. Run the Hello World samples using the steps outlined in "Running the Hello World samples" on page 57.

Chapter 12. Understanding JVMs

CICS provides the support you need to run a Java program in a Java Virtual Machine (JVM) executing under the control of a CICS region. CICS support for JVMs allows you to run CICS application programs written in the Java language and compiled to bytecode by any standard Java compiler.

CICS TS 4.1 supports the JVM provided by the IBM SDK for z/OS, Java Technology Edition, Version 6. 31-bit and 64-bit versions of the IBM SDK for z/OS, Java Technology Edition are available. CICS TS 4.1 supports only the 31-bit version. You can find more information about Java on the z/OS platform and download a suitable version of the SDK at http://www.ibm.com/servers/eserver/zseries/software/java/.

When you run Java applications under CICS, CICS owns and manages a number of JVMs. Each JVM is used by only one Java program at a time, so Java programs running concurrently are isolated from each other. When each Java program has finished using its JVM, the JVM can be reused by a subsequent program.

JVMs in a CICS region have different characteristics depending on the options specified in their JVM profiles, and they can also be in different execution keys. Each Java program must use the appropriate type of JVM, as specified in the PROGRAM resource definition for the program.

"The structure of a JVM" tells you what you need to know about the structure of a JVM in order to use JVMs with CICS.

CICS performs the following management tasks relating to JVMs:

- CICS manages the pool of JVMs in the CICS region, starting and terminating JVMs as needed. This process is described in "How CICS manages JVMs in the JVM pool" on page 79.
- CICS allocates JVMs to applications that need to run a Java program. This process is described in "How CICS allocates JVMs to applications" on page 82.
- Most JVMs can be reused once an application has finished using them to run a Java program. "How JVMs are reused" on page 88 explains how this happens.
- CICS supports a shared class cache so that some of the JVMs in the CICS region can share commonly-used class files and compiled classes. CICS also provides an interface so that you can manage the shared class cache. The shared class cache enables faster JVM startup and reduces the cost of class loading. "The shared class cache" on page 90 describes this.

Chapter 11, "Setting up Java support," on page 65 tells you how to set up and use JVMs in your CICS system.

The structure of a JVM

There are several things you should know about the structure of a JVM in order to use JVMs with CICS.

For further information about Version 6 of the IBM SDK for z/OS, Java Technology Edition, see the *IBM 31-bit and 64-bit SDKs for z/OS, Java Technology Edition, Version*

6 SDK and Runtime Environment User Guide. The document is available to download from www.ibm.com/servers/eserver/zseries/software/java/javaintr.html.

Classes and class paths in JVMs

Three types of classes and native libraries are used by a JVM running under CICS.

The types of classes and native libraries are as follows:

1. The z/OS JVM code, which provides the base services in the JVM. These classes are *system classes* and *standard extension classes*, which are known collectively as *primordial classes*.
2. Native C dynamic link library (DLL) files that are used by the JVM. These files have the extension .so in z/OS UNIX. Some libraries are necessary for the JVM to run, and additional native libraries might be loaded by application code or services. For example, the additional native libraries might include the DLL files needed to use the DB2 JDBC drivers.
3. The Java classes for the applications that run in the JVM. These classes are known as *application classes*. This group includes classes that are part of user-written applications. It also includes some classes supplied by IBM or by another vendor to provide services that access resources, such as the JCICS interfaces classes, JDBC, and JNDI, which are not included in the standard JVM setup for CICS. When application classes have been loaded, they are kept across JVM reuses so that they can be used by other transactions.

The JVM understands the purpose of each of these items and determines how the class or native library is loaded by the JVM and where it is stored.

The class paths for a JVM are defined by options in the JVM profile and in the JVM properties file that the JVM profile references.

In the continuous JVM, the class paths on which classes or native libraries can be included are as follows:

1. The *library path* is for all the native C dynamic link library (DLL) files that are used by the JVM, including the files required to run the JVM and additional native libraries loaded by application code or services. Only one copy of each DLL file is loaded, and all the JVMs share it, but each JVM has its own copy of the static data area for the DLL.

   The base library path for the JVM is built automatically using the directories specified by the **CICS_HOME** and **JAVA_HOME** options in the JVM profile. The base library path is not visible in the JVM profile. It includes all the DLL files required to run the JVM and the native libraries used by CICS. You can extend the library path using the **LIBPATH_SUFFIX** option or the **LIBPATH_PREFIX** option. **LIBPATH_SUFFIX** adds items to the end of the library path, after the IBM-supplied libraries, which is the normal placement. **LIBPATH_PREFIX** adds items to the beginning, which are loaded in place of the IBM-supplied libraries if they have the same name. You might have to do this for problem determination purposes.

   Any DLL files that you include on the library path for use by your applications should be compiled and linked with the XPLink option for optimum performance. The DLL files supplied on the base library path, and the DLL files used by services such as the DB2 JDBC drivers, are built with the XPLink option.
2.

The *standard class path* is for all application classes that are used by the applications that run in the JVM. All Java .class and .jar files are placed on the *standard class path*. You can add classes to the standard class path using the **CLASSPATH_SUFFIX** option in the JVM profile or the **CLASSPATH_PREFIX** option.

CICS also builds a base class path for the JVM automatically, using the `/lib` subdirectories of the directories specified by the **CICS_HOME** and **JAVA_HOME** options in the JVM profile. This class path contains the JAR files supplied by CICS and by the JVM. It is not visible in the JVM profile.

You do not have to include the system classes and standard extension classes (the primordial classes) on a class path, because they are already included on the boot class path in the JVM.

Enterprise beans are a special case. You do not have to add the deployed JAR files (DJARs) for your enterprise beans to the class path. CICS manages the loading of the classes included in these files with the DJAR definitions. However, if your enterprise beans use any classes, such as classes for utilities, that are not included in the deployed JAR file, you do have to include these classes on the class path that will be used by the JVM for the request processor program.

"Adding application classes to the class paths for a JVM" on page 145 tells you how to add native libraries and application classes to the different class paths.

Changes to class paths in JVM profiles: middleware classes

In a continuous JVM, you now place the classes formerly treated as middleware classes on the same class path as user application classes. You specified these classes on the trusted middleware class path options TMPREFIX and TMSUFFIX in the JVM profile.

To help you to upgrade, the trusted middleware class path options TMPREFIX and TMSUFFIX are still accepted, but CICS issues a warning message when they are used.

When you are creating, changing, or upgrading JVM profiles, place the classes formerly treated as middleware classes on the standard class path. The standard class path is defined by the CLASSPATH_SUFFIX option in the JVM profile for the JVM where the application will run. When you have placed the classes on the standard class path, remove the TMPREFIX and TMSUFFIX options from your JVM profiles.

Changes to class paths in JVM profiles: standard class path

From CICS Transaction Server for z/OS, Version 3 Release 2 onwards, the standard class path is constructed in a new way. Use the CLASSPATH_SUFFIX option to specify any nonshareable application classes.

CICS builds a base standard class path for the JVM using the `/lib` subdirectories of the directories specified by the CICS_HOME and JAVA_HOME options in the JVM profile. This standard class path contains the JAR files supplied by CICS and by the JVM. It is not visible in the JVM profile.

The CLASSPATH option in the JVM profile is no longer used. To help you to upgrade, it is still accepted, but CICS issues a warning message when it is found (DFHSJ0523).

Use the CLASSPATH_SUFFIX option to place classes on the standard class path. When you are creating, changing, or upgrading JVM profiles, specify any items that you added to the standard class path in previous CICS releases using CLASSPATH_SUFFIX.

If you are changing JVM profiles from resettable (REUSE=RESET) to continuous (REUSE=YES), place application classes on the standard class path. With IBM SDK for z/OS, Java Technology Edition Version 6, there is no shareable application class path, so you must use the standard class path whether or not you are using the shared class cache. Classes on the standard class path are placed into the shared class cache. The shareable application class path was the recommended choice for a resettable JVM, because it enabled the classes to be cached in the JVM and reinitialized when the JVM was reset, whereas classes on the standard class path were discarded and reloaded. However, in a continuous JVM, classes on the standard class path are cached in the JVM and kept across reuses.

For CICS Transaction Server for z/OS, Version 4 Release 1, you must upgrade to use Version 6 of the IBM SDK for z/OS, Java Technology Edition for Java support.

Changes to class paths in JVM profiles: library path

From CICS Transaction Server for z/OS, Version 3 Release 2 onwards, the base library path is not visible in the JVM profile. You specify only any additional dynamic link library (DLL) files that you added to the library path. The option to use for this is LIBPATH_SUFFIX.

The base library path for the JVM is built automatically using the directories specified by the CICS_HOME and JAVA_HOME options in the JVM profile. It includes all the DLL files required to run the JVM, and the native libraries used by CICS. In previous CICS releases, you specified the base library path explicitly in the JVM profile, but now that is not required.

The LIBPATH option in the JVM profile is no longer used. To help you to upgrade, it is still accepted, but CICS issues a warning message when it is found (DFHSJ0538). If you leave any classes specified on this option, they are placed on the library path after the base library path.

You can extend the library path using the LIBPATH_SUFFIX option. When CICS builds the library path, these items are placed on the library path after the base library path directories. When you are creating, changing, or upgrading JVM profiles, any items that you added to the library path in previous CICS releases, such as the DLL files required to use the DB2-supplied JDBC drivers, you now specify using LIBPATH_SUFFIX. The CICS-supplied `/lib` and `/ctg` directories, and the IBM JVM-supplied `/bin` and `/bin/classic` directories, which you specified on the library path in the CICS-supplied sample JVM profiles in earlier CICS releases, are not now specified explicitly in the JVM profile. These directories are now part of the base library path.

The option LIBPATH_PREFIX is available if you need to place items before the base library path, but use this option only under the guidance of IBM support.

Changes to class paths in JVM profiles: shareable application class path

In Java 6, the shareable application class path is not used for class sharing. To share Java classes when using Java 6, place the classes on the standard class path for the JVM.

When you upgrade to using Java 6 in a CICS region, if you have any classes on the shareable application class path in your JVM profiles, you must put them on the standard class path. CICS still accepts the shareable application class path but places the classes on the standard class path instead.

With Java 6, the shared class cache does not have a special shareable application class path. If you request class sharing to take place with Java 6 JVMs, all of the classes in the JVMs are shared, and all must be placed on the standard class path, which is defined by the CLASSPATH_SUFFIX option in the JVM profile.

Storage heap in JVMs

The runtime storage in JVMs for IBM SDK for z/OS, Java Technology Edition Version 6 is managed by a single storage heap.

The heap for each JVM are allocated from the storage in the Language Environment enclave for the JVM. The size of each heap is determined by options in the JVM profile.

The single storage heap is just known as "the heap", or sometimes as "the garbage-collected heap". Its initial storage allocation is set by the `-Xms` option in a JVM profile, and its maximum size is set by the `-Xmx` option. There is no separate system heap in this JVM. The JVM supplied by Version 6 of the SDK does have another memory area called the process (or native) heap, but this memory area is used only for the underlying implementation of particular Java objects such as JIT-compiled code, and it is not used for storage of any system classes or application classes.

You can tune the size of a heap to achieve optimum performance for your JVMs. Tuning JVM storage heaps and garbage collection, in the *CICS Performance Guide* tells you how to do this.

Migrating storage settings in JVM profiles from resettable JVMs

You will probably need to adjust and tune the storage-related options in your JVM profiles when you migrate applications to run in continuous JVMs.

When you migrate an application from a resettable JVM to run in a continuous JVM, initially deal with each storage option that you have specified in the JVM profile as shown in Table 4.

Table 4. Migrating storage options in JVM profiles

Option (if specified)	Action
-Xmx	Use the setting from the resettable JVM profile
-Xinitth	Comment out (no longer used)
-Xms	Take the setting from the resettable JVM profile and increase it by the values of **-Xinitth** and **-Xinitacsh** from the resettable JVM profile
-Xinitacsh	Comment out (no longer used)
-Xinitsh	Comment out (no longer used)

These suggestions assume that the continuous JVM is running the same application or applications as the resettable JVM; that is, you are changing an existing

resettable JVM profile to become a continuous JVM profile. If the mix of applications running in the continuous JVM is different, your choice of storage settings will not fit this model.

These suggestions also assume that the storage settings for the resettable JVM were correctly tuned for the needs of your applications. If that is not the case, migrating the storage settings according to this model will not improve that situation.

Use your new settings as a starting point for the continuous JVM. The way in which storage is used in a continuous JVM differs in some respects from the way it is used in a resettable JVM. In particular, bear in mind that the storage heaps in continuous JVMs are not automatically cleaned up after each program invocation. Because of this, depending on the application design and the extent to which each program cleans up after itself, compared to a resettable standalone JVM running the same workload, the continuous JVM might require either larger storage heap sizes or more frequent garbage collection.

Where JVMs are constructed

When a JVM is needed, the CICS launcher program for JVMs requests storage from MVS™, sets up a Language Environment enclave, and launches the JVM in the Language Environment enclave. Each JVM is constructed in its own Language Environment enclave, to ensure isolation between JVMs running in parallel.

The Language Environment enclave is created using the Language Environment preinitialization module, CEEPIPI, and the JVM runs as a z/OS UNIX process. The JVM therefore uses MVS Language Environment services rather than CICS Language Environment services. The storage used for a JVM is MVS storage, obtained by calls to MVS Language Environment services. This storage resides within the CICS address space, but is not included in the CICS dynamic storage areas (DSAs).

The Language Environment enclave for a JVM can expand, depending on the storage needs of the JVM. The Language Environment runtime options used by CICS for a Language Environment enclave control the initial size of, and incremental additions to, the Language Environment enclave heap storage.

You can tune the runtime options that CICS uses for a Language Environment enclave, so that the amount of storage CICS requests for the enclave is as close as possible to the amount of storage specified by your JVM profiles. This makes the most efficient use of MVS storage. Tuning Language Environment enclave storage for JVMs, in the *CICS Performance Guide* tells you how to do this.

Execution key for JVMs

A Java program needs to run in a JVM that is in the correct execution key. JVMs can be in one of two execution keys: user key or CICS key. Running applications in user key extends CICS storage protection, so most of your Java programs should run in a JVM in user key. However, if a Java program is part of a transaction that specifies **TASKDATAKEY**(CICS), the program needs to run in a JVM in CICS key.

When you set the **EXECKEY** parameter on the PROGRAM resource definition for a Java program to USER, CICS gives the program a JVM that is in user key. A J9 TCB is used to run the JVM, and MVS storage is obtained in user key. When you set the **EXECKEY** parameter to CICS, CICS gives the program a JVM that is in CICS key. A J8 TCB is used to run the JVM, and MVS storage is obtained in CICS key.

The default for the **EXECKEY** parameter is USER. Before CICS Transaction Server for z/OS, Version 2 Release 3, the **EXECKEY** parameter was ignored for Java programs. CICS always made them run in JVMs in CICS key, because user key was not available for JVMs. You might find that in most cases, the PROGRAM resource definitions for Java programs that you created for earlier releases of CICS are still set to the default of **EXECKEY**(USER). For CORBA stateless objects and enterprise beans, CIRP (the default transaction for REQUESTMODEL definitions) specifies **TASKDATAKEY**(USER), and the PROGRAM resource definition for DFJIIRP (the default request processor program) specifies **EXECKEY**(USER), so by default CORBA stateless objects and enterprise beans run in user key.

You do not need to make any other changes if you change the **EXECKEY** parameter for a Java program. CICS can use the same JVM profile to create JVMs in both execution keys. A single CICS task can include Java programs running in CICS key, and Java programs running in user key. However, bear in mind that a JVM can only be reused by programs that specify the same execution key and JVM profile on their PROGRAM resource definition. If most of your JVMs are created in the same execution key, CICS has more opportunities for giving a program an existing JVM to reuse, rather than creating a new JVM.

JVMs and the z/OS shared library region

The shared library region is a z/OS feature that enables address spaces to share dynamic link library (DLL) files.

This feature enables your CICS regions to share the DLLs that are needed for JVMs, rather than each region having to load them individually. This can greatly reduce the amount of real storage used by MVS, and the time it takes for the regions to load the files.

The storage that is reserved for the shared library region is allocated in each CICS region when the first JVM is started in the region. The amount of storage that is allocated is controlled by the SHRLIBRGNSIZE parameter in z/OS. Tuning the z/OS shared library region, in the *CICS Performance Guide* tells you how to tune the amount of storage that is allocated for the shared library region.

How CICS manages JVMs in the JVM pool

CICS uses the open transaction environment (OTE) to run JVMs. Each JVM runs on an MVS TCB, which is allocated from a pool of J8- and J9-mode open TCBs, managed by CICS in the CICS address space. This pool of open TCBs is called the JVM pool.

The priority of the J8- and J9-mode open TCBs in the JVM pool is set lower than that of the main CICS QR TCB, to ensure that J8- and J9-mode activity does not affect the main CICS workload that is being processed on the CICS QR TCB.

CICS normally manages the startup of JVMs, creating TCBs and JVMs in response to the demand from applications. You can also start JVMs using CICS commands if you need to. The **CEMT INQUIRE JVMPOOL** command (or the equivalent **EXEC CICS** command) tells you how many JVMs are currently present in the CICS region.

JVMs can be in one of two execution keys: user key or CICS key. JVMs that are in user key need to run on a J9 TCB. JVMs that are in CICS key need to run on a J8 TCB. Statistics are collected separately for each of the modes, so you can see what proportions of each mode are in the JVM pool.

JVMs can be created with any of the JVM profiles that have been defined for your CICS region. The JVM profile determines characteristics of the JVM. You can define different JVM profiles that fit the needs of your Java programs. The JVM profile and execution key are independent of each other, so two JVMs could have the same profile but a different execution key.

You use the PROGRAM resource definition for a Java program to specify the appropriate execution key and JVM profile for the JVM that the program uses. When CICS receives a request to run the program, it might either create a suitable JVM, or assign an existing JVM that is not currently being used.

In the JVM pool, at any one time, some JVMs and their TCBs might be currently allocated to tasks; that is, transactions are using them to run Java programs. When a JVM has finished running a Java program, CICS does not discard it immediately, unless it is a single-use JVM. Instead, CICS keeps the JVM in the pool in case it can be reused to run another Java program. So the JVM pool might also contain some JVMs and their TCBs that are not currently allocated to tasks, but are waiting to be reused.

MAXJVMTCBS: Limit for JVMs in the JVM pool

The total number of TCBs that can be created for JVMs is limited by the **MAXJVMTCBS** system initialization parameter. This parameter therefore limits the number of JVMs that you can have in the JVM pool in your CICS region. The default value for **MAXJVMTCBS** is 5. The minimum permitted value is 1, meaning that CICS is always able to create at least 1 TCB in the JVM pool.

MAXJVMTCBS specifies the maximum total number of J8 and J9-mode TCBs in the JVM pool. You cannot specify the proportions of J8 and J9 TCBs that are in the JVM pool; CICS decides how many should be J8 TCBs and how many should be J9 TCBs, according to the number of requests that specify each execution key. JM TCBs, used for the shared class cache, do not count towards the **MAXJVMTCBS** limit.

Each JVM runs in its own Language Environment enclave, and uses MVS storage. For this reason, you need to choose a **MAXJVMTCBS** limit for your CICS region that takes into account not just the processor time used by the JVMs, but also:

- The amount of MVS storage used by each of your JVMs.
- The amount of MVS storage available for the use of the region.

If you set a **MAXJVMTCBS** limit that is too high, CICS might attempt to create too many JVMs for the available MVS storage, resulting in an MVS storage constraint.

CICS has a storage monitor for MVS storage, which notifies it when MVS storage is constrained or severely constrained, so that it can take short-term action to reduce the number of JVMs in the JVM pool. (The storage monitor uses exits in Language Environment routines; it is not a monitoring transaction.) However, the action that CICS takes when MVS storage is constrained only solves the problem on a temporary basis. When you receive operator messages relating to MVS storage constraints, to provide a long-term solution, you need to work out an appropriate **MAXJVMTCBS** limit that will prevent the problem from recurring. Managing your JVM pool for performance, in the *CICS Performance Guide*, explains more about the action CICS takes to deal with MVS storage constraints, and tells you how to work out an appropriate setting for the **MAXJVMTCBS** system initialization parameter.

Automatic termination of inactive JVMs

If there are too many JVMs in the JVM pool waiting to be reused, and the workload does not require them, CICS terminates them automatically. If a JVM is not used by any application during the period of time specified in the IDLE_TIMEOUT option in its JVM profile, it becomes eligible for automatic termination. The next time CICS checks on the idle JVMs, some of the JVMs that have reached their timeout thresholds and are still idle will be destroyed, together with their TCBs.

CICS does not immediately terminate all of the JVMs that have timed out; instead, they are terminated progressively over a period of time, so that a balanced level of capacity is maintained in the JVM pool. JVMs that have timed out and have not yet been terminated are still available to be reused by applications if there is an increase in demand, and if a JVM is reused it ceases to be eligible for automatic termination. CICS never automatically terminates the last JVM in the JVM pool.

You need to choose an appropriate IDLE_TIMEOUT value for JVMs with each JVM profile. You might prefer CICS to terminate inactive JVMs more quickly in order to free up system resources, and create new JVMs if there is an increase in demand from Java applications. In this case you should select a shorter timeout threshold. Alternatively, you might prefer CICS to keep unused JVMs available for a longer period, so that capacity is always available to meet your peak workloads, without incurring the CPU costs of JVM startup. In this case, you should select a longer timeout threshold.

The default timeout threshold is 30 minutes. You can specify a longer timeout threshold of up to 7 days. You can also specify a timeout threshold of zero, which means that JVMs with that profile are never terminated automatically because of inactivity. Under normal conditions, JVMs with a timeout threshold of zero are only terminated if they are selected for stealing or mismatching.

The process of automatic termination of inactive JVMs operates when conditions in the CICS region are normal. If MVS storage becomes constrained or severely constrained, CICS takes immediate action to manage that situation. During that process unused JVMs are destroyed regardless of their timeout thresholds, even if the timeout threshold is zero.

Example JVM pool

Figure 4 on page 82 shows an example JVM pool. The **MAXJVMTCBS** limit for this JVM pool is 5, and the JVM pool contains 5 JVMs, so CICS has already created the maximum possible number of JVMs in this JVM pool.

The JVM pool contains:

- A JVM (JVM 1) created with the JVM profile DFHJVMPR, in CICS key (so running on a J8 TCB)
- A JVM (JVM 2) created with the JVM profile USERJVM1, in user key (so running on a J9 TCB)
- A JVM (JVM 3) created with the JVM profile DFHJVMCD, the JVM profile for the default request processor program, in user key (so running on a J9 TCB)
- A JVM (JVM 4) created with the JVM profile USERJVM1, in CICS key (so running on a J8 TCB)
- A JVM (JVM 5) created with the JVM profile DFHJVMPR, in user key (so running on a J9 TCB)

JVMs 1, 4 and 5 are currently allocated to tasks, but JVMs 2 and 3 are waiting to be reused.

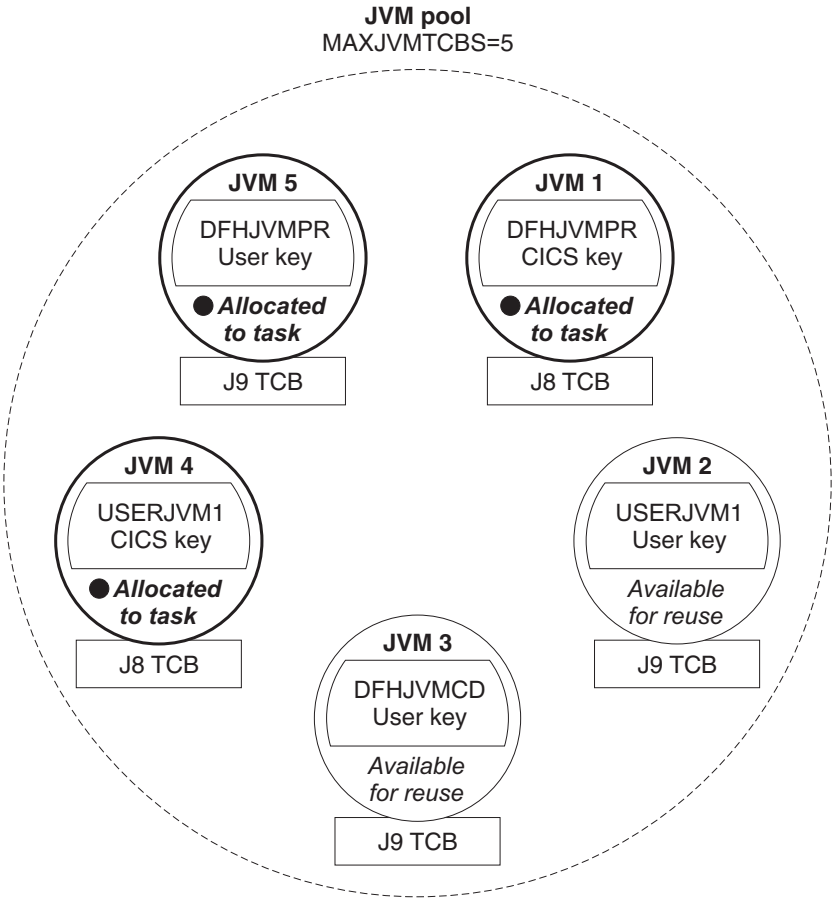

Figure 4. An example JVM pool

How CICS allocates JVMs to applications

When an application requests execution of a Java program, CICS first tries to find a suitable JVM that is available for reuse in the JVM pool. If a suitable JVM, with the correct JVM profile and execution key, is not available, CICS either creates a new JVM if possible, or uses its selection mechanism to decide on an alternative course of action.

An application can reuse an available JVM if the JVM was created using the JVM profile and the execution key (USER or CICS) that are specified in the Java program's PROGRAM resource definition. If a suitable JVM is available, CICS assigns the JVM to the request.

If a suitable JVM, with the correct JVM profile and execution key, is not available, and the limit set by the **MAXJVMTCBS** system initialization parameter has not yet been reached, and MVS storage is not severely constrained, CICS creates a new JVM for the Java program. The new JVM has the correct profile and execution key for the program.

If CICS cannot find a suitable JVM, and a new JVM cannot be created because the **MAXJVMTCBS** limit has been reached, or because MVS storage is severely constrained and CICS is acting as though the **MAXJVMTCBS** limit had been reached, then CICS

must decide on the best way to provide the application with a JVM. This involves assessing the need of the application for a JVM, against the need for different types of JVM in the CICS region. CICS can fulfil an application's request for a JVM by:

- Taking a free JVM that has the right execution key but the wrong profile for the request, destroying the JVM, and re-initializing (that is, re-creating) the JVM on the old JVM's TCB, with the correct profile. This is called a *mismatch*.
- Destroying a free JVM and its TCB that are in the wrong execution key, and replacing it with a JVM and TCB in the correct execution key. This situation is known as a *steal*, or *stealing*, as the TCB has been "stolen" from one TCB mode (J8 or J9) to another TCB mode.

Both a mismatch and a steal are expensive, so before taking one of these courses of action, CICS tries to decide if it is worthwhile. In terms of the need for different types of JVM in the CICS region, it might be more economical for overall system performance for CICS to make the application wait until a suitable JVM is available, and to keep the free JVMs for requests that can benefit more from them. CICS has a selection mechanism to make this decision.

Figure 5 on page 84 shows this process happening. Our example JVM pool is in the state shown above in Figure 4 on page 82, with a **MAXJVMTCBS** limit of 5, and 5 JVMs in the pool. CICS receives two of the requests described above in "Setting up a PROGRAM resource definition for a Java program to run in a JVM" on page 144.

Request B specifies the PROGRAM resource definition for the default request processor program DFJIIRP, which names the JVM profile DFHJVMCD, and the execution key USER. CICS checks the JVM pool, and finds that JVM 3 has the correct JVM profile and execution key to match the request, and it is available for reuse. CICS assigns JVM 3 to Request B.

Request D specifies the PROGRAM resource definition for PROG1, which names the JVM profile USERJVM2, and the execution key CICS. CICS checks the JVM pool. There is a free JVM, JVM 2, but it has the wrong profile and execution key for Request D. As the **MAXJVMTCBS** limit has been reached, CICS cannot create a new JVM for Request D. So CICS must use the selection mechanism to decide if it should destroy JVM 2 and its TCB, and replace it with a JVM and TCB that matches Request D; or if it should make Request D wait, and keep JVM 2 for a request that can benefit more from it. If Request D is made to wait, it is queued along with any other requests that are waiting for a JVM.

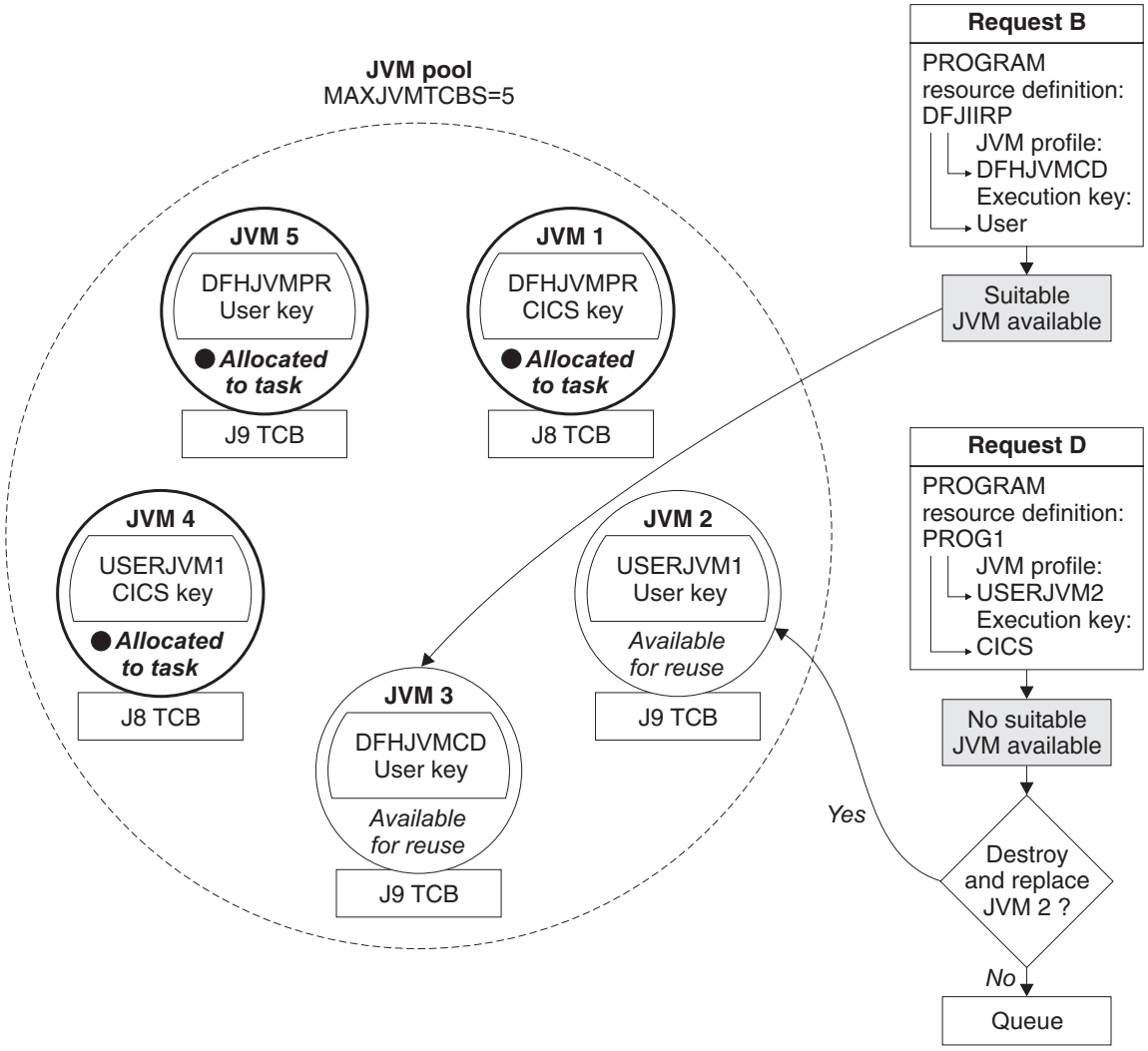

Figure 5. Dealing with requests for JVMs: example

Now let's look in more detail at the whole process. CICS makes its decision to assign a JVM to an application in two stages:

- It takes one set of actions to deal with incoming requests for a JVM
- It takes another set of actions when it has a queue of requests waiting for a JVM.

How CICS deals with incoming requests for a JVM

To deal with incoming requests for a JVM, CICS takes the actions summarized here.

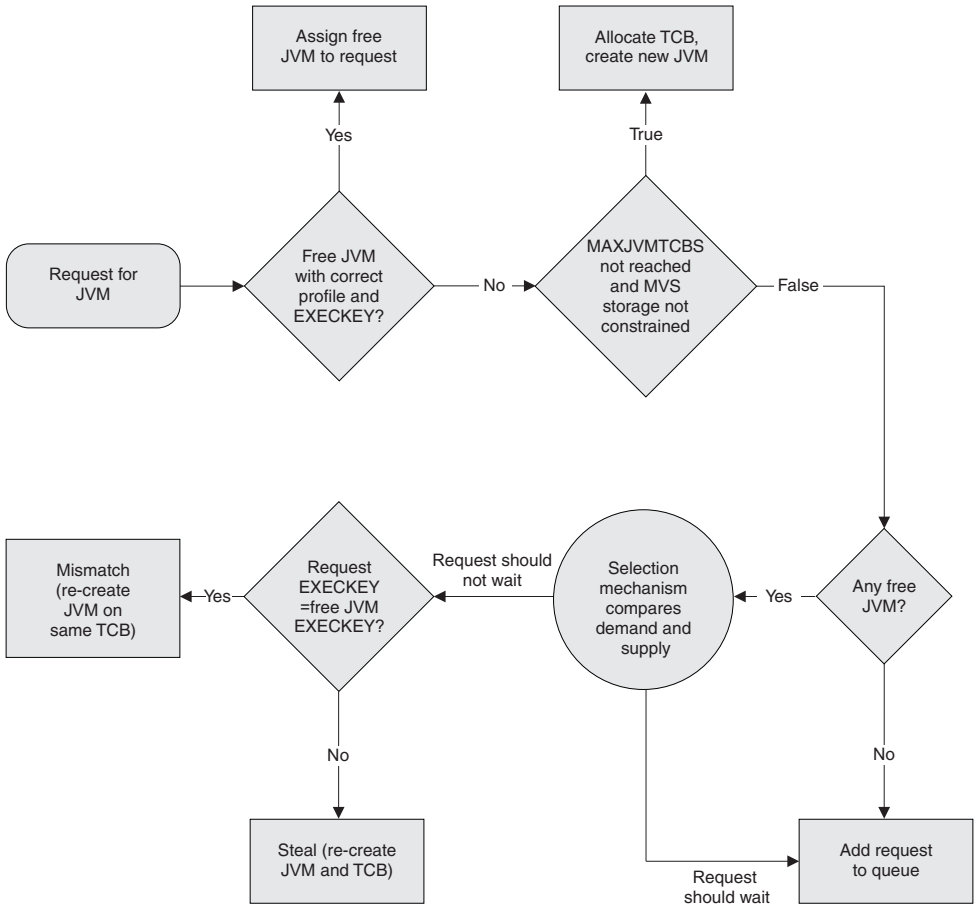

Figure 6. Dealing with incoming requests for JVMs

1. When CICS receives a request for a JVM, and a JVM of the correct profile and execution key is free, CICS assigns the JVM to the incoming request.
2. If CICS receives a request for a JVM when either:
   - there are no free JVMs
   - there are free JVMs, but they are not of the correct profile and execution key for the request

   and CICS **is** able to create more JVMs (because the **MAXJVMTCBS** limit has not been reached and MVS storage is not severely constrained), then a TCB is allocated and a new JVM is created for the request.
3. If CICS receives a request when there are free JVMs, but they are not of the correct profile and execution key, and CICS is **not** able to create more JVMs (because the **MAXJVMTCBS** limit has been reached or MVS storage is severely constrained), the selection mechanism is used. The selection mechanism decides whether the request should wait for a suitable JVM, or whether it should receive one of the free JVMs.
   a. If the request receives one of the free JVMs, there will be either a mismatch or a steal, and the JVM and possibly the TCB will need to be re-initialized, so the selection mechanism avoids this where it makes sense to do so. If the selection mechanism does decide that the request should receive one of the free JVMs, CICS checks whether the execution key specified by the request matches the execution key of the JVM. If the execution key does not match,

the JVM and its TCB are destroyed and reinitialized (a steal). If the execution key does match, and only the JVM profile is incorrect, the JVM is reinitialized on the same TCB (a mismatch).

b. If the selection mechanism decides that the request should wait rather than receiving one of the free JVMs, the request is placed on the queue to wait for a suitable JVM to become free.

4. If CICS receives a request when there are no free JVMs, and CICS **is not** able to create more JVMs (because the **MAXJVMTCBS** limit has been reached or MVS storage is severely constrained), the request is placed on the queue to wait for a JVM to become free.

How CICS deals with a queue of requests waiting for a JVM

When CICS has a queue of requests waiting for a JVM, it takes these actions.

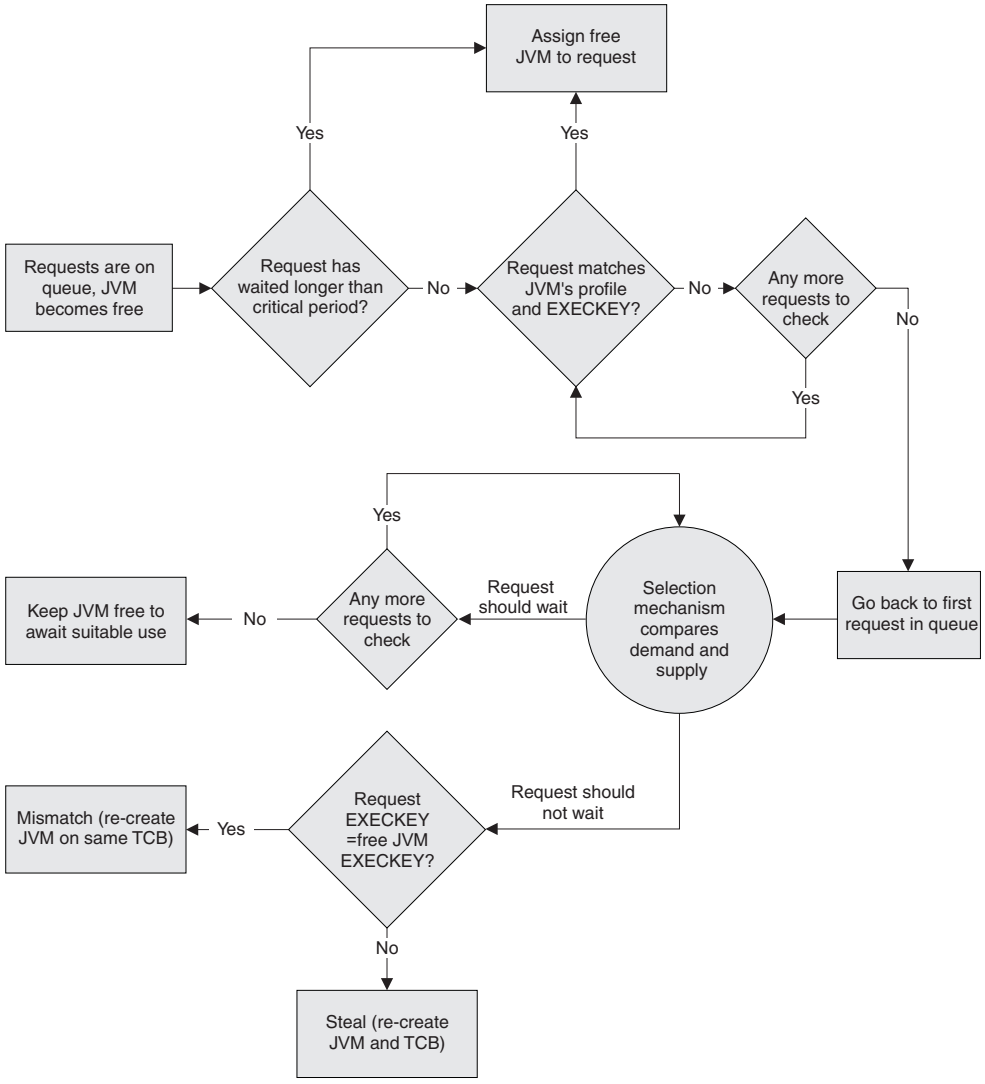

Figure 7. Dealing with a queue of requests waiting for a JVM

1. If any request that is waiting for a JVM to become free has been waiting longer than a critical period (which CICS determines), CICS gives it the next available JVM, whatever the profile and execution key of the JVM. This applies both to

requests that have been placed on the queue because no JVMs are free, and requests that have been placed on the queue because the free JVMs have the wrong profile or execution key. There will be either a mismatch or a steal, and the JVM and possibly the TCB are likely to be re-initialized (unless the request is in a queue and the next free JVM happens to have the correct profile and execution key), but the action is worth taking, as the request should not wait any longer.

2. If requests are queueing and a JVM becomes free, but no requests have been waiting longer than the critical period, CICS scans through the queue to find the longest-waiting request that requires a JVM with that profile and execution key. It gives the free JVM to the longest-waiting request that specifies the correct profile and execution key. So in this situation, the JVM does not need to be re-initialized, and a mismatch or steal is avoided.
3. If CICS cannot find a request that matches the profile and execution key of the free JVM, it scans through the queue again and uses the selection mechanism to look for a request where it will be an advantage to destroy and re-initialize the free JVM, and re-initialize it as a JVM with the profile and execution key that the request needs. A mismatch or a steal occurs, but the selection mechanism ensures that it occurs for a deserving request.
4. If CICS does not find a request in the queue where it will be an advantage to destroy and re-initialize the free JVM, the JVM is kept free to await a more appropriate use. For example, CICS might receive a request that needs a JVM with the profile and execution key of the free JVM; or the first request in the queue might wait longer than the critical period, and so be given the free JVM; or CICS might receive a request where it is an advantage to destroy and re-initialize the free JVM.

The selection mechanism

The selection mechanism is used when CICS needs to know if an incoming request should wait for a more suitable JVM, or when CICS has a queue of requests that do not match a free JVM, and needs to know if one of them deserves to take, destroy and re-initialize the JVM.

In these situations, the mechanism looks at the complete picture of the need for different types of JVM in the CICS region. It compares the demand for, and supply of, JVMs with each profile and execution key, by looking at:

- The historical data relating to recent requests for each type of JVM (the demand).
- The number of each type of JVM in the pool, and the time for which tasks kept these JVMs (the supply).

The selection mechanism uses this data to work out whether a given request should wait for a JVM of the correct profile and execution key, or whether it should be given a free JVM. The same answer is valid for a request that is waiting in a queue for a JVM to become free, or for a request that is made when there are free JVMs but they are not of the correct profile or execution key. In both cases, a request is made to wait if the data indicates that the demand for the type of JVM (that is, a JVM with that profile and execution key) which the request wants, is generally *lower* than the supply, and so it is not worth destroying and re-creating the free JVM as a JVM of that type. When the selection mechanism is examining a queue of requests, it continues down the queue until it reaches a request where the data indicates that the demand for the type of JVM that the request wants is generally *higher* than the supply. For this request, the selection mechanism decides that because JVMs of that type are needed in the CICS region, it is worth destroying and re-creating the free JVM as a JVM of that type, and assigns the free

JVM to the request. If the free JVM had the wrong profile but the correct execution key, this is a mismatch, and the JVM is re-initialized. If the free JVM had the wrong execution key, this is a steal, and both the TCB and JVM are destroyed and re-created. So although the overhead of re-initializing the JVM, and if necessary re-creating the TCB, has still been incurred, the selection mechanism has ensured that the new JVM and TCB are of a type that is likely to be used in the future.

Under certain circumstances, there could be an unusually large number of requests for JVMs that have been waiting longer than the critical period. For example, this could happen when a system dump has just been taken, which delays all processing. In this case, rather than abandon matching and give each of the waiting requests the next available JVM, as would normally happen when a request has been waiting longer than the critical period, CICS temporarily increases the critical period value for the JVM pool. This enables it to perform matching for the waiting requests, and avoids incurring abnormal overhead. Once the situation has passed, CICS lowers the critical period value again.

How JVMs are reused

Every Java program that is run in CICS, runs in a JVM that has been assigned to run that program alone. This ensures that every transaction involving a JVM is isolated from every other concurrent transaction involving a JVM. However, when a Java program has finished using its JVM, the JVM can be reassigned to another, subsequent program and reused for that program.

The type of reusable JVM used in CICS TS 4.1 is the continuous JVM provided by the IBM SDK for z/OS, Java Technology Edition. A continuous JVM has the option REUSE=YES in its JVM profile. This JVM can be reused many times by Java applications in CICS, either by a different Java program in the same transaction, or by another transaction. This model is suited to CICS transaction processing, which is characterized by short, repetitive transactions, usually processed in high volumes.

You can modify a JVM to run as a single-use JVM and not attempt serial reuse. A single-use JVM has the option REUSE=NO in its JVM profile. A single-use JVM is initialized, is used to run a single Java program, and then is automatically destroyed. The single-use JVM is not recommended for running Java applications in a production environment. A single-use JVM is only beneficial for Java applications that were originally designed to run in a single-use JVM, and have not been made suitable for running in a JVM that is intended for reuse.

From CICS Transaction Server for z/OS, Version 3 Release 2, resettable JVMs, which were reset between each use, are no longer supported. Any Java programs that ran in resettable JVMs must be migrated to run in continuous JVMs. Continuous JVMs generally perform better because they are not reset between each use, and they are also compatible with future versions of Java. The migration process involves checking that the Java programs do not contain any code which might have an unwanted effect on serial isolation when the continuous JVM is reused by a subsequent program.

Continuous JVMs (REUSE=YES)

The continuous JVM is kept in the JVM pool for reuse. It is initialized once, and is reused many times. A continuous JVM has the option REUSE=YES in its JVM profile.

Compared to older types of JVMs used with CICS, the behavior of the continuous JVM is more consistent with the behavior of JVMs on platforms other than CICS, which can be an advantage when executing Java programs designed for use in a generic reusable Java environment.

Programs that run in a continuous JVM are fully isolated from concurrent activity elsewhere in CICS. However, the application code that runs in the next Java program or transaction is not automatically isolated from the actions of the previous program invocation (that is, serial isolation is not automatic). You need to ensure that your Java application programs do not change the state of a continuous JVM in undesirable ways, or leave any unwanted state in the JVM. You can also exploit this characteristic of the continuous JVM to your advantage, for example, by creating persistent items that might be of use to future executions of the same application in the same JVM.

A continuous JVM maintains the content of its storage heaps between one program invocation and the next. Static or dynamic state persist in a continuous JVM's storage heaps, and threads that are not quiesced will persist, along with their related storage. All application classes that have been loaded into the JVM are kept intact. The application can choose to clean up any unwanted items and retain any desirable items.

A continuous JVM can use the shared class cache. JVMs that use the shared class cache start up more quickly, and have lower storage requirements, than JVMs that do not.

"Programming considerations for continuous JVMs" on page 133 explains the programming considerations for applications that run in a continuous JVM.

Single-use JVMs (REUSE=NO)

The single-use JVM is not kept in the JVM pool for reuse. With this type of JVM, the JVM is initialized, is used to run a single Java program, and then is automatically destroyed. A single-use JVM has the option REUSE=NO in its JVM profile.

The single-use JVM is like the earlier JVM that was supported by CICS in CICS TS 1.3, for which support was removed in CICS TS 2.3 (see Migration for Java applications).

The single-use JVM is not recommended for running Java applications in a production environment, and it should not be used for Java applications comprising enterprise beans or which are started by IIOP requests. It is only beneficial for Java applications that were originally designed to run in a single-use JVM, and have not been made suitable for running in a JVM that is intended for reuse. To improve performance, you should redesign these Java programs as soon as you can, so that the programs can run in a continuous JVM.

The single-use JVM has poor performance in terms of transaction throughput, because the JVM must be initialized for each use, which is an expensive process.

If you use a single-use JVM, you can invoke the user-replaceable program DFHJVMAT to change options in the JVM profile, as you could in CICS TS 1.3. This user-replaceable program cannot be invoked for a continuous JVM.

"Programming considerations for single-use JVMs" on page 141 explains the programming considerations for applications that run in a single-use JVM.

The shared class cache

The IBM SDK for z/OS provides a class sharing facility for the JVM, where multiple JVMs can share a single cache of class files that have already been loaded. CICS supports this facility and provides an interface for you to manage the shared class cache.

JVMs that use the shared class cache start up more quickly, and have lower storage requirements, than JVMs that do not. The overall cost of class loading is also reduced when JVMs use the shared class cache. When a new JVM that shares the class cache is initialized, it uses the preloaded classes instead of reading them from the file system. A JVM that shares the class cache still owns all the working data (objects and variables) for the applications that run in it. This helps to maintain the isolation between the Java applications being processed in the system.

The shared class cache can support the majority of the JVMs in each CICS region. A JVM that uses the shared class cache can be continuous or single-use, and it can be used for debugging if required. If you want to isolate some JVMs in a CICS region from the shared class cache, these JVMs can run as standalone JVMs, and have their own cache of classes in their storage heaps.

There is no difference in the JVM options for a JVM that uses the shared class cache and a JVM that does not, except for the CLASSCACHE option. When you specify CLASSCACHE=YES in the JVM profile for a JVM, the JVM uses the shared class cache. Otherwise, the JVM profiles are set up in the same way, and the same class paths are used.

CICS supports one active shared class cache in each region. (A region might also contain old shared class caches that are being phased out.) You can manage the shared class cache and monitor its status using CICS commands.

CICS uses the CICS-supplied sample profile DFHJVMCD to initialize and terminate the shared class cache. DFHJVMCD must always be available and configured for use in your CICS region, but you do not need to make any additional changes to it for use with the shared class cache.

CICS uses one or more JM TCBs, a type of open TCB, for shared class cache management functions. JM TCBs are not used for any other purpose, and they do not count towards the **MAXJVMTCBS** limit for the JVM pool.

Contents of the shared class cache

The shared class cache contains all the classes that are needed by the JVMs that use the shared class cache. There is no distinction between shareable and nonshareable classes, as there was with older types of JVM used in CICS. All the application classes needed by your Java programs are placed on the standard class path in your JVM profiles, and they are all eligible to be loaded into the shared class cache. (In some exceptional scenarios, discussed in the *IBM Developer Kit and Runtime Environment, Java Technology Edition, Version 6 Diagnostics Guide*, some classes might not be eligible to be loaded into the shared class cache.)

The shared class cache does **not** store these items:

- Native C dynamic link library (DLL) files specified on the library path in JVM profiles. A single copy of each DLL file is used by all the JVMs that need it.
- Working data for applications (objects and variables). This is stored in the individual JVMs.
- Compiled classes produced by just-in-time (JIT) compilation. These are stored in individual JVMs, not in the shared class cache, because the compilation process can vary for different workloads.

The shared class cache updates its contents automatically if you change any application classes or JAR files, or add new items to the class paths in your JVM profiles, and restart the appropriate JVMs.

If the shared class cache becomes full, JVMs can continue to use the classes that are already present in it, and any further classes are loaded into the individual JVMs. A warning message is issued if you have requested verbose output, but the JVMs can continue to run applications as they did before. You can use CICS commands to phase in a new, larger shared class cache to replace the old one. The **JVMCCSIZE** system initialization parameter specifies the initial size of the shared class cache.

Lifespan of the shared class cache

The shared class cache in a CICS region normally starts when the first JVM needs to use it. When you specify CLASSCACHE=YES in the JVM profile for a JVM, the JVM uses the shared class cache that is currently active in that CICS region, or creates one, if it is the first JVM in the CICS region that requires the shared class cache. Although the first JVM creates the shared class cache, that JVM does not own the shared class cache.

The shared class cache is normally persistent across warm and emergency CICS starts, except in some circumstances such as an IPL of z/OS, so there is no startup cost to the first JVM in the CICS region at those times. The shared class cache is destroyed on a cold or initial start, and normally starts again automatically when it is required. The **JVMCCSTART** system initialization parameter controls the startup behavior of the shared class cache at CICS region initialization.

It is possible to disable the shared class cache from starting automatically (autostart), and use CICS commands to start it manually. You can change the autostart status of the shared class cache while CICS is running.

Because the shared class cache updates its contents automatically, you should not normally need to terminate it manually while CICS is running, unless you introduce new workload and the shared class cache becomes full. In this case, you can phase in a new shared class cache with a larger size. The old shared class cache remains in the system until all the JVMs that are using it have been terminated, and then it is deleted. New JVMs use the new shared class cache.

The shared class cache is named `CICS_sharedcc_&APPLID;_n`, where `&APPLID;` is the applid of the CICS region, and `n` is a generation number starting at zero. In Java 6, it is possible to have multiple shared class caches available for use at the same time, but CICS Transaction Server for z/OS, Version 4 Release 1 does not provide support for this. A CICS region contains multiple shared class caches while an old shared class cache is being phased out, but all new JVMs must use the new shared class cache. The generation number is used to differentiate the name of the new shared class cache.

Chapter 13. Setting up Java support and using JVMs

You need to understand how to customize JVM profiles and properties files, manage your JVMs and shared class cache, and how to identify problems with your Java applications and JVMs.

Before you begin

Before you begin, verify that the Java components are correctly installed using the tasks outlined in Setting up Java support.

About this task

When you have verified that the Java components are correctly installed, you are ready to use the following topics.

Setting up JVM profiles

CICS requires JVM profiles to create JVMs. CICS supplies samples that you can copy and customize, or you can create your own JVM profiles based on the samples.

JVM profiles

JVM profiles contain Java launcher options and system properties, which determine the characteristics of JVMs. When CICS receives a request to run a Java program, the PROGRAM resource definition names an appropriate JVM profile for the Java program.

What are JVM profiles?

JVM profiles are text files that contain lists of options and comments. You can edit JVM profiles using any standard text editor. You can also specify any UNIX System Services environment variables in a JVM profile; these variables apply only to JVMs created with that profile.

A JVM profile lists the Java launcher options used by the CICS launcher for Java. Some of the options are standard for the JVM runtime environment and some are nonstandard JVM options, which might be subject to change in future releases of the Java language specification. Some CICS-specific options are required only by CICS. Here are some examples of JVM characteristics controlled by the JVM profile:

- The initial size of the storage heap in the JVM, and how far it can expand.
- Whether the JVM can be reused (a continuous JVM) or is not reusable (a single-use JVM).
- Whether the JVM uses the shared class cache.
- The destinations for messages and dump output produced by the JVM.
- The timeout threshold after which an inactive JVM is eligible for automatic termination.

As well as determining the characteristics of a JVM, you use the JVM profiles to specify the class paths. Class paths contain the directories that the JVM searches for the application classes and resources that are needed for your applications.

When CICS receives a request to run a Java program, the name of the JVM profile is passed to the Java launcher. The Java program is given a JVM, which was created using the options in the JVM profile and the JVM properties file, if one is specified.

What are JVM properties files?

JVM properties files are text files that can list the system properties for the JVM. From CICS TS 4.1 onwards, JVM properties files are optional because all system properties can be included in the JVM profile. Samples of JVM properties files are not provided.

Although properties files are optional, you might find it convenient to use separate JVM properties files, for these reasons:

- To specify system properties that are used by more than one type of JVM.
- To separate out sensitive information and apply extra security controls to it.
- To continue using files that you created in a previous release.

You can create and edit JVM properties files using any standard text editor. The name of a JVM properties file can be any length, but, for ease of use, it is generally a short name with some similarity to the name of the JVM profile that references it. Do not give it a name beginning with DFH, because these characters are reserved for use by CICS. The name can be any name that is a valid name for a Unix System Services file.

The rules for coding JVM properties files are the same as the rules for coding JVM profiles. For more information on the rules, see "Creating your own JVM profiles" on page 103.

If you create a JVM properties file, store it in any z/OS UNIX directory. Ensure that CICS has read and execute access on z/OS UNIX for your JVM properties file and the directory containing it. "Giving CICS regions permission to access z/OS UNIX directories and files" on page 66 tells you how to do this. Specify the full path name for the JVM properties file, using the JVMPROPS option, in all the JVM profiles that you want to reference that JVM properties file. For example, a JVM profile that states `JVMPROPS=/u/myuser/myprops/myjvm.props` references the JVM properties file `myjvm.props`, in the directory `/u/myuser/myprops`.

When you specify the name of your JVM properties file anywhere in CICS, you must use the same combination of uppercase and lowercase characters that is present in the z/OS UNIX file name of the JVM properties file. If you change the location of JVM properties files, you must specify the correct path in the JVM profiles that reference those JVM properties files.

System properties are key name and value pairs that contain basic information about the JVM and its environment, such as the operating system in which the application is running. Here are some examples of information that can be supplied by a JVM properties file:

- The name of the Java security manager to be used and the names of security policy files that define the security properties for the JVM. Use these system properties to enable the Java security policy mechanism for the JVM.

- The names of the JDBC drivers supplied by DB2, and also the DataSource interface, so that your Java applications running in CICS can access DB2 data.
- The name server to be used for JNDI references.
- Security information for access to an LDAP name server.

Security caution:

1. Ensure that JVM properties files are secure, with update authority restricted to system administrators, if they are used to define sensitive JVM configuration options, such as the security policy file.
2. In particular, if you specify that a secure LDAP server is to be used, by coding **-Djava.naming.security.authentication** in the JVM properties files, you must specify **-Djava.naming.security.principal** and **-Djava.naming.security.credentials**. These properties hold the user ID and password that CICS requires to access the secure LDAP service, so you must give particular attention to the access controls in force at your installation for the JVM properties files and for any other copies of this information that you have.

Where are JVM profiles located?

JVM profiles are stored in z/OS UNIX System Services. You must ensure that CICS region user ID has read access to the JVM profiles and execute access to the directories containing them.

CICS looks for JVM profiles in the z/OS UNIX System Services directory that is specified by the **JVMPROFILEDIR** system initialization parameter. **JVMPROFILEDIR** specifies the full path of the z/OS UNIX directory, and this path can be up to 240 characters long.

When you install CICS, the CICS-supplied sample JVM profiles are placed in the `/usr/lpp/cicsts/cicsts41/JVMProfiles directory`. The `/usr/lpp/cicsts/cicsts41` directory is the installation directory for CICS files on z/OS UNIX. This directory is specified by the **USSDIR** parameter in the DFHISTAR installation job, which is passed to the *uss_path* variable used by the DFHIJVMJ job, which creates the sample profiles.

The supplied setting for the **JVMPROFILEDIR** system initialization parameter is `/usr/lpp/cicsts/cicsts41/JVMProfiles`, which is the installation location for the sample JVM profiles. This directory is not a safe place to store your customized JVM profiles, because you risk losing your changes if the sample JVM profiles are overwritten when program maintenance is applied. So you must always change **JVMPROFILEDIR** to specify a different z/OS UNIX directory where you can store your JVM profiles. Choose a directory where you can give appropriate permissions to the users who must customize the JVM profiles.

Before you start to work with the CICS-supplied sample JVM profiles, copy them from their installation location to the z/OS UNIX directory that you specified for the **JVMPROFILEDIR** system initialization parameter. Make your customization changes to these copies of the files.

If you create your own JVM profiles, you can also store them in the z/OS UNIX directory that you specified for the **JVMPROFILEDIR** system initialization parameter. As an alternative, you can link to the new JVM profiles from the directory specified by **JVMPROFILEDIR** (using UNIX symbolic links). In this way, you can store your profiles in any place in the z/OS UNIX file system.

A JVM profile can reference a JVM properties file by using its full path name specified on the JVMPROPS option in the JVM profile, so CICS does not need to know the location of any JVM properties files. The **JVMPROFILEDIR** system initialization parameter does not apply to JVM properties files.

To locate a particular JVM profile in z/OS UNIX, you can use the **EXEC CICS INQUIRE JVMPROFILE** command to find the full path name of the z/OS UNIX file for the JVM profile, provided that the JVM profile has been used during the lifetime of the CICS region. (This command has no CEMT equivalent.)

Working with JVM profiles.

When you are setting up Java support in a CICS region, you set the **JVMPROFILEDIR** system initialization parameter to the location on z/OS UNIX where you want to store the JVM profiles used by the CICS region.

You then copy the CICS-supplied sample JVM profiles into this directory, so that you can use them to verify your installation and customize them later on.

You must ensure that CICS has read and execute access on z/OS UNIX for the JVM profiles, any optional JVM properties files, and for the other resources needed to create JVMs. For more information, see Chapter 11, “Setting up Java support,” on page 65.

CICS supplies two sample JVM profiles. DFHJVMPR is used if a Java program is defined as using a JVM but no JVM profile is specified, and it is used for sample programs. DFHJVMCD is used by CICS-supplied system programs, including the default request processor program (DFJIIRP) and the program that CICS uses to publish and retract deployed JAR files (DFJIIRQ, the CICS-key equivalent of DFJIIRP). CICS also uses DFHJVMCD to initialize and terminate the shared class cache. These two JVM profiles must always be available to CICS in the directory specified for the **JVMPROFILEDIR** system initialization parameter. If you create any JVM properties files, you must have the correct path specified in the corresponding JVM profile and you must set them up correctly for your CICS region.

You associate every Java program that you want to run under CICS with an appropriate JVM profile for the requirements of the Java program. In some cases, you might find that the options in the copies that you have made of the sample JVM profiles must be changed to fit the needs of a particular application or your CICS region. As an alternative, you can create your own JVM profiles.

If you already have JVM profiles, which you set up in a previous CICS release, you might want to upgrade these for use with the new CICS release rather than setting up new profiles based on the new samples. The settings that are suitable for use in JVM profiles can change from one CICS release to another, so check the CICS documentation for any significant changes, and compare your existing JVM profiles with the latest CICS-supplied samples. Make a copy of your JVM profiles in a new location on z/OS UNIX to use with the new CICS release, and make the changes that are required to upgrade them; for example, changing the path for the home directory for CICS files on z/OS UNIX. Do not try to use JVM profiles with more than one CICS release at the same time, because the settings are not compatible.

In the PROGRAM resource definition for each Java program, you name a suitable JVM profile. The JVM profile can reference a JVM properties file. When CICS

receives a request to run a Java program, it either creates a new JVM using these options and assigns it to the program or assigns the program an existing JVM that was created using these options.

You also add the classes and native libraries used by each Java program to the class paths specified in the JVM profile and the JVM properties file, if one is specified, that you have chosen for it. In this way, the JVM can load the classes for your program.

Whenever you introduce new JVM profiles or JVM properties files, or if you change the directory specified by **JVMPROFILEDIR**, ensure that CICS has read and execute access on z/OS UNIX for the files and directories involved.

If program maintenance is applied to update the CICS-supplied sample JVM profiles, this maintenance is applied to the samples in their installation location, and not to your JVM profiles or copies of the samples stored in any other location. Examine any changes introduced by the program maintenance, and consider making similar changes to your customized copies of the samples or to your own JVM profiles and any JVM properties files.

The CICS-supplied sample JVM profiles

CICS supplies three sample JVM profiles to help you configure your Java environment. They are tailored for your system during the CICS installation process. These files are used by CICS as defaults or for system programs.

You can copy the samples and use them for your own applications, customizing them as necessary, or you can create your own files based on them.

The sample JVM profiles include symbols for the variable part of the name of the installation directory for CICS files on z/OS UNIX (&CICS_HOME) and for the installation directory for the IBM SDK for z/OS, Java Technology Edition, which provides Java support (&JAVA_HOME). As part of the CICS installation process, you run the DFHIJVMJ job, which is described in the *CICS Transaction Server for z/OS Installation Guide* . The DFHIJVMJ job substitutes your own values for the symbol names and produces sample files that are tailored for your system.

The base library path and base class path for the JVM, which are not visible in the JVM profile, are built automatically using these directories.

The text provided in the CICS documentation shows the sample files as they appear after the default values are substituted for the symbol names; that is, cicsts41 for the &CICS_HOME symbol, and java/J6.0 for the &JAVA_HOME symbol.

When you install CICS, the CICS-supplied sample JVM profiles are placed in the directory `/usr/lpp/cicsts/cicsts41/JVMProfiles`. The `/usr/lpp/cicsts/cicsts41` directory beneath which the sample JVM profiles are stored is the installation directory for CICS files on z/OS UNIX. This directory is specified by the USSDIR parameter in the DFHISTAR installation job, which is passed to the *uss_path* variable used by the DFHIJVMJ job that creates the sample profiles. CICS does not supply any sample JVM properties files.

Before you start to work with the CICS-supplied sample JVM profiles, copy them from their installation location to the z/OS UNIX directory that you specified with the **JVMPROFILEDIR** system initialization parameter. The sample JVM profiles in their

original installation location are overwritten if you apply an APAR that includes changes to these files. To avoid losing your modifications, you must always copy the samples to a different location before adding your own application classes or changing any options.

Table 5 explains the key characteristics of each of the sample JVM profiles.

Table 5. CICS-supplied sample JVM profiles

JVM profile	Characteristics
DFHJVMPR	Profile DFHJVMPR is the default if no JVM profile is specified in the PROGRAM resource definition of a Java program. JVMs created with the profile DFHJVMPR use the shared class cache. (The profile specifies CLASSCACHE=YES.) DFHJVMPR is the default if no other JVM profile is specified, and it is used for sample programs. Make sure that it is set up correctly for your CICS region.
DFHJVMAX (reserved for the use of CICS)	The JVM server has its own JVM profile to control the startup of the JVM. The DFHJVMAX profile is specified on the JVMSERVER resource definition. CICS uses DFHJVMAX to initialize the JVM server for system processing. Do not specify this profile in PROGRAM resource definitions that you set up for your own applications. However, you must make sure that it is set up correctly for your CICS region.
DFHJVMCD (reserved for the use of CICS)	CICS-supplied system programs have their own JVM profile, DFHJVMCD, to make them independent of any changes that you make to the default JVM profile, DFHJVMPR. In particular, the PROGRAM resource definition for the default request processor program, DFJIIRP, specifies DFHJVMCD. The CICS-supplied default is that JVMs created with the profile DFHJVMCD do not use the shared class cache (the profile specifies CLASSCACHE=NO), but you can change that. Do not specify this profile in PROGRAM resource definitions that you set up for your own applications. However, you must make sure that it is set up correctly for your CICS region. CICS uses DFHJVMCD to initialize and terminate the shared class cache as well as using it for CICS-supplied system programs.

If you are setting up standard Java programs or your own request processor program definition, choose an appropriate JVM profile to specify on the PROGRAM resource definition. Profile DFHJVMPR is the default if no JVM profile is specified in the PROGRAM resource definition of a Java program.

In many cases, you might find that you can use your copies of the sample JVM profile with most of the options that are already set in them, and just add your own application classes to the class paths. In some cases, you might have to change the options to fit the needs of a particular application or of your CICS region. As an alternative, you can create your own JVM profiles.

What you can change in JVM profiles

You might have to change the options in JVM profiles to fit the needs of a particular Java application or of your CICS region.

Some key options that you might want to change are explained here:

- Enable Java security for the JVM. The Java security policy mechanism protects Java applications running in a JVM, and particularly enterprise beans, from performing a potentially unsafe action. You can enable Java security by naming a security manager, using the **-Djava.security.manager** system property, and to state the location of one or more security policy files that the security manager will use to determine the security policy for the JVM, using the **-Djava.security.policy** system property. "Protecting Java applications in CICS by using the Java security policy mechanism" on page 345 describes the changes that you need to make to enable Java security, how to set up a security policy file, and the CICS-supplied sample security policy file `dfjejbpl.policy`, which defines security properties that are suitable for JVMs used by enterprise beans.
- Change the amount of storage available for the application, by changing the size of the heap in the JVM, using the **-Xmx** option in the JVM profile. The value specified in the supplied sample JVM profile DFHJVMPR is `16M`, which is adequate for most purposes. If you have large Java applications, you might want to increase this value. Tuning JVM storage heaps and garbage collection, in the *CICS Performance Guide*, has more information about the storage-related JVM options and how to determine suitable values for them.
- Change the timeout threshold for the JVM, using the IDLE_TIMEOUT option in the JVM profile. The default is that an inactive JVM becomes eligible for automatic termination by CICS after a 30-minute period. If you prefer to keep unused JVMs available for a longer period, you can select a timeout threshold up to 7 days or set the JVM to never time out. See "Automatic termination of inactive JVMs" on page 81 for more detail.
- Change the destination for messages and output from the JVM. You can change the name and location of the `stdin`, `stdout`, and `stderr` files and Java dumps, and use symbols to make these files unique to each JVM. During application development, you can redirect messages from JVM internals and output from Java applications using the USEROUTPUTCLASS option in the JVM profile. "Controlling the location for JVM stdout, stderr and dump output" on page 156 tells you more about the changes that you can make.
- Change your work directory using the WORK_DIR option in the JVM profile. The WORK_DIR default is `/tmp`.
- Set up the JDBC drivers supplied by DB2 and the DataSource interface, so that your Java applications can access DB2 data. See Using JDBC and SQLJ to access DB2 data from Java programs and enterprise beans written for CICS , in the *CICS DB2 Guide*, for more information. You use various options, which are described in that topic.
- For CORBA stateless objects and enterprise beans, specify the information that is necessary to configure the name server to be used for JNDI references using the **-Dcom.ibm.cics.ejs.nameserver** system property, and further information that is necessary if you are using an LDAP name server. See the procedures described in Chapter 16, "Configuring CICS for IIOP," on page 181.

"JVM profiles: options and samples" on page 105 documents a selection of relevant options that you can specify.

If any changes are required to fit with the configuration of your CICS region (for example, if you are required to enable Java security), you must make the same changes to your copies of the supplied sample JVM profiles, DFHJVMPR and DFHJVMCD, in the directory specified by **JVMPROFILEDIR**, and any optional associated JVM properties files. CICS uses these supplied sample JVM profiles for a number of functions, so you must configure both of these JVM profiles so that they can be used in your CICS region.

03 If you want to change any of the options in the JVM profiles, see "Customizing or
03 creating JVM profiles."

If you do not want to change any of the options specified in the JVM profiles and you have specific applications (standard Java programs, CORBA stateless objects, or enterprise beans) to run, "Enabling applications to use a JVM" on page 142 tells you how to set up applications to use a JVM profile and how to add the classes for the application to the class paths.

Customizing or creating JVM profiles

If you want to change any of the options specified in the JVM profiles, you can either customize your copies of the CICS-supplied sample files in the directory specified by **JVMPROFILEDIR** or create your own JVM profiles.

Before you start to work with the CICS-supplied sample JVM profiles, make sure that they have been copied from their installation location to the z/OS UNIX directory that you specified with the **JVMPROFILEDIR** system initialization parameter. If you set up Java support following the procedure described in Chapter 11, "Setting up Java support," on page 65, you have already done so. Make your customization changes to these copies of the files. Working with copies of the files ensures that you do not lose your changes if the sample JVM profiles are overwritten when program maintenance is applied.

A selection of relevant options that you can specify in JVM profiles, and their possible values, are documented in "JVM profiles: options and samples" on page 105. Also, if you want to enable Java security, "Protecting Java applications in CICS by using the Java security policy mechanism" on page 345 tells you which options to specify.

For single-use JVMs, that is, with a JVM profile that specifies the option REUSE=NO, instead of customizing the JVM profile, you can override the options in it, using the user-replaceable program DFHJVMAT. This program is called at JVM initialization if you specify INVOKE_DFHJVMAT=YES as an option on the JVM profile that you want to override. DFHJVMAT cannot be used with any type of JVM other than the single-use JVM. Typically, a JVM profile provides sufficient flexibility to configure a JVM as required. If you have to make unusual modifications, the *CICS Customization Guide* has more information about using DFHJVMAT. Continuous JVMs perform much better than single-use JVMs, so it is generally better to customize a JVM profile rather than using DFHJVMAT to override it.

Customizing DFHJVMCD

The JVM profile DFHJVMCD is reserved for use by CICS-supplied system programs, in particular the default request processor program, DFJIIRP, used by the CICS-supplied CIRP request processor transaction. CICS also uses DFHJVMCD to initialize and terminate the shared class cache.

Before you begin

DFHJVMCD must be set up correctly for your CICS region, but customize it only when necessary. DFHJVMCD can have an associated JVM properties file, but this is optional.

Make sure that you are working with a copy of DFHJVMCD in the z/OS UNIX directory that you specified on the **JVMPROFILEDIR** system initialization parameter, and not with the original file in its installation location.

About this task

The options that you can change are indicated in the text of DFHJVMCD. Do not make any other changes to the files.

For detailed information about the options in DFHJVMCD that you can change, and the purpose of changing them, see "Options for JVMs in a CICS environment" on page 111, and "JVM system properties" on page 119.

To customize DFHJVMCD:

Procedure

1. Open DFHJVMCD in a standard text editor.
2. If you have a shared class cache in your CICS region, and you want JVMs created using DFHJVMCD to use the shared class cache, change the CLASSCACHE option to CLASSCACHE=YES. The default, CLASSCACHE=NO, means that they are standalone JVMs.
3. If the values for the CICS_HOME and JAVA_HOME options do not match your installation directory for CICS files on z/OS UNIX and your IBM SDK for z/OS, Java Technology Edition installation location on z/OS UNIX, change them to the correct values.
4. To change the working directory on z/OS UNIX that is used by JVMs with the DFHJVMCD profile, change the WORK_DIR option to specify your preferred directory.
5. To change the names of the z/OS UNIX files to be used for `stderr`, `stdin`, and `stdout`, change the STDERR, STDIN, and STDOUT options.
6. To use an output redirection class to intercept and redirect output and messages from the JVM, use the USEROUTPUTCLASS option to specify the name of the class. Do not use this option in a production environment.
7. To tune the heap size for JVMs with the DFHJVMCD profile, to fit better with the needs of your applications, change the **-Xms** or **-Xmx** options.
8. If you have enterprise beans that use JDBC, add the relevant DB2 libraries and files as specified in the sample profile DFHJVMPR to the LIBPATH_SUFFIX and CLASSPATH_SUFFIX options in DFHJVMCD.
9. To the CLASSPATH_SUFFIX option in DFHJVMCD, add any classes, such as classes for utilities, that are required by your enterprise beans but are *not* included in the deployed JAR files for the enterprise beans. You do not have to add the deployed JAR files for enterprise beans to a class path.
10. Specify the system properties necessary to configure your JNDI nameserver; that is, the **-Dcom.ibm.cics.ejs.nameserver** system property and further system properties if you are using an LDAP nameserver.
11. Enable the Java security policy mechanism (the **-Djava.security.policy** system property) if required by your installation.
12. Save your copies of DFHJVMCD. Confirm that your customized copy of DFHJVMCD is in the z/OS UNIX directory that you specified on the **JVMPROFILEDIR** system initialization parameter. If you use an optional properties file, ensure that DFHJVMCD specifies the correct path to the file.

What to do next

Do not specify DFHJVMCD in PROGRAM resource definitions that you set up for your own applications. You might want to make similar customization changes to

a copy of the other CICS-supplied sample JVM profile, DFHJVMPR, for use by your applications.

Customizing DFHJVMPR

Follow this procedure if you want to keep the existing name for the JVM profile that you are customizing. When you keep the existing name for the file, you do not change the PROGRAM resource definitions for applications that are already set up to use that JVM profile.

About this task

If you want to change the name of the file, follow the procedure in "Creating your own JVM profiles" on page 103 instead. If you do change the name, applications will not use your new JVM profile unless you make changes to inform the applications of the new file name.

Make sure that you are working with a copy of the CICS-supplied sample JVM profile that you are customizing, and not the original file in its install location. "Customizing or creating JVM profiles" on page 100 explains where to place these copies.

Note that DFHJVMPR is the default if no JVM profile is specified in a PROGRAM resource definition, and it is used by sample programs. Make sure that all your Java programs that specify DFHJVMPR or no JVM profile in their PROGRAM resource definitions are suited to the changes that you are making. If they are not suitable, set up a new JVM profile based on DFHJVMPR with a different name.

Procedure

1. Open the JVM profile in a standard text editor and alter the options that you want to change, using the lists of options in "JVM profiles: options and samples" on page 105 for reference. Each parameter or property is specified on a separate line, and the parameter or property value is delimited by the end of the line. Follow the coding rules in "Rules for coding JVM profiles" on page 107.
2. If you want to enable Java security, you need to specify some options and set up one or more security policy files to define security properties for the JVM. "Protecting Java applications in CICS by using the Java security policy mechanism" on page 345 tells you which options to specify, how to set up a security policy file, and about the CICS-supplied sample security policy file `dfjejbpl.policy`, which defines security properties that are suitable for JVMs that are used by enterprise beans.
3. Save the customized JVM profile in the z/OS UNIX directory that is specified by the **JVMPROFILEDIR** system initialization parameter for your CICS region. CICS loads the JVM profiles from this directory. Confirm that CICS has read and execute access on z/OS UNIX for your JVM profile and the directory containing it.
4. If you have customized JVM profiles for a CICS region in which JVMs are already running, issue the CEMT PERFORM JVMPOOL PHASEOUT command for each JVM profile that is affected. This command marks all the existing JVMs with your chosen profile for deletion. The existing JVMs were built with the old version of the JVM profile. When each old JVM has finished running its current Java program, it stops. If requests are waiting, CICS starts a new JVM

in its place, or you can start new JVMs manually using the CEMT PERFORM JVMPOOL START command. The new JVMs use your new versions of the JVM profiles.

What to do next

"Enabling applications to use a JVM" on page 142 tells you how to set up applications to use a JVM profile and how to add the classes for the application to the class paths. If you are following a procedure to set up IIOP support or support for enterprise beans, and you do not yet have any specific applications to run, you can go to the procedure "Setting up the host system for IIOP" on page 181 or Chapter 19, "Setting up an EJB server," on page 239.

Creating your own JVM profiles

Follow this procedure if you want to create a JVM profile with a different name from the supplied sample files.

About this task

When you create a JVM profile with a new name, you specify the profile name in the PROGRAM resource definition for any applications that will use your new JVM profile.

To minimize administration, if you want to set up JVM profiles that are to be used by most of your applications, you might prefer to customize the supplied sample files and keep their existing names, following the procedure in "Customizing DFHJVMPR" on page 102. However, if you want to set up a JVM profile that is to be used by a small number of applications, or if you want to ensure that the default JVM profile, DFHJVMPR, is not affected by your modifications, you might want to create a file with a new name.

Create your own JVM profiles:

Procedure

1. Base your JVM profile on one of the supplied sample JVM profiles. "The CICS-supplied sample JVM profiles" on page 97 lists and describes these files. Note that the supplied sample JVM profile, DFHJVMPS, is not recommended for use with new Java applications and especially enterprise beans, so if you are creating a profile for a JVM in which these applications will run, do not base it on DFHJVMPS.
2. Create the JVM profile in a standard text editor, using the lists of options in "JVM profiles: options and samples" on page 105 for reference. Each parameter or property is specified on a separate line, and the parameter or property value is delimited by the end of the line. Follow the coding rules in "Rules for coding JVM profiles" on page 107.
3. If you want to enable Java security, you specify some options and set up one or more security policy files to define security properties for the JVM. "Protecting Java applications in CICS by using the Java security policy mechanism" on page 345 tells you which options to specify, how to set up a security policy file, and about the CICS-supplied sample security policy file `dfjejbpl.policy`, which defines security properties that are suitable for JVMs that are used by enterprise beans.
4. Give your JVM profile a suitable name. "JVM profiles" on page 93 states the rules for names and has some important information about case.

a. Do not give the JVM profile a name beginning with DFH, because these characters are reserved for use by CICS.
b. Note that the names of JVM profiles are case-sensitive. In particular, if you give a JVM profile a name that includes lowercase characters, you must be aware of your terminal UCTRAN setting when you enter the name on the CEDA command line or in another CICS transaction such as CEMT or CECI.

5. Store your JVM profile in the z/OS UNIX directory that is specified by the **JVMPROFILEDIR** system initialization parameter for your CICS region. CICS loads the JVM profiles from this directory.
6. Ensure that CICS has read and execute access on z/OS UNIX for your JVM profile and the directory containing it. See "Giving CICS regions permission to access z/OS UNIX directories and files" on page 66.
7. Specify the name of your JVM profile on the JVMPROFILE option of the PROGRAM resource definitions for the Java programs that you want to use this JVM profile. Alternatively, you can use a **CEMT SET PROGRAM JVMPROFILE** command or the equivalent **EXEC CICS** command to change the JVM profile from that specified on the installed PROGRAM resource definitions. Make sure that you use the same combination of uppercase and lowercase characters that is present in the z/OS UNIX file name of the JVM profile.
8. Add the classes that the Java programs use to the class paths specified in the JVM profile. "Enabling applications to use a JVM" on page 142 provides more information.
9. If you have created new JVM profiles to be used by Java programs that are already running in the CICS region, issue the **CEMT PERFORM JVMPOOL PHASEOUT** command for each JVM profile that the affected programs currently use. This command marks all the existing JVMs with your chosen profile for deletion. When each old JVM has finished running its current Java program, it stops. If requests are waiting, CICS starts a new JVM in its place, or you can start new JVMs manually using the **CEMT PERFORM JVMPOOL START** command. The new JVMs use your new JVM profiles.

What to do next

"Enabling applications to use a JVM" on page 142 tells you how to set up applications to use a JVM profile and how to add the classes for the application to the class paths. If you are following a procedure to set up IIOP support or support for enterprise beans, and you do not yet have any specific applications to run, you can go to the procedure "Setting up the host system for IIOP" on page 181 or Chapter 19, "Setting up an EJB server," on page 239.

Validation of JVM profile options

CICS carries out a number of checks on key options specified in your JVM profiles whenever you start JVMs. These checks enable the early detection of problems in your JVM setup.

CICS carries out checks relating to the following JVM profile options:

CICS_HOME

CICS checks the following points for this directory:

- The directory exists in z/OS UNIX.
- CICS has at least *read* permission to access the directory.
- The CICS_INSTALL_OK file is present in the directory, indicating a completed installation of the CICS files in this location in z/OS UNIX.

- The CICS_INSTALL_OK file contains the correct CICS version number, indicating that you are not inadvertently using the installed files from a previous CICS release (which might happen if you used a JVM profile created for a previous CICS release and did not update this option).

If any problems are found, CICS issues an error message and does not start the JVM. CICS also issues a warning message if you use the old CICS_DIRECTORY option instead of the CICS_HOME option.

JAVA_HOME

CICS checks similar points for this directory:

- The directory exists in z/OS UNIX.
- CICS has at least *read* permission to access the directory.
- The JDK_INSTALL_OK file is present in the directory, indicating a completed installation of the IBM SDK for z/OS, Java Technology Edition files in this location.
- The Java release number in the JDK_INSTALL_OK file is a version supported by CICS.

If any problems are found, CICS issues an error message and does not start the JVM.

Deprecated class path options: LIBPATH, CLASSPATH, TMPREFIX, and TMSUFFIX

A warning message is issued at JVM startup if one of these options is found in a JVM profile, to prompt you to transfer its value to an appropriate class path. The message advises on the correct option to use instead.

JVM profiles: options and samples

CICS provides sample JVM profiles that contain a selection of options for IBM JVMs that are used in a CICS environment. Some of these options are specific to the CICS environment and are not used for JVMs in other environments. Other options are standard or nonstandard Java options, which can be used for IBM JVMs in any environment.

You can specify any JVM option or system property in a JVM profile, and it is passed to the JVM. No central repository of all options and system properties for the JVM exists. Here are some recommended sources of information:

- The documentation for the IBM SDK for z/OS, Java Technology Edition, Version 6.
- The *IBM Developer Kit and Runtime Environment, Java Technology Edition Diagnostics Guide*, which is available to download from www.ibm.com/developerworks/java/jdk/diagnosis/. This guide documents system properties that are used for JVM trace and problem determination.

The Java class libraries include other system properties, and applications might have their own system properties. With all options or system properties available for the IBM JVM that are not specific to the CICS environment, regard the IBM JVM documentation as the primary source of information and the CICS documentation as a secondary source of information.

You can specify any JVM options and system properties in a JVM profile, and it is not necessary to have a separate JVM profile and JVM properties file. From CICS TS 4.1 onwards, sample JVM properties files are not supplied.

The summary table, Table 6, lists options used in the CICS-supplied sample JVM profiles and some further options that you might use to complete tasks described in the CICS documentation. The table indicates the default for each option if it is not specified in the JVM profile.

Version 6 of the SDK does not use a master JVM, so the JVM options for a JVM that uses the shared class cache and a JVM that does not are the same, except for the CLASSCACHE option.

Table 6. JVM options reference table for JVMs in a CICS environment

Option	Default	Comments
JVM type		
CLASSCACHE	NO	YES makes JVM use shared class cache, NO does not
REUSE	YES	YES makes continuous, NO makes single-use
Directories		
CICS_HOME	None	Required, sample profiles include this directory
JAVA_HOME	None	Required, sample profiles include this directory
WORK_DIR	/tmp	
Class paths		
CLASSPATH_SUFFIX	None	
LIBPATH_SUFFIX	None	
Timeout threshold		
IDLE_TIMEOUT	30 minutes	Continuous only
THREAD_TIMEOUT	60 seconds	Used in a JVM server profile only.
Further settings and facilities for the JVM		
JVMPROPS	None	
INVOKE_DFHJVMAT	NO	Single-use only
Initialization classes		
SETUP_CLASSES	None	Used in a JVM server profile only.
TERMINATION_CLASSES	None	Used in a JVM server profile only.
Storage heap sizes		
-Xms		For information on the **-Xms** default value, see the reference information at http://www.ibm.com/servers/eserver/zseries/software/java/reference
-Xmx		For information on the **-Xmx** default value, see the reference information at http://www.ibm.com/servers/eserver/zseries/software/java/reference

Table 6. JVM options reference table for JVMs in a CICS environment (continued)

Option	Default	Comments
Garbage collection threshold		
GC_HEAP_THRESHOLD	85%	Continuous only
Output from the JVM		
LEHEAPSTATS	NO	
STDERR	dfhjvmerr	
STDIN	dfhjvmin	
STDOUT	dfhjvmout	
USEROUTPUTCLASS	None	
Problem determination and application debugging		
JAVA_DUMP_OPTS	YES	
USE_LIBJVM_G	NO	
-Xdebug	NO	
PRINT_JVM_OPTIONS	NO	Set YES only temporarily
DISABLE_EJB	NO	Disables CICS provided EJB support

UNIX System Services environment variables

In addition to the JVM options and system properties that are used to construct the JVM, you can specify any UNIX System Services environment variables in a JVM profile. Any name and value pair in a JVM profile that is not recognized as a JVM option or system property is treated as a UNIX System Services environment variable and is exported. UNIX System Services environment variables specified in a JVM profile apply only to JVMs created with that profile.

The JAVA_DUMP_OPTS and JAVA_DUMP_TDUMP_PATTERN options used in the CICS-supplied sample JVM profiles are UNIX System Services environment variables. Another example is the TZ environment variable, which can be specified to change the time zone for the JVM.

UNIX System Services environment variables can be specified only in a JVM profile, not in a JVM properties file.

Rules for coding JVM profiles

You can edit JVM profiles using any standard text editor. Follow these rules when coding your JVM profiles.

- The name of a JVM profile can be up to 8 characters in length. The name of a JVM properties file can be any length, but, for ease of use, it is generally a short name with some similarity to the name of the JVM profile that references it.
- The name of a JVM profile or JVM properties file can be any name that is valid for a file in Unix System Services. Do not use a name beginning with DFH, because these characters are reserved for use by CICS.
- Because JVM profiles and JVM properties files are UNIX files, case is important. When you specify the name anywhere in CICS, you must enter it using the same combination of uppercase and lowercase characters that is present in the z/OS UNIX file name.
- Do not use quotes when specifying values for directories in a JVM profile.

- The CEDA panels accept mixed case input for the JVMPROFILE field irrespective of your terminal UCTRAN setting. However, you need to enter the name of a JVM profile in mixed case when you use CEDA from the command line or when you use another CICS transaction, such as CEMT or CECI, ensure that the terminal you use is correctly configured, with uppercase translation suppressed. You can use the CICS-supplied CEOT transaction to alter the uppercase translation status (UCTRAN) for your own terminal, for the current session only.

Follow these rules when coding JVM options or system properties:

Case sensitivity
All parameter keywords and operands are case-sensitive, and must be specified exactly as shown in "Options for JVMs in a CICS environment" on page 111 and "JVM system properties" on page 119.

Class path separator character
Use the **:** (colon) character to separate the directory paths that you specify on a class path option, such as CLASSPATH_SUFFIX.

Continuation
For JVM options or system properties, the value is delimited by the end of the line in the text file. If a system property or JVM option, such as a class path, that you are entering or editing is too long for an editor window, you can break the line to avoid scrolling. To continue on the next line, terminate the current line with the backslash and a blank (\ ) continuation characters, as in this example:

```
CLASSPATH_SUFFIX=/u/example/pathToJarOrZipFile/jarfile.jar:\
/u/example/pathToRootDirectoryForClasses
```

Comments
To add comments or to comment out an option instead of deleting it, begin each line of the comment with a # symbol. Comment lines are ignored when the file is read by the JVM launcher.

Blank lines are also ignored. You can use blank lines as a separator between options or groups of options.

Character escape sequences
In a property element string, you can code the escape sequences shown in Table 7

Table 7. Escape sequences

Escape sequence	Character value
\b	Backspace
\t	Horizontal tab
\n	Newline
\r	Carriage return
\"	Double quote
\'	Single quote
\\	Backslash
xxx	The character corresponding to the octal value *xxx*, where *xxx* is between 000 and 377
\uxxxx	The Unicode character with encoding *xxxx*, where *xxxx* is 1 - 4 hexadecimal digits. (See note below for more information.)

Note: Unicode \u escapes are distinct from the other escape types. The Unicode escape sequences are processed before the other escape sequences described in Table 7 on page 108. A Unicode escape is an alternative way to represent a character that might not be displayable on non-Unicode systems. The character escapes, however, can represent special characters in a way that prevents the usual interpretation of those characters.

Multiple instances of options
You can use each option only once in a JVM profile. If more than one instance of the same option is included in a JVM profile, the value for the last option found is used, and previous values are ignored.

Storage sizes
When specifying storage-related options in a JVM profile, specify storage sizes in multiples of 1024 bytes. Use the letter K to indicate kilobytes, the letter M to indicate megabytes, and the letter G to indicate gigabytes. For example, to specify 6 291 456 bytes as the initial size of the heap, code **-Xms** in one of the following ways:

```
-Xms6144K
-Xms6M
```

Unique JVM number and other identifying information in output file names

You can use symbols in output file names, in a JVM profile or JVM properties file, to help identify the output of each JVM.

&JVM_NUM; symbol

When you use the &JVM_NUM; symbol in a value in a JVM profile or JVM properties file, such as an output file name, CICS substitutes the unique JVM number for the symbol at runtime. The &APPLID; symbol is used in the same way to include the CICS region applid in a value. The symbols &DATE; and &TIME; 03 can be used to insert the data and time, respectively, in a value. As an alternative 03 for `stdout` and `stderr` JVM output files, you can use the **-generate** option to 03 include the unique JVM number, a time stamp, and the CICS region applid as part 03 of the file name.

At runtime, CICS replaces the &JVM_NUM; symbol with the JVM number, which 03 is unique to the JVM. Using the unique JVM number means that you can 03 distinguish the output of each JVM from the output of other JVMs in the CICS 03 region and locate the output files for a JVM that is currently running.The JVM number used in CICS is the same number that is used to identify the JVM in the z/OS UNIX environment, where it is known as the process id (PID) for the JVM.

The &JVM_NUM; symbol can be specified for any type of output from the JVM. The CICS-supplied sample JVM profiles demonstrate some possible uses for the &JVM_NUM; symbol:

- With the WORK_DIR option, as part of the name of the working directory for the CICS region. In this way, each CICS region has a different working directory. If you use this option, ensure that you have created all the relevant directories on z/OS UNIX and given the CICS regions read, write, and execute access to them.
- With the STDOUT and STDERR options, as part of the names of the z/OS UNIX files to be used for `stdout` (JVM output) and `stderr` (JVM error messages).

- With, for example, the JAVA_DUMP_TDUMP_PATTERN option, as part of the file name for TDUMPs from the JVM. Note that, in this context, CICS might have to modify the JVM number to conform to MVS data set naming standards.

Using the &JVM_NUM; symbol guarantees that each JVM in the CICS region has its own unique output files during the lifetime of the CICS region.

The JVM number used for the &JVM_NUM; symbol is the same as the JVM number used on the EXEC CICS INQUIRE JVM and CEMT INQUIRE JVM commands to identify individual JVMs. You can use these commands to browse the JVMs in the JVM pool, identify their JVM numbers, and see which JVM is currently assigned to which task. If a task has a problem, you can use the relevant JVM number to locate the output files for the task JVM.

&DATE; symbol

At runtime, CICS replaces the &DATE; symbol with the current date in the format Dyymmdd.

The &DATE; symbol can be specified for any type of output from the JVM including the WORK_DIR, STDOUT and STDERR, and JAVA_DUMP_TDUMP_PATTERN options.

&TIME; symbol

At runtime, CICS replaces the &TIME; symbol with the current time in the format Thhmmss.

The &TIME; symbol can be specified for any type of output from the JVM including the WORK_DIR, STDOUT and STDERR, and JAVA_DUMP_TDUMP_PATTERN options.

&APPLID; symbol

At runtime, CICS replaces the &APPLID; symbol with the applid of the CICS region. The applid is always in uppercase. You can specify the symbol for any type of output from the JVM.

Specifying the CICS region applid is helpful if you are using the same JVM profiles for multiple CICS regions. By using the &APPLID; symbol, you can share the same set of JVM profiles across CICS regions, and still have region-specific output destinations or working directories. You can use the symbol in these ways:

- With the WORK_DIR option, as part of the name of the working directory for the CICS region. In this way, each CICS region has a different working directory. If you use this option, ensure that you have created all the relevant directories on z/OS UNIX and given the CICS regions read, write, and execute access to them.
- With the STDOUT and STDERR options, as part of the names of the z/OS UNIX files to be used for `stdout` (JVM output) and `stderr` (JVM error messages), in combination with the &JVM_NUM; option.
- With the JAVA_DUMP_TDUMP_PATTERN option, as part of the file name for TDUMPs from the JVM, again with the &JVM_NUM; option.

Using the &APPLID; symbol with the &JVM_NUM; symbol guarantees that output files are unique not only in the CICS region but also across multiple CICS regions.

-generate option

The **-generate** option can be specified for the names of the z/OS UNIX files to be used for `stdout` (JVM output) and `stderr` (JVM error messages).

The **-generate** option appends the unique JVM number (as with the &JVM_NUM; symbol), the CICS region applid (as with the &APPLID; symbol), and also some additional qualifiers, to the file name that you have specified for the STDOUT or STDERR option in the JVM profile. Specify the qualifiers in this order:

- The applid of the CICS region
- The unique JVM number
- The time when the file was created (at JVM startup), in the form `yydddhhmmss`
- The suffix `.txt`, a literal string suffix to indicate that the file contains readable data

A typical output file name for a `stdout` file created with the **-generate** option might be like this:

```
dfhjvmout.IYK2ZIK1.0067240142.06004165342.txt
```

where:

- `dfhjvmout` is the fixed part of the file name.
- `IYK2ZIK1` is the applid of the CICS region.
- `0067240142` is the unique JVM number.
- `06004165342` is the time stamp showing when the JVM was created.
- `.txt` is the file suffix.

When you use the **-generate** option, the &APPLID; and &JVM_NUM; options are not required in the file name, because **-generate** supplies these pieces of information automatically.

Because the **-generate** option includes the JVM number, the resulting output file is unique to the JVM and can be matched with the JVM number identified from the EXEC CICS INQUIRE JVM and CEMT INQUIRE JVM commands. Because it includes the CICS region applid, it is also unique across multiple CICS regions.

Options for JVMs in a CICS environment

The options in a JVM profile are used by CICS, the IBM SDK for z/OS, Java Technology Edition, or UNIX System Services, to start up JVMs.

Note these points when you specify the options:

- Specify each option according to the coding rules described in "Rules for coding JVM profiles" on page 107.
- Some options in a JVM profile for CICS take the form of a keyword and value separated by an = sign.
- Other options are specified with the option immediately followed by the value, and no = sign.
- Any option that begins with a hyphen (-) character is either a Java standard option or a Java nonstandard option, and is passed to the JVM without any parsing by CICS.

- You can also specify any UNIX System Services environment variables in a JVM profile. Any name and value pair in a JVM profile that is not recognized as a JVM option or system property is treated as a UNIX System Services environment variable and is exported.
- The JVM system properties, which begin with **-D**, can also be specified in a JVM profile. They are listed separately in "JVM system properties" on page 119.

The following symbols can be used in the values of options in a JVM profile or JVM properties file for CICS, as demonstrated in the CICS-supplied sample JVM profiles.

&APPLID;
When you use this symbol, the applid of the CICS region is substituted at runtime. In this way, you can use the same profile or properties file for all regions, and still have region-specific working directories or output destinations.

&DATE;
When you use this symbol, the symbol is replaced with the current date in the format Dyymmdd at runtime. The &DATE; symbol can be specified for any type of output from the JVM, including the WORK_DIR, STDOUT and STDERR, and JAVA_DUMP_TDUMP_PATTERN options.

&JVM_NUM;
When you use this symbol, the unique number of the JVM is substituted at runtime. Use this symbol to create unique output or dump files for each JVM.

The CICS region applid and unique JVM number are included automatically in the names of the z/OS UNIX files to be used for `stdout` (JVM output) and `stderr` (JVM error messages), if you specify the **-generate** option for the STDOUT or STDERR option in the JVM profile. "Unique JVM number and other identifying information in output file names" on page 109 has more information about the &APPLID; and &JVM_NUM; symbols and the **-generate** option.

In the list of options that follows, any default value indicated for an option, is the default value that CICS uses when the option is *not* specified in the JVM profile. However, some or all of the CICS-supplied sample JVM profiles might specify a value that is different from the default value.

&TIME;
When you use this symbol, the symbol is replaced with the current time in the format Thhmmss at runtime. The &TIME; symbol can be specified for any type of output from the JVM including the WORK_DIR, STDOUT and STDERR, and JAVA_DUMP_TDUMP_PATTERN options.

CICS_HOME=/usr/lpp/cicsts/cicsts41/

Specifies the path for the home directory for CICS files on z/OS UNIX. The value of this option is used to build the base library path and the base class path for the JVM. By default, this directory is `/usr/lpp/cicsts/`*`cicsts41`*`/`, where *cicsts41* is defined by the USSDIR installation parameter when you installed CICS TS for z/OS, Version 4.1.

CLASSCACHE={YES,NO}
Specifies whether this JVM is to use the shared class cache described in "The shared class cache" on page 90. The default for CLASSCACHE, if it is not specified, is NO. The IBM SDK for z/OS, Version 6 for Java has no master JVM, so the options for a JVM that uses the shared class cache are the same as for a JVM that does not.

CLASSPATH_PREFIX, CLASSPATH_SUFFIX=*class_pathnames*

The standard class path specifies directory paths, JAR files, and zip files to be searched by the JVM for application classes and resources. You can specify entries on separate lines by using a \ (backslash) at the end of each line that is to be continued.

CLASSPATH_PREFIX adds class path entries to the beginning of the standard class path, and CLASSPATH_SUFFIX adds them to the end of the standard class path.

With Version 6 of the SDK, all application classes are placed on the standard class path, and they are all eligible to be loaded into the shared class cache.

Use the CLASSPATH_PREFIX option with care. Classes in CLASSPATH_PREFIX take precedence over classes of the same name supplied by CICS and the Java runtime and the wrong classes might be loaded.

CICS builds a base class path for the JVM using the `/lib` subdirectories of the directories specified by the CICS_HOME and JAVA_HOME options in the JVM profile. This base class path contains the JAR files supplied by CICS and by the JVM. It is not visible in the JVM profile. You do not specify these files again in the class paths in the JVM profile.

DISABLE_EJB={YES|NO}

Specifies whether the EJB support provided by CICS is included by the JVM at startup. The default is NO. If specified then the following CICS and Java supplied JAR files will not be included in the class path: `dfjorb.jar`, `dfjcont.jar`, `dfjjts.jar`, `dfjadjr.jar`, `omgcos.jar`, `dfjname.jar`, `websphere.jar`, `dfjcci.jar`, `dfjejbdd.jar`, `ejb20.jar`, `jta_spec.jar`, `connector.jar`.

DISPLAY_JAVA_VERSION=

If this option is set to YES, whenever a JVM is started by an application CICS writes message DFHSJ0901 to the MSGUSER log, showing the version and build of the IBM Software Developer Kit for z/OS, Java Technology Edition that is in use.

04 **GC_HEAP_THRESHOLD=**

04 Specifies the heap utilization limit for the JVM heap. When this percentage of
04 the storage in the active part of the heap is used, CICS schedules a garbage
04 collection. CICS checks heap utilization after every Java program execution. If
04 the limit is reached, the garbage collection transaction CJGC is scheduled to
04 run in the JVM immediately after the current use of the JVM ends.

04 The default heap utilization limit is 85 (85%). The minimum is 50. The
04 maximum if you want CICS to schedule garbage collections is 99. If you
04 specify a heap utilization limit of 100, CICS never schedules garbage
04 collections, and all garbage collections result from allocation failures while
04 applications are running.

04 This option does not apply to a single-use JVM, which is destroyed after a
04 single Java program has run in it.

IDLE_TIMEOUT={30|*number***}**

Specifies the timeout threshold, in minutes, for JVMs with this JVM profile. If a JVM is inactive, that is, it is not used by an application for the specified amount of time, it becomes eligible for automatic termination. The next time CICS checks on the idle JVMs, if the JVM is still inactive, the JVM and its J8 or J9 TCB might be destroyed. (CICS does not immediately stop all the JVMs that have timed out; they are stopped progressively over a period of time.)

The default timeout threshold is 30 minutes, and the maximum is 10 080 minutes (7 days). You can also specify a timeout threshold of zero, so that JVMs with that profile are never stopped automatically because of inactivity. JVMs with a timeout threshold of zero might be stopped if they are selected for stealing or mismatching, or if MVS storage becomes constrained. If you specify an unacceptable value, CICS uses the default instead.

This option is irrelevant for a single-use JVM, which is destroyed after a single Java program has run in it.

INVOKE_DFHJVMAT={NO|YES}
Specifies whether the user replaceable module, DFHJVMAT, is called before creating a new JVM. DFHJVMAT can be used only for single-use JVMs; that is, where the option REUSE=NO is specified in the JVM profile. INVOKE_DFHJVMAT is ignored for a continuous JVM, with REUSE=YES in the JVM profile.

JAVA_DUMP_OPTS=
03 A UNIX System Services environment variable. Specifies a set of Java dump options to obtain diagnostics for an abend in the JVM. Information about Java dump options can be found in the *IBM Developer Kit and Runtime Environment, Java Technology Edition Diagnostics Guide*, which is available to download from www.ibm.com/developerworks/java/jdk/diagnosis/.

JAVA_DUMP_TDUMP_PATTERN=
03 A UNIX System Services environment variable. Specifies the file name to be used for transaction dumps (TDUMPs) from the JVM. Java TDUMPs are written to a data set destination in the event of a JVM abend. You can use the symbols &APPLID; (CICS region applid) and &JVM_NUM; (unique JVM number) in this value, as shown in the CICS-supplied sample JVM profiles, to create unique dump file names for each JVM.

When you use the &JVM_NUM; symbol here, CICS might have to modify the JVM number to conform to MVS data set naming standards. The number is formatted as an 8-digit hexadecimal value. If the first character is numeric, it has to be changed: 0 is changed to G, 1 is changed to H, and so on through 9 which is changed to P.

JAVA_HOME=/usr/lpp/java/J6.0/ (note case-sensitivity java not Java)
Specifies the installation location for IBM SDK for z/OS, Java Technology Edition in z/OS UNIX. This location contains subdirectories and JAR files required for Java support.

The CICS-supplied sample JVM profiles contain a path that was generated using the JAVADIR parameter in the DFHISTAR CICS installation job. The default for the JAVADIR parameter is java/J6.0/, which produces a JAVA_HOME setting in the JVM profiles of /usr/lpp/java/J6.0/, which is the default install location for the IBM SDK for z/OS, Java Technology Edition.

Use JAVA_HOME to specify the installation location of Version 6 of the SDK for Java support in the CICS region. The default installation location for Version 6 of the SDK is /usr/lpp/java/J6.0/.

JVMPROPS=*path/file_name*
Specifies the path and name of an optional JVM properties file, which is a z/OS UNIX file that can be used to contain system properties for this JVM. "JVM system properties" on page 119 tells you what you can specify in a JVM properties file.

LEHEAPSTATS={YES|NO}
Specifies whether statistics are to be collected for the amount of Language

Environment heap storage that is used by the JVM. The default is NO. The statistics appear as the field "Peak Language Environment heap storage used" in the JVM Profile statistics. Collecting these statistics affects the performance of the JVM, so you must specify LEHEAPSTATS=YES only while you are tuning the heap sizes for your JVMs. "Java applications using a Java virtual machine (JVM): improving performance" in the *CICS Performance Guide* explains this process. In a production environment, specify LEHEAPSTATS=NO.

LIBPATH_PREFIX, LIBPATH_SUFFIX=*pathnames*
Specifies directory paths to be searched for native C dynamic link library (DLL) files that are used by the JVM; that have the extension `.so` in z/OS UNIX, including those required to run the JVM and additional native libraries loaded by application code or services.

The base library path for the JVM is built automatically using the directories specified by the CICS_HOME and JAVA_HOME options in the JVM profile. The base library path is not visible in the JVM profile. It includes all the DLL files required to run the JVM and the native libraries used by CICS.

You can extend the library path using the LIBPATH_SUFFIX option. This option adds directories to the end of the library path, after the base library path. Use this option to specify directories containing any additional native libraries that are used by your applications or by any services that are not included in the standard JVM setup for CICS. For example, the additional native libraries might include the DLL files needed to use the DB2 JDBC drivers.

The LIBPATH_PREFIX option adds directories to the beginning of the library path, before the base library path. Use this option with care, because, if DLL files in the specified directories have the same name as DLL files on the base library path, they are loaded in place of the CICS-supplied files. You might need to do this for problem determination purposes.

Any DLL files that you include on the library path for use by your applications must be compiled and linked with the XPLink option for optimum performance. The DLL files supplied on the base library path and the DLL files used by services such as the DB2 JDBC drivers are built with the XPLink option.

PRINT_JVM_OPTIONS=
If set to YES, whenever a JVM is started, all the options passed to the JVM at startup are printed to SYSPRINT. The output is produced every time a JVM is started with this option in its profile. So you add the option to the appropriate JVM profile, wait for a JVM to be started with the profile, or issue the PERFORM JVMPOOL command to manually start a JVM with the profile, and then immediately remove the option from the profile. You can use this option to check the contents of the class paths for a particular JVM profile, including the base library path and the base class path built by CICS, which are not visible in the JVM profile.

REUSE={YES|NO}
Specifies whether the JVM is reusable or not reusable.

- REUSE=YES, which is the default, creates a JVM that is reused many times by Java applications. This type of JVM is known as a continuous JVM.
- REUSE=NO creates a JVM that is not reused, but instead is destroyed after a single Java program has run in it. This type of JVM is known as a single-use JVM.

STDERR={dfhjvmerr|*file_name*} [-generate]
Specifies the name of the z/OS UNIX file to be used for `stderr`. If the file does not exist, it is created in the directory specified by the WORK_DIR option. If the file already exists, output is appended to the end of the file. When the JVM stops, if the `stderr` file is empty and it has been created for the specific JVM, it is deleted. Otherwise, the file is kept.

The default name for the file is `dfhjvmerr`. Note that for a fixed file name, the output from multiple JVMs is appended to the named file, and the output is interleaved. To create unique output files for each JVM, either use the &JVM_NUM; and &APPLID; symbols in your file name, as demonstrated in the CICS-supplied sample JVM profiles, or specify the **-generate** option. The **-generate** option appends the unique JVM number, the applid of the CICS region, and additional identifying information to the file name. **-generate** must be preceded by one blank. "Unique JVM number and other identifying information in output file names" on page 109 has more information about the &APPLID; and &JVM_NUM; symbols and the **-generate** option.

If you specify the USEROUTPUTCLASS option on a JVM profile, the Java class named on that option handles the System.err requests instead. The z/OS UNIX file named by the STDERR option might still be used if the class named by the USEROUTPUTCLASS option cannot write data to its intended destination; as is the case when you use the CICS-supplied sample class com.ibm.cics.samples.SJMergedStream. You can also use the file if output is directed to it for any other reason by a class named by the USEROUTPUTCLASS option.

STDIN={dfhjvmin|*file_name*}
Specifies the name of the z/OS UNIX file to be used for `stdin`. If the file does not exist, it is created in the directory specified by the WORK_DIR option.

STDOUT={dfhjvmout|*file_name*} [-generate]
Specifies the name of the z/OS UNIX file that is to be used for output to the `stdout` file. If the file does not exist, it is created in the directory specified by the WORK_DIR option. If the file already exists, output is appended to the end of the file. When the JVM stops, if the `stdout` file is empty and it has been generated for the specific JVM, it is deleted. Otherwise, the file is kept.

The default name for the file is `dfhjvmout`. Note that for a fixed file name, the output from multiple JVMs is appended to the named file, and the output is interleaved. As with the STDERR option, to create unique output files for each JVM, either use the &JVM_NUM; and &APPLID; symbols in your file name, as demonstrated in the CICS-supplied sample JVM profiles, or specify the **-generate** option, which must be preceded by one blank.

If you specify the USEROUTPUTCLASS option on a JVM profile, the Java class named on that option handles the System.out requests instead. The z/OS UNIX file named by the STDOUT option might still be used if the class named by the USEROUTPUTCLASS option cannot write data to its intended destination; for example, when you use the CICS-supplied sample class com.ibm.cics.samples.SJMergedStream. You can also use the file if output is directed to it for any other reason by a class named by the USEROUTPUTCLASS option.

USE_LIBJVM_G={YES|NO}
Specifying USE_LIBJVM_G=YES enables the debug libraries for the JVM. If you specify NO or omit the option, the optimized libraries are used. When you are using the debug libraries, extra checking takes place, so performance is greatly reduced compared to the use of the normal, optimized libraries.

Use this option only under the direction of IBM service, because these libraries are not shipped with the JVM.

USEROUTPUTCLASS={*classname***}**
Specifies the fully qualified name of a Java class that intercepts the output from the JVM and messages from JVM internals. You can use this Java class to redirect the output and messages from your JVMs, and you can add time stamps and headers to the output records.

"Controlling the location for JVM stdout, stderr and dump output" on page 156 has more information about what this class can do and about the sample classes com.ibm.cics.samples.SJMergedStream and com.ibm.cics.samples.SJTaskStream provided by CICS.

If you do not specify the USEROUTPUTCLASS option in a JVM profile, or if you specify it as null, the z/OS UNIX files named by the STDOUT and STDERR options in the profile are used for output from the JVM. If you specify the USEROUTPUTCLASS option in a JVM profile, the z/OS UNIX files named by the STDOUT and STDERR options in the profile might be used if the class named by the USEROUTPUTCLASS option cannot write data to its intended destination.

Specifying the USEROUTPUTCLASS option has a negative effect on the performance of JVMs. For best performance in a production environment, do not use this option. However, specifying the USEROUTPUTCLASS option during application development enables developers using the same CICS region to keep their own JVM output separate and to direct it to an identifiable destination.

WORK_DIR={.|*directory_name***}**
Specifies the working directory on z/OS UNIX that the CICS region uses for Java-related activities. The CICS JVM interface uses this directory when creating the `stdin`, `stdout`, and `stderr` files. A period (.) is defined in the CICS-supplied JVM profiles, indicating that the home directory of the CICS region user ID (that is, the z/OS UNIX directory `/u/`*`CICS region userid`*) is to be used as the working directory. This directory can be created during CICS installation. If the CICS region user ID does not have this home directory, or if WORK_DIR is omitted, `/tmp` is used as the z/OS UNIX directory name.

You can create a subdirectory in this z/OS UNIX directory to hold the output files, by specifying the subdirectory name after the period. For example, if you specify:

```
WORK_DIR=./javaoutput
```

the output files from all the JVMs in that CICS region are created in the subdirectory `javaoutput` in the home directory of the CICS region user ID.

If you do not want to use this home directory as the working directory for Java-related activities, or if your CICS regions are sharing the same z/OS user identifier (UID) and so have the same home directory, you can create a different working directory for each CICS region. You specify a directory name that uses the &APPLID; symbol, for which CICS substitutes the actual CICS region applid. So you can have a unique working directory for each region, even if all the CICS regions share the same set of JVM profiles. For example, if you specify:

```
WORK_DIR=/u/&APPLID;/javaoutput
```

each CICS region using that JVM profile has its own working directory. Ensure that you have created all the relevant directories on z/OS UNIX and given the CICS regions read, write, and execute access to them.

You can also specify a fixed name for the working directory, again ensuring that you have created the relevant directory on z/OS UNIX and given the CICS regions the correct access. When you use a fixed name for the working directory, the output files from all the JVMs in the CICS regions that share the JVM profile are created in that directory. If you have also used fixed file names for your output files, the output from all the JVMs in those CICS regions is appended to the same z/OS UNIX files. To avoid appending to the same files, use the &JVM_NUM; symbol, the &APPLID; symbol, or the **-generate** option with the appropriate JVM profile options to produce unique output and dump files for each JVM in each CICS region.

You are recommended not to define your working directories in the CICS directory on z/OS UNIX, which is the home directory for CICS files (the directory specified by the CICS_HOME option in the JVM profile, which by default is `/usr/lpp/cicsts/cicsts41/`).

You can also use the option USEROUTPUTCLASS to name a Java class that intercepts, redirects, and formats the `stderr` and `stdout` output from a JVM. The CICS-supplied sample classes for output redirection use the directory specified by WORK_DIR in some circumstances.

-Xdebug
Specifies whether or not debugging support is to be enabled in the JVM.

For more information, see "Debugging an application that is running in a CICS JVM" on page 160. See also the Java Platform Debugger Architecture (JPDA) description at `http://java.sun.com/products/jpda/doc/`.

To ensure a clean end to the debug session, specify REUSE=NO when debugging support is enabled.

-Xms
Specifies the initial size of the heap.

Specify *size* as a number of kilobytes or megabytes. See "Specifying storage sizes." The CICS-supplied sample JVM profiles specify **-Xms16M**. For information on the default value, see http://www.ibm.com/servers/eserver/zseries/software/java/reference

-Xmx
Specifies the maximum size of the heap. This fixed amount of storage is allocated by the JVM during JVM initialization.

Specify *size* as a number of kilobytes or megabytes. See "Specifying storage sizes." The CICS-supplied sample JVM profiles specify **-Xmx16M**.

Note:
For information on the -Xmso, -Xiss, and -Xss JVM options and all the default values, see the reference information at http://www.ibm.com/servers/eserver/zseries/software/java/reference

Specifying storage sizes

Specify storage sizes in multiples of 1024 bytes. Use the letter K to indicate kilobytes, the letter M to indicate megabytes, and the letter G to indicate gigabytes. For example, to specify 6 291 456 bytes as the initial size of the heap, code **-Xms** in one of the following ways:

```
-Xms6144K
-Xms6M
```

JVM system properties

System properties are key name and value pairs that contain basic information about the JVM and its environment, such as the operating system in which the application is running. Consider these system properties that are particularly relevant for JVMs in a CICS environment, including some that are defined by CICS.

You can specify JVM system properties either in a JVM profile or in an optional JVM properties file, referenced by the JVMPROPS option in JVM profiles. You might want to set up a separate JVM properties file if you have system properties that are common to more than one type of JVM in your CICS region, or if you need to separate out sensitive information and apply extra security controls to it. CICS passes all the system properties in a JVM profile or JVM properties file to the JVM unchanged.

The JVM can support a much wider range of system properties than those documented here. "JVM profiles: options and samples" on page 105 lists some recommended sources of information about system properties. The list below includes a selection of relevant system properties and describes how you can use them in a CICS environment. The system properties that begin **-Dcom.ibm.cics** are specific to the IBM JVM in a CICS environment. Those that begin **-Dcom.ibm** or **-Djava** are used more widely.

Specify each system property according to the coding rules described in "Rules for coding JVM profiles" on page 107.

CAUTION:

If you use JVM properties files, ensure that the files are secure, with update authority restricted to system administrators, if they are used to define sensitive JVM configuration options, such as the security policy file.

In particular, if you specify that a secure LDAP server is to be used, by coding -Djava.naming.security.authentication in the JVM properties files, you must also specify -Djava.naming.security.principal and -Djava.naming.security.credentials . These properties hold the user ID and password that CICS requires to access the secure LDAP service, so you must give particular attention to the access controls in force at your installation for the JVM properties files, and any other copies of this information that you have.

-Dcom.ibm.cics.datasource.path=
Specifies the name and subContext of a CICS-compatible DataSource that you have deployed to generate JDBC connections for Java applications in CICS that access DB2. The *CICS DB2 Guide* has more information about this.

-Dcom.ibm.cics.ejs.nameserver=
Specifies the URL and TCP/IP port number of the name server that you use for JNDI references.

- For an LDAP name server, specify something like this:

```
-Dcom.ibm.cics.ejs.nameserver=ldap://myldserv.hursley.ibm.com:389
```

myldserv.hursley.ibm.com is the URL of the name server and 389 is the port number on which it is configured to listen. Your LDAP administrator can supply the correct URL and port number.

- For a standard COS Naming Directory Server, specify something like:

  ```
  -Dcom.ibm.cics.ejs.nameserver=iiop://mycsserv.hursley.ibm.com:900
  ```

  The relevant administrator in your organization can supply the correct name and port number.

  If you are using the COS Naming Directory Server supplied with WebSphere Application ServerVersion 5 or later, specify something like this:

```
-Dcom.ibm.cics.ejs.nameserver=iiop://mycsserv.hursley.ibm.com:2809/domain/legacyRoot
```

  because, in WebSphere Application Server, these conditions apply:

  – The default TCP/IP port used by the COS Naming Directory Server is 2809.

  – CICS objects must be published to a specially architected location in the WebSphere naming structure called "domain/legacyRoot". CICS publishes objects to a context defined by the JNDIPREFIX option of the CORBASERVER definition, where the JNDI prefix is a relative path. If you do not specify the /domain/legacyRoot path from the root node of the name space, CICS tries to publish objects to the JNDI prefix location relative to the root node itself. With the COS Naming Directory Server supplied with WebSphere Application Server, this attempt fails.

If you are using a COS naming service, and you have chosen to specify it in **-Djava.naming.provider.url**, do not specify it again here.

-Dcom.ibm.cics.ejs.loadjndiproperties=
Sets up a file called jndi.properties to contain JNDI nameserver configuration properties that are common across a set of CICS regions. By default, CICS does not attempt to locate a jndi.properties file. Include the following system property to cause CICS to load jndi.properties for this JVM:

```
-Dcom.ibm.cics.ejs.loadjndiproperties=true
```

Place the directory containing the jndi.properties file on the standard class path in the JVM profile, in all the relevant JVM profiles, for all the regions that you want to share the same nameserver settings.

-Dcom.ibm.cics.iiop.CSIv2Enabled=true
Enables CICS support for the Common Secure Interoperability Version 2 (CSIv2) protocol for identity assertion. To activate this support, specify this system property in all of the JVM profiles or JVM properties files used in the CICS region. This support is required if a CICS CorbaServer needs to support asserted identity authentication for IIOP messages sent from WebSphere Application Server for z/OS Version 6.1 or later. (Release 6.1.0.13 or later of WebSphere Application Server for z/OS is required to support this function.)

-Dcom.ibm.cics.soap.validation.local.CCSID=
Specifies the local code page to use when validating SOAP messages if validation is enabled for a CICS WEBSERVICE resource. If you do not specify a local CCSID, the default USS code page for your installation is assumed when validating the SOAP message.

-Dcom.ibm.websphere.naming.jndicache.cacheobject={populated \|none}
Turns the JNDI cache on or off. The JNDI cache stores the results of JNDI

lookups in local storage, so that, if an application does the same lookup twice, perhaps in different tasks, the results are already available. Note that the cache has these characteristics:

- Is JVM-specific. That is, a separate cache for each JVM.
- Works with an IBM JNDI name server only.
- Stores only object references and not other things, such as DataSources.

populated
The JNDI cache is active.

none The JNDI cache is not used.

-Dcom.ibm.websphere.naming.jndicache.maxcachelife={20 *vmins***}**
Specifies, in minutes, the "time to live" of the JNDI cache. If the cache is accessed after this time is exceeded, the entire cache is flushed of its contents.

See also the **-Dcom.ibm.websphere.naming.jndicache.cacheobject** property.

-Dcom.ibm.ws.naming.ldap.containerdn=
Specifies the Container Distinguished Name for the LDAP name server. For example:

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

Your LDAP administrator can supply the correct value for your installation.

The Container Distinguished Name is the root of the system namespace.

This property is not required if you specify a COS naming service.

-Dcom.ibm.ws.naming.ldap.noderootrdn=
Specifies the Noderoot Relative Distinguished Name for the LDAP name server. For example:

```
-Dcom.ibm.ws.naming.ldap.noderootrdn=ibm-wsnName=legacyroot,
ibm-wsnName=PLEX2,ibm-wsnName=domainRoots
```

Your LDAP administrator can supply the correct value.

This property is not required if you specify a COS naming service.

-Djava.naming.security.authentication=
Specifies the type of security authentication in use for naming operations. You might need this property if you are using an LDAP name server.

CICS must have write access into the LDAP namespace. If the LDAP service is set up securely, these three properties, authentication, credentials and principal, are required. If the LDAP service is set up so that any user can write to it, these three are not needed. Your LDAP administrator can tell you whether you need to include these properties in your JVM profile or optional JVM properties file.

Simple is the only value for this property that is supported by CICS. Specifying **-Djava.naming.security.authentication=simple** indicates that the LDAP name server is running in secure mode.

Important:

If you do specify this property, you must also specify **-Djava.naming.security.principal** and **-Djava.naming.security.credentials**.

Because these properties specify the user ID and password that CICS requires to access the secure LDAP service, give particular attention to the file permissions controlling access to all the files containing these system properties.

-Djava.naming.security.credentials=
Specifies the password required for the **principal**, which is described in java.naming.security.principal, to access to the LDAP name server.

This property is required if you specified **-Djava.naming.security.authentication=simple**. Your LDAP administrator provides the value that you specify. Here is an example:

```
-Djava.naming.security.credentials=secret
```

.

-Djava.naming.security.principal=
Specifies the **principal** required for access to the LDAP name server.

This property is required if you specified **-Djava.naming.security.authentication=simple**. Your LDAP administrator provides the value that you specify. For example, **-Djava.naming.security.principal=cn=CICSUser,c=uk** .

-Djava.security.manager={default| "" | |*other_security_manager***}**
Specifies the Java security manager to be enabled for the JVM. To enable the default Java security manager, include this system property in one of the following formats:

```
-Djava.security.manager=default
```

or

```
-Djava.security.manager=""
```

or

```
-Djava.security.manager=
```

All these statements enable the default security manager. If you do not include the **-Djava.security.manager** system property in your JVM profile, the JVM runs without Java security enabled. To disable Java security for a JVM, comment out this system property.

-Djava.security.policy=
Describes the location of additional policy files that you want the security manager to use to determine the security policy for the JVM. A default policy file is provided with the JVM in `/usr/lpp/java/J6.0/lib/security/java.policy`, where the `java/J6.0` subdirectory names are the default values when you install the IBM SDK for z/OS, Java Technology Edition. The default security manager always uses this default policy file to determine the security policy for the JVM, and you can use the **-Djava.security.policy** system property to specify any additional policy files that you want the security manager to take into account as well as the default policy file.

To enable CICS Java applications and enterprise beans to run successfully when Java security is active, specify, as a minimum, an additional policy file that gives CICS the permissions it needs to run the enterprise beans container and gives applications the permissions outlined in the *Enterprise JavaBeans* specification, Version 1. The CICS-supplied enterprise beans policy file, `dfjejbpl.policy`, contains the permissions that you need for this purpose. To specify this policy file, include the system property:

```
-Djava.security.policy=/usr/lpp/cicsts/cicsts41/lib/security/dfjejbpl.policy
```

where *cicsts41* is your chosen value for the USSDIR installation parameter that you defined when you installed CICS TS.

"Protecting Java applications in CICS by using the Java security policy mechanism" on page 345 has more information about specifying security policy files and about `dfjejbpl.policy`.

-Djdbc.drivers=

Specifies one or more JDBC drivers. Setting the driver names as a system property is an alternative to the Java application itself loading the drivers using the **Class.forName("driver_name");** command. Separate each driver name in a list by a : (colon).

To specify the DB2-supplied JDBC drivers, set the system property:

```
-Djdbc.drivers=COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver
```

This common name works for all levels of the JDBC driver supplied by DB2, including the DB2 Universal JDBC Driver.

Using JDBC and SQLJ to access DB2 data from Java programs and enterprise beans written for CICS has more information about using JDBC.

DFHJVMPR, JVM profile

The JVM profile DFHJVMPR is a sample JVM profile for JVMs that use the shared class cache. The file is used as the default if you do not specify a JVM profile in the PROGRAM resource definition of a Java program.

JVM options in DFHJVMPR JVM profile

```
############################################################
# JVMProfile: DFHJVMPR
############################################################
#
# This is a sample CICS JVM Profile for JVMs that use the
# Shared Class Cache. This profile is the default profile
# for use with all CICS PROGRAMs defined with JVM(YES)
# unless specified otherwise.
#
######
#
# Symbol Substitution:
#
# If you use any of the following variable symbols in any of
# the variables below, they will be replaced with appropriate
# values. The variable symbols may be specified in upper or
# lower case.
#
#  Symbol       Replacement value
#  ------       -----------------
#
# &APPLID;      The APPLID of the CICS region
# &JVM_NUM;     The Unix Systems Services Process ID (pid)
#               of the JVM. This is guaranteed to be unique
# &DATE;        The current date in the format Dyymmdd
# &TIME;        The current time in the format Thhmmss
#
# With this substitution, for example
#     STDERR=dfhjvmerr.&APPLID;.&JVM_NUM;.&DATE;.&TIME;
# becomes
#     STDERR=dfhjvmerr.ABCDEF.0084214386.D081220.T135323
#
######
```

```
#
# ********* CICS-specific parameters ***********
#
CICS_HOME=/usr/lpp/cicsts/cicsts41
JAVA_HOME=/usr/lpp/java/java6_31/J6.0
WORK_DIR=.
REUSE=YES
CLASSCACHE=YES
#
# A JVM Properties file can optionally be used by supplying its
# full path and file name on the JVMPROPS option.
# See "Java Applications in CICS" for more information on JVM
# Properties Files.
#
# JVMPROPS=/u/example/pathToProperties/myJVMProps.data
#
STDIN=dfhjvmin
STDOUT=dfhjvmout
STDERR=dfhjvmerr
#
DISPLAY_JAVA_VERSION=NO
# Percentage of heap full which will trigger a scheduled GC
GC_HEAP_THRESHOLD=85
# Timeout value in minutes after which a JVM and its TCB become
# eligible for termination
IDLE_TIMEOUT=30
#
# Specify any directories containing DLLs needed at runtime.
# For example, to use the IBM DB2 Driver for JDBC and SQLJ,
# add the directory containing the native DLLs to the
# LIBPATH_SUFFIX. See the DB2 Application and Programming
# Guide for Java relevant to the level of DB2 being used.
#
#LIBPATH_PREFIX=
#LIBPATH_SUFFIX=
#
# Specify any directories containing application Java classes
# and jar files. (Uncomment the lines below if needed)
#
#CLASSPATH_SUFFIX=/u/example/pathToJarOrZipFile/jarfile.jar:\
#                 /u/example/pathToRootDirectoryForClasses
#
# Uncomment the line below to use the specified output redirection
# class.
#
#USEROUTPUTCLASS=com.ibm.cics.samples.SJMergedStream
#
######
#
# ********* Unix System Services Environment Variables ***********
#
# Java Dump Options. See "IBM Developer Kit and Runtime Environment,
# Java Technology Edition, Diagnostics Guide" for information on all
# Java runtime options.
JAVA_DUMP_OPTS="ONANYSIGNAL(JAVADUMP,CEEDUMP,SYSDUMP),ONINTERRUPT(NONE)"
#
# Specify where JVM dumps should be written to
#JAVA_DUMP_TDUMP_PATTERN=DUMP.JVM.TDUMP.&APPLID;.&JVM_NUM;.LATEST
#
# Specify the local timezone
#TZ=CET-1CEST,M3.5.0,M10.5.0
#
######
#
# ********* JVM options **************
#
-Xms16M
```

```
-Xmx16M
-Xmso128K
-Xiss64K
-Xss256K
```

DFHJVMAX, JVM profile reserved for the JVM server

The JVM profile DFHJVMAX is a CICS-supplied JVM profile that is reserved for use by a JVM server. Make sure that DFHJVMAX is set up correctly for your CICS region.

JVM options in DFHJVMAX JVM profile

```
############################################################
# JVMProfile: DFHJVMAX
############################################################
#
# This is a sample CICS JVM Profile for a JVM server.
#
######
#
# Symbol Substitution :
#
# If you use the symbol &APPLID; in any of the values below,
# the applid of the CICS region is substituted at runtime.
# So you can use the same profile for all regions, and still
# have unique region-specific working directories or output
# destinations.
#
# If you use the symbol &JVMSERVER; in any of the values
# below, the name of the JVMSERVER resource is substituted
# at runtime.
#
# With this substitution
#     ENV_VAR=dfhjvmerr.&APPLID;.&JVMSERVER;.data
# becomes
#     ENV_VAR=dfhjvmerr.ABCDEF.JSERVER1.data
# for a JVMSERVER resource with the name JSERVER1 in a CICS
# region with applid ABCDEF. Applids are always in uppercase.
#
######
#
# ********* CICS-specific parameters ***********
#
# When using a JVM server, standard CICS options (which are usually
# specified in this section) are ignored. However, some new options
# that are specific to JVM servers have been introduced. These are
# detailed below.
#
# The SETUP_CLASSES and TERMINATION_CLASSES options are used to
# specify one or more Java classes that will be called when the JVM
# is initialized and before the JVM is terminated. These lists of
# classes must be semi-colon separated and must not be fully qualified.
#
# Arguments can be passed to the setup and termination classes,
# by enclosing them in a pair of curly braces, comma-separated.
# For example, the following class declaration:
#
#  SETUP_CLASSES=com.ibm.cics.setUpClass1{"string1", "string2"}
#
# will result in the setup class com.ibm.cics.setUpClass1 being
# invoked when the JVM is initialized, with the string arguments
# "string1" and "string2" being passed in to its main method.
#
# If the class is in a directory that is not the correct working
# directory, it must be added to the classpath using the
# -Djava.class.path option, which is specified in the JVM server
```

```
# properties section of this profile.
#
# N.B. Setup and termination classes must have a main method and
# must not be long running, because an internal timeout of 120
# seconds will be imposed on them.
#
#SETUP_CLASSES=com.ibm.cics.setUpClass1;com.ibm.cics.setUpClass2
#
#TERMINATION_CLASSES=com.ibm.cics.cleanUpClass
#
# This option is used to specify, in seconds, how long setup and
# termination classes are allowed ot run for before timing out. The
# value specified must be in the range 1 - 60000. If it falls outside
# of this range, then the value defaults to 60 seconds.
#
#THREAD_TIMEOUT=60
#
# ********* Unix System Services Environment Variables ***********
#
# Java Dump Options. See "IBM SDK for z/OS platforms, Java Technology
# Edition, SDK Guide" or "IBM Developer Kit and Runtime Environment,
# Java Technology Edition Diagnostics Guide" for information on all
# Java runtime options.
#
#
# JAVA_DUMP_OPTS="ONANYSIGNAL(JAVADUMP,CEEDUMP,SYSDUMP),ONINTERRUPT(NONE)"
#
# Specify where JVM dumps are to be written:
#
#JAVA_DUMP_TDUMP_PATTERN=DUMP.JVM.TDUMP.&APPLID;.&JVMSERVER;.LATEST
#
# Specify the local timezone:
#
#TZ=CET-1CEST,M3.5.0,M10.5.0
# Set the libpath environment variable - this must specify the
# correct install location of the Java SDK, for your system
#
LIBPATH=/usr/lpp/java/J6.0/bin/j9vm:\
        /usr/lpp/java/J6.0/bin
#
# Set the current working directory. If this environment variable
# is set, a change to the directory specified is issued before
# the JVM is initialized, and the STDIN, STDOUT, and STDERR streams
# are allocated to this directory.
#
# If it is not specified, the current working directory is left
# unchanged and the STDIN, STDOUT, and STDERR streams are allocated
# to the directory /tmp.
#
WORK_DIR=.
#
######
#
# ********* JVM options **************
#
# See "IBM SDK for z/OS platforms, Java Technology Edition, SDK Guide"
# SDK Guide" or "IBM Developer Kit and Runtime Environment, Java
# Technology Edition, Diagnostics Guide" for information on all JVM
# options.
#
# N.B. JVM options that print output and then exit must not be
# specified, because you need to continue to create the JVM. These
# are options such as: -version, -help, -?, -assert, and -X.
#
-Xms16M
-Xmx32M
-Xoss4M
```

```
-Xss512K
#
# The -Xgcthreads option sets the maximum number of helper threads
# allowed for garbage collection. If this option is not specified,
# the default is set to the number of CPUs - 1.
#
-Xgcthreads1
#
# ********* Properties for the JVM server ***********
#
# JVM system properties for a JVM server can be
# specified here, in this profile file. Properties
# are key name and value pairs that contain basic
# information about the JVM and its environment. They
# are always prefixed with -D. For example:
# -Dcom.ibm.cics.some.property=some_value
#
# Set the Java classpath to the current directory
# and include any other directories that are needed,
# such as those containing setup or termination classes.
#
# e.g. -Djava.class.path=.:/usr/lpp/setup_classes:\
#                          /usr/lpp/term_classes
#
-Djava.class.path=.:\
                    /usr/lpp/cicsts///&CICS_HOME///libdfjwrap.jar:\
                    /usr/lpp/cicsts///&CICS_HOME///lib/dfjcicsras.jar:\
                    /usr/lpp/cicsts///&CICS_HOME///lib/ras.jar:\
                    /usr/lpp/cicsts///&CICS_HOME///lib/dfjorb.jar:\
                    /usr/lpp/cicsts///&CICS_HOME///lib/dfjcont.jar:\
                    /usr/lpp/cicsts///&CICS_HOME///lib/dfjcsi.jar:\
                    /usr/lpp/cicsts///&CICS_HOME///lib/dfjcics.jar:\
                    /usr/lpp/cicsts///&CICS_HOME///lib/dfjcdmn.jar:\
                    /usr/lpp/cicsts///&CICS_HOME///lib/dfjjts.jar:\
                    /usr/lpp/cicsts///&CICS_HOME///lib/dfjadjr.jar:\
                    /usr/lpp/cicsts///&CICS_HOME///lib/omgcos.jar:\
                    /usr/lpp/cicsts///&CICS_HOME///lib/dfjname.jar:\
                    /usr/lpp/cicsts///&CICS_HOME///lib/websphere.jar:\
                    /usr/lpp/cicsts///&CICS_HOME///lib/dfjcci.jar:\
                    /usr/lpp/cicsts///&CICS_HOME///lib/dfjejbdd.jar:\
                    /usr/lpp/cicsts///&CICS_HOME///lib/dfjoutput.jar:\
                    /usr/lpp/cicsts///&CICS_HOME///lib/wsdl/dfjwsdl.jar:\
                    /usr/lpp/cicsts///&CICS_HOME///lib/wsdl/wsdl4j.jar:\
                    /usr/lpp/cicsts///&CICS_HOME///lib/wsdl/woden.jar:\
                    /usr/lpp/cicsts///&CICS_HOME///lib/wsdl/XmlSchema.jar:\
#
```

DFHJVMCD, JVM profile reserved for CICS-supplied system programs

The JVM profile DFHJVMCD is a CICS-supplied JVM profile that is reserved for use by CICS-supplied system programs, in particular the default request processor program DFJIIRP, which is used by the CICS-supplied CIRP request processor transaction. CICS also uses DFHJVMCD to initialize and stop the shared class cache. Make sure that DFHJVMCD is set up correctly for your CICS region, but customize it only if necessary.

"Customizing DFHJVMCD" on page 100 has instructions for customizing the options in this JVM profile.

JVM options in DFHJVMCD JVM profile

```
############################################################
# JVMProfile: DFHJVMCD
############################################################
```

```
#
# This is the CICS JVM Profile for use by CICS programs.
# It must have valid values for CICS_HOME and JAVA_HOME.
# It must always be available in the directory specified by
# the JVMPROFILEDIR SIT parameter.
#
######
#
# Symbol Substitution:
#
# If you use any of the following variable symbols in any of
# the variables below, they will be replaced with appropriate
# values. The variable symbols may be specified in upper or
# lower case.
#
#  Symbol       Replacement value
#  ------       -----------------
#
# &APPLID;      The APPLID of the CICS region
# &JVM_NUM;     The Unix Systems Services Process ID (pid)
#               of the JVM. This is guaranteed to be unique
# &DATE;        The current date in the format Dyymmdd
# &TIME;        The current time in the format Thhmmss
#
# With this substitution, for example
#     STDERR=dfhjvmerr.&APPLID;.&JVM_NUM;.&DATE;.&TIME;
# becomes
#     STDERR=dfhjvmerr.ABCDEF.0084214386.D081220.T135323
#
######
#
# ********* CICS-specific parameters ***********
#
CICS_HOME=/usr/lpp/cicsts/cicsts41
JAVA_HOME=/usr/lpp/java/java6_31/J6.0
WORK_DIR=.
REUSE=YES
CLASSCACHE=NO
#
STDIN=dfhjvmin
STDOUT=dfhjvmout
STDERR=dfhjvmerr
######
#
# ********* Unix System Services Environment Variables ***********
#
# Java Dump Options. See "IBM Developer Kit and Runtime Environment,
# Java Technology Edition, Diagnostics Guide" for information on all
# Java runtime options.
JAVA_DUMP_OPTS="ONANYSIGNAL(JAVADUMP,CEEDUMP,SYSDUMP),ONINTERRUPT(NONE)"
#
# Specify where JVM dumps should be written to
#JAVA_DUMP_TDUMP_PATTERN=DUMP.JVM.TDUMP.&APPLID;.%JVM_NUM;.LATEST
#
# Specify the local timezone
#TZ=CET-1CEST,M3.5.0,M10.5.0
#
######
#
# ********* JVM options **************
#
-Xms4M
-Xmx4M
-Xmso128K
-Xiss64K
-Xss256K
```

Managing the shared class cache

Once you have set up a shared class cache in your CICS region, you might need to perform the management tasks described in this section.

"The shared class cache" on page 90 explains how the shared class cache works, and how JVMs benefit from using it.

Starting the shared class cache

By default, the shared class cache starts automatically as soon as CICS receives a request to run a Java application in a JVM whose profile requires the use of the shared class cache. If at any time you stop the shared class cache and disable autostart, and you want to restart it again, re-enable autostart or use CICS commands to carry out the restart.

Procedure

- To maintain the normal startup behavior of the shared class cache, where the shared class cache starts as soon as a JVM needs it, leave the **JVMCCSTART** system initialization parameter with the default setting AUTO. On a warm or emergency start, if a shared class cache was active when the system shut down, it normally persists (except in some circumstances such as an IPL of z/OS). This means it is available before the first JVM needs it.
- Because CICS now supports two versions of the IBM SDK for z/OS, you can no longer use the **JVMCCSTART**=YES system initialization parameter to make the shared class cache start up during CICS initialization on an initial or cold start, as CICS cannot tell what version is required. If you require this behavior, you can write an initialization program (PLTPI program) and define it to CICS in a program list table (PLT) to run immediately after CICS initialization is complete. In the program, use the PERFORM JVMPOOL command to manually start a JVM whose profile specifies the correct version of the SDK and requires the use of the shared class cache. This makes the shared class cache start up.
- If you have stopped the shared class cache and disabled autostart, and you want to restart the shared class cache while CICS is running, use one of the following methods:
  1. To restart the shared class cache immediately, use the **CEMT PERFORM CLASSCACHE START** command (or the equivalent **EXEC CICS** command). If you also want to re-enable autostart, use the AUTOSTARTST option on the command to specify this. You can use the CACHESIZE option on this command if you want to change the size of the shared class cache.
  2. To set the shared class cache to start when it is required by a JVM, use the **CEMT SET CLASSCACHE AUTOSTARTST** command (or the equivalent **EXEC CICS** command) to enable autostart while CICS is running. The shared class cache is restarted when CICS receives a request to run a Java application in a JVM whose profile requires the use of the shared class cache. Subsequent warm or emergency CICS starts use this setting for autostart, unless you have specified the **JVMCCSTART** system initialization parameter as an override at startup.

Related information

Writing initialization and shutdown programs

"Manually starting and terminating JVMs and disabling the JVM pool" on page 152

CICS starts up JVMs in response to the requirements of applications, and reduces the number of available JVMs automatically if the workload does not require them. You can also control the JVM pool using CICS commands; you can start up and terminate JVMs, and disable the JVM pool temporarily. This manual control lets you implement changes to JVM profiles or suspend activity in the JVM pool. You can also use it to create JVMs in advance of application requests.

Adjusting the size of the shared class cache

When the shared class cache starts up, the amount of storage in the cache is fixed. The default size is 24MB. When the storage in the shared class cache becomes full, the classes that are already present in it can still be used, but no new classes can be added to it. In this situation, you need to increase the size of the shared class cache.

About this task

For the shared class cache, when the storage is full, any further classes are loaded into the individual JVMs. A warning message is issued if you have requested verbose output, but the JVMs can continue to run applications as they did before. In this situation, you need to increase the size of the shared class cache.

Use the CEMT INQUIRE CLASSCACHE command, or the equivalent EXEC CICS command, to report on the size of the shared class cache and amount of free storage in the shared class cache by specifying the CACHESIZE and CACHEFREE options respectively.

You can obtain further shared class cache statistics by running the following command in a z/OS UNIX System Services shell:

```
java -Xshareclasses:name=CICS_sharedcc_&APPLID_n,printStats
```

where *&APPLID* is the VTAM® APPLID of the CICS system, and *n* is the current generation number for the shared class cache. For more information on the Java shared classes utility, see the Version 6 *IBM Developer Kit and Runtime Environment, Java Technology Edition Diagnostics Guide*, which is available to download from www.ibm.com/developerworks/java/jdk/diagnosis/.

The **JVMCCSIZE** system initialization parameter specifies the initial size of the shared class cache. If you have made the initial size of the shared class cache too small (or too large), you can change it while CICS is running:

Procedure

Use the **CEMT PERFORM CLASSCACHE RELOAD** command or the equivalent **EXEC CICS** command to create a new shared class cache, and specify the size for the new shared class cache by using the CACHESIZE option on the command. This causes the least disruption to JVMs that are using the shared class cache.

Results

When you specify a new size for the shared class cache while CICS is running, subsequent CICS restarts use the new value, unless CICS is initial or cold started.

In these cases, the value from the **JVMCCSIZE** system initialization parameter is used. The shared class cache persists across warm or emergency starts, so the **JVMCCSIZE** parameter never applies to it in this situation, even if you specify it as an override.

The size that you specify for the shared class cache must be sufficient to contain all the classes for your applications, as specified on the standard class path (the CLASSPATH_PREFIX and CLASSPATH_SUFFIX options in the JVM profile) for all the JVMs that use the shared class cache. The shared class cache does not make a distinction between shareable and nonshareable application classes, and it does not contain JIT-compiled code.

Either estimate the storage required for your application classes, or for better results, run the applications in a test environment to identify the total space required in the shared class cache.

1. Run each application repeatedly in a test environment, using the shared class cache.
2. While you are running the application, monitor the amount of free space in the shared class cache.
3. Run the application until the amount of free space has stabilized, which should happen almost immediately.
4. Repeat this process for each application that will be using the shared class cache.
5. Add together the amount of storage used by each application, and add on a suitable safety margin to allow for any future application changes.

This gives you an approximate size for the shared class cache.

Terminating the shared class cache

You can terminate the shared class cache and prevent it from restarting, and terminate any JVMs that are using it.

About this task

When you terminate the shared class cache, if autostart is enabled, a new shared class cache is created as soon as a JVM requests its use. If you want to prevent this, and terminate the shared class cache without restarting it, then you need to disable autostart as well as terminating the shared class cache.

Note: If you terminate the shared class cache and it is not restarted, either by a command or by the autostart feature, JVMs that need to use the shared class cache cannot run.

When you change the autostart status of the shared class cache while CICS is running, subsequent CICS restarts use the most recent setting that you made using the **SET CLASSCACHE** command or the **PERFORM CLASSCACHE** command, unless the system is INITIAL or COLD started, or the **JVMCCSTART** system initialization parameter is specified as an override at startup. In these cases, the setting from the system initialization parameter is used.

Procedure

1. Use the **CEMT INQUIRE CLASSCACHE** command (or the equivalent **EXEC CICS** command) to check the current status of autostart for the shared class cache.

2. If you do not want the shared class cache to restart when you terminate it, disable autostart. You can disable autostart for the shared class cache in three ways:
   - Before you enter the command to terminate the shared class cache, use the **CEMT SET CLASSCACHE AUTOSTARTST** command (or the equivalent **EXEC CICS** command) to disable autostart.
   - When you are entering the **CEMT PERFORM CLASSCACHE** command (or the equivalent **EXEC CICS** command) to terminate the shared class cache, use the AUTOSTARTST option to disable autostart.
   - To disable autostart for the next CICS execution, set the **JVMCCSTART** system initialization parameter to NO. This setting always prevents autostart on an initial or cold start of CICS. If a Version 6 shared class cache was active when the system shut down, it persists across a warm or emergency start, even if you specify **JVMCCSTART** as an override.
3. Use the **CEMT PERFORM CLASSCACHE PHASEOUT**, **PURGE** or **FORCEPURGE** command (or the equivalent **EXEC CICS** command) to terminate the shared class cache and any JVMs that are using it. You can choose to purge or forcepurge the JVMs, or allow them to finish running their current Java programs before they are deleted. JVMs that are not using the shared class cache (standalone JVMs) are not affected by this command.
4. If you do not want to restart the shared class cache, and the JVMs that are using it remain active for too long, repeat the **CEMT PERFORM CLASSCACHE PHASEOUT**, **PURGE** or **FORCEPURGE** command (or the equivalent **EXEC CICS** command), to attempt to purge the tasks that are using the JVMs. You should only repeat the command if autostart for the shared class cache is **disabled**. The command operates on both the most recent shared class cache, and any old shared class caches in the system that still have JVMs using them. If autostart is enabled, and you repeat the command to terminate the shared class cache, the command could operate on the new shared class cache that has been started by the autostart facility, and terminate it.

Monitoring the shared class cache

You can use CICS commands to report on the status of the shared class cache and of the JVMs in the pool.

Procedure

- To report on the status of the shared class cache, use the **CEMT INQUIRE CLASSCACHE** command (or the equivalent **EXEC CICS** command). The command tells you if the shared class is being initialized (STARTING), ready for use (STARTED), being reloaded (RELOADING), or not active (STOPPED). The command also tells you information such as the status of autostart, the size of the shared class cache, and the amount of free space in it. If there are any old shared class caches in the CICS region which are being phased out, the command reports these.
- To report on the status of the JVMs in the JVM pool, use the **CEMT INQUIRE JVM** command (or the equivalent **EXEC CICS** command). The command tells you about a specified JVM or about each JVM in the pool, indicating the task to which it is allocated, whether its execution key is USER or CICS, and whether or not it is using the shared class cache.

Programming for JVMs in CICS

When developing Java applications for JVMs in CICS, you need to consider the way in which CICS reuses JVMs and the requirements for transaction isolation in CICS.

Programming considerations for continuous JVMs

Java programs for CICS need to be careful not to leave any unwanted state in the JVM or change the state of the JVM in undesirable ways. They also need to close any DB2 connections that they open. You can use facilities available with CICS and Java to check or enforce isolation between your Java applications.

Besides the considerations detailed in this topic, other APIs available in Java also have the potential to create issues with transaction isolation for JVMs in a CICS region. For example, there are some Java operations which CICS cannot undo if the CICS task is backed out, such as alterations to z/OS UNIX files or directories.

Protect the state of a continuous JVM

The continuous JVM does not automatically isolate invocations of Java programs from changes made to the JVM by previous invocations of programs in the same JVM. Application classes that run in a continuous JVM are able to change the state of the JVM in ways that might affect subsequent program invocations. For example, a program might reset the default time-zone, and do calculations based on this time-zone. Subsequent invocations of the program would use the new default time-zone, which might not be appropriate. If your program changes the state of the JVM, you should ensure that the program also resets to the original state.

Control static state in a continuous JVM

Invocations of Java programs in a continuous JVM are able to pass on state to subsequent invocations of programs in the same JVM. You must therefore take care when designing and coding your applications that you do not leave any unwanted state in a continuous JVM.

Because the static storage is not reinitialized for each invocation of the program in a continuous JVM, your program must reinitialize its own static storage, if it depends on the state of a changeable class field. The values of static variables in a continuous JVM persist within the JVM, even though it is serially dispatched to a number of CICS tasks. This is true for static variables in all classes, both application and system classes, and includes classes which might affect the application, but are not used explicitly (including those used in static initializers).

In most cases, static variables are used to avoid re-initialization of storage, and allowing them to persist across JVM uses can improve performance. However, if the application requires that the value of these variables is reset between JVM uses, then for use in a continuous JVM, the application must reset the value itself. Try to identify and eliminate any changeable class fields and static initializers that have not been included deliberately as part of the application's design. Consider the following guidelines:

- Define a class field as private and final whenever possible. Be aware that a native method can write to a final class field, and a non-private method can obtain the object referenced by the class field and can change the state of the object or array.

- Be aware of system-loaded classes that use changeable class fields.

You can use the ability to pass on state to your advantage in designing your Java applications if you want information to persist from one program invocation to the next. Static state and object instances referenced through static state are kept across JVM reuses in a continuous JVM, so it is permissible for applications to create persistent items that might be of use to future executions of the same application in the same JVM.

Imagine an operation that reads DB2 information in order to construct a complex data structure; this might be an expensive operation that should not be repeated more times than absolutely necessary. With a continuous JVM, the complex data structure can be stored in application static and be accessible to later executions of the application in the same JVM, thus avoiding unnecessary initialization. (If objects are anchored in static, that is, in the static class fields, then they are never candidates for garbage collection.)

If you design an application in this way, remember that there is no guarantee that subsequent executions of an application (or even executions of a different Java program within the same transaction), will be assigned a JVM containing the items that were created by the first execution of the application. The subsequent executions of the application might be assigned a newly created JVM, or a JVM that has been re-initialized following a mismatch or a steal, or a JVM that has been used by a different application which cleared the JVM's storage heaps. Your application should not rely on the presence of the persistent items that you create in the JVM; it should check for their presence in order to avoid unnecessary initialization, but it should be prepared to initialize them if they are not found in the present JVM.

Close DB2 connections and other task lifetime system resources after use

After a Java application running in a continuous JVM has accessed DB2, it is important that it closes the DB2 connection. This is because subsequent executions of the same application in the continuous JVM will try to open a new DB2 connection. This fails if a previous connection has not been closed. The same applies to any other task lifetime system resources used by the application, which must be released after use.

Test applications for possible issues with transaction isolation

You can use the CICS JVM Application Isolation Utility to audit the use of static variables in your Java applications. The utility inspects Java bytecodes and reports on the static variables used by each class. You can use this information to help you check your source code. Make sure that the application is resetting the static variable correctly in each case. “Auditing Java applications for the use of static variables” on page 137 explains how to use the utility.

If a Java application works correctly on its first use in a given JVM, but does not behave correctly on subsequent uses, then the problem is likely to be due to isolation issues. In this case, using the CICS JVM Application Isolation Utility as part of your problem determination work might help to identify the cause of the problem.

Consider applying a Java 2 security policy

If you want to monitor and police any potentially unsafe actions in a continuous JVM, consider enabling the Java 2 security policy mechanism.

By default, CICS does not enforce a Java 2 security policy. When you enable the security manager for a JVM, you can specify security policy files to give applications permission only for actions which you consider safe. CICS provides a Java 2 security policy file, `dfjejbpl.policy`, which can be used to restrict the permitted operations for a Java application in CICS to only those operations permitted for enterprise beans. You may choose to use this policy file, and to provide further policies of your own, if wanted. "Protecting Java applications in CICS by using the Java security policy mechanism" on page 345 has more information about applying a security policy.

Consider using JVM profiles to enforce isolation between different applications

When you are specifying JVM profiles for continuous JVMs, bear in mind that if more than one application uses the same JVM profile that creates a continuous JVM, the applications could see each other's persistent state.

If you need to ensure that an application that uses a continuous JVM does not have any contact with the persistent state from another application, you should create separate JVM profiles for the applications to use. (The JVM profiles can be identical in content, provided that they have different eight-character names.)

Possible Java application behavior changes in continuous JVMs

Because there is no reset operation in the continuous JVM, applications that were designed to execute in a resettable JVM might exhibit changed behavior when they execute in a continuous JVM. You might have to make changes to an application to preserve its original behavior while running in a continuous JVM.

In a resettable JVM, the state of the JVM was reset after each use, so that no application transaction (that is, code other than trusted middleware code) could affect the operation of subsequent transactions. The JVM reset cleaned up the JVM's storage heaps, reinitialized shareable application classes, and discarded and reloaded nonshareable application classes, meaning that no objects other than trusted static middleware objects could persist in the JVM from one use of the JVM to the next.

The continuous JVM does not reset the JVM's state between uses. This continuity enables the persistence of static objects across tasks, which can be a powerful tool when used deliberately. For example, an application developer can use caching techniques to avoid reinitializing objects on each use. It can also, however, be a source of unexpected and erroneous behavior unless it is handled carefully.

Example 1: Altering static variables

The most common type of state change that an application can make is to alter the value of a static variable. static variables are shared by all instances of a class, unlike non-static variables, which are allocated separately for each instance.

In a resettable JVM, when a class is first loaded, the JVM takes a copy of the initial value of each static variable and uses it to restore the variable to its original state at the end of each transaction. Consider the following trivial case:

```
public class HelloWorld
{
    public static int count = 0;

    public static void main(String args[])
    {
        count++;
        System.out.println("Hello World, count is " + count);
    }
}
```

In a resettable JVM, the static variable count is reset to zero by the JVM after each invocation of the HelloWorld main() method. The message therefore shows that count is 1 each time HelloWorld is invoked.

In a continuous JVM, however, count is not reset to its original value before the next invocation of the main() method, and the old, shared, value persists. The message therefore shows the count increasing by 1 on each invocation in subsequent transactions.

To preserve the original behavior while running in a continuous JVM, the HelloWorld class could be changed to make count an instance variable and initialise it on each invocation in a constructor:

```
public class HelloWorld
{
    public int count = 0;

    public static void main(String args[])
    {
        HelloWorld hw = new HelloWorld();
        hw.count++;
        System.out.println("Hello World, count is " + hw.count);
    }

    HelloWorld()
    {
        count = 0;
    }
}
```

Example 2: Altering the contents of static objects

A more subtle type of issue can arise when the static variable is an object reference whose internal state might change, as in this example:

```
import java.util.Hashtable;
import java.util.Enumeration;

class StaticHash
{
    private static final Hashtable myHashtable = new Hashtable();

    public static void main(String[] args)
    {
        int count = myHashtable.size();
        myHashtable.put("key" + count, "value" + count);

        Enumeration keys = myHashtable.keys();
        while (keys.hasMoreElements())
        {
```

```
            Object key = keys.nextElement();
            System.out.println("Found this key in the Hashtable: " + key);
        }
    }
}
```

In a resettable JVM, a new instance of myHashtable is created every time the JVM is reset, and it will only ever contain a single key, “key0”. In a continuous JVM, however, only one instance of myHashtable is created, and each time the class is run, a new key is added to it.

You can solve the problem in a similar manner to the first example, by making myHashtable an instance variable and creating the new Hashtable in a constructor. Alternatively, myHashtable could be left as a static reference and be reset each time by adding a constructor containing an invocation of myHashtable.clear().

Auditing Java applications for the use of static variables

The CICS JVM Application Isolation Utility helps system administrators and application programmers to discover static variables in Java applications that they use or plan to use in their CICS regions. Application developers then review the findings of the utility and determine whether or not the application might exhibit unintended behavior when it runs in a continuous JVM. You can use the utility when migrating Java workloads from resettable to continuous JVMs.

Before you begin

The CICS JVM Application Isolation Utility is shipped with CICS Transaction Server for z/OS, Version 4 Release 1 as a JAR file named `dfhjaiu.jar`. It runs under z/OS UNIX System Services as a standalone utility. You do not need to have a CICS Transaction Server for z/OS, Version 4 Release 1 region or any other CICS region running when you use the utility.

About this task

The CICS JVM Application Isolation Utility is a code analyzer tool, which inspects Java bytecodes in Java Archive (JAR) files and class files. It does not alter any Java bytecodes. It is provided as a means to help identify potential issues before they arise in a continuous JVM under CICS. The Java application does not need to be running in a CICS region when it is inspected.

To inspect Java applications using the CICS JVM Application Isolation Utility, follow these steps:

Procedure

1. Confirm that the CICS-supplied file `dfhjaiu.jar`, which is the CICS JVM Application Isolation Utility, is present in the `/utils/isolation` subdirectory of the home directory for CICS files on z/OS UNIX. The default name for the home directory is `/usr/lpp/cicsts/cicsts41/`, where `cicsts41` is defined by the USSDIR installation parameter when you installed CICS TS for z/OS, Version 4.1. You can add the `/utils/isolation` directory to the PATH environment variable in z/OS UNIX System Services, so that you do not need to give the full path to the file when you run the utility.
2. Confirm that the shell script DFHIsoUtil, which is used to run the CICS JVM Application Isolation Utility, is also present in the `/utils/isolation` subdirectory of the home directory for CICS files on z/OS UNIX. Check that

the script file specifies the correct value for the CICS_HOME environment variable, and edit the file to change this if necessary.

3. Identify the class files or JAR files that you want to specify to the utility for inspection. Bear these points in mind:
   a. A Java application can involve classes and JAR files that are specified on several different class paths in the JVM profile or JVM properties file. Make sure you include all of them in your inspections.
   b. You can use wildcard characters in the file names, to inspect all the class files or JAR files in a given directory.
   c. When you specify a JAR file for inspection, the utility inspects all the classes contained in the JAR file.
   d. If you specify an individual class file for inspection, the utility inspects only the named class. If the class includes inner classes, the utility does not automatically inspect these. Specifying JAR files, or using wildcards to inspect a whole directory, ensures that any inner classes are included in the inspection.
4. Log in to a z/OS Unix System Services shell, and enter the command

   ```
   DFHIsoUtil [-verbose] filename [filename ... filename]
   ```

   In this command:
   a. DFHIsoUtil is the name of the script file which runs the CICS JVM Application Isolation Utility. If you have not set an appropriate PATH environment variable and you are not working in the directory containing the script file, give the full path to the file, for example `/usr/lpp/cicsts/cicsts41/utils/isolation/DFHIsoUtil`.
   b. The **-verbose** option makes the utility provide additional information. See "The -verbose option" on page 140.
   c. *filename* specifies the names of one or more class files or JAR files that you have identified for the utility to inspect. Separate each file name with a space. Give the full path to the files if necessary. You can use wildcard (glob) characters in the file names.

   For example, to inspect the class file HelloWorld and obtain the standard report (not the verbose report), enter the command

   ```
   DFHIsoUtil HelloWorld.class
   ```
5. The report produced by the CICS JVM Application Isolation Utility is written to System.out. You can redirect it to another destination as required.
6. Review the findings of the utility and then examine the source code for your Java applications. The reports produced by the utility identify some potential issues, but you must check if these affect the behavior of the application when it runs in a continuous JVM.

Example 1: Report showing alteration of `static` variables

When the CICS JVM Application Isolation Utility is used to inspect the HelloWorld class file used in Example 1 in "Possible Java application behavior changes in continuous JVMs" on page 135, the report looks like this:

```
CicsIsoUtil: CICS JVM Application Isolation Utility

Copyright (C) IBM Corp. 2006

Reading Class File: HelloWorld.class

    Method: public static void main(java.lang.String[])
      Static fields written in this method:
        public static int count
```

```
    Method: <clinit> (Class Initialization)
      Static fields written in this method:
        public static int count

  Number of methods inspected       : 3
  Total static writes for this class: 2

Number of Jar Files inspected    : 0
Number of Class Files inspected  : 1
```

The report shows that the static field count is written to during Class Initialization and in the main() method. The report indicates that count might behave differently when the class is used in a continuous JVM, rather than in a resettable JVM. The application programmer must examine the source code to decide whether count really does behave differently.

Example 2: Report showing alteration of the contents of `static` objects

When the CICS JVM Application Isolation Utility is used to inspect the StaticHash class file used in Example 2 in "Possible Java application behavior changes in continuous JVMs" on page 135, the report looks like this:

```
CicsIsoUtil: CICS JVM Application Isolation Utility

Copyright (C) IBM Corp. 2006

Reading Class File: StaticHash.class

    Method: <clinit> (Class Initialization)
      Static fields written in this method:
        private static final java.util.Hashtable myHashtable

  Number of methods inspected       : 3
  Total static writes for this class: 1

Number of Jar Files inspected    : 0
Number of Class Files inspected  : 1
```

Note that the static variable myHashtable is only written to during Class Initialization, but the internal state of the Hashtable changes on each invocation.

This problem is more difficult to assess. The output of the utility identifies that a static object exists. In this case, the object is a hash table; other items such as arrays might also be in this situation. The application developer must check the source code of the application to ensure that the state of the static object is not changed in a way that unintentionally affects subsequent invocations of the class in a continuous JVM.

You must also check the entire graph of other objects that might be referenced from the original static object. Any static object can contain state of its own. This state can include further objects that are not defined as static, but are included within the static context of the parent object. A large graph of objects can be built up in this way, where only the root object is declared as static, but state held in any of the objects might be available for use by subsequent applications, because of the static root object. The application developer must check for application isolation problems at every level of the object graph, in addition to checking at the root level.

The -verbose option

Normally, the CICS JVM Application Isolation Utility does not print details of methods which do not write to static variables, or details of static final String variables. With the **–verbose** option specified, the utility does print these extra details and also lists all static method invocations made.

This additional information can identify other potential problems with your applications. For example, this extract from a report shows code relating to the resettable JVM:

```
Static methods invoked by this method:
        boolean isResettableJVM()
           (defined in class: com.ibm.jvm.ExtendedSystem)
```

All methods in the com.ibm.jvm.ExtendedSystem class are related to the resettable JVM. They are all deprecated, and you should remove them from any application code.

Threads and sockets in Java applications for CICS

For a Java application running in a JVM in a CICS region, threads and sockets should be used with caution. These Java features could affect the isolation of CICS tasks, and interfere with JVM phaseout.

Threads

The main thread under which a JVM starts is called the Initial Process Thread (IPT). Application code that uses the JCICS API must execute under the IPT. CICS ensures that the public static main method in any Java program (from the Java class specified by the JVMCLASS attribute in the PROGRAM resource definition) executes under the IPT, and this is also the case for enterprise beans and stateless CORBA applications.

It is possible for application code running in a JVM to start a new thread, or call a library which starts a thread on its behalf. Threads started by user code cannot make use of CICS services; if you attempt to do this, the JVM abends with an 0501 user abend code. An application could start a thread and use it for purposes other than interacting with CICS. However, the use of threads in a Java application for CICS can have undesirable consequences.

If an application running in a continuous JVM starts threads, the CICS task completes when the IPT has finished its activity, but other threads can continue executing after the IPT has returned control to CICS. The threads might carry on executing while the JVM is not assigned to a CICS task, and might even be running when a JVM is assigned to a new task. This damages isolation for a CICS JVM, and can also cause problems when CICS attempts to phase out the JVM, because the phaseout process might be blocked waiting for the user threads to end.

For these reasons, the use of threads in a continuous JVM should be treated with extreme caution. In general, it is recommended that applications running in a continuous JVM do not start any threads at all. If you really need to start threads, the application needs to ensure that they are not allowed to execute beyond the lifetime of the CICS task which starts them.

Sockets

Threads started by application code might be used to manage sockets created using classes in the `java.net` package. Sockets created using the `java.net` classes use the JVM's native sockets capabilities, rather than the CICS sockets domain. These sockets are not managed by CICS, and the user is responsible for handling and managing them. CICS is not capable of transactionally managing or monitoring any communications performed using these sockets.

In a continuous JVM, when the CICS task ends, threads started by application code could still be listening on the sockets in order to process new workload, and the sockets are not automatically closed. In this situation, the threads could continue executing beyond the lifetime of the CICS task, and interfere with isolation or with JVM phaseout.

You could consider using the Java 2 security policy mechanism to prevent applications from starting threads or from creating sockets using the `java.net` classes. Note that the CICS-supplied enterprise beans policy file, `dfjejbpl.policy`, does allow the use of sockets, because this is recommended in the *Enterprise JavaBeans* specification. You should only consider removing this permission if you do not use enterprise beans.

Programming considerations for single-use JVMs

New Java applications should not be developed in such a way that they can only run in a single-use JVM. You should only use this type of JVM for older Java programs that previously ran in a single-use JVM, and cannot at present be redesigned to run safely in a continuous JVM. To improve performance, you should redesign these Java programs as soon as you can.

Because single-use JVMs are not reused by further programs, but are destroyed after use, there are no considerations about transaction isolation. A Java program that runs in a single-use JVM can change the state of the JVM or leave unwanted state in the JVM.You cannot have more than one invocation of a Java program in a single-use JVM, so these programs cannot pass on state to subsequent invocations of the same program. However, because single-use JVMs have far inferior performance to continuous JVMs, they should not be used for repeated transactions in a production environment.

The single-use JVM is the only type of JVM that should be configured for debug using the Java Platform Debugger Architecture (JPDA). A JVM that has been run in debug mode is not a candidate for reuse. “Debugging an application that is running in a CICS JVM” on page 160 has more information about this.

Encoding with Java in CICS

When you write Java programs that interact with CICS or port Java programs to CICS, carefully consider the encoding that you use.

For Java programs that interact with other parts of CICS using EBCDIC-encoded text, you can generate byte array objects that contain text data in a specific encoding. For example, your Java program might supply data for use with a COMMAREA or a VSAM file. The following code fragment returns a byte array encoded in code page 037:

```
byte[] ebcdicData = "Some Text Value".getBytes("Cp037");
```

The code fragment is platform-independent because it always generates data in code page 037 for all platforms on which it runs. However, a common alternative is as follows:

```
byte[] localEncodingData = "Some Text Value".getBytes();
```

In this second example, the code fragment does not specify the code page for the data encoding. Java uses the default encoding for the platform. The default encoding for the z/OS operating system is a variant of EBCDIC. On many other platforms, the default encoding is ISO 8859-1 or a similar ASCII-based encoding. This technique is also platform-independent because Java generates data in the encoding that is most suitable for the platform.

However, this second example can cause problems if the code is written on a platform that uses an ASCII-based default encoding and is then deployed to the CICS Java environment. For example, if the application communicates with a process on a different platform, the protocol might require data to be sent in an ASCII-based encoding. In CICS, the data is sent in an EBCDIC-based encoding. You must therefore use caution when porting Java code to CICS to ensure that it encodes the data correctly.

You can use the Java system property **-Dfile.encoding** to change the default data encoding for the JVM. However, the JCICS API expects this property to indicate an EBCDIC encoding. If you change the default encoding to an ASCII-based encoding, you might experience problems when communicating with CICS using JCICS. For more information about encoding on z/OS, see http://www-03.ibm.com/servers/eserver/zseries/software/java/faq/.

Enabling applications to use a JVM

Just as for non-Java applications, CICS requires that you define the resources needed to run a Java program in a JVM. Also, CICS needs to know where to find the classes that the application will use.

Standard Java programs

To enable a standard Java program (one that is not a CORBA stateless object or enterprise bean) to use a JVM, you need to:

1. Select, or create, an appropriate JVM profile for each Java program to use. “The CICS-supplied sample JVM profiles” on page 97 summarizes the considerations you need to take into account, and the changes that you might want to make to the JVM profile.
2. Set the appropriate Java attributes on the PROGRAM resource definition for the Java program. These attributes specify that the program needs a JVM, what the JVM profile and execution key for that JVM must be, and what the main class in the program is.“Setting up a PROGRAM resource definition for a Java program to run in a JVM” on page 144 tells you how to do this.
3. Add the classes that the application uses to the class paths for the JVM, which are set by using options in the JVM profile and JVM properties file for the JVM. “Adding application classes to the class paths for a JVM” on page 145 tells you how to do this.

When you have set up a PROGRAM resource definition for your Java program, and added the application classes to a class path, the Java program is ready to run. Remember that if the JVM profile for the JVM specifies the use of the shared class cache (CLASSCACHE=YES), then for the Java program to run, the shared class

cache must be started, or autostart must be enabled so that the shared class cache can be started when the application requests it. “Starting the shared class cache” on page 129 tells you how to start the shared class cache or enable autostart.

CORBA stateless objects and enterprise beans

CORBA stateless objects and enterprise beans do not have their own PROGRAM resource definitions. A method request for an enterprise bean or CORBA stateless object involves a JVM, because the request processor that handles it executes in a JVM. (A request processor is a program that manages the execution of an IIOP request, including calling the container to process the method.) When CICS receives the method request, it compares it to installed REQUESTMODEL resource definitions, finds the one that best matches the request, and uses the transaction identifier from that request model to determine the PROGRAM resource definition.

Sometimes, IIOP requests are processed using an existing request processor transaction, that already has a JVM assigned to it. CICS only looks at the transaction identifier in any matching request model when a *new* request processor transaction is required.

To enable CORBA stateless objects and enterprise beans to use a JVM, you need to:

1. Identify the JVM profile that is used for the request processor program that will handle the CORBA stateless object or enterprise bean. This is specified on the PROGRAM resource definition for the request processor program. The default request processor program, which is named by the default CIRP transaction on REQUESTMODEL definitions, is DFJIIRP. The supplied PROGRAM resource definition for DFJIIRP specifies the JVM profile DFHJVMCD. If you set up your own request processor program, you can specify a different JVM profile in the resource definition for that program.

   You do not need to set up any further PROGRAM resource definitions or select any JVM profiles for the individual CORBA stateless objects and enterprise beans. They all use the JVM profile that is specified for the request processor program that handles them. Chapter 16, “Configuring CICS for IIOP,” on page 181 explains how to configure CICS as a CORBA participant, and Chapter 19, “Setting up an EJB server,” on page 239 explains how to set up a CICS EJB server and how to deploy enterprise beans. Both these procedures include setting up a suitable request processor program.
2. For CORBA stateless objects only, add the JAR file for the application to the appropriate class path in the JVM profile that is used by the JVM for the request processor program.

   If the application uses any classes, such as classes for utilities, that are not included in its JAR file, these classes also need to be added to the appropriate class path. “Adding application classes to the class paths for a JVM” on page 145 tells you how to do this.
3. For enterprise beans, you do not need to add the deployed JAR files (DJARs) for your enterprise beans to the class path. CICS manages the loading of the classes included in these files by means of the DJAR definitions. However, if your enterprise beans use any classes, such as classes for utilities, that are not included in the deployed JAR file, you do need to include these classes on the class path that will be used by the JVM for the request processor program, as explained in “Adding application classes to the class paths for a JVM” on page 145.

Setting up a PROGRAM resource definition for a Java program to run in a JVM

You need to specify various attributes on the PROGRAM resource definition to enable a Java program to run in a JVM. Only standard Java programs need their own individual PROGRAM resource definitions, so if you are setting up CORBA stateless objects or enterprise beans, skip this topic.

When an application makes a request to run a Java program, it can make the request in various ways. For a standard Java program, the request could be one of the following:

- A 3270 or **EXEC CICS START** request that specifies a transaction identifier.
- An **EXEC CICS LINK** request, or an ECI or EXCI call that names the Java program directly.
- An entry in a program list table (PLT).

For **EXEC CICS LINK** requests or ECI or EXCI calls, and for entries in a program list table, CICS is given the name of the PROGRAM resource definition directly. However, for 3270 or START requests, CICS determines the PROGRAM resource definition by looking at the transaction identifier.

, in the *CICS Resource Definition Guide*, tells you how to set up a PROGRAM resource definition for a program. The attributes you need to specify on the PROGRAM resource definition to enable a Java program to run in a JVM are as follows:

EXECKEY
Specify EXECKEY(USER) if you want the program to run in a JVM that executes in user key. The default for the EXECKEY parameter is USER. Before CICS Transaction Server for z/OS, Version 2 Release 3, the EXECKEY parameter was ignored for Java programs, so you might find that in most cases, the PROGRAM resource definitions for any Java programs that you created for earlier releases of CICS are still set to the default of EXECKEY(USER). EXECKEY(USER) is suitable for most Java programs, because it improves storage protection. However, if the program is part of a transaction that specifies TASKDATAKEY(CICS), the program needs to run in a JVM in CICS key, so in this case, specify EXECKEY(CICS). “Execution key for JVMs” on page 78 explains more about the effects of setting the execution key.

JVM
Specify YES to state that the program is a Java program that has to run in a JVM.

JVMCLASS
Specify the name of the main class in the Java program that is to run in the JVM. If the program has been built as a package (that is, compiled using a Java package statement), you need to specify the fully qualified name, which is the Java class name qualified by the package name, with a period (.) used as a separator. For example, the package `example.HelloWorld` contains the class HelloCICSWorld; in this case, the fully qualified class name is example.HelloWorld.HelloCICSWorld. If the program has not been built as a package, you only need to specify the class name, with no qualifiers.

The names are case-sensitive and must be entered with the correct combination of upper and lower case letters. For example, com.ibm.cics.iiop.RequestProcessor is the class specified for the CICS IIOP request processor program, DFJIIRP. The CEDA panels accept mixed case input for the JVMCLASS field irrespective of your terminal's UCTRAN setting.

However, this does not apply when values for this field are supplied on the CEDA command line, or by using another CICS transaction such as CEMT or CECI. If you need to enter a class name in mixed case when you use CEDA from the command line or when you use another CICS transaction, ensure that the terminal you use is correctly configured, with upper case translation suppressed.

You can use the **CEMT SET PROGRAM JVMCLASS** command or the **EXEC CICS SET PROGRAM JVMCLASS** command to change the name of the main class from that specified on the installed PROGRAM resource definition. (If you use an **EXEC CICS** command to set the JVMCLASS field, the value is always accepted in mixed case.)

If the program uses a single-use JVM (that is, with a JVM profile that specifies the option REUSE=NO), you can also use the user-replaceable program DFHJVMAT to override the JVMCLASS specified on the installed PROGRAM resource definition. On the PROGRAM resource definition, the limit for the JVMCLASS attribute is 255 characters, but you can use DFHJVMAT to specify a class name longer than 255 characters.

JVMPROFILE
Specify the name (up to eight characters) of the profile that CICS is to use for the JVM that will run this program. The default is DFHJVMPR. CICS looks for JVM profiles in the z/OS UNIX directory that is specified by the **JVMPROFILEDIR** system initialization parameter. "Setting up JVM profiles" on page 93 tells you how to select or create JVM profiles and their associated JVM properties files.

As JVM profiles are z/OS UNIX files, case is important. When you specify the name of the JVM profile, you must enter it using the same combination of upper and lower case characters that is present in the z/OS UNIX file name. As for the JVMCLASS field, the CEDA panels accept mixed case input for the JVMPROFILE field irrespective of your terminal's UCTRAN setting. However, this does not apply when values for this field are supplied on the CEDA command line, or by using another CICS transaction such as CEMT or CECI. If you need to enter the name of a JVM profile in mixed case when you use CEDA from the command line or when you use another CICS transaction, ensure that the terminal you use is correctly configured, with upper case translation suppressed.

You can use the **CEMT SET PROGRAM JVMPROFILE** command or the **EXEC CICS SET PROGRAM JVMPROFILE** command to change the JVM profile from that specified on the installed PROGRAM resource definition. (If you use an **EXEC CICS** command to set the JVMPROFILE field, the value is always accepted in mixed case.) This enables you to change the JVM profile that a program uses during a CICS run, without having to re-install the PROGRAM resource definition. Any instances of the program that are currently running in a JVM with the old JVM profile are unaffected, and are allowed to finish running their current Java program. New instances of the program will use a JVM with the new JVM profile that you have specified.

Adding application classes to the class paths for a JVM

The class paths for a JVM are defined by options in the JVM profile, and in the JVM properties file that the JVM profile references. For each Java program, when you have specified the JVM profile that it will use (on the JVMPROFILE attribute of the PROGRAM resource definition), you need to locate the JVM profile and its associated JVM properties file, and add the application classes for the program to the class paths.

About this task

"Classes and class paths in JVMs" on page 74 explains the items that are present in a JVM: system classes and standard extension classes, application classes, and native libraries. That topic also explains the class path to which you can add the classes and native libraries that your application needs. The class path on which each class or native library is placed determines how the item is loaded by the JVM, and where it is stored.

Generally speaking, when you are preparing Java applications that will run in a JVM, you need to ensure that all the application classes required by the application are included on the class path in the JVM profile. You also need to ensure that any native C dynamic link library (DLL) files that are required for the application are included on the library path in the JVM profile.

For CORBA stateless objects and enterprise beans, you need to include the following items on the class path that will be used for the request processor program:

- The JAR files for CORBA stateless objects.
- Any classes, such as classes for utilities, that are used by CORBA stateless objects or enterprise beans, but are not included in the JAR files.

Use the standard class path.

You do not need to include the deployed JAR files (DJARs) for enterprise beans on the class path.

When you add any class to a class path, remember:

- You can edit JVM profiles and JVM properties files in a standard text editor.
- The name of the class itself (including the name of the package, if the program has been built as a package) is not specified in the JVM profile or JVM properties file. The options in the JVM profile or JVM properties file specify the *path* to the class—that is, the full path of the z/OS UNIX directory in which a class loader will be able to find the class or the package containing the class. The rule is to *stop* specifying the path, at the point where you would *start* specifying the name of the class in the JVMCLASS attribute in a PROGRAM resource definition (see "Setting up a PROGRAM resource definition for a Java program to run in a JVM" on page 144).
- If any of your Java application programs are built as a package (that is, compiled using a Java package statement), and you would use the Java class name qualified by the package name (the fully qualified class name) in the PROGRAM resource definition, do not include the package name as part of the path. For example, the source of the CICS `HelloCICSWorld` sample program begins with:

  ```
  package examples.HelloWorld;
  ```

  In this case, the package name should be included in the class name in the PROGRAM resource definition; for example, as JVMCLASS(examples.HelloWorld.HelloCICSWorld). When you create the sample program using the supplied `HelloCICSWorld.mak` makefile, it is installed in the `/examples/HelloWorld/` subdirectories. When you specify an entry on the class path for this Java program, you need to omit the `/examples/HelloWorld/` subdirectories. For example, if the samples have been created in the location

  ```
  /u/MySamples/examples/HelloWorld/
  ```

the correct entry to enable the JVM to find this Java program is

```
CLASSPATH_SUFFIX=/u/MySamples
```

omitting the package name.

- If the program has not been built as a package, and you would just use the class name in the PROGRAM resource definition, then the path must specify all the subdirectories, including the subdirectory containing the class.
- Where classes or packages have been placed in JAR files (with the extension `.jar`), include the name of the JAR file on the class path as if it were the name of a directory. (Remember that *deployed* JAR files do not need to be placed on a class path.)
- Use a colon as the separator between paths that you specify on a class path. To include line breaks, use a backslash and a blank (\ ). "Rules for coding JVM profiles" on page 107 has a full explanation of how to code class paths and other items in a JVM profile or JVM properties file.

Whenever you set up a new Java program:

Procedure

1. Locate the JVM profile, and its associated JVM properties file, for the JVM which the Java program will use. The JVM profile is specified on the JVMPROFILE attribute of the PROGRAM resource definition, and the JVM properties file is referenced by the JVMPROPS option in the JVM profile. CICS looks for JVM profiles in the z/OS UNIX directory that is specified by the **JVMPROFILEDIR** system initialization parameter.
2. Identify any native C dynamic link library (DLL) files that are required by application code. Middleware and tooling supplied by IBM or by vendors might require DLL files to be added to the library path; for example, DLL files are needed to use the DB2 JDBC drivers. You might also have native code associated with a class that you have written.
3. Include the native libraries on the library path, by listing them under the LIBPATH_SUFFIX option in the JVM profile for the JVM in which the program will run. "Options for JVMs in a CICS environment" on page 111 has more information about the LIBPATH_SUFFIX option.
4. Place the application classes on the standard class path. The standard class path is defined by the CLASSPATH_SUFFIX option in the JVM profile for the JVM in which the program will run. For CORBA stateless objects or enterprise beans, use the profile for the JVM for the request processor program. The default request processor program is DFJIIRP, and the default JVM profile for DFJIIRP is DFHJVMCD
5. If you have added classes or native libraries to JVM profiles or properties files in a CICS region where JVMs are already running, issue the **CEMT PERFORM JVMPOOL PHASEOUT** command for each JVM profile that is affected. This command marks all the existing JVMs with your chosen profile for deletion. The existing JVMs were built with the old versions of the JVM profile or properties file. When each old JVM has finished running its current Java program, it terminates. If requests are waiting, CICS starts a new JVM in its place, or you can start new JVMs manually using the **CEMT PERFORM JVMPOOL START** command. The new JVMs use your new versions of the JVM profiles or properties files, including the new classes and native libraries. The shared class cache updates itself automatically.

What to do next

If you need to check the contents of the class paths for a particular JVM profile (including the base library path and the base class path built by CICS, which are not visible in the JVM profile), you can temporarily specify the PRINT_JVM_OPTIONS=YES option in the JVM profile. When this option is specified, all the options passed to the JVM at startup, including the contents of the class paths, are printed to SYSPRINT. The output is produced every time a JVM is started with this option in its profile, so you should add the option to the appropriate JVM profile, wait for a JVM to be started with the profile (or issue the **PERFORM JVMPOOL** command to manually start a JVM with the profile), and then immediately remove the option from the profile.

Including CORBA stateless objects and enterprise beans on the class path

For CORBA stateless objects and enterprise beans, you need to include the following items on the class path that will be used for the request processor program.

- The JAR files for CORBA stateless objects.
- Any classes, such as classes for utilities, that are used by CORBA stateless objects or enterprise beans, but are not included in the JAR files.

Do not include the deployed JAR files (DJARs) for enterprise beans on the class path.

1. Identify the JVM profile to change. Normally your stateless CORBA objects and enterprise beans run under the `DFHJVMCD` JVM profile. However, if you are using REQUESTMODEL resources to cause CICS to process IIOP requests under a different transaction then you may be using a different JVM profile. CICS normally runs IIOP workloads using the CIRP transaction which in turn points to the `DFJIIRP` program. `DFJIIRP` specifies a JVMPROFILE of DFHJVMCD. If you have tailored CICS to use a different request processing transaction then you will have to follow the chain of resource definitions to find the JVM profile that will be used.
2. When you are adding these items to the class path, remember:
   - The name of the class itself is not specified. The options in a JVM profile or JVM properties file specify the *path* to the class, that is, the full path of the HFS directory in which a class loader will be able to find the class or the package containing the class. Where classes or packages have been placed in JAR files (with the extension `.jar`), this means that you need to include the name of the JAR file on the class path as if it were the name of a directory. (Remember that *deployed* JAR files do not need to be placed on a class path.) If you need to add any utility classes, see the guidance given earlier in "Adding application classes to the class paths for a JVM" on page 145.
   - Use a colon as the separator between paths that you specify on a class path. To include line breaks, use a backslash and a blank (\ ). "Rules for coding JVM profiles" on page 107 has a full explanation of how to code class paths and other items in a JVM profile or JVM properties file.
3. Specify these items on the standard class path by using the CLASSPATH_SUFFIX option in the JVM profile that you identified in step 1 (usually DFHJVMCD).

Managing your JVMs

CICS performs many of the tasks needed to manage the JVMs in your JVM pool, including creating new JVMs, reusing free JVMs, and terminating inactive JVMs. CICS provides some operator facilities which you can use to monitor and manage the JVM pool, including manual startup and termination of JVMs.

"How CICS manages JVMs in the JVM pool" on page 79 and "How CICS allocates JVMs to applications" on page 82 explain how CICS performs its tasks. You can:

- Select an appropriate **MAXJVMTCBS** limit for your JVM pool, to prevent MVS storage constraints. "How CICS manages JVMs in the JVM pool" on page 79 explains the issues associated with **MAXJVMTCBS**, and what happens when an MVS storage constraint occurs. Managing your JVM pool for performance, in the *CICS Performance Guide*, tells you how to work out an appropriate setting for the **MAXJVMTCBS** system initialization parameter.
- Specify an appropriate timeout threshold for each type of JVM using your JVM profiles. "How CICS manages JVMs in the JVM pool" on page 79 explains how the timeout threshold works. If you change a timeout threshold while CICS is running, you can implement the change by terminating the JVMs with that profile.
- Monitor your JVM pool, the JVMs in it, and the JVM profiles that they use, and collect statistics about JVMs and JVM profiles. See "Monitoring JVM activity."
- Start up or terminate JVMs in the JVM pool, or disable the JVM pool so that it cannot service new requests. See "Manually starting and terminating JVMs and disabling the JVM pool" on page 152.
- Update the classes or JAR files used by Java applications, if you change these or introduce new ones. See "Changing classes or JAR files for Java applications" on page 154.
- Tune the JVM pool as a whole, and your individual JVMs, to achieve optimum performance. Java applications using a Java virtual machine (JVM): improving performance, in the *CICS Performance Guide*, tells you how to do this.

Monitoring JVM activity

You can use CICS commands and statistics to monitor JVM activity in your CICS region.

You can monitor:

- The JVM pool.
- The JVMs that CICS has in the JVM pool, and how CICS assigns them to requests.
- The JVM profiles that have been used to create JVMs, and the activity for each JVM profile.
- The Java programs that run in JVMs.

You can also monitor the TCBs used by JVMs, using the CICS dispatcher monitoring function. For example, the CEMT INQUIRE DISPATCHER command displays the number of active JVM TCBs and the maximum number of JVM TCBs.

Monitoring the JVM pool

You can use the **CEMT INQUIRE JVMPOOL** command (or the equivalent **EXEC CICS** command) to find out information about the JVM pool.

The command tells you about:

- The number of JVMs in the pool.
- The number of those JVMs that have been marked for deletion, but are still being used by a task.
- Whether the JVM pool is enabled or disabled (that is, whether it can service new requests or not).
- What trace options apply for the JVMs in the pool (this option is only available on the **EXEC CICS** version of the command).

Monitoring JVMs in the JVM pool

You can use the **EXEC CICS INQUIRE JVM** command or the **CEMT INQUIRE JVM** command to identify and report the status of each JVM in the JVM pool. You can also monitor the activity in the JVM pool using the CICS statistics.

Using the **EXEC CICS INQUIRE JVM** command, you can inquire on a specific JVM, or you can browse through all the JVMs in the JVM pool. Using the CEMT INQUIRE JVM command, you can list all the JVMs in the JVM pool, or inquire on all JVMs in a specified state.

The commands tell you about:

- The JVM profile and execution key of the JVMs in the pool.
- Which of the JVMs in the pool use the shared class cache.
- The age of each JVM.
- The task to which a JVM is allocated, and the time it has been allocated to the task.
- JVMs that are being phased out as a result of a **CEMT SET JVMPOOL PHASEOUT**, **PURGE** or **FORCEPURGE** command, or a **CEMT PERFORM CLASSCACHE PHASEOUT**, **PURGE** or **FORCEPURGE** command (or the equivalent **EXEC CICS** commands).

You can also monitor the activity in the JVM pool using the CICS statistics. Use the **EXEC CICS COLLECT STATISTICS** command, or the **CEMT PERFORM STATISTICS** command, with the relevant options to collect these statistics. Some useful statistics are the JVM pool statistics (JVMPOOL option), the TCB Mode statistics (DISPATCHER option), the JVM profile statistics (JVMPROFILE option), and the JVM program statistics (JVMPROGRAM option). These statistics can tell you, among other things:

- How many JVMs of a particular profile, on a particular TCB mode, are in the JVM pool (from the JVM profile statistics).
- How many requests were made for a JVM of a particular profile, on a particular TCB mode (from the JVM profile statistics).
- How many times a request for a JVM had to wait because there was no JVM available with an execution key and profile matching the request (from the TCB pool statistics for the JVM pool). This includes both requests that were eventually assigned a suitable JVM, and requests to which CICS decided to assign a mismatching or stolen JVM, rather than make them wait any longer. This figure can also include serialization waits, that is, time spent waiting to obtain any required locks.
- How long these requests spent waiting (from the TCB pool statistics for the JVM pool).
- How many times a request for a JVM was assigned a JVM that had the wrong profile or the wrong execution key (from the JVM profile statistics). These incidents of mismatching and stealing are broken down by JVM profile, so you can see if a particular profile is causing excess stealing activity.

Monitoring the use of JVM profiles

You can use the **EXEC CICS INQUIRE JVMPROFILE** command in browse mode to find out what JVM profiles have been used in this CICS execution. You can also collect CICS statistics for JVM profiles.

INQUIRE JVMPROFILE only finds JVM profiles that have been used during the lifetime of the CICS region. The command returns each 8–character JVM profile name, as used in a PROGRAM resource definition, and the full path name of the z/OS UNIX file for that JVM profile. (Note that there is no CEMT equivalent for this command.) The command also tells you whether or not JVMs with that profile use the shared class cache.

You can collect statistics for JVM profiles by using the **EXEC CICS COLLECT STATISTICS** command, or the **CEMT PERFORM STATISTICS** command, with the JVMPROFILE option. The statistics are broken down by JVM profile and execution key, and they show, among other things:

- The number of requests made by applications for JVMs of this profile.
- The total, current and peak number of JVMs of this profile that were in the JVM pool.
- The number of JVMs of this profile that were destroyed because CICS was short on storage.
- The incidence of TCB stealing by, and from, JVMs of this profile.
- The Language Environment heap storage and JVM heap storage used by JVMs of this profile.

Interpreting JVM statistics, in the *CICS Performance Guide*, has more information about JVM statistics, and tells you how to find the full listings and reports for these statistics.

Monitoring JVM programs

You can use the **EXEC CICS COLLECT STATISTICS** command, or the **CEMT PERFORM STATISTICS** command, with the JVMPROGRAM option, to collect statistics on Java programs that run in a JVM.

CICS does not collect statistics for these programs when a **COLLECT** or **PERFORM STATISTICS PROGRAM** command is issued, because the JVM programs are not loaded by CICS.

The JVM program statistics show, for each program:

- The JVM profile that the program requires (as specified in the JVMPROFILE attribute of the PROGRAM resource definition).
- The execution key that the program requires (CICS key or user key, as specified in the EXECKEY attribute of the PROGRAM resource definition).
- The main class in the program (the Java class whose public static main method is to be invoked, as specified in the JVMCLASS attribute of the PROGRAM resource definition).
- The number of times that the program has been used.

Interpreting JVM statistics, in the *CICS Performance Guide*, has more information about JVM statistics, and tells you how to find the full listings and reports for these statistics.

Manually starting and terminating JVMs and disabling the JVM pool

CICS starts up JVMs in response to the requirements of applications, and reduces the number of available JVMs automatically if the workload does not require them. You can also control the JVM pool using CICS commands; you can start up and terminate JVMs, and disable the JVM pool temporarily. This manual control lets you implement changes to JVM profiles or suspend activity in the JVM pool. You can also use it to create JVMs in advance of application requests.

CICS normally manages the startup and termination of JVMs in order to achieve a balanced level of capacity in the JVM pool to meet the demand from applications. CICS has sophisticated mechanisms to manage the number and type of JVMs in the pool, particularly when there is a need to optimize the performance of complex workloads at times of peak demand.

You might want to start up or terminate JVMs manually in certain situations:

- You need to update JVMs if you make changes to your JVM profiles or JVM properties files while CICS is running, including adding new classes or JAR files to class paths.
- If your Java workload is regular, predictable, and involves a limited number of different JVM profiles, you could consider starting up JVMs in advance of the demand from applications, so that they are ready for use as soon as they are required.

Starting JVMs using CICS commands

To start up JVMs manually, use the **EXEC CICS** or **CEMT PERFORM JVMPOOL** command. You need to specify the number of JVMs to be started, and the JVM profile and execution key that is to be used for them.

The number that you specify, added to the number of JVMs that already exist in the JVM pool, must not exceed the **MAXJVMTCBS** limit for the CICS region. You can check this by issuing the **EXEC CICS** or **CEMT INQUIRE DISPATCHER** command. **MAXJVMTCBS** shows the limit, and ACTJVMTCBS shows the number of JVMs that currently exist.

CICS does not start all the JVMs at once, but schedules the starts over a short period of time. Each JVM is available for use by an application as soon as it has been started. If a JVM is not used by an application, then like any other idle JVM, it becomes eligible for automatic termination at the timeout threshold that you have specified in the JVM profile.

If you have just terminated JVMs in order to implement changes to JVM profiles, and application activity in the CICS region is low, you can use the **PERFORM JVMPOOL** command to start a JVM of the type where you applied the changes. This enables you to confirm, without waiting for an application request, that the JVM is able to start with the changed profile, and that the classes specified on your class paths can be loaded.

If the Java workload in your CICS region is regular and predictable, you might want to use the manual startup facility to create a JVM pool that anticipates the needs of your applications, rather than allowing CICS to do this in response to demand. This strategy might reduce the delay time for applications in periods when workload is increasing.

By configuring the timeout threshold (which defaults to 30 minutes), and starting up JVMs in advance of need, you could structure a JVM pool that always has enough capacity available for your requirements. For example, you could start up a sufficient number of JVMs to handle your peak workloads, with their timeout thresholds set so that they are only eligible for automatic termination after 24 hours of idleness. (You might want to set up a task that starts the appropriate number of JVMs when the CICS region is started.) With a JVM pool like this, CICS would not terminate the JVMs automatically at times of the day when the workload is reduced. They would only be terminated if the system was idle for an extended period, or if your workload reduced over the long term.

When you start up JVMs manually with a particular JVM profile, they are eligible for mismatching or stealing in the same way as JVMs started by CICS. Mismatching and stealing change the JVM profile or user key, so the JVM can no longer be used by the applications for which you originally started it up. Mismatching and stealing also involve restarting the JVM, which can negate any benefit you experience from starting the JVMs in advance. The possibility of mismatching and stealing increases with the number of different JVM profiles in the CICS region, so if you want to structure a JVM pool manually, the benefit is likely to be greatest if your applications use only one or a small number of JVM profiles.

Terminating JVMs

To terminate JVMs, use the CEMT or **EXEC CICS PERFORM JVMPOOL** command. You can choose to terminate all the JVMs in the JVM pool, or you can specify a JVM profile to terminate only the JVMs with that profile.

You need to terminate JVMs to implement changes to JVM profiles or to add new application classes. Changes to existing classes on the standard class path do not require termination of the JVMs. The standard class path, is the recommended choice for standalone JVMs, but if you are in the process of migrating from resettable to continuous JVMs, you might still have classes on the shareable application class path in standalone JVMs.

The **PERFORM JVMPOOL** command does not terminate the shared class cache. The shared class cache updates itself automatically when classes are changed or new classes are added, so you do not need to terminate it in this situation.

To minimize disruption to your applications, try to terminate only those JVM profiles where you have made changes to the JVM profile, its associated JVM properties file, or the applications that use it. Terminating a subset of the JVM pool is more efficient than terminating the whole JVM pool. Make sure that you do terminate all the JVMs affected by your changes. For example, a shared Java class which you have changed might be listed on the class path in more than one JVM profile. In certain unusual circumstances, an application class might be used by JVMs with more than one profile, but this might not be obvious from the JVM profiles. This might be an issue, for example, if you use custom classloaders, or instantiate classes through reflection, or have enterprise beans which call other enterprise beans. If you are not sure whether an application class is used by JVMs with more than one profile, you might prefer to be safe and terminate the whole JVM pool.

CICS starts up new JVMs as soon as it receives requests from applications for each type of JVM. If you prefer, you can start JVMs manually using the **PERFORM JVMPOOL** command. If you have made any changes to the JVM profiles, the new

JVMs use the changed options. If you have made any changes to your Java applications, the new JVMs load the new or changed classes.

Disabling the JVM pool

To suspend all activity in the JVM pool, use the **EXEC CICS** or **CEMT SET JVMPOOL** command to set the status to DISABLED. In this state, the JVM pool cannot service new requests.

When you disable the JVM pool, the JVMs in it are retained, but new Java programs cannot use them until you enable the JVM pool again. Java programs that are already using a JVM are allowed to finish running. To re-enable the JVM pool, use the **EXEC CICS** or **CEMT SET JVMPOOL** command to set the status to ENABLED.

Changing classes or JAR files for Java applications

If you change any of your Java applications, you need to terminate and restart the JVMs that run those applications in order to load the changed resources. You also need to terminate and restart the JVMs if you make any changes to the class paths, including adding new resources to the class paths or changing the names of any of the files, because JVMs do not recognize changes to their profiles; new JVMs must be started to pick up the changes.

Before you begin

First compile and package the changed or new application, and update the changed files in your z/OS UNIX file system.

Procedure

1. If you have changed the contents of a class or JAR file but kept the same name:
   a. Issue the CEMT PERFORM JVMPOOL PHASEOUT command for each JVM profile that lists the changed file. Other JVMs that do not run this application can continue to run.
   b. If requests are waiting for JVMs with the profiles that you phased out, CICS starts new JVMs, or you can start new JVMs manually using the CEMT PERFORM JVMPOOL START command. A Version 6 shared class cache updates itself automatically when the new JVMs load the changed classes, so you do not have to restart it.
2. If you change the names of any of the files for your Java applications, or introduce new Java programs, follow the instructions in "Adding application classes to the class paths for a JVM" on page 145. That topic explains how to add the new classes to the class paths in the appropriate JVM profile or properties file, and how to phase out and restart the JVMs in order to load the new classes.

Problem determination for JVMs

Many of the usual sources of CICS diagnostic information contain information that applies to JVMs. In addition to this CICS-supplied information, there are a number of interfaces specific to the JVM that you can use for problem determination.

The CICS diagnostic information that applies to JVMs includes:

- Abend codes and messages. The *CICS Messages and Codes* lists the messages that apply to the SJ (JVM) domain. These are in the format DFHSJxxxx.

- Statistics. “Monitoring JVM activity” on page 149 lists the statistics information that CICS collects for JVMs.
- Monitoring data. Managing your JVM pool for performance, in the *CICS Performance Guide*, lists the monitoring data fields that relate to JVMs.
- The trace points for the SJ (JVM) domain. “CICS SJ domain tracing for JVMs” on page 160 and *CICS Trace Entries* have details of these trace points.

When the first JVM is started in a CICS region after initialization, CICS issues message DFHSJ0540, showing the version of Java in use. If you want details of the build of the IBM SDK for z/OS, Java Technology Edition in use, specify the option DISPLAY_JAVA_VERSION in a JVM profile. When a JVM starts with this option, CICS displays message DFHSJ0901 to show the SDK version and build information.

If you need to check the contents of the class paths for a particular JVM profile (including the base library path and the base class path built by CICS, which are not visible in the JVM profile), you can temporarily specify the PRINT_JVM_OPTIONS=YES option in the JVM profile. When this option is specified, all the options passed to the JVM at startup, including the contents of the class paths, are printed to SYSPRINT. The output is produced every time a JVM is started with this option in its profile, so you should add the option to the appropriate JVM profile, wait for a JVM to be started with the profile (or issue the PERFORM JVMPOOL command to manually start a JVM with the profile), and then immediately remove the option from the profile.

The JVM's own diagnostic tools and interfaces give you more detailed information about what is happening within the JVM than CICS can, because CICS is unaware of many of the activities within a JVM. Messages and diagnostic information from the JVM are written to the `stderr` log file for the JVM. If you encounter a Java problem, you should always consult this file, because it might contain useful information. For example, if CICS issues a message to indicate that the JVM has abended, the `stderr` log file is the primary source of diagnostic information. “Controlling the location for JVM stdout, stderr and dump output” on page 156 tells you how to control the location of output from the JVM, and how to redirect messages from JVM internals and output from Java applications running in a JVM.

The CICS documentation provides information about some more of the JVM's own diagnostic tools and interfaces:

- Defining and activating tracing for JVMs tells you how you can use the JVM's internal trace facility through the interfaces provided by CICS. The JVM's internal trace facility can provide detailed tracing of entry, exit, and event points within the JVM. This information is output as CICS trace.
- “Debugging an application that is running in a CICS JVM” on page 160 tells you how you can use a remote debugger to step through the application code for a Java application that is running in a JVM. CICS also provides a set of interception points (or “plugins”) in the CICS Java middleware, which allows additional Java programs to be inserted immediately before and after the application Java code is run, for debugging, logging, or other purposes. These plugins are described in “The CICS JVM plugin mechanism” on page 163.

Many more diagnostic tools and interfaces are available for the JVM. The *IBM Developer Kit and Runtime Environment, Java Technology Edition Diagnostics Guide*, which is available to download from www.ibm.com/developerworks/java/jdk/diagnosis/ has information about further facilities that can be used for problem determination for JVMs. You might find the following facilities especially useful:

- The JVM's internal trace facility can be used directly, without going through the interfaces provided by CICS. The *Diagnostics Guide* has information about the system properties that you can use to control the JVM's internal trace facility and to output JVM trace information to various destinations. You can use these system properties to output trace from any method or class within the JVM, and to find the value of any parameters and return types on the method call.
- If you experience memory leaks in the JVM, you can request a Heapdump from the JVM. A Heapdump generates a dump of all the live objects (objects still in use) that are in the JVM's heap.
- The HPROF profiler, which is shipped with the IBM SDK for z/OS, Java Technology Edition, provides performance information for applications that run in the JVM, so you can see which parts of a program are using the most memory or processor time.
- The JVM provides interfaces for monitoring, profiling, and RAS (Reliability, Availability and Serviceability).

With all interfaces, options or system properties available for the IBM JVM which are not specific to the CICS environment, the IBM JVM's own documentation should be considered the primary source of information, and the CICS documentation should be considered a secondary source of information.

When developing Java applications for JVMs in CICS, it is important to consider the way in which CICS reuses JVMs and the requirements for transaction isolation in CICS. "Programming for JVMs in CICS" on page 133 explains some key considerations to help you avoid problems in this area. If a Java application works correctly on its first use in a given JVM, but does not behave correctly on subsequent uses, then the problem is likely to be due to isolation issues. In this case, using the CICS JVM Application Isolation Utility as part of your problem determination work might help to identify the cause of the problem. "Auditing Java applications for the use of static variables" on page 137 explains how to use the utility.

If you are using enterprise beans, Chapter 26, "Dealing with CICS enterprise bean problems," on page 337 has more information about issues that apply specifically to them.

Controlling the location for JVM stdout, stderr and dump output

Output from Java applications running in a JVM is normally written to the z/OS UNIX files that are named by the STDOUT and STDERR options in the JVM profile for the JVM. JAVADUMP files are written to the JVM's working directory on z/OS UNIX, and the more detailed Java TDUMPs are written to the file named by the JAVA_DUMP_TDUMP_PATTERN option. Most of these file names can be customized at runtime to uniquely identify the JVMs that produced them. During application development, you can also redirect the output from the JVM and messages from JVM internals using a Java class.

In the standard setup for a CICS JVM, the file named by the STDOUT option in the JVM profile is used for System.out requests, and the file named by the STDERR option is used for System.err requests. The output files are z/OS UNIX files located in the working directory named by the WORK_DIR option in the JVM profile.

You can specify a fixed file name for the `stdout` and `stderr` files. However, if you use a fixed file name, the output from all the JVMs which were created with that

JVM profile is appended to the same file, and the output from different JVMs is interleaved with no record headers. This is not helpful for problem determination.

A better choice is to specify a variable file name for the `stdout` and `stderr` files. When you do this, the files can be made unique to each individual JVM during the lifetime of the CICS region. You can also include additional identifying information.

- The unique JVM number differentiates the JVM from any other JVMs in the CICS region. The JVM number used in CICS is the same number that is used to identify the JVM in the z/OS UNIX environment, where it is known as the process id (PID) for the JVM. You can specify this number as part of the file name using the &JVM_NUM; symbol, or using the **-generate** option.
- You can include the CICS region applid in the file name by using the &APPLID; symbol, or the **-generate** option.
- You can include a time stamp in the file name using the **-generate** option.

Other identifying information in file names includes the &DATE; and &TIME; symbols.

&DATE; is replaced by the current date in the form Dyymmdd

&TIME; is replaced by the current time in the format Thhmmss.

The location for JAVADUMP files output from the JVM is the working directory on z/OS UNIX named by the WORK_DIR option in the JVM profile. JAVADUMP files are uniquely identified by a timestamp in their names, and you cannot customize the names for these files.

TDUMPs output from the JVM, which contain more detailed dump output including the JVM's address space, are written to a data set destination. The name of the destination is specified by the JAVA_DUMP_TDUMP_PATTERN option in the JVM profile. You can use the &APPLID;, &DATE; &JVM_NUM;, and &TIME; symbols in this value to make the name unique to the individual JVM, as shown in the CICS-supplied sample JVM profiles. Note that in this context, CICS might have to modify the JVM number to conform to MVS dataset naming standards.

The JVM writes information to its `stderr` file when it generates a JAVADUMP or a TDUMP. The *IBM Developer Kit and Runtime Environment, Java Technology Edition Diagnostics Guide*, which is available to download from www.ibm.com/developerworks/java/jdk/diagnosis/ has more information about the contents of JAVADUMP and TDUMP files.

During application development, you can use the USEROUTPUTCLASS option in a JVM profile to name a Java class that intercepts and redirects the output from the JVM and messages from JVM internals. You can add time stamps and headers to the output records, and identify the output from individual transactions running in the JVM. CICS supplies sample classes which perform these tasks. Specifying this option has a negative effect on the performance of JVMs, so it should not be used in a production environment.

Redirecting JVM stdout and stderr output during application development (USEROUTPUTCLASS)

The USEROUTPUTCLASS option enables developers using the same CICS region to separate out their own JVM stdout and stderr output, and direct it to an identifiable destination of their choice. You can use a Java class to redirect the output, and you can add time stamps and headers to the output records. Dump output cannot be intercepted by this method.

Specifying the USEROUTPUTCLASS option has a negative effect on the performance of JVMs. For best performance in a production environment, you should not use this option.

Output written to System.out() or System.err(), either by an application or by system code, can be redirected by the output redirection class. The z/OS UNIX files named by the STDOUT and STDERR options in the JVM profile are still used for some messages issued by the JVM, or if the class named by the USEROUTPUTCLASS option is unable to write data to its intended destination. You should therefore still specify appropriate file names for these files.

To use the USEROUTPUTCLASS option, specify `USEROUTPUTCLASS=[java class]` in a JVM profile, naming the Java class of your choice. (The class extends java.io.OutputStream.) The CICS-supplied sample JVM profiles DFHJVMPR and DFHJVMCD contain the commented-out option `USEROUTPUTCLASS=com.ibm.cics.samples.SJMergedStream`, which names the CICS-supplied sample class. Uncomment this option to use the com.ibm.cics.samples.SJMergedStream class to handle output from JVMs with that profile. CICS also supplies an alternative sample Java class, com.ibm.cics.samples.SJTaskStream.

The source for the CICS-supplied user output classes is provided as samples, so you can modify the classes as you want, or write your own classes based on the samples. The *CICS Customization Guide* tells you how to do this.

The class that you are using must be present in a directory on an appropriate class path in the JVM profile. The CICS-supplied sample class is automatically included on an appropriate class path and you do not need to specify it explicitly in the JVM profile. If you supply your own output redirection class, add the directory to the standard class path, using the CLASSPATH_SUFFIX option, in the JVM profile where you specified the USEROUTPUTCLASS option.

The CICS-supplied sample classes com.ibm.cics.samples.SJMergedStream and com.ibm.cics.samples.SJTaskStream

For Java applications executing on the initial process thread (IPT), which are able to make CICS requests, the intercepted output from the JVM can be written to a transient data queue, and you can add time stamps, task and transaction identifiers, and program names. This enables you to create a merged log file containing the output from multiple JVMs. You can use this log file to correlate JVM activity with CICS activity. The CICS-supplied sample class, **com.ibm.cics.samples.SJMergedStream**, is set up to create merged log files like this.

The com.ibm.cics.samples.SJMergedStream class directs output from the JVM to the transient data queues CSJO (for stdout output), and CSJE (for stderr output, internal messages, and unresettable event logging). These transient data queues are supplied in group DFHDCTG, and they are indirected to CSSL, but they can be redefined if necessary.

In particular, note that the length of messages issued by the JVM can vary, and the maximum record length for the CSSL queue (133 bytes) might not be sufficient to contain some of the messages you receive. If this happens, the sample output redirection class issues an error message, and the text of the message might be affected.

If you find that you are receiving messages longer than 133 bytes from the JVM, you should redefine CSJO and CSJE as separate transient data queues. Make them extrapartition destinations, and increase the record length for the queue. You can allocate the queue to a physical data set or to a system output data set. You might find a system output data set more convenient in this case, because you do not then need to close the queue in order to view the output. The *CICS Resource Definition Guide* tells you how to define transient data queues. If you redefine CSJO and CSJE, ensure that they are installed as soon as possible during a cold start, in the same way as for transient data queues that are defined in group DFHDCTG.

If the transient data queues CSJO and CSJE cannot be accessed, output is written to the z/OS UNIX files `/work_dir/applid/stdout/CSJO` and `/work_dir/applid/stderr/CSJE`, where `work_dir` is the directory specified on the WORK_DIR option in the JVM profile, and `applid` is the applid identifier associated with the CICS region. If these files are unavailable, the output is written to the z/OS UNIX files named by the STDOUT and STDERR options in the JVM profile.

As well as redirecting the output, the class adds a header to each record containing applid, date, time, transid, task number and program name. The result is two merged log files for JVM output and for error messages, in which the source of the output and messages can easily be identified.

For Java applications executing on threads other than the initial process thread (IPT), which are not able to make CICS requests, the output from the JVM cannot be redirected using CICS facilities. The com.ibm.cics.samples.SJMergedStream class still intercepts the output and adds a header to each record containing applid, date, time, transid, task number and program name. The output is then written to the z/OS UNIX files `/work_dir/applid/stdout/CSJO` and `/work_dir/applid/stderr/CSJE` as described above, or if these files are unavailable, to the z/OS UNIX files named by the STDOUT and STDERR options in the JVM profile.

As an alternative to creating merged log files for your JVM output, you can direct the output from a single task to z/OS UNIX files, and add time stamps and headers, to provide output streams that are specific to a single task. The CICS-supplied sample class, **com.ibm.cics.samples.SJTaskStream** is set up to do this. The class directs the output for each task to two z/OS UNIX files, one for stdout output and one for stderr output, that are uniquely named using a task number (in the format `YYYYMMDD.Task.`*`tasknumber`*). The z/OS UNIX files are stored in the directory `/work_dir/applid/stdout` for stdout output, or `/work_dir/applid/stderr` for stderr output, where `work_dir` is is the directory specified on the WORK_DIR option in the JVM profile, and `applid` is the applid identifier associated with the CICS region. The process is the same for both Java applications executing on the IPT, and Java applications that are executing on other threads.

When an error is encountered by the CICS-supplied sample output redirection classes, one or more error messages are issued reporting this. If the error occurred while processing an output message, then the error messages are directed to System.err, and as such are eligible for redirection. However, if the error occurred while processing an error message, then the new error messages are sent to the file named by the STDERR option in the JVM Profile. This is done to avoid a recursive loop in the Java class. The classes do not return exceptions to the calling Java program.

The classes are shipped as a class file `dfjoutput.jar`, which is in the directory `/usr/lpp/cicsts/cicsts41/lib`, where `/usr/lpp/cicsts/cicsts41` is the install

directory for CICS files on z/OS UNIX. The source for the classes is also provided as samples, so you can modify the classes as you want, or write your own classes based on the samples. The *CICS Customization Guide* tells you how to customize these classes, or write your own classes based on the samples.

Control of Java dump options

The JAVA_DUMP_OPTS option in JVM profiles specifies the Java dump options for the JVM.

You can use this option to set the Java dump options of your choice.

Information about Java dump options can be found in the *IBM Developer Kit and Runtime Environment, Java Technology Edition Diagnostics Guide*, which is available to download from www.ibm.com/developerworks/java/jdk/diagnosis/.

CICS SJ domain tracing for JVMs

As well as the trace points produced by JVMs, CICS provides some standard trace points in the SJ (JVM) domain, to trace the actions that CICS takes in setting up and managing JVMs and the shared class cache. These are available at CICS trace levels 0, 1 and 2.

You can activate the SJ domain trace points at levels 0, 1 and 2 using the CETR Component Trace screens. Selecting tracing by component, in the *CICS Problem Determination Guide*, explains how to do this.

The SJ domain includes a level 2 trace point SJ 0224, which shows you a history of the programs that have used each JVM.

"JVM domain trace points", in the *CICS Trace Entries* manual, has details of all the standard trace points in the SJ domain.

Debugging an application that is running in a CICS JVM

The JVM in CICS supports the Java Platform Debugger Architecture (JPDA), which is the standard debugging mechanism provided in the Java 2 Platform. This architecture provides a set of APIs that allow the attachment of a remote debugger to a JVM.

About this task

A number of third party debug tools are available that exploit JPDA and can be used to attach to and debug a JVM that is running an enterprise bean, CORBA object or Java program. Typically the debug tool provides a graphical user interface that runs on a workstation and allows you to follow the application flow, setting breakpoints and stepping through the application source code, as well as examining the values of variables.

There is more information about JPDA at http://java.sun.com/javase/technologies/core/toolsapis/jpda/.

Attaching a debugger to a CICS JVM

To run a JVM in debug mode and allow a JPDA remote debugger to be attached, you need to set some options in the JVM profile for the JVM.

"Customizing or creating JVM profiles" on page 100 explains the procedure for customizing options in a JVM profile.

The specific options required for debugging are as follows:

-Xdebug
This is needed to start the JVM in debug mode (that is, with the JPDA interfaces active).

-Xrunjdwp:<option>=<value>, ...
This option specifies the details of the connection between the debugger and the CICS JVM. These details include the TCP/IP address to be used for the connection, and the sequence in which connection occurs. Different debuggers have different connection requirements and capabilities; refer to the documentation provided with the debugger. Some typical example settings are as follows:

-Xrunjdwp:transport=dt_socket,server=y,address=9876
This set of suboptions specifies that:
- The standard TCP/IP socket connection mechanism is used
- The server starts first (`server=y`) and waits for the debugger to attach to it
- The CICS JVM listens on TCP/IP port 9876 for a debugger to attach to it.

The CICS JVM waits after initialization for instructions from the debugger before executing the application code.

If you are using the Java debugger supplied with WebSphere Studio Enterprise Edition, you should specify the `-Xrunjdwp` option in your JVM profile. In addition, in WebSphere Studio you must create a Remote Java Application definition, within the Debug Perspective, that specifies:
- The IP address (or host name) of the z/OS system that hosts the CICS region.
- The TCP/IP port number (called "address" in the `-Xrunjdwp` syntax) that the CICS JVM is using. (This is the same number specified to CICS on the `-Xrunjdwp` option.)
- That a standard TCP/IP socket connection (Socket Attach) is to be used.

-Xrunjdwp:transport=dt_socket,address=bos.hurs.ibm.com:6780
This set of suboptions specifies that:
- The standard TCP/IP socket connection mechanism is used
- Omitting the `server` option defaults to `server=no`, which means the debugger starts first and waits for the JVM to attach to it
- The JVM attaches to a debugger that is running on a machine called `bos.hurs.ibm.com` on port number 6789.

After initialization the JVM waits for instructions from the debugger before executing the application code.

If your debugger is WebSphere Studio, you must specify `server=y`.

REUSE=NO
A JVM that has been run in debug mode is not a candidate for reuse. Set this option to `NO` to ensure that the JVM is discarded after the debug session.

"Options for JVMs in a CICS environment" on page 111 has full information about the options available in a JVM profile.

When you set these options in a JVM profile, any CICS JVM program that uses that profile runs in debug mode (and waits for attach from, or attempts to attach to a debugger). You should therefore ensure that the JVM profile applies only to programs that you wish to debug. Remember:

- **Never configure for debug** the JVM profiles that are involved with the shared class cache; that is, JVM profiles that specify CLASSCACHE=YES.JVM debugging is not supported for shared classes, and if you configure these JVM profiles for debug, CICS ignores your setting.
- **Avoid configuring for debug** the CICS-supplied sample JVM profile DFHJVMPR, and the JVM profile DFHJVMCD for CICS-supplied system programs. It is possible to configure these profiles for debug, provided they have not been changed to specify CLASSCACHE=YES, but because they are used as defaults within CICS, there is a strong risk that they will be used for programs other than those you want to debug.

Instead of configuring any of the CICS-supplied sample profiles for debug, you should create a separate JVM profile specifically for debug use, and set the appropriate CICS PROGRAM resource definition to use this debug JVM profile.

For enterprise beans, you need to specify the debug JVM profile in the PROGRAM definition for the request processor program that is used by the enterprise bean. The default request processor program, which is named by the default CIRP transaction on REQUESTMODEL definitions, is DFJIIRP. To modify CICS-supplied definitions for this purpose, such as those in CSD group DFHIIOP, you have to copy the definitions to your own group first. DFHIIOP is locked and cannot be modified. However, bear in mind that if you modify the PROGRAM definition for the default request processor program to use the debug JVM profile, there is a strong risk that it will be used for programs other than those you want to debug. It is safer to set up a different PROGRAM definition to be used by the enterprise beans that you want to debug.

Errors during initialization of the debug connection (for example incorrect TCP/IP host or port values) result in messages on the JVM standard output and standard error streams. “Controlling the location for JVM stdout, stderr and dump output” on page 156 tells you how to set the destination for these messages.

The debugger should give an indication that it has successfully attached to the CICS JVM. The initial state of the JVM (such as the identity of threads that have started, and system classes that are loaded) is visible in the debugger user interface. The JVM will have suspended execution, and the Java application in CICS (enterprise bean, CORBA object or Java program) will not yet have started. Your next action is normally to set a breakpoint at a suitable point in the Java application by specifying the full Java class name and source code line number. As the application class will not usually have been loaded at this point, the debugger indicates that activation of this breakpoint is deferred until the class is loaded. You should then let the JVM run through the CICS middleware code to the application breakpoint, at which point it suspends execution again. You can then examine loaded classes, and variables, set further breakpoints and step through code as required.

To terminate the debug session you can let the application run to completion, at which point the connection between the debugger and the CICS JVM closes. Some debuggers support forced termination of the JVM. This normally results in an abend and error messages on the CICS system console.

To fully enable the capabilities of a Java source code debugger, the Java code to be debugged must be compiled using the `-g` option on the Java compiler (`javac` command). Additional symbolic information is then preserved in the `.class` file, which is used when the debugger is attached at run time. IDEs usually support this compiler option via a user setting .

The CICS JVM plugin mechanism

In addition to the standard JPDA debug interfaces in the JVM, CICS provides a set of interception points in the CICS Java middleware, which can be of value to developers of debugging applications. These interception points (or plugins) allow additional Java programs to be inserted immediately before and after the application Java code is run.

Information about the application (for example classname and method name) is made available to the plugin programs. The plugin programs can also use the JCICS API to obtain information about the application. These interception points can be used in conjunction with the standard JPDA interfaces to provide additional CICS-specific debug facilities. They can also be used for purposes other than debugging, in a similar way to user exit points in CICS.

There are three Java exit points:

- A CICS EJB container plugin providing methods that are called immediately before and after an EJB method is invoked.
- A CICS CORBA plugin providing methods that are called before and after a CORBA method is invoked.
- A CICS Java Wrapper plugin providing methods that are called immediately before and after a Java program is invoked

Debug plugins can be used with continuous and single-use JVMs (with REUSE=YES or REUSE=NO in the JVM profile). When you use plugin programs to debug Java applications, you need to specify the classes on the standard class path for the JVM which will be used by the application that is to be debugged. The standard class path is specified by the CLASSPATH_SUFFIX option in the JVM profile. “Adding application classes to the class paths for a JVM” on page 145 tells you how to do this; classes for plugin programs can be added in the same way as classes for ordinary applications.

The programming interface consists of two Java interfaces. **DebugControl** (full name: `com.ibm.cics.server.debug.DebugControl`) defines the method calls that can be made to a user-supplied implementation, and **Plugin** (full name: `com.ibm.cics.server.debug.Plugin`) provides a general purpose interface for registering the plugin implementation. These interfaces are supplied in `dfjwrap.jar`, and documented in JAVADOC HTML (see “The JCICS class library” on page 17 for more information).

The code fragment in Figure 8 on page 164 shows an example implementation of the DebugControl interface.

```
public interface DebugControl
{
    // called before an application object method or program main is invoked
    public void startDebug(java.lang.String className,java.lang.String methodName);

    // called after an application object method or program main is invoked
    public void stopDebug(java.lang.String className,java.lang.String methodName);

    // called before an application object is deleted
    public void exitDebug();

}
public interface Plugin
{
    // initaliser, called when plugin is registered
    public void init();
}
```

Figure 8. Definitions of the DebugControl and Plugin interfaces

The code fragment in Figure 9 shows an example implementation of the DebugControl and Plugin interfaces.

```
import com.ibm.cics.server.debug.*;

public class SampleCICSDebugPlugin
    implements Plugin, DebugControl
{
    // Implementation of the plugin initialiser
    public void init()
    {
        // This method is called when the CICS Java middleware loads and
        // registers the plugin. It can be used to perform any initialisation
        // required for the debug control implementation.
    }

    // Implementations of the debug control methods
    public void startDebug(java.lang.String className,java.lang.String methodName)
    {
        // This method is called immediately before the application method is
        // invoked. It can be used to start operation of a debugging tool. JCICS
        // calls such as Task.getTask can be used here to obtain further
        // information about the application.
    }

    public void stopDebug(java.lang.String className,java.lang.String methodName)
    {
        // This method is called immediately after the application method is
        // invoked. It can be used to suspend operation of a debugging tool.
    }

    public void exitDebug()
    {
        // This method is called immediately before an application object is
        // deleted. It can be used to terminate operation of a debugging tool.
    }
}
```

Figure 9. Sample implementation of the DebugControl and Plugin interfaces

In order to activate a debug plugin implementation you need to set one or more of the following system properties in the JVM properties file for the JVM:

-
```
-Dcom.ibm.cics.server.debug.EJBPlugin=<fully qualified classname,
   for example com.ibm.cics.server.debug.SampleCICSDebugPlugin>
```

This is the EJB container debug plugin. If this is set, the supplied plugin is registered by Java code in the CICS EJB server layer when the EJB container is initialized.

- ```
  -Dcom.ibm.cics.server.debug.CORBAPlugin=<fully qualified classname,
     for example com.ibm.cics.server.debug.SampleCICSDebugPlugin>
  ```

  This is the CORBA debug plugin. If this is set, the supplied plugin is registered by Java code in the CICS ORB when the ORB is initialized.

- ```
  -Dcom.ibm.cics.server.debug.WrapperPlugin=<fully qualified classname,
     for example com.ibm.cics.server.debug.SampleCICSDebugPlugin>
  ```

  This is the CICS Java debug plugin. If this is set, the supplied plugin is registered by additional Java code in the JCICS wrapper when the Java program is run.

Note that more than one plugin interface may be triggered when a Java application is run. For example, if plugin implementations are registered for all three interfaces, and an enterprise bean method is run, the JCICS wrapper, CORBA and EJB plugins will be triggered in succession.

Part 4. CICS and IIOP

This section tells you what you need to know to configure CICS to support distributed IIOP applications.

Chapter 14. IIOP support in CICS

The Internet Inter-ORB protocol (IIOP) is a TCP/IP based implementation of the General Inter-ORB Protocol (GIOP) that defines formats and protocols for distributed applications.

It is part of the Common Object Request Broker Architecture (CORBA). Both client and server systems require a CORBA Object Request Broker (ORB) to implement IIOP interoperability.

The Common Object Request Broker Architecture (CORBA) is a specification for a standard object-oriented architecture for distributed applications. It was defined by a consortium of over 500 information technology organizations called The Object Management Group (OMG). You can read the CORBA *Architecture and Specification* document at their Web site: http://www.omg.org/

CICS provides an ORB and support for IIOP defined by CORBA 2.3.

The Object Request Broker (ORB)

CORBA uses a **broker**, or intermediary, to handle requests between clients and servers in the system. The broker chooses the best server to meet the client's request and separates the **interface** that the client sees from the **implementation** of the server.

The broker, known as the ORB, intercepts client method calls and is responsible for finding objects that can implement requests, passing them parameters, invoking their methods, and returning results. The client does not need to know where the object is located, its programming language, its operating system, or any other system aspects that are not part of the object's interface.

In this way, the ORB provides interoperability between applications on different machines in heterogeneous distributed environments, and interconnects multiple object systems.

The CICS ORB implements the following level of function:

- Support for CORBA Version 2.3, *except for*:
  - Stateful CORBA objects (only stateless CORBA objects are supported).

    Note: The only exception to this rule is stateful session beans—which *are* supported.
  - The Dynamic Invocation Interface (DII).
  - The Dynamic Skeleton Interface (DSI).
  - GIOP 1.1 fragments.
  - The Portable Object Adapter (POA).
  - Bi-directional GIOP
- Support for IIOP 1.2—including GIOP 1.2 fragments.
- Support for both inbound and outbound IIOP requests. IIOP applications can act as both client and server.
- Support for **transactional objects**. CICS method invocations may participate in Object Transaction Service (OTS) distributed transactions. If a client calls an IIOP application within the scope of an OTS transaction, information about the

transaction flows as an extra parameter on the IIOP call. If the client ORB sends an OTS Transaction Service Context and the target stateless CORBA object implements `CosTransactions::TransactionalObject`, the object is treated as transactional.

Note: An **OTS transaction** is a distributed unit of work, not a CICS transaction instance or resource definition. For a description of a CICS transaction, see "CICS transactions" on page 13.

ORB function is implemented in CICS by:

- The CICS sockets domain listener
- The CICS IIOP request receiver
- The CICS IIOP request processor

CICS IIOP application models

IIOP applications are client/server object-oriented programs executing in a TCP/IP network.

CICS supports the following types of IIOP application:

`Stateless CORBA objects`
Stateless CORBA objects are Java server applications that communicate with a client application using the IIOP protocol. No state is maintained in object attributes between successive invocations of methods; state is initialized at the start of each method call and referenced by explicit parameters.

Stateless CORBA objects can receive inbound requests from a client and can also make outbound IIOP requests.

CICS stateless CORBA objects execute in a CICS JVM.

You can read more about CICS stateless CORBA objects in Chapter 29, "Stateless CORBA objects," on page 371.

`Enterprise beans`
Enterprise beans are portable Java server applications that use interfaces defined by Sun Microsystem's *Enterprise JavaBeans Specification, Version 1.1*. CICS has implemented these interfaces by mapping them to underlying CICS services.

Enterprise beans communicate using the Java Remote Method Invocation (RMI) interface. CICS supports RMI over IIOP, mediated by a CORBA Object Request Broker (ORB).

Enterprise beans can link to other CICS programs using the **CCI Connector for CICS TS**. You can also develop enterprise beans that use the JCICS class library to access CICS services or programs directly, but these server applications are not portable to a non-CICS platform.

Enterprise beans execute in a CICS JVM.

You can read more about enterprise beans in Chapter 18, "What are enterprise beans?," on page 213.

Some common CORBA terminology

These terms are used throughout this information segment.

CORBA
The Common Object Request Broker Architecture. An architecture and a specification for distributed, object-oriented, computing.

GIOP The General Inter-Orb Protocol. The CORBA data representation specification and interoperability protocol. It defines how different ORBs communicate; it does not define which transport protocol to use.

IDL Interface Definition Language. A definition language that is used in CORBA to describe the characteristics and behavior of a kind of object, including the operations that can be performed on it.

IIOP The Internet Inter-Orb Protocol. Defines how to send GIOP messages over a TCP/IP transport layer. IIOP is GIOP over TCP/IP.

Interface
Describes the characteristics and behavior of a kind of object, including the operations that can be performed on those objects. This maps to a Java **class**. In CORBA terminology, the client request specifies, in IDL, an interface that defines the server object.

IOR Interoperable Object Reference. A "stringified" reference to a remote CORBA object. It is published by the server ORB. The client application must have access to the IOR at runtime. The client ORB can deconstruct the IOR to determine (among other things) the location of the remote ORB and object, the maximum version of GIOP supported by the remote ORB, and any relevant CORBA services supported by the remote ORB.

Module
An IDL packaging construct containing interfaces. This maps to a Java **package**.

OMG The Object Management Group. The consortium of software organizations that has defined the CORBA architecture.

Operation
An action that can be performed on an object. This maps to a Java method. In CORBA terminology, the client requests an operation, defined in IDL, that is mapped to a method on the server object.

ORB The Object Request Broker. A CORBA system component that acts as an intermediary between the client and server applications. Both client and server platforms require an ORB; each is tailored for a specific environment, but supports common CORBA protocols and IDL.

RMI-IIOP
The Remote Method Invocation (RMI) over IIOP specification and protocol. The specification defines how to make the Java-specific RMI application architecture inter-operate, using CORBA protocols. This is the communication protocol used by enterprise beans.

Skeleton
A piece of code generated by the server IDL compiler. It is used by the server ORB to parse a message into a method call on a local (to the server) object.

Stub or proxy
A piece of code generated by the client IDL or RMI compiler. It is used by the client application to invoke methods on the remote object. The stub class calls methods on the client ORB, which in turn sends remote method requests to the server ORB. The stub class must be generated for the specific client ORB it is to be used with. If you use client ORBs from

different vendors, you should ensure that you are using client-side stubs generated using the tools provided with the correct client ORB.

Tie A piece of code generated by the RMI compiler. It is used by the server ORB to parse a message into a method call on a local (to the server) object.

Chapter 15. The IIOP request flow

This diagram shows the execution flow of an incoming request.

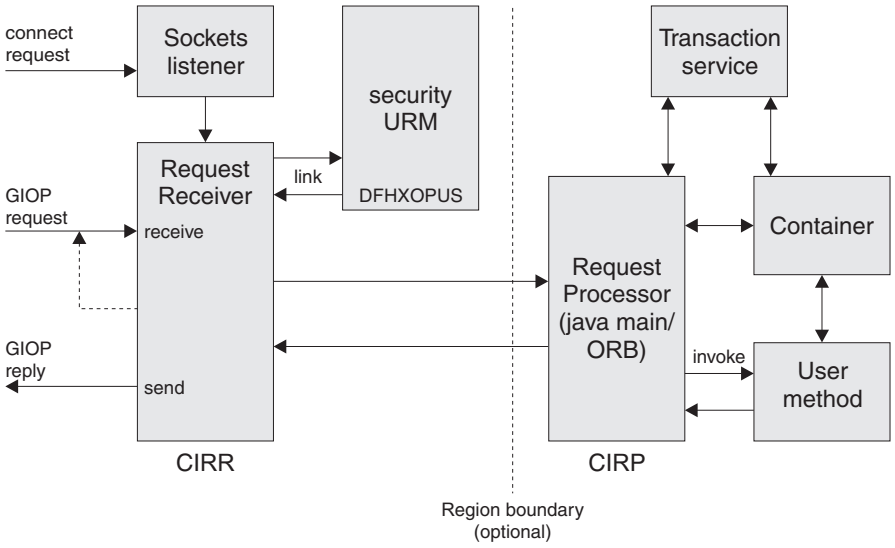

Figure 10. IIOP request execution flow

The TCP/IP listener

The CICS TCP/IP listener monitors specified ports for inbound requests. You specify IIOP ports and configure the listener by defining and installing TCPIPSERVICE resources.

The listener receives the incoming request and starts the transaction specified in the TCPIPSERVICE definition for that port. For IIOP services, this transaction resource definition must have the program attribute set to DFHIIRRS, the **request receiver** program. The default transaction name is **CIRR**.

Request receiver

The request receiver retrieves the incoming request and examines the contents of the GIOP formatted message stream. The following GIOP message types can be received and are handled as follows:

Request

- A CICS USERID is determined from Secure Sockets Layer (SSL) parameters, or by calling a CICS user-replaceable program specified by the TCPIPSERVICE resource definition. The CICS USERID is used for authorization of the request by the request processor.
- A CICS TRANSID is determined, from the message content, by comparison with installed REQUESTMODEL resource definitions. The CICS TRANSID defines execution parameters that are used if a new request processor instance is created to handle the request.
- The request is passed to the request processor using an associated **request stream**, which is an internal CICS routing mechanism. The object key in the request, or any transaction service context, determines if the request must be sent to an existing processor.

Note: A *transaction* in this context means a unit of work defined and managed using the **Object Transaction Service** (OTS) specification.

The request-handling logic uses a directory to determine if an IIOP request should be routed to an existing request processor instance (by means of its associated request stream). The directory, DFHEJDIR, relates request streams (and request processor instances) to OTS transactions and the object keys of stateful session beans that manage their own transactions. DFHEJDIR is a recoverable CICS file.

- Incoming GIOP 1.1 Fragments are rejected with a GIOP MessageError message.

`LocateRequest`
Locate requests have no **operation** or parameters. They are passed to a new instance of the request processor.

`CancelRequest`
A cancel request notifies a server that the client is no longer expecting a reply to a specified pending Request or LocateRequest message. This is an advisory message only, no reply is expected. A cancel request received during fragment processing causes the request in progress to be terminated. All other cancel requests are ignored.

`MessageError`
A message error indicates that the client has not recognized a reply that the request receiver has sent to it. This error is recorded for diagnostic purposes and a CloseConnection message sent to end the connection.

`Fragments`
A fragment is a continuation of a Request or a Reply. It contains a GIOP message header followed by data. Incoming GIOP 1.1 fragments are rejected with a GIOP MessageError message.

Linkage from the request receiver to the request processor can exploit CICS dynamic routing services to provide load balancing within the CICSplex.

The CIRR request receiver terminates when it has no further work to do. (That is, CIRR terminates when there are no outstanding GIOP requests to read from the TCPIPSERVICE and no outstanding responses to send from earlier requests. Should further workload arrive for the TCPIPSERVICE after the CIRR task has been terminated, a new CIRR task is started.)

`Request processor`
The request processor manages the execution of the IIOP request. It :

- Locates the object identified by the request
- For an enterprise bean request, calls the container to process the bean method
- For a request for a stateless CORBA object, processes the request itself (although the transaction service may also be involved)

The request processor instance that handles each IIOP request is configured by a CORBASERVER resource definition.

IIOP in a sysplex

You can implement a CICS CORBA server in a single CICS region.

However, in a sysplex it's likely that you'll want to create a server consisting of multiple regions. Using multiple regions makes failure of a single region less critical and enables you to use workload balancing. A **CICS logical server** consists of one or more CICS regions configured to behave like a single server.

Typically, a CICS logical server consists of:

- A set of cloned **listener regions** defined by identical TCPIPSERVICE resource definitions to listen for incoming IIOP requests.
- A set of cloned application-owning regions (AORs), each of which supports an identical set of IIOP applications or enterprise bean classes in an identically-defined CorbaServer. Multiple methods for the same OTS transaction are directed to the same AOR. Each AOR must have TCPIPSERVICE definitions that match those in the corresponding listener regions.

Note:

The listener regions and AORs may be separate or combined into listener/AORs. You must specify the following system initialization parameters:

IIOPLISTENER=YES
: Specify this value in a listener region, or in a combined listener/AOR. YES is the default value.

IIOPLISTENER=NO
: Specify this value in an AOR that is not also a listener region.

Workload balancing of IIOP requests

To balance client connections across the listener regions, you can use either IP routing or connection optimization by means of Domain Name System (DNS) registration.

To balance OTS transactions across a set of cloned AORs, you use distributed routing. To implement distributed routing, you can use either CICSPlex SM or a customized version of the CICS distributed routing program, DFHDSRP.

Domain Name System (DNS) connection optimization
: Connection optimization is a technique that uses DNS to balance IP connections in a sysplex domain. With DNS, multiple CICS systems are started to listen for IIOP requests on the same port (using Virtual IP addresses), and registered with MVS Workload Manager (WLM). Each client IIOP request contains a generic host name and port number. This host name is resolved to an IP address by DNS and WLM services.

  Connection Optimization using the WLM is described in the *z/OS Communication Server: IP Configuration Guide*.

Distributed routing
: Distributed routing is used to balance method calls for enterprise beans and CORBA stateless objects across a set of CICS application owning regions (AORs). The dynamic selection of the target is made by the workload manager—CICSPlex SM or a user-written distributed routing program—which selects the least loaded or most efficient application region. CICS invokes the workload manager for method requests that will run under a new, or no, OTS transaction, but not for method requests that will run under an existing OTS transaction; these are directed automatically to the AOR in which the existing OTS transaction runs. See Writing a distributed routing program , in the *CICS Customization Guide*, for guidance on writing a customized distributed routing

program. See Workload management and dynamic routing, in the *CICSPlex System Manager Managing Workloads* manual, for information about CICSPlex SM Workload Management.

The following diagram shows a CICS logical server. In this example, the listener regions and AORs are in separate groups, connection optimization is used to balance client connections across the listener regions, and distributed routing is used to balance OTS transactions across the AORs.

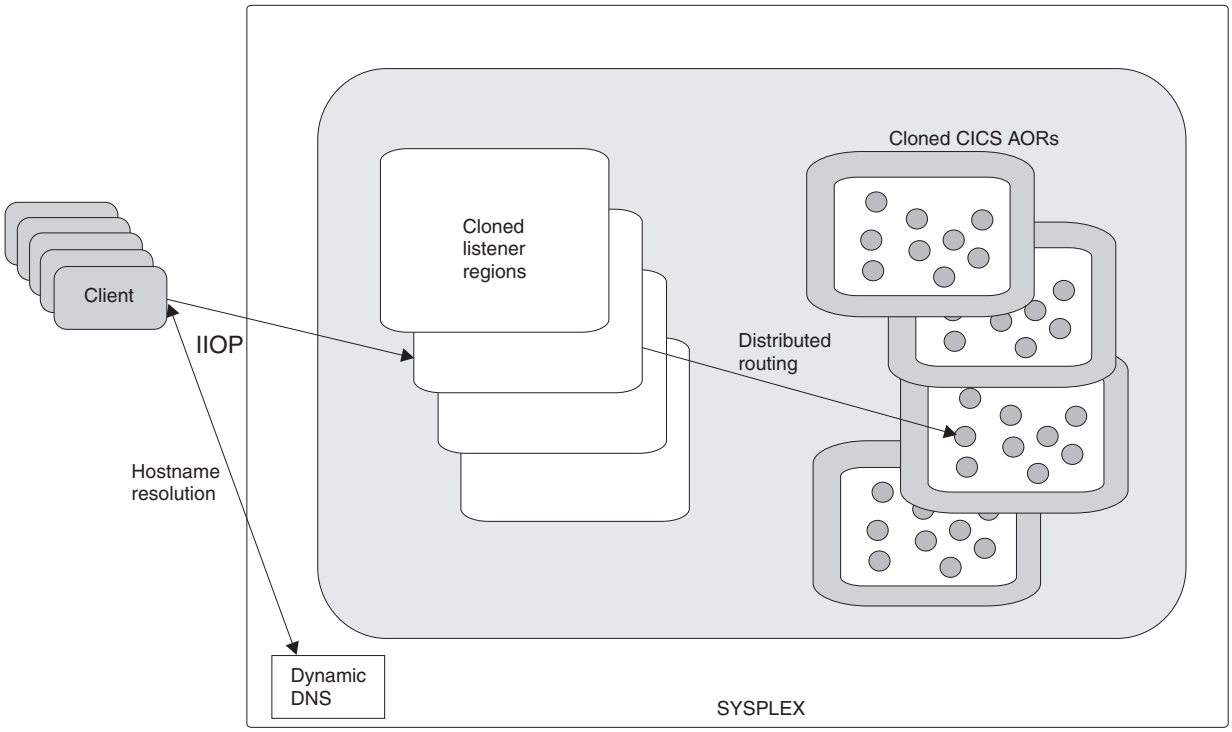

Figure 11. A CICS logical server. In this example, the logical server consists of a set of cloned "listener" regions and a set of cloned AORs. Connection optimization by means of dynamic DNS registration is used to balance client connections across the listener regions. Distributed routing is used to balance OTS transactions across the AORs.

Domain Name System (DNS) connection optimization

Connection optimization is a technique that uses DNS to balance IP connections and workload in a sysplex domain.

In DNS terms, a sysplex is a subdomain that you add to your DNS namespace. Connection optimization extends the concept of a "DNS host name" to clusters, or groups of server applications or hosts. Server applications within the same group are considered to provide equivalent service. Connection optimization uses load-based ordering to determine which addresses to return for a given cluster.

Connection optimization registration

Server applications register with the MVS Workload Manager (WLM), which quantifies the availability of server resources within a sysplex.

The WLM must be configured in goal mode on all hosts within the sysplex. TCP/IP stacks can also register with the WLM to provide information on the

started IP addresses, or static definitions can be used if stacks do not support registration. When registering, server applications provide the following information:

Group name
This is the name of a cluster of equivalent server applications in a sysplex. It is the name within the sysplex domain that client applications use to access the server applications. CICS uses the DNSGROUP parameter of the TCPIPSERVICE resource definition as the group name to register with the WLM.

Server name
This is the name of the server application instance. The server name must be unique among all servers that share the same group name. A server application instance can belong to more than one group. CICS registers with WLM using the specific APPLID of the region as specified by the APPLID system initialization parameter.

Host name
This is the host name of the TCP/IP stack on which the server application runs. During startup, CICS calls the TCP/IP function *gethostbyaddr* to determine the host name of the machine on which it is running, and passes it to the WLM for registration.

Name resolution example

This example shows a CICSplex consisting of four CICS regions, each executing on separate OS/390 machines within a sysplex.

The MVS systems are named MVS1A, MVS1B, MVS1C and MVS1D, with the CICS

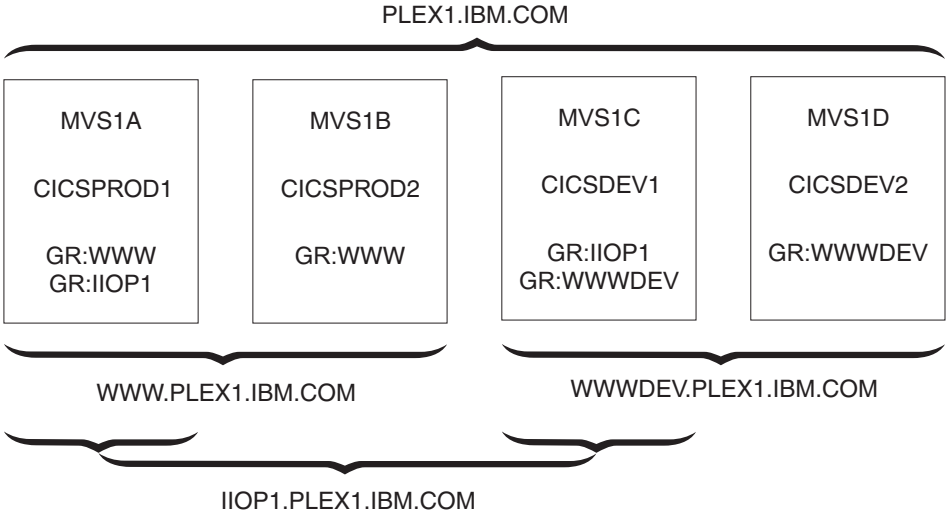

Figure 12. CICSplex using DNS connection optimization

regions having APPLIDs of CICSPROD1, CICSPROD2, CICSDEV1 and CICSDEV2

The sysplex is defined to the DNS to have the name PLEX1 and each MVS machine has a single IP address. The above diagram describes the names that a client machine could use to access the CICS regions based on the following resource definitions installed on each CICS:

- The region CICSPROD1 running on machine MVS1A has twoTCPIPSERVICE definitions, one specifying a group_name of WWW and the second specifying a group_name of IIOP1.

- The region CICSPROD2 running on machine MVS1B has one TCPIPSERVICE definition, specifying a group_name of WWW.
- The region CICSDEV1 running on machine MVS1C has two TCPIPSERVICE definitions, one specifying a group_name of IIOP1 and the second specifying a group_name of WWWDEV.
- The region CICSDEV2 running on machine MVS1D has one TCPIPSERVICE definition, specifying a group_name of WWWDEV.

The names that a client can access are:

- PLEX1.IBM.COM—returns the IP address of any of the machines in the sysplex.
- WWW.PLEX1.IBM.COM—returns either the address of MVS1A or MVS1B.
- IIOP1.PLEX1.IBM.COM—returns either the address of MVS1A or MVS1C.
- WWWDEV.PLEX1.IBM.COM—returns either the address of MVS1C or MVS1D.

You can also address individual CICS regions within a group by using their APPLIDs (or server names). For example, CICSPROD1.WWW.PLEX1.IBM.COM will return the address of MVS1A. This is equivalent to MVS1A.PLEX1.IBM.COM, but the client does not have to know the machine on which the CICSPROD1 server is running, only that CICSPROD1 is part of the WWW group.

Since these names dynamically become available as CICS regions register with the WLM, adding more CICS regions and more MVS machines does not result in any more administration. Using the generic host names (such as WWWDEV.PLEX1.IBM.COM) decouples client applications from specific CICS regions and MVS hosts, which enhances availability and scalability.

Resource definition for DNS connection optimization

These TCPIPSERVICE options must be defined for TCP/IP ports that use DNS connection optimization.

DNSGROUP
specifies the location parameter passed on the IWMSRSRG register call to Workload Manager. The value may be up to 18 characters in length, with trailing blanks ignored.

This parameter is referred to as `group_name` by the OS/390 TCP/IP DNS documentation. It is the generic name of a cluster of equivalent server applications in a sysplex. It is also the name within the sysplex domain that clients use to access the CICS TCPIPSERVICE.

More than one TCPIPSERVICE is allowed to specify the same group name.

The register call is made to WLM when the first service with this group name specified is opened. Subsequent services with the same group name do not cause more register calls to be made.

The deregister action is dictated by the GRPCRITICAL attribute, as described below. It is also possible to explicitly deregister CICS from a group by issuing the master terminal (CEMT) or **EXEC CICS** command **SET TCPIPSERVICE DNSSTATUS DEREGISTERED**, or by using the equivalent CICSPlex SM command.

GRPCRITICAL
marks the service as a critical member of the DNS group such that this service closing or failing causes a deregister call to be made to WLM for this group name.

The default is NO, allowing two or more services in the same group to fail independently and CICS still to remain registered to the group. Only when the last service in a group is closed is the deregister call made to WLM, if it has not already been done so explicitly.

Multiple services with the same group name can have different grpcritical settings. The services specifying GRPCRITICAL(NO) can be closed or fail without causing a deregister. If a service with GRPCRITICAL(YES) is closed or fails, the group is deregistered from WLM.

To implement DNS connection optimization for IIOP requests (including requests for enterprise beans), the following CORBASERVER options must be defined:

- The HOSTNAME option of the CORBASERVER definition must specify a generic host name. This generic hostname is the DNSGROUP value from the TCPIPSERVICE definition, suffixed by the domain or subdomain name managed by the nameserver on MVS. This domain name is established by the TCP/IP administrator. For example, in the previous example, WWW.PLEX1.IBM.COM could be used to route to CICSPROD1 and CICSPROD2.
- The CORBASERVER with the generic hostname (or the DJARS within it) must be published to the nameserver.

The nameserver must be configured to allow it to look up and resolve the generic host name.

Avoiding Domain Name System (DNS) problems

Important

To avoid difficulties in using nameservers, you should be aware of the following:

- Lookups for dynamic names should not be cached. If you use a client that caches nameserver lookup results you cannot be certain that you continue to work with the correct IP address. This might result in the client continuously attempting to call a server region that has been closed, rather than obtaining the address of another server region that has taken over the role previously fulfilled by the other server.
- A problem can arise due to stress on the nameserver being used. Some lookups succeed, others fail with a `NameNotFoundException`.

  When the number of concurrent lookups becomes high, perhaps when a client or bean does repeated lookups without caching, the likelihood of encountering one of these nameserver "blips" increases. Possible measures to consider are:

  – Install a machine of higher capacity to run the name server.
  – Code your applications to recognize this possibility and to retry when this error is encountered.
  – Setup the MVS system so that the most commonly used addresses are included in its `/etc/hosts` file. This bypasses the nameserver lookup for these names and uses the address coded in the file.
  – Rather than specify IP addresses by name, specify them by number. (However, this solution is not advisable in a production environment.)

The IIOP user-replaceable security program

This is an optional identification mechanism.

It is *not* an authentication mechanism, but a way to supply a CICS USERID. To use it, you must specify the name of your security program on the URM option of the TCPIPSERVICE definition for the IIOP port. If you do so, your security program is called by the IIOP request processor.

On invocation, the security program is primed with the value defined by the system initialization parameter DFLTUSER (which defaults to CICSUSER), but can override it. Before routing the IIOP request to a request processor, CICS checks with RACF® that the request receiver transaction is allowed to initiate work on behalf of the USERID generated by the security program.

You can write your own program to supply a USERID, or use the sample security program, DFHXOPUS. See "Using the IIOP user-replaceable security program" on page 203.

CONNECTION authentication

The client USERID is transmitted from the listener region to the AOR only if ATTACHSEC(IDENTIFY) is specified in the CONNECTION definition in the AOR.

See Link security with MRO, in the *CICS RACF Security Guide*, for more information.

IIOP users are recommended to specify SEC=YES and ATTACHSEC(IDENTIFY).

Chapter 16. Configuring CICS for IIOP

This chapter describes what you need to do to configure CICS as a CORBA participant. You need to do this to run all IIOP-based applications, including enterprise beans.

Important

If you are setting up a CICS EJB server (to support enterprise beans) we recommend that you start at Chapter 19, "Setting up an EJB server," on page 239, which contains the specific requirements for enterprise bean support, rather than here.

Configuration of CICS to support IIOP inbound and outbound requests requires setup of the CICS system, and also setup of the host z/OS system environment. server or client, set up the following host software environment:

- A z/OS system, with UNIX Systems Services and its file system.
- Language Environment configured and active.
- CICS.
- .IBM SDK for z/OS, Java Technology Edition. You can download this product, and find out more information about it, at http://www.ibm.com/servers/eserver/zseries/software/java/.

You may also need:

- Java Naming and Directory Interface (JNDI) Version 1.2.
- DB2 with Java Data Base Connectivity (JDBC) Version 1.2 extensions.

Perform the following steps:

- "Setting up the host system for IIOP"
- "Setting up TCP/IP for IIOP" on page 193
- "Setting up CICS for IIOP" on page 195

You might also need to perform one of these steps:

- "Setting up an LDAP server" on page 184
- "Setting up a COS Naming Directory Server" on page 193

If you choose "Setting up an LDAP server" on page 184, you should read "The LDAP namespace structure" on page 189.

Setting up the host system for IIOP

To support IIOP, perform these system tasks:

About this task

Procedure

1. Giving CICS regions access to z/OS UNIX System Services. As part of this task, you will:
   a. Give CICS access to the z/OS UNIX directories and files that are needed to create JVMs

b. Create and give CICS access to the z/OS UNIX working directory that you have specified for input, output, and messages from the JVMs

2. "Setting up JVM profiles" on page 93. During this task, you will:
   a. Enable CICS to locate JVM profiles and any associated JVM properties files.
   b. Choose appropriate JVM profiles for your CORBA stateless objects and enterprise beans.
   c. If necessary, customize the JVM profiles to fit the requirements of your CICS region. (In the course of setting up CICS as a CORBA server, you will need to add some further information.)

   Bear in mind when reading "Setting up JVM profiles" on page 93 that, *for CORBA stateless objects and enterprise beans*:

   - The JVM profile used is that specified on the PROGRAM definition of the **request processor** program.
   - As for all CICS Java programs, if you use a JVM properties file it must be specified on the JVM profile.
   - The default JVM profile, specified on the PROGRAM definition of the default request processor program, is DFHJVMCD.
   - If you plan to use the default JVM profile with your CORBA stateless object and enterprise bean requests, then you need only to locate DFHJVMCD and customize the profile for your CICS region, as described in "Setting up JVM profiles" on page 93.

     If you plan to use customized JVM profiles, you should still make the changes to DFHJVMCD that are required to fit with the setup of your CICS region, because DFHJVMCD is used internally by CICS, as well as being used for the default request processor program.

3. "Defining a shelf directory." The shelf directory is used for deployed JAR files.
4. "Defining name servers." This step is necessary only if you need to define name servers for the purposes described in that procedure.

Defining a shelf directory

Every CORBASERVER definition must specify the name of a shelf directory on z/OS UNIX.

When a DJAR definition is installed, CICS copies the deployed JAR file into a sub-directory of the shelf root directory. (Also, when a PERFORM CORBASERVER PUBLISH command is issued, the IOR of the CorbaServer is written to the sub-directory.)

You can call your shelf directory anything you like. However, it's recommended that you create it somewhere under the `/var` directory. For example, you might create a z/OS UNIX directory called `/var/cicsts/`. Having created the shelf directory, you must give the CICS region userid full access to it—read, write, and execute. See Giving CICS regions access to z/OS UNIX System Services for guidance.

Defining name servers

You might need to define name servers for two purposes:

1. If you are using Domain Name system connection optimization, the listener regions need to be configured to talk to the same name server on z/OS that the MVS Workload Manager is configured to use.

You can define the name server to be used by TCP/IP by providing a SYSTCPD DD statement in the CICS startup JCL for the listener region, as described in Enabling TCP/IP in a CICS region , in the *CICS Transaction Server for z/OS Installation Guide* manual.

2. A client application can locate an IIOP server application using object references that have been registered in a name server. For example, a Java client can use the JNDI interface to obtain a reference to a server application object such as an instance of the home interface of an enterprise bean. Object references can be registered in a name server from CICS by issuing the commands PERFORM CORBASERVER PUBLISH, or PERFORM DJAR PUBLISH.

Enabling JNDI references

To enable your applications to obtain references using a JNDI Interface, set up a name server that supports the Java Naming and Directory Interface (JNDI) V 1.2.

You can use either of the following:

A Lightweight Directory Access Protocol (LDAP) server
If you use an LDAP name server on z/OS, enterprise beans from CICS and WebSphere can interoperate more readily in a shared namespace. See "Setting up an LDAP server" on page 184.

A Corba Object Services (COS) Naming Directory Service.
COS Naming Servers run on an external machine.

Any industry-standard COS Naming Service that supports JNDI Version 1.2 can be used. See "Setting up a COS Naming Directory Server" on page 193.

Specifying the location of the JNDI name server

To enable Java code running under CICS to issue JNDI API calls, and CICS to publish references to the home interfaces of enterprise beans or IORs of stateless CORBA objects, you must define the location of the name server.

About this task

Specify the Web address (URL) and TCP/IP port number of your name server using the **-Dcom.ibm.cics.ejs.nameserver** system property. "JVM system properties" on page 119 has more detailed information.

Important:

1. You must specify the location of your name server on the **-Dcom.ibm.cics.ejs.nameserver** system property in all the JVM profiles or optional properties files that are used by your CORBA stateless objects or enterprise beans.
2. In particular, be sure to specify the location of your name server in the DFHJVMCD JVM profile. The DFHJVMCD profile is used by CICS-defined programs, including the default request processor program and the program that CICS uses to publish and retract deployed JAR files.
3. You also need to specify the location of your name server in any other JVM profiles that you choose to use for CORBA stateless objects or enterprise beans. These might be CICS-supplied sample JVM profiles or your own JVM profiles. For CORBA stateless objects and enterprise beans, the JVM profiles are named in the PROGRAM resource definitions for your request processor programs.
4. For detailed information about defining the location of your name server, see "JVM system properties" on page 119.

Setting up an LDAP server

Either use an existing LDAP server configured for WebSphere, or configure a new one.

If you have an existing LDAP server configured for WebSphere

If the nameserver that you have chosen for use by CICS has already been configured for WebSphere Application Server for z/OS, there is likely to be very little configuration needed to enable CICS to use it.

Correct operation of the EJB support in CICS requires the chosen LDAP namespace to be configured with a WebSphere System Namespace - the publish and retract mechanisms of CICS both attempt to operate within a System Namespace structure. However, once inside an EJB method or if executing a regular Java transaction in CICS, you can communicate with any LDAP namespace regardless of whether it supports a System Namespace.

When you use an LDAP server that is not configured with a WebSphere System Namespace, use an alternative directory service, such as the SUN LDAP service supplied as part of the IBM Developer Kit for the Java Platform 5.0 base, rather than the WebSphere context factory supplied with CICS. See “SUN LDAP Context Factory” on page 296 for details of using the SUN LDAP factory.

An understanding of the WebSphere naming structure that exists on the LDAP server (see “The LDAP namespace structure” on page 189) makes it easier for you or your LDAP administrator to determine suitable values for the six key properties a CICS region needs to know: These are described in “JVM system properties” on page 119. The three security properties are only necessary if the LDAP namespace is setup in a secure manner. On some LDAP servers it may be the case that all users have write access and neither the principal or credentials properties need to be set for the CICS region.

If the structure laid out in the namespace by WebSphere is suitable for your needs, no further configuration is necessary.

The values for nameserver, containerdn and noderootrdn can be obtained by understanding the System Namespace structure and observing the structure in place on your chosen LDAP server, the final part of this section discusses how to determine the property values if you are browsing an existing namespace.

Reasons for further configuration

You might need to proceed with LDAP server configuration, even though the server is already configured for WebSphere Application Server for z/OS, for any of these reasons.

1. The security configuration needs changing to cope with the CICS regions being introduced. See “The LDAP namespace structure” on page 189 and “Security considerations” on page 191 for further information about the LDAP structure and security issues.
2. CICS needs to run in a separate *domain* from WebSphere. If you are building a new, separate, domain, WebSphere Application Server for z/OS and CICS will not easily be able to locate each other's enterprise beans. However, if you just intend to build a new domain the only configuration steps you need to execute are Step 4. “Build the legacyRoot node” and Step 5. “Apply security at CICS region level”.

3. CICS needs to run in an entirely different system namespace structure on the LDAP server. That is, CICS needs to have a `containerdn` that points to somewhere other than the existing namespace root location on the server. In this case, start the configuration procedure at Step 2. “Add a new suffix”. In this case, it is not possible for CICS and WebSphere Application Server for z/OS systems working with the differing container settings to locate each other's Enterprise Beans.

Configuring a new LDAP server

If you do not have an existing LDAP server configured for WebSphere Application Server for z/OS, perform these steps to configure a new LDAP server.

About this task

1. Install the WebSphere naming schema
2. Add a new suffix
3. Build the system namespace root node (containerdn)
4. Build the legacyRoot node below the namespace root node (noderootrdn)
5. Optionally, apply security measures at the CICS region level.

In order to perform many of the steps you are likely to need access to a LDAP principal that has suitable authority on your LDAP server to create new entries at the *root* level.

When these steps are completed, you can determine the values of the system properties that are needed in your JVM properties files to enable CICS to operate with the LDAP server, and add these system properties to all the relevant JVM properties files.

The steps in the following example enable you to configure an LDAP server with the following values for the system properties in your JVM properties files:

```
-Dcom.ibm.cics.ejs.nameserver=ldap://wibble.example.com:389
-Dcom.ibm.ws.naming.ldap.containerdn=ibm-wsnTree=t1,o=WASNaming,c=US
-Dcom.ibm.ws.naming.ldap.noderootrdn=ibm-wsnName=legacyRoot,ibm-wsnName=PLEX2,
      ibm-wsnName=domainRoots
-Djava.naming.security.authentication=simple
-Djava.naming.security.principal=cn=CICSSystems,c=US
-Djava.naming.security.credentials=secret
```

Similar values are given for the example system properties in the CICS-supplied sample JVM properties files.

An example

There are notes throughout the configuration files that are used in this example which guide you to tailor this set of properties to your particular needs.

The one most likely to change is *noderootrdn*, you will probably have some domain other than PLEX2 as the grouping for your nodes - this value is input into the system at Step 4. “Build the legacyRoot node”.

Notice that the example assumes a principal of '`cn=admin`' exists on the LDAP server, with password '`adminpwd`' and that this principal is authorized to perform any operation on the LDAP server.

1. Install the WebSphere naming schema.

If the LDAP server to be configured already has the WebSphere naming schema, this step can be skipped. An LDAP name server configured for WebSphere will already have this schema.

If it is any other LDAP server, install the WebSphere naming schema. The schema is shipped with CICS as `/usr/lpp/cicsts/cicsts41/utils/namespace/WebSphereNamingSchema.ldif` on z/OS UNIX.

Note: The `WebSphereNamingSchema.ldif` file requires that `RFC2256.ldif` and `RFC2713.ldif` be loaded first. This is because the definition of the **ibm-wsnEntry** object class refers to the **javaClassName** attribute type. When using the LDAP server on z/OS, these prerequisite LDAP files are not loaded by default when the LDAP server is set up.

The LDAP server on OS/390 and z/OS needs to store the schema entries in the back-end store to which they apply. This is achieved by adding a suffix to the dn of each schema entry. The supplied `WebSphereNamingSchema.ldif` file does not specify a suffix on the schema entries, so you must add one. For example, if the suffix for the back-end store is "`c=US`", you should change every instance of "`dn:cn=schema`" in the `ldif` file to "`dn:cn=schema,c=US`".

Apply the schema to the nameserver using the **ldapmodify** command :

```
ldapmodify -h <hostname>
           -p <portnumber>
           -D <authorized_principal>
           -w <authorized_principal_password>
           -f WebSphereNamingSchema.ldif
```

Where `hostname` and `portnumber` are those for the LDAP server and the authorized principal is the distinguished name of a user with sufficient authority on the nameserver to write entries.

The **ldapmodify** command must be available for your chosen LDAP server. If it is not, consult your LDAP server documentation to determine how a new schema (in ldif form) should be installed.

A specific example might be:

```
ldapmodify -h wibble.example.com
           -p 389
           -D cn=admin
           -w adminpwd
           -f WebSphereNamingSchema.ldif
```

2. Add a new suffix.

   To build a new hierarchy in the namespace it is necessary to create a new base distinguished name suffix. In this example configuration the suffix is *c=US*, and the new hierarchy is to be *ibm-wsnTree=t1,o=WASNaming,c=US*. The procedure for adding a suffix varies between the different LDAP providers. Your LDAP documentation should indicate how to do this for your chosen provider. As an example, here is the procedure for adding a suffix to a Secureway installation on Windows 32:

   - Start the LDAP Administration interface on a Web browser by typing `http://[hostname]/ldap`, where hostname is the host name of the machine where the LDAP directory is installed. The Administration logon window displays.
   - Type the administrator user ID (for example, in the format cn=root) and password.
   - Make sure that the LDAP server is running.
   - In the left navigation pane, click the Settings folder, and then click Suffixes.

- Type the name of the Base DN to be used as the suffix (in our example, "c=US"), and click Update.
- After the Base DN suffix is added, stop and restart the LDAP server.

The suffix now exists on your LDAP system

On a z/OS system, update the `slapd.conf` file to introduce your new suffix to the system, then restart the nameserver. The extra line to add to `slapd.conf` is:

```
suffix "c=US"
```

3. Build the system namespace root node (containerdn).

   An ldif file to build the root of the system namespace (a node called the containerdn) is supplied with CICS in `utils/namespace/dfhsns.ldif`. This file contains comments describing how to tailor it for your environment. If it is used without alteration, it creates a containerdn of `ibm-wsnTree=t1,o=wasnaming,c=US` and also two CICS users on the LDAP namespace. The first CICS user has a distinguished name of `cn=CICSSystems,c=US` and the second is `cn=CICSUser,c=US`.

   Two user IDs are defined. To understand how they are used, see "Security considerations" on page 191.

   The ldapmodify command must be available for your chosen LDAP server, if it is not, consult your LDAP server documentation to determine how the root of the system namespace should be built.

   This LDIF file can be applied to the LDAP server as follows:

```
ldapmodify-h <hostname>
        -p <portnumber>
        -D <authorized_principal>
        -w <authorized_principal_password>
        -f dfhsns.ldif
```

   Where *hostname* and *portnumber* are those for the LDAP server and the authorized principal is the distinguished name of a user with sufficient authority on the nameserver to write entries.

   A specific example is:

```
ldapmodify-h wibble.example.com
        -p 389
        -D cn=admin
        -w adminpwd
        -f dfhsns.ldif
```

4. Build the legacyRoot node below the namespace root node (noderootrdn).

   The legacyRoot node in the namespace is the point where CICS is usually configured to position itself when called to create a new InitialContext. For this step , the script `DFHBuildSNS` is shipped with CICS in the directory `utils/namespace`.

   The syntax is :

```
DFHBuildSNS   -ldapserver  <server_url>
             [-node        <node within the domain>]
              -domain      <domain_name>
              -containerdn <Root of the namespace>
              -principal   <principal authorized to write to the namespace>
              -credentials <password for that principal>
             [-force]
```

   For example:

```
DFHBuildSNS    -ldapserver ldap://wibble.example.com:389
               -domain PLEX2
               -containerdn ibm-wsnTree=t1,o=WASNaming,c=US
               -principal cn=admin
               -credentials adminpwd
```

(The *-force* option is only used with the *-node* flag, but neither are used in a CICS environment.

5. Optionally apply the additional measures described in "Security at the CICS region level" on page 192.

After running this script, the values of the system properties required in your JVM properties files can be determined and you can add them to all the relevant JVM properties files.

Determining the values for the LDAP system properties

These system properties relate to the use of an LDAP namespace for JNDI.

"JVM system properties" on page 119 has full descriptions of each of these system properties.

- If you have just set up this LDAP namespace, you know the values that you used. Some of these are the ones required for setting the CICS properties.
- If you are using or reusing an existing system namespace, ask your LDAP administrator for suitable values for these properties.
- If you do not have access to the LDAP administrator or the values are unavailable, you might be able to determine them, with the help of the following information, by browsing the namespace.

  You are unlikely to discover that the security principal or credentials by browsing the namespace.

-Dcom.ibm.cics.ejs.nameserver
: Is the URL for the LDAP server being configured. In the example in "Configuring a new LDAP server" on page 185, it is *ldap://wibble.example.com:389*

-Dcom.ibm.ws.naming.ldap.containerdn
: Is the value specified in the `dfhsns.ldif` file. The default is *ibm-wsnTree=t1,o=WASNaming,c=US* if you did not tailor the ldif file. If you are seeking this value by browsing an existing namespace, look for a node of type ibm-wsnTree; the path to this node is a possible value for containerdn.

-Dcom.ibm.ws.naming.ldap.noderootrdn
: Can be determined from the domain that you specified on the DFHBuildSNS call. In the example, the noderootrdn is *ibm-wsnName=legacyRoot,ibm-wsnName=PLEX2,ibm-wsnName=domainRoots*. If you are seeking this value by browsing an existing namespace, look for the path from the chosen containerdn to the legacyRoot entry.

-Djava.naming.security.authentication
: Is set to *simple* if CICS must authenticate itself to LDAP to bind (or write) to it. Using the the defaults in the supplied scripts, authentication is necessary because the `dfhsns.ldif` script removed default write access for the ANYBODY group, and granted write access to the new principal `cn=CICSUser,c=US` that it created. If CICS does not have to authenticate itself to LDAP to write to it, do not set a value for this system property.

Important: If you do specify this system property, you must also specify **-Djava.naming.security.principal** and **-Djava.naming.security.credentials**. Because these system properties hold the UserID and password that CICS requires to access the secure LDAP service, you must give particular attention to the access controls in force at your installation for the files containing these system properties. You must ensure that the files are secure, with update authority restricted to system administrators.

-Djava.naming.security.principal
Is a principal with the authority to bind to the namespace. You might choose the system principal that has write access to the entire namespace if security is not a real concern. However, you are advised to use at least the *cn=CICSUser,c=US* distinguished name specified in `dfhsns.ldif`, because that ID can write to only a particular area of the LDAP namespace (the containerdn and below).

If you want even tighter security, the principal can be *cn=CICSSystems,c=US*. If you use this ID, you must perform an extra LDAP configuration. See "Security considerations" on page 191 for a full discussion of CICS LDAP security configuration.

-Djava.naming.security.credentials
Is the password for the principal. The default if you did not tailor`dfhsns.ldif` is *secret*.

When you have determined the values of these system properties, you specify them in all the JVM profiles or optional JVM properties files that are used by CORBA applications or enterprise beans.

In particular, specify them in the DFHJVMCD JVM profile or referenced properties file. The DFHJVMCD profile is used by CICS-defined programs, including the default request processor program and the program that CICS uses to publish and retract deployed JAR files.

You must also specify these system properties in the JVM profiles or properties files referenced by any other JVM profiles that you choose to use for CORBA stateless objects or enterprise beans. These profiles might be CICS-supplied sample JVM profiles or your own JVM profiles. For CORBA stateless objects and enterprise beans, the JVM profiles are named in the PROGRAM resource definitions for your request processor programs.

The LDAP namespace structure

The LDAP namespace structure used by WebSphere Application Server Version 4 for z/OS and OS/390, is a convenient structure for use in a CICS environment.

Note: WebSphere Application Server Version 5 and later use a COS Naming Server by default and support LDAP only for backwards compatibility with WebSphere Application Server Version 4.

There are two important nodes in the LDAP namespace structure used by WebSphere, the container root, and the legacy root.

The container root

The container root is a node of type ibm-wsnTree. By default, this is called: *ibm-wsnTree=t1, o=wasnaming, c=us* However, this is customisable by changing the *bboldif.cb* file shipped with WebSphere.

The legacy root

The legacy root is a node of type *ibm-wsnName* some way below the container root .

A typical name for this might be: *ibm-wsnName=legacyRoot,ibm-wsnName=PLEX2,ibm-wsnName=domainRoots,ibm-wsnTree=t1,o=WASNaming,c=us* The names legacyRoot and domainRoots are fixed. The only variable is the middle name, in this example PLEX2.

There may be several legacyRoot nodes, each with a different name. Each of these is a "domain". The WebSphere Application Server for z/OS configuration maps a domain to a sysplex. It is configured when the sysplex name is entered into the customization dialog when WebSphere Application Server for z/OS is installed.

Domains

A domain contains a number of servers.

In WebSphere Application Server for z/OS, each server has a node below legacyRoot, for example a server called BBOSV1 would have a name *ibm-wsnName=BBOSV1,ibm-wsnName=PLEX2* relative to the legacy root, and the objects it publishes would be below this node.

When CICS is configured to use the same LDAP server as WebSphere, each CICS CorbaServer has a node directly below legacyRoot. So if a CorbaServer has a JNDI prefix of CICS1, there will be a node *ibm-wsnName=CICS1* relative to the legacy root, and CICS publishes the CorbaServer's objects below this node. When a new InitialContext is created in WebSphere Application Server for z/OS, or in CICS configured as above, the InitialContext will be based on the legacyRoot node. This makes it easy for enterprise beans in CICS to look up objects published by WebSphere, and for enterprise beans or servlets in WebSphere to look up objects published by CICS.

Note: Any JNDI sub-context below a CICS region's initial JNDI context (which is typically the legacyRoot node) may be transient. This is the case if CICS has write access to the initial context node.

A CorbaServer's JNDI sub-context is specified on the JNDIPREFIX option of the CORBASERVER definition. CICS creates the sub-context (if it has the necessary write permission and the sub-context does not already exist in the namespace structure) when an enterprise bean is published from the CorbaServer. However, if all the enterprise beans in the CorbaServer are retracted, CICS may delete the sub-context from the namespace structure. Where multiple CorbaServers share part of a prefix hierarchy, CICS never removes contexts that are still in use by any of them. But if the contexts in the prefix are empty they are removed, as far back as the initial context.

If you want to protect the top-level node of the sub-context hierarchy from deletion, do not give CICS write access to the initial context node. (This means that you must create the top-level node of the sub-context manually.) If you want to

protect several higher levels of the sub-context hierarchy, give CICS write permission only to the lower levels. (This means that you must create the higher-level nodes of the sub-context manually.) For more information, see "Security at the CICS region level" on page 192.

Versions of WebSphere Application Server for distributed platforms have a similar concept of domain, but that concept does not relate to a sysplex.

Nodes

There is another concept, that of a ***node***. A domain represents a number of nodes, and you can navigate your way to a domain by knowledge of the nodename rather than the domain name. Thus a node is a sort of alias for a domain.

Nodes are used in versions of WebSphere Application Server for distributed platforms, but not in WebSphere Application Server for z/OS and OS/390. They are not used by CICS. However, part of the structure for support of nodes is built when you set up a new LDAP server for use by CICS. Since WebSphere Application Server for z/OS and OS/390 does not use nodes, the nodename is an optional parameter to the DFHBuildSNS utility, which under CICS builds the system namespace.

Security considerations

If you specified that CICS must authenticate itself to LDAP in order to write to it, by coding the system property **-Djava.naming.security.authentication=***simple* in your JVM properties files, you now have a choice between

- "Security at the containerdn level" on page 192, or
- "Security at the CICS region level" on page 192.

To help you decide, a very simplified view of part of the LDAP namespace is shown in Figure 13.

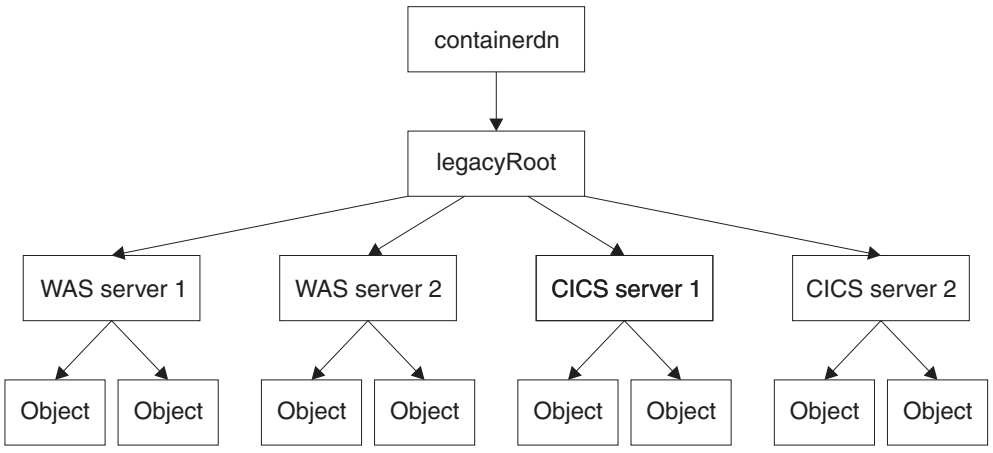

Figure 13. Simplified view of part of an LDAP namespace

If you use security at the containerdn level, CICS has write access to containerdn and all nodes below it. This allows CICS, or a CICS application using the JNDI interfaces, to write to all these nodes, including those that belong to WebSphere Application Server for z/OS and OS/390. If you use security at the CICS region level, then CICS and CICS applications are only able to write to the specific CICS nodes in the tree.

Security at the containerdn level

To use security at the containerdn level, use the CICS administration principal (cn=CICSUser,c=us) created by the *dfhsns.ldif* file (see Step 3. "Build the system namespace root node"). Give this principal access to the containerdn node when you create it. Ensure that this userid and its password appear in the system properties **-Djava.naming.security.principal**and **-Djava.naming.security.credentials** in your JVM properties files.

Security at the CICS region level

Give this principal access to the containerdn node when you create it. Ensure that this userid and its password appear in the system properties **-Djava.naming.security.principal**and **-Djava.naming.security.credentials** in your JVM properties files.

To use security at the CICS region level, use the CICS runtime principal (cn=CICSSystems,c=US) created by the *dfhsns.ldif* file, see Step 3. "Build the system namespace root node". This involves some additional steps. Ensure that this userid and its password appear in the system properties **-Djava.naming.security.principal**and **-Djava.naming.security.credentials** in your JVM properties files. Additionally, as CICS does not have write access to legacyRoot, CICS will be unable to create its own node (called CICS server 1 in Figure 13 on page 191), so you must do it manually, and then give the CICS runtime principal (cn=CICSSystems,c=US) write access to this node. This is described below.

To configure a CICS region in this way and then use the new subcontext:

- Choose a suitable subcontext, we shall call it *cicsabcd*.
- Create that subcontext below the legacyRoot for use by a CICS system (see "Creating a subcontext").
- Ensure the CICS runtime principal can write to it.
- Specify the CICS runtime principal and credentials using the system properties **-Djava.naming.security.principal** and **-Djava.naming.security.credentials** in the JVM properties files that are in use in the region.
- Ensure that any CORBASERVER definitions created in the CICS region have JNDIPREFIX attributes which start with *cicsabcd*. This means that references which they publish, are published *under* the new subcontext *cicsabcd* under legacyRoot.

Security configuration is now complete. A user browsing the LDAP namespace is able to locate this context *cicsabcd* below legacyRoot, and relate it to the CORBASERVER definitions.

Creating a subcontext: To create the subcontext *cicsabcd* below the legacyRoot in the LDAP namespace, and to set suitable Access Control Lists (ACLs) for it, use the LDIF file supplied with CICS in *utils/namespace/dfhNewCICSSubcontext.ldif*.

- The LDIF file contains comments to explain the steps involved, and the values that are likely to need altering for a particular LDAP System Name Space configuration.
- The LDIF file can be applied to the LDAP server using the ldapadd command:

```
Ldapadd -h wibble.example.com
        -p 389
        -D cn=CICSUser,c=us
        -w CICSUserpwd
        -f dfhNewCICSSubcontext.ldif
```

where `CICSUserpwd` is the password for `CICSuser` established when `CICSuser` was set up.

This command needs to be run with a principal (and credentials) that can write to the legacyRoot node. In the example we are using, that is *cn=CICSUser,c=US id*, which has been created for this purpose.

- The most important line of the LDIF file to change is the distinguished name of the node being created, assuming the LDAP System Namespace was configured using all the default scripts supplied with CICS, the distinguished name is:

  ```
  ibm-wsnName=cicsabcd,ibm-wsnName=legacyRoot,ibm-wsnName=PLEX2,
  ibm-wsnName=domainRoots,ibm-wsnTree=t1,o=wasnaming,c=US
  ```

- The rest of the LDIF sets the Access Control Lists appropriately for the new node.
- The comments in this LDIF file are important, they explain other things that you might have to consider. For example, there might be some additional ACL entries that are appropriate in your installation depending on which principals currently have write access to the System Namespace.
- Once the LDIF is applied, the new node exists on the LDAP server below the legacyRoot, and the Access Control Lists are set such that the CICS runtime principal has write access.

Other considerations: You might want to consider the following:

- You could create several different CICS runtime principals for different regions, and so reduce scope of the access granted to each principal.
- If you are using this process within an existing system namespace, there may be other principals (and credentials) in use. They need to be given write access to the new subcontext created by *dfhNewCICSSubcontext*. The comments in the dfhNewCICSSubContext LDIF file discuss ways to check if this is so, and how to tailor the LDIF file appropriately before executing the ldapadd.

Setting up a COS Naming Directory Server

The most convenient way to set up a COS Naming Directory Server is to use IBM WebSphere Application Server running on an external Windows NT or Windows 2000 machine.

About this task

The most convenient way to set up a COS Naming Directory Server is to use IBM WebSphere Application Server running on an external Windows NT or Windows 2000 machine. Follow the installation instructions supplied with it.

Setting up TCP/IP for IIOP

To configure a CICS region as a TCP/IP Listener to accept and send IIOP requests, you need to make these definitions in CICS.

About this task

1. In the CICS startup jobstream for every CICS region where the Listener is required, set the following system initialization parameters:
   - **IIOPLISTENER** to **YES**
   - **TCPIP** to **YES**
2. Define and install TCPIPSERVICE resource definitions in the Listener region for every port that the Listener will monitor, specifying:

- PROTOCOL(IIOP)
- The port or IP address on which CICS will listen for incoming IIOP requests

  Note: If the SSL connection fails, some clients will attempt to retry on an associated non-SSL port. CICS TS defines this port to be SSL port–1. You should ensure that this port (SSL port–1) is not defined for any other purpose. The well-known IIOP ports are 683(non-SSL) and 684(SSL).
- The CICS transaction to start when a request arrives. For an IIOP service, this should be set to the CICS IIOP Request Receiver, CIRR.
- The level of Secure Sockets Layer (SSL) authentication to be used.
- The DNSGROUP name if DNS connection optimization is to be used. See "Resource definition for DNS connection optimization" on page 178
- The name of the user-replaceable program to be called to associate this request with a CICS USERID for security or workload management purposes. If omitted, no user-replaceable program is called. A sample user-replaceable program, DFHXOPUS, is supplied—see "Using the IIOP user-replaceable security program" on page 203.

For example:

```
DEFINE TCPIPSERVICE(IIOPNSSL) GROUP(DFH$IIOP)
     DESCRIPTION(IIOP TCPIPSERVICE with no SSL support)
     URM(DFHXOPUS)          BACKLOG(10)            PORTNUMBER(683)
     TRANSACTION(CIRR)      SSL(NO)
     STATUS(CLOSED)         PROTOCOL(IIOP)
```

Important: In a multi-region server, the TCPIPSERVICE definitions must be installed in *all* the regions (both listeners and AORs) of the logical server. In the listener regions, the IIOPLISTENER system initialization parameter must be set to 'YES'. In the AORs, it must be set to 'NO'. In a combined listener/AOR, it must be set to 'YES'.

See the *CICS Resource Definition Guide* for the full syntax of the TCPIPSERVICE resource definition.

Using DNS connection optimization

To use DNS connection optimization with IIOP, you need to define a DNSGROUP name in the IIOP TCPIPSERVICE resource definition.

All CICS regions providing the same TCPIPSERVICE, with the same DNSGROUP name are registered with MVS Workload Management (WLM) with the same *group-name*, as candidates for client requests requiring the same service. This registration also includes the region's *Host name*, obtained by the TCP/IP function **gethostbyaddr**, and a unique *Server name*, which CICS obtains from the specific APPLID of the region as specified by the APPLID system initialization parameter.

Listener regions need to be configured to talk to the same DNS name server on z/OS that the MVS Workload Manager is configured to use. You can define the name server to be used by TCP/IP by providing a SYSTCPD DD statement in the CICS startup JCL, as described in Enabling TCP/IP in a CICS region, in the *CICS Transaction Server for z/OS Installation Guide*.

Note:

1. Both the client and the CICS server must use the same TCP/IP name server.
2. The name server must be able to perform a reverse look-up, that is, it must be able to translate the IP address of the server into a full hostname.

Setting up CICS for IIOP

To support IIOP you must define a CICS start-up jobstream, and define and install some CICS resources.

About this task

Defining CICS start-up jobstream

A list of parameters you must define in the start-up jobstream for a CICS region that supports IIOP:

JCL parameter

REGION
1000M minimum is recommended

CICS system initialization parameters

EDSALIM
500M minimum is recommended.

IIOPLISTENER

- Specify IIOPLISTENER=YES if the CICS region is an IIOP listener region, or a combined listener and application owning region (AOR).
- Specify IIOPLISTENER=NO if the CICS region is an IIOP application owning region. TCPIPSERVICE definitions installed in the region that specify PROTOCOL(IIOP) cannot be opened.

JVMPROFILEDIR
Set to the z/OS UNIX directory containing the JVM profiles that you are using for your applications. "Setting the location for the JVM profiles" on page 65 tells you how to do this.

KEYRING
Required if you are using Secure Sockets Layer (SSL) authentication with certificates registered to RACF.

MAXJVMTCBS
Specify the number of JVMs that your CICS region can support. Managing your JVM pool for performance, in the *CICS Performance Guide*, tells you how to work out an appropriate setting for the **MAXJVMTCBS** system initialization parameter.

TCPIP Set to YES.

DD statements for CICS datasets
Sample local VSAM data set definitions are provided in the CICS-supplied RDO group DFHEJVS. These data sets must be authorized with RACF for UPDATE access. See Authorizing access to CICS data sets, in the *CICS RACF Security Guide*.

DFHEJDIR
A recoverable shared file containing the request streams directory. This can be a VSAM file or a coupling facility data table. CICS supplies sample JCL to help you create this file, in the DFHDEFDS member of the SDFHINST library.

Note: In most cases, the RECORDSIZE parameter in the supplied JCL should not require modification. However, if you intend to install more

than 40 CorbaServers in your logical EJB/CORBA server, see "Specifying the RECORDSIZE of DFHEJDIR and DFHEJOS."

DFHEJOS
A non-recoverable shared file used by CICS when CorbaServers are installed and to store stateful session beans that have been passivated. This can be a VSAM file or a coupling facility data table. CICS supplies sample JCL to help you create this file, in the DFHDEFDS member of the SDFHINST library.

Note: In most cases, the RECORDSIZE parameter in the supplied JCL should not require modification. However, if you intend to install more than 40 CorbaServers in your logical EJB/CORBA server, see "Specifying the RECORDSIZE of DFHEJDIR and DFHEJOS."

Specifying the RECORDSIZE of DFHEJDIR and DFHEJOS

The maximum number of CorbaServers that can be defined to a CICS EJB/CORBA logical server is controlled by the RECORDSIZE values of the request streams directory file, DFHEJDIR, and the EJB object store file, DFHEJOS.

The RECORDSIZE attributes in the supplied JCL and FILE definitions for DFHEJDIR specify a RECORDSIZE of 1017 bytes. The RECORDSIZE attributes in the supplied JCL and FILE definitions for DFHEJOS specify a RECORDSIZE of 8185 bytes. Normally, these values should not require modification. Only if you intend to install more than 40 CorbaServers in your logical EJB/CORBA server do you need to change these values.

Both DFHEJDIR and DFHEJOS contain a control record which is made up of a 24-byte header and a repeating group of CorbaServer control fields, each 24 bytes long. The default length of 1017 for DFHEJDIR effectively limits the logical server to 41 CorbaServers: (1 + 41) * 24 = 1008 bytes. If you need to install more CorbaServers than this into your logical server, calculate the required RECORDSIZE for DFHEJDIR like this:

1. Multiply the required number of CorbaServers by 24.
2. Add 24 bytes for the control record header. This gives the absolute minimum record size.
3. Round up the last value to the next multiple of 512 to get the minimum control interval size.
4. Subtract 7 to get the value for the RECORDSIZE parameter.

Make the RECORDSIZE value for DFHEJOS greater than that of DFHEJDIR. Too short a length will result in collisions when passivating beans. (The supplied definitions make the RECORDSIZE of DFHEJOS almost 8 times that of DFHEJDIR.)

Note: The sample JCL for DFHEJDIR and DFHEJOS is in the DFHDEFDS member of the SDFHINST library. Sample FILE resource definitions for DFHEJDIR and DFHEJOS are in the DFHEJVS RDO group, with sample coupling facility FILE definitions in the DFHEJCF group, and sample VSAM RLS FILE definitions in the DFHEJVR group.

Defining CICS resources

A list of CICS resources you must be define and install.

You can define CICS resources online using CEDA (see Resource definition online (RDO) transaction CEDA, in the *CICS Resource Definition Guide*); from a CICS application using **EXEC CICS CREATE PROGRAM**, in the *CICS System Programming Reference*); using the DFHCSDUP offline utility (see System definition file utility program (DFHCSDUP), in the *CICS Operations and Utilities Guide*); or by using CICSPlex SM (see the *CICSPlex System Manager Concepts and Planning* manual).

FILE

Provide and install FILE resource definitions for the following files required by CICS:

The "EJB Directory", DFHEJDIR
is a file containing a request streams directory; the directory is used in the routing of method requests for both enterprise beans and CORBA stateless objects. You must define DFHEJDIR as recoverable.

The "EJB Object Store", DFHEJOS
is a file of stateful session beans that have been passivated. (It is also used when CorbaServers are installed.) You must define it as non-recoverable.

In a single-region CICS EJB/CORBA server, it is acceptable to define DFHEJDIR and DFHEJOS as local files. However, in a multiple-region CICS EJB/CORBA server:

- DFHEJDIR must be shared by all the regions (listeners and AORs) in the server.
- DFHEJOS must be shared by all the AORs in the server.

To enable DFHEJDIR and DFHEJOS to be shared across multiple regions, you can define them in one of the following ways:

- As remote files in a file-owning region (FOR)
- As coupling facility data tables
- Using VSAM RLS.

There are sample FILE definitions for DFHEJDIR and DFHEJOS in the CICS-supplied RDO group, DFHEJVS. There are sample coupling facility FILE definitions for DFHEJDIR and DFHEJOS in the CICS-supplied RDO group, DFHEJCF. There are sample VSAM RLS FILE definitions for DFHEJDIR and DFHEJOS in the CICS-supplied RDO group, DFHEJVR. (DFHEJVS, DFHEJCF, and DFHEJVR are not included in the default CICS startup group list, DFHLIST.)

Note: In most cases, the values of the RECORDSIZE attributes in the supplied FILE definitions should not require modification. However, if you intend to install more than 40 CorbaServers in your logical EJB/CORBA server, see "Specifying the RECORDSIZE of DFHEJDIR and DFHEJOS" on page 196.

For reference information about FILE definitions, see the *CICS Resource Definition Guide*.

TRANSACTION and PROGRAM

CORBA stateless objects and enterprise beans don't have PROGRAM resource definitions as such. The PROGRAM resource definition that is relevant to a CORBA stateless object or enterprise bean is that for the request processor program.

Required default TRANSACTION and PROGRAM definitions for the CICS-supplied request receiver and request processor programs are in resource group DFHIIOP, which is included in the default CICS startup group list, DFHLIST.

Normally, you should not need to replace the default TRANSACTION and PROGRAM definitions for the request receiver (CIRR and DFHIIRRS, respectively). This is the definition of CIRR in DFHIIOP:

```
DEFINE TRANSACTION(CIRR)      GROUP(DFHIIOP)
       PROGRAM(DFHIIRRS)      TWASIZE(0)
       PROFILE(DFHCICST)      STATUS(ENABLED)
       TASKDATALOC(ANY)       TASKDATAKEY(USER)
       RUNAWAY(SYSTEM)        SHUTDOWN(ENABLED)
       PRIORITY(1)            TRANCLASS(DFHTCL00)
       DTIMOUT(NO)            TPURGE(NO)
       SPURGE(YES)            ISOLATE(NO)
       RESSEC(NO)             CMDSEC(NO)
       RESTART(NO)
       DESCRIPTION(Default CICS IIOP Request Receiver transaction)
```

One reason for creating your own TRANSACTION and PROGRAM definitions for the request processor program is to specify a JVM profile other than the default. The name of the JVM profile to be used is specified on the JVMPROFILE option of the PROGRAM definition for the request processor program. The default PROGRAM definition for the request processor (DFJIIRP in DFHIIOP) specifies the JVM profile DFHJVMCD. This is the definition of DFJIIRP in DFHIIOP:

```
DEFINE PROGRAM(DFJIIRP)       GROUP(DFHIIOP)
     DESCRIPTION(CICS IIOP Request Processor)
     JVM(YES)
     JVMCLASS(com.ibm.cics.iiop.RequestProcessor)
     JVMPROFILE(DFHJVMCD)
     LANGUAGE(LE370)
     RELOAD(NO)
     EXECKEY(USER)
     RESIDENT(NO)
     USAGE(NORMAL)
     USELPACOPY(NO)
     STATUS(ENABLED)
     CEDF(NO)
     DATALOCATION(ANY)
     DYNAMIC(NO)
```

Note: The CEDF attribute can be set to YES for debugging purposes. See "Using EDF with enterprise beans" on page 301.

If you do create your own PROGRAM definition for the request processor, you can provide one with any name, but the JVMCLASS parameter must be set to **com.ibm.cics.iiop.RequestProcessor**. Choose another JVM profile for the request processor to use, and specify the name of your JVM profile on the JVMPROFILE option. CICS supplies sample JVM profiles in the `/usr/lpp/cicsts/cicsts41/JVMProfiles` z/OS UNIX directory, where `/usr/lpp/cicsts/cicsts41` is the install directory for CICS files on z/OS UNIX. "Setting up JVM profiles" on page 93 tells you how to locate, choose and customize JVM profiles.

TCPIPSERVICE

Provide and install TCPIPSERVICE resource definitions to configure the CICS Listener to receive IIOP requests and call the IIOP *request receiver*. The TCPIPSERVICE resource definition also specifies load-balancing and security options. See "Setting up TCP/IP for IIOP" on page 193.

CICS supplies, in resource group DFH$EJB, a TCPIPSERVICE definition for use with the EJB installation verification program (IVP) and the EJB "Hello World" sample application. If you are setting up a CICS EJB server, we suggest that you follow the step-by-step example of how to configure this definition in "Actions required on CICS" on page 241.

CORBASERVER

Provide and install a CORBASERVER resource definition. Note that the DFHEJDIR file must be defined, installed, and available before a CORBASERVER can be installed.

CICS supplies, in resource group DFH$EJB, a CORBASERVER definition for use with the EJB IVP program and the EJB "Hello World" sample application. If you are setting up a CICS EJB server, we suggest that you follow the step-by-step example of how to configure this definition in "Actions required on CICS" on page 241.

REQUESTMODEL

Provide and install REQUESTMODEL resource definitions to enable the *request receiver* to match the incoming request to a CICS transaction, to define execution parameters that are used if a new request processor instance is created to handle the request. The default TRANSID on REQUESTMODEL definitions is CIRP, which specifies the default request processor program DFJIIRP. If you choose to use your own TRANSACTION definition, you must define and install it; it must specify a PROGRAM definition with the JVMCLASS parameter set to **com.ibm.cics.iiop.RequestProcessor**. See "Obtaining a CICS TRANSID" on page 204.

Note:

1. You need to provide REQUESTMODEL definitions only if the default TRANSID, CIRP, is unsuitable, or if you want to segregate your IIOP workload by transaction ID (for monitoring purposes, for example).
2. The TRANSACTION definition for CIRP specifies DYNAMIC(NO). If you want to use dynamic routing of method requests for enterprise beans and CORBA stateless objects, you must provide one or more TRANSACTION definitions that specify DYNAMIC(YES), and specify them on your REQUESTMODEL definitions.
3. After the CorbaServer is operational, you can use the CREA CICS-supplied transaction to display the transaction IDs associated with particular enterprise beans and bean-methods in the CorbaServer. You can change the transaction IDs, apply the changes, and save the changes to new REQUESTMODEL definitions. This is an easier method than building REQUESTMODEL definitions by hand.
4. In a multi-region CICS logical server, it's recommended that you install your REQUESTMODEL definitions on the AORs as well as the listener regions—see Figure 14 on page 200. The REQUESTMODEL definitions in the AORs are required for outbound requests to local objects. If a CORBA stateless object or enterprise bean makes a call to another object, and that object is available on the local AOR, CICS does not send the request to a listener region. Instead, it either runs the called method in the current task ("tight loopback") or starts another request processor in the local AOR ("normal loopback"). Where normal loopback is used, it's preferable that the new request processor task should use the same REQUESTMODEL as that used for the call to the first object—otherwise, unpredictable results

may occur. If your CORBA stateless objects and enterprise beans make no outbound calls, the REQUESTMODELs on the AOR are not strictly required.

DJAR
Provide and install DJAR resource definitions for any enterprise beans.

Note: DJAR definitions are typically created and installed by the CICS scanning mechanism (see Defining deployed JAR files using the CICS scanning mechanism, in the *CICS Resource Definition Guide*).

Figure 14 shows the RDO definitions required to define a CICS logical server. It shows which definitions are required in the listener regions, which in the AORs, and which in both.

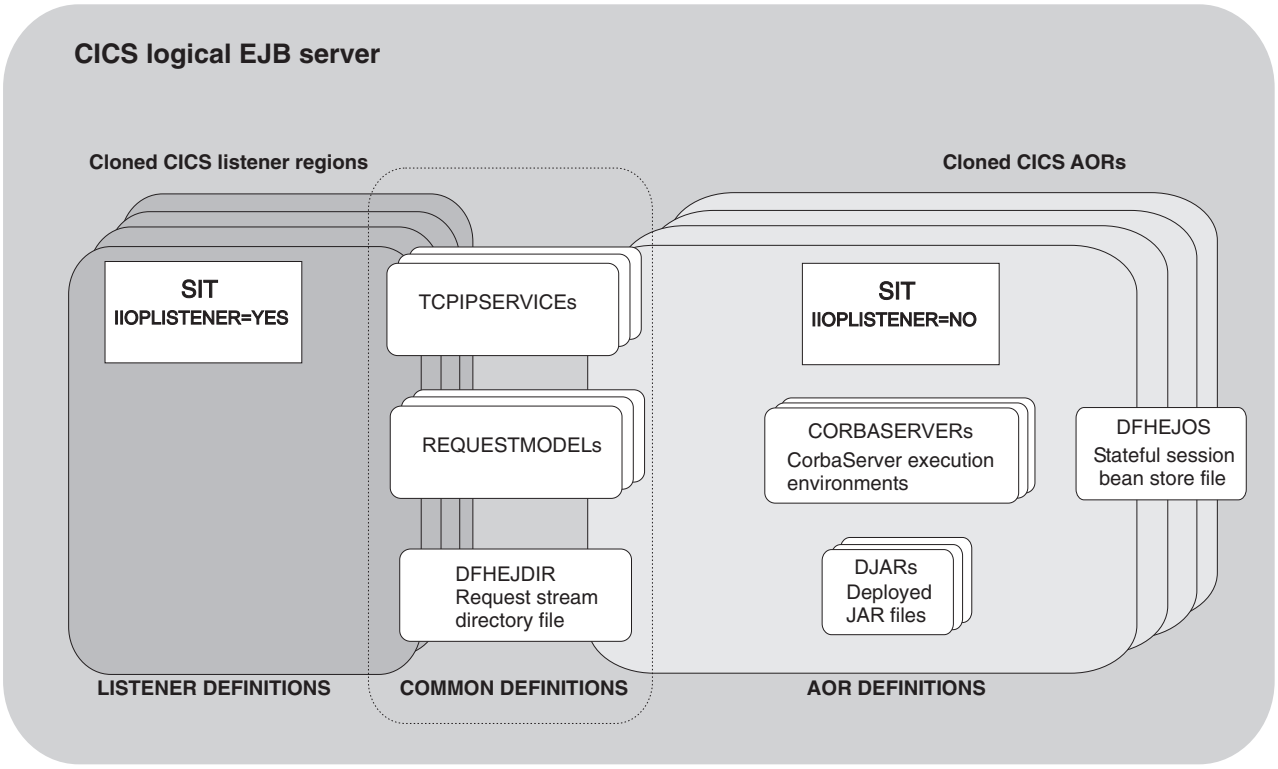

Figure 14. Resource definitions in a CICS logical server. The picture shows which definitions are required in the listener regions, which in the AORs, and which in both.

Chapter 17. Processing IIOP requests

The CICS request receiver derives a CICS USERID and TRANSID that establish CICS execution parameters for the request, before passing control to the IIOP request processor to invoke the target methods.

Obtaining a CICS user ID

For IIOP requests, you can authenticate and identify the user in the following ways.

About this task

1. Using Secure Sockets Layer (SSL) client authentication. See the *CICS RACF Security Guide* for more information.
2. If SSL authentication does not provide a user ID, you can use the IIOP user-replaceable security program to provide one. Specify the name of your IIOP security program on the URM attribute of the TCPIPSERVICE definition for the port. See "Using the IIOP user-replaceable security program" on page 203 for more information.
3. If neither of these mechanisms provides a user ID, the default user ID is used.

If you specify the name of a security program on the TCPIPSERVICE definition, but omit the PROGRAM resource definition for it, CICS tries to build a resource definition for it (autoinstall); if this fails, or your security program does not return a USERID, CICS uses the user ID associated with the SSL client certificate, if there is one. Otherwise, the default user ID is used.

The following communications area is passed to the user-replaceable program. This structure is based on the format of an IIOP message defined in *The Common Object Request Broker: Architecture and Specification* obtainable from the OMG Web site at http://www.omg.org/library

Offset Hex	Type	Len	Name
(0)	STRUCTURE	80	sXOPUS
(0)	CHARACTER	4	standard_header
(4)	FULLWORD	4	pIIOPData
(8)	FULLWORD	4	lIIOPData
(C)	FULLWORD	4	pRequestBody
(10)	FULLWORD	4	lRequestBody
(14)	CHARACTER	4	corbaserver
(18)	FULLWORD	4	pBeanName

Offset Hex	Type	Len	Name
(1C)	FULLWORD	4	lBeanName
(20)	FULLWORD	4	BeanInterfaceType
(24)	FULLWORD	4	pModule
(28)	FULLWORD	4	lModule
(2C)	FULLWORD	4	pInterface
(30)	FULLWORD	4	lInterface
(34)	FULLWORD	4	pOperation
(38)	FULLWORD	4	lOperation
(3C)	CHARACTER	8	userid
(44)	FULLWORD	4	transid
(48)	FULLWORD	4	flag_bytes
(4C)	FULLWORD	4	return_code
(50)	FULLWORD	4	reason_code

standard_header
contains a standard header with the following format:

function
1–byte field set to X'00'

domain
2–character field containing II

* 1–character reserved field

pIIOPData
contains the address of the first megabyte of the unconverted IIOP buffer.

lIIOPData
contains the length of the unconverted IIOP buffer.

pRequestbody
contains the address of the incoming IIOP request.

lRequestbody
contains the length of the incoming IIOP request.

corbaserver
contains the name of the CorbaServer associated with this request.

pBeanName
contains a pointer to the EBCDIC bean name.

lBeanName
contains the length of the bean name.

BeanInterfaceType
contains an enumerated value. X'00' indicates home; X'01' indicates remote.

pModule
contains a pointer to the EBCDIC Module name.

lModule
contains the length of the Module name.

pInterface
contains a pointer to the EBCDIC Interface name.

lInterface
contains the length of the Interface name.

pOperation
contains a pointer to the EBCDIC Operation name.

lOperation
contains the length of the Operation.

userid
contains the input and output user ID. The output user ID must be exactly 8 characters long. If it is shorter than 8 characters it must be padded with blanks.

transid
contains the input TRANSID

Flag_bytes
contains the following indicators::

littleEndian
1–byte field showing byte-order, where **1** indicates TRUE and **0** indicates FALSE

sslClientUserid
1–byte field showing the derivation of the USERID if SSLTYPE CLIENTAUTH is specified in the TCPIPSERVICE definition, where:

0 USERID set from DFLTUSER

1 USERID set from SSL CERTIFICATE

* 2–byte reserved field

return_code
contains the return code.

reason_code
contains the reason code.

RETNCODE is set to RCUSRID (X'01') if a USERID is being returned. The user-replaceable program should return all other fields unchanged, or unpredictable results will occur.

See the *CICS Customization Guide* for information about installing user-replaceable programs.

Using the IIOP user-replaceable security program

You may optionally provide an IIOP security program to examine elements of the incoming IIOP request and generate a USERID.

You must specify the name of your security program on the URM attribute of the TCPIPSERVICE resource definition, and also supply a PROGRAM resource definition for it. If you do not specify a value for URM on the TCPIPSERVICE, no program is called.

The IIOP security program is called only if CICS cannot obtain a user ID using SSL client authentication. See SSL authentication, in the *CICS RACF Security Guide*, for more information.

A sample IIOP security program, DFHXOPUS, is supplied

Your security program may use CICS services, such as a task-related user exit to access DB2, and application parameters encoded within the body of the request.

Using DFHXOPUS

The CICS supplied sample user-replaceable program, DFHXOPUS, accepts the RACF USERID associated with the client certificate, if there is one.

If there is no RACF USERID associated with a certificate:

- For SSL(CLIENTAUTH), DFHXOPUS uses the first eight characters of the COMMONNAME extracted from the client certificate.
- For SSL(YES) or SSL(NO), DFHXOPUS uses the first eight characters of the IIOP Principal, if there is one.

  Note: Versions of the General Inter-ORB Protocol (GIOP) from 1.2 onwards do not support the IIOP Principal field in request headers. So DFHXOPUS will only ever return a user ID derived from the IIOP Principal when the request is in GIOP 1.1, or earlier, format.

If a USERID has not been found using these procedures, DFHXOPUS returns the USERID specified in the CICS system initialization DFLTUSERDFLTUSER system initialization parameter.

The security exit program returns the user ID in the `userid` field of the communications area. If the user ID is less than 8 characters long, the exit program pads the field with blanks. Because a user ID is being returned, the `return_code` field is set to RCUSRID (X'01') .

If you write your own security exit program, it should return all fields other than `userid` and `return_code` unchanged, or unpredictable results may occur.

Obtaining a CICS TRANSID

To associate the incoming GIOP request with a CICS transaction ID, you need to provide and install a REQUESTMODEL resource definition.

You should supply REQUESTMODEL resources for all possible requests that should run under a non-default transaction ID. At run-time, when CICS receives a GIOP request it compares fields in the request with predefined values in the REQUESTMODELs, to find the REQUESTMODEL that most exactly matches the request. The selected REQUESTMODEL provides the TRANSID name that is used to process the request. If no match is found, a default TRANSID (CIRP) is used. REQUESTMODELs can be used with enterprise beans, stateless CORBA objects, or both. They specify:

- CORBA MODULE and INTERFACE patterns to match against requests for stateless CORBA objects
- Bean names for matching enterprise beans.
- OPERATION patterns to match against:
  - Enterprise bean method names
  - CORBA stateless object method names
  - IDL operations (CORBA stateless objects only)

  Note: The OPERATION field is subject to the Java-to-IDL name-mangling rules described in "Name-mangling of the OPERATION field" on page 206.
- The CICS transaction to be started when a matching request is received. The default is CIRP, which specifies the default DFJIIRP program. If you choose to use your own transaction definition, you should base it on CIRP and provide a TRANSACTION resource definition with the PROGRAM parameter set to the name of a CICS program that is defined with the JVMCLASS parameter set to **com.ibm.cics.iiop.RequestProcessor**. The following default resource definitions are provided by CICS in the DFHIIOP group:

```
DEFINE TRANSACTION(CIRP)    GROUP(DFHIIOP)
     PROGRAM(DFJIIRP)       TWASIZE(0)
     PROFILE(DFHCICST)      STATUS(ENABLED)
     TASKDATALOC(ANY)       TASKDATAKEY(USER)
     RUNAWAY(SYSTEM)        SHUTDOWN(ENABLED)
     PRIORITY(1)            TRANCLASS(DFHTCL00)
     DTIMOUT(NO)            TPURGE(NO)
     SPURGE(YES)            ISOLATE(YES)
     RESSEC(YES)            CMDSEC(YES)
     RESTART(NO)
     DESCRIPTION(Default CICS IIOP Request Processor transaction)


DEFINE PROGRAM(DFJIIRP)       GROUP(DFHIIOP)
     DESCRIPTION(CICS IIOP Request Processor)
     JVM(YES)
     JVMCLASS(com.ibm.cics.iiop.RequestProcessor)
     JVMPROFILE(DFHJVMCD)
     LANGUAGE(LE370)        RELOAD(NO)             EXECKEY(USER)
     RESIDENT(NO)           USAGE(NORMAL)          USELPACOPY(NO)
     STATUS(ENABLED)        CEDF(NO)               DATALOCATION(ANY)
     DYNAMIC(NO)
```

  See "Dynamic routing" on page 207 if the request is to be routed to an AOR.
- The name of the CorbaServer that will process the request

See the *CICS Resource Definition Guide* for full details of the REQUESTMODEL resource definition.

Note: To simplify the process of creating REQUESTMODEL definitions for enterprise beans, use the CREA CICS-supplied transaction.

Pattern matching

All requests are compared with installed REQUESTMODEL values for CORBASERVER and TYPE.

A TYPE value of CORBA indicates a request for a stateless CORBA object; a TYPE value of EJB indicates a request for an enterprise bean, and a TYPE value of GENERIC can indicate either type of request. Further matching is then performed, based on the TYPE value:

Stateless CORBA objects

For stateless CORBA objects, (TYPE=CORBA, or GENERIC), the matching process compares the **MODULE** name, **INTERFACE** and **OPERATION** fields contained within the IIOP message, against the patterns defined in each installed REQUESTMODEL, until the closest match is found. INTERFACE, MODULE, and OPERATION can be defined as generic patterns. The rules for pattern matching are summarized as follows:

- Double colons are used as component separators. Each component must be between 1 and 16 characters long
- Generic patterns can consist of zero or more characters followed by *.

If several different generic patterns match a given string, the longest generic pattern results in the most specific match.

Enterprise beans

For enterprise beans, the matching process compares the BEANNAME, OPERATION, and INTFACETYPE fields within the IIOP message, against those defined in each installed REQUESTMODEL.

Name-mangling of the OPERATION field

The OPERATION field of the REQUESTMODEL definition is used to supply the name of the remote method that is to be matched by this request model.

The GIOP request received at run-time includes an operation field which is compared to the OPERATION field on the request model. However, the value of the operation field is not always the same as the method name, as used on the stateless CORBA object or enterprise bean. If RMI-IIOP is being used (as always happens with enterprise beans and may happen with stateless CORBA objects), the method name undergoes a process known as "*mangling*" to change the method name into a canonical form suitable for transmission using IIOP. This mangled method name may not be the same as the original method name. The operation field in the REQUESTMODEL must supply the mangled version of the method name (or a pattern, using wildcard characters, that matches it).

The CICS-supplied CREA transaction can be used to create REQUESTMODEL definitions for enterprise beans that automatically deal with this name-mangling issue.

This mangling and de-mangling knowledge is compiled into the application's stub and tie classes generated using the RMI compiler (RMIC).

For more information about mangling, see "Name mangling for Java" on page 207.

REQUESTMODEL examples

This is an example of a stateless CORBA object REQUESTMODEL:

```
DEFINE REQUESTMODEL(DFJ$IIRH)  GROUP(DFH$IIOP)
       CORBASERVER(IIOP)
       TYPE(Corba)
       MODULE(hello)
       INTERFACE(HelloWorld)
       OPERATION(*)
       TRANSID(IIHE)
       DESCRIPTION(Hello world java server sample)
```

Dynamic routing

If the method invocation is to be routed to another region (AOR), you must define the TRANSID specified in the REQUESTMODEL as dynamically routable in the Listener region (using the DYNAMIC parameter). If you use the supplied default TRANSACTION definition, CIRP, then you will need to change it.

Name mangling for Java

Name mangling is a term that denotes the process of mapping a name that is valid in a particular programming language to a name that is valid in the CORBA Interface Definition Language (IDL). This topic explains why mangling is necessary for Java names, how the names are mangled, and how mangling affects your CICS system.

Why mangling is necessary for Java names

Java client programs use Java Remote Method Invocation (RMI) to invoke methods in a server.

RMI in turn uses one of two communication protocols between client and server:

Java Remote Method Protocol (JRMP)
RMI uses JRMP when both client and server applications are written in Java. CICS does not use JRMP.

Internet Inter-ORB Protocol (IIOP)
RMI uses in an environment when client and server applications may be written in different languages. When IIOP is used as the communications protocol, Java client applications can use the RMI to invoke server programs in another language (C++, for example), as well as to invoke remote Java programs.

IIOP uses Interface Definition Language (IDL) to specify interfaces between objects in a language-independent way. When a Java client makes a remote method call, the Java method name, and its arguments, are converted to the equivalent IDL for transmission to the server using IIOP. It is at this point that mangling may be necessary, because there are many differences in the rules for Java names and IDL names. Some of these differences are:

- Java names are case-sensitive, IDL names are not
- Java supports overloaded methods, IDL does not
- Java names can contain Unicode characters, IDL names cannot
- Some valid Java names may collide with IDL keywords
- Java names can start with a leading underscore, IDL names cannot

In these cases, and others, Java names that are not permitted in IDL, or that are permitted but may be ambiguous, are mangled into an acceptable form.

How Java names are mangled

The rules by which a Java method call is mapped to an IDL name are not simple, and depend upon the circumstances.

Here is one example:

A Java remote interface has methods save, Save and SAVE. These names are distinct in Java, but - because IDL names are not case sensitive - IDL cannot distinguish between them. Therefore, the names are mangled to make them distinct. The mangled names are save_, Save_0 and SAVE_0_1_2_3. However, if the Java remote interface had just one method - save - the name would not be mangled, because there is no possibility of ambiguity.

This example illustrates two important principles:

- It is not possible to determine the mangled name of a given method without knowing what other methods exist.
- Adding or removing a method can affect the mangled names of other methods.

Other cases where mangling is necessary are handled differently. For detailed information about the mapping between Java and IDL, see *Java Language to IDL Mapping*, which is published by the Object Management Group (OMG) (`http://www.omg.org`).

How mangling affects CICS

Although the support for IIOP within CICS contains code that implements the mangling rules, there is very little visible effect on the way you configure and use your CICS system.

There are just two situations in which you need to be aware that mangling takes place. They are:

When defining REQUESTMODELs
REQUESTMODEL resource definitions map inbound IIOP request to CICS transactions. When an inbound request initiated by a Java remote method invocation is received, the OPERATION attribute in the REQUESTMODEL is compared with the mangled name in the inbound request to determine if the REQUESTMODEL matches the request. If it is possible that mangling can take place, do not specify a method name in the OPERATION attribute of the REQUESTMODEL, but specify a generic operation instead.

When creating debugging profiles for Java programs
Debugging profiles specify which program instances are to run under the control of a debugger. When an inbound request initiated by a Java remote method invocation is received, the method field of the debugging profile is compared with the mangled name in the inbound request to determine if the profile matches the request. If it is possible that mangling can take place, do not specify a method name in the debugging profile, but specify a generic method instead.

CAUTION: Although - in theory - its is possible to deduce the mangled names corresponding to each method, it is not a simple task, and is not advisable. To do so, you will need a thorough knowledge of the mangling rules, and of all the method names used in your application. There is also a risk that small changes to an application can change a mangled name.

Handling IIOP diagnostics

If a remote method that is invoked over IIOP fails, the client code will receive a CORBA exception. This includes all enterprise bean exceptions.

CORBA exceptions are defined in the CORBA documentation, which can be obtained from the CORBA Web site: `http://www.omg.org`.

In many instances, the exception includes a CICS specific minor code to aid in problem determination. CICS currently uses the following minor codes:

Table 8. CICS specific CORBA minor codes

Code	**CICS component detecting problem**
1229111296	CICS IIOP request receiver
1229111297	Elsewhere in CICS II domain
1229111298	ORB component of CICS OT domain
1229111299	JTS component of CICS OT domain
1229111300	CSI component of CICS OT domain
1229111301	CSI component of CICS EJ domain

If the client receives a CORBA exception containing any of the CICS minor codes, you should examine the CICS message logs for further information about the error.

Part 5. Using enterprise beans

This section tells you what you need to know to develop and use enterprise beans in CICS.

Chapter 18. What are enterprise beans?

This chapter describes CICS support for the **Enterprise JavaBeans** (**EJB**) architecture.

This chapter is intended as an introduction to CICS support for Enterprise JavaBeans. It does not attempt to describe the Enterprise JavaBeans architecture in depth. If you need a full description of the EJB architecture, see Sun Microsystem's *Enterprise JavaBeans Specification, Version 1.1*, which is available at `http://www.javasoft.com/products/ejb`.

The chapter covers the following topics:

Enterprise beans—the big picture

This section shows you the "big picture"—what CICS support for Enterprise JavaBeans means in general terms. The sections that follow fill in the details.

Sun Microsystem's *Enterprise JavaBeans Specification, Version 1.1*, defines a model for the development of reusable Java server components (known as **enterprise beans**) that can be used in any application server that provides the services and interfaces defined by the specification.

You can configure CICS as an **EJB server**. CICS provides a run-time environment where requests for EJB services are mapped to existing or enhanced CICS services.

You can write enterprise beans that give Java clients access to your past investment in CICS applications and data. For example, you can write enterprise beans that:

- Use the JCICS classes to access CICS resources.

  Note: Enterprise beans that use the JCICS classes are not portable to a non-CICS environment.

- Use JCICS or the CCI Connector for CICS TS to link to existing CICS programs written in procedural languages such as COBOL. (For information about the CCI Connector for CICS TS, see page Chapter 25, "The CCI Connector for CICS TS," on page 319.)

Figure 15 shows, in simplified form, a CICS EJB application server interacting with its environment. It shows enterprise beans that have been developed on a workstation being installed into the EJB server by a process known as **deployment**. Once installed in the server, the enterprise beans are executed in a Java Virtual Machine (JVM) at the request of a client program.

Note: The details of Figure 15 are explained in the sections that follow.

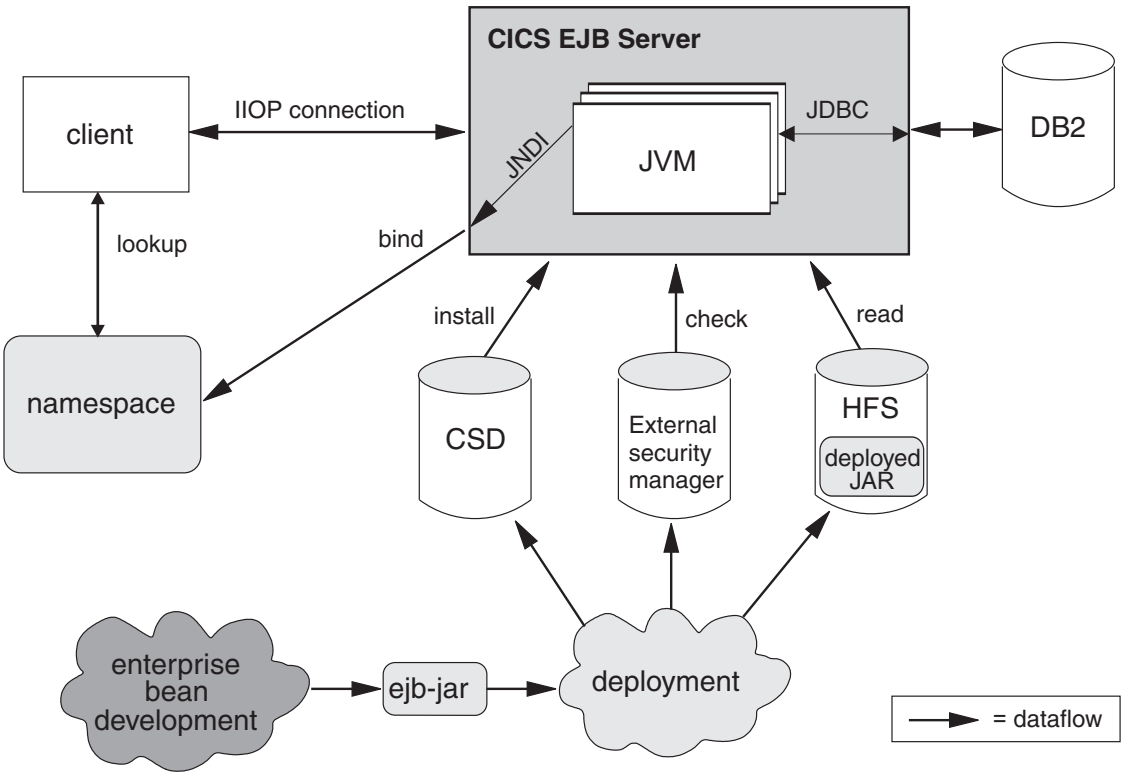

Figure 15. A CICS EJB application server. Enterprise beans developed on a workstation are installed into the EJB server by a process known as deployment. They are executed in a JVM at the request of a client program. The details of this picture are explained in the sections that follow.

JavaBeans and Enterprise JavaBeans

JavaBeans and Enterprise JavaBeans are component architectures for the Java language.

Components

A **component** is a reusable software building block; a pre-built piece of encapsulated application code that can be combined with other components and with handwritten code to produce a custom-built application rapidly.

An application developer can make use of a component without requiring access to its source code. Components can be customized to suit the specific requirements of an application through a set of external property values. For example, a button

component has a property that specifies the caption that should appear on the button. An account management component has a property that specifies the location of the account database.

Components execute within a construct called a **container**, which (among other things) provides an operating system process in which to execute the component.

The **component model** defines the interfaces by which the component interacts with its container and with other components. The developer of a component may code it using a variety of internal methods and properties but, to ensure that it can be used with other components, he or she must implement the interfaces defined in the component model. These interfaces also allow components to be loaded into rapid application development (RAD) tools, such as WebSphere Studio Application Developer.

JavaBeans

A **JavaBean** is a self-contained, reusable software component, written in Java, usually intended for use in a *desktop or client* application.

Typically, desktop JavaBeans have a visual element, and execute within some type of visual container, such as a form, panel, or Web page. Examples might range from a simple button to a fully-featured software CD player.

Bean developers can use a visual tool, such as WebSphere Studio Application Developer, to create JavaBeans. Application developers can use such tools to "wire" JavaBeans together into a larger application, and to set the properties of individual beans.

Enterprise JavaBeans

The **Enterprise JavaBeans architecture** supports *server components*. Server components are application components that run in an application server such as CICS. Unlike desktop components, they do not have a visual element and the container they run in is not visual.

Server components written to the Enterprise JavaBeans specification are known as **enterprise beans**. They are portable across any EJB-compliant application server.

To be useful, server components require access to the application server's infrastructure services, such as its distributed communication service, naming and directory services, transaction management service, data access and persistence services, and resource-sharing services. Different application servers implement these infrastructure services using different technologies. However, an EJB-compliant application server provides an enterprise bean with access to these services through standard interfaces, and manages many of them on behalf of the bean.

Bean developers can use a visual tool, such as WebSphere Studio Application Developer, to create enterprise beans. Application developers can combine method calls to enterprise beans with desktop JavaBeans, Web servlets, and handwritten code to form client/server applications.

The EJB server—overview

An EJB-compliant application server is known as an *EJB server*.

An EJB server could be a transaction processing monitor such as CICS, a Web server, a database, or some other type of server. Note that a CICS EJB server may comprise multiple CICS regions, as described in "Logical servers—enterprise beans in a sysplex" on page 229.

An EJB server provides a standard set of services to support enterprise bean components. These services include:

- Support of the Java Remote Method Invocation (RMI) interface that is used by enterprise beans for communication. RMI has two transport protocol options—JRMP for Java-to-Java interoperation and IIOP for interlanguage interoperation, mediated using a CORBA Object Request Broker (ORB). (For a description of the CICS ORB, see "The Object Request Broker (ORB)" on page 169.)

  CICS Transaction Server for z/OS, Version 4 Release 1 supports RMI over IIOP (RMI-IIOP), but not JRMP. (JRMP is a proprietary protocol that cannot be used to interoperate with non-Java components. CICS does not support distributed transactions over JRMP.)
- A container, called an **EJB container**, which provides management services for enterprise beans.
- A distributed transaction management service that implements the javax.transaction.UserTransaction interface of the Java Transaction API (JTA). The javax.transaction.UserTransaction interface is used by session beans that manage their own transactions.
- Security services.
- Support for the Java Naming and Directory Interface (JNDI). The JNDI API provides directory and naming functionality for Java applications. It enables a client to locate an enterprise bean.
- Support for the Java Data Base Connectivity (JDBC) interface.

The EJB container—overview

Whereas desktop JavaBeans usually run within a visual container such as a form or a Web page, an enterprise bean runs within a container provided by the application server.

The EJB container creates and manages enterprise bean instances at run-time, and provides the services required by each enterprise bean running in it.

The EJB container supports a number of implicit services, including life cycle, state management, security, and transaction management:

Life cycle
Individual enterprise beans do not need to manage process allocation, thread management, object activation, or object passivation explicitly. The EJB container automatically manages the object life cycle on behalf of the enterprise bean.

State management
Individual enterprise beans do not need to save or restore object state between method calls explicitly. The EJB container automatically manages object state on behalf of the enterprise bean.

Security
Individual enterprise beans do not need to authenticate users or check authorization levels explicitly. The EJB container can automatically perform all security checking on behalf of the enterprise bean.

Transaction management
Individual enterprise beans do not need to specify transaction demarcation code to participate in distributed transactions. The EJB container can automatically manage the start, enrollment, commitment, and rollback of transactions on behalf of the enterprise bean.

The execution environment

Before enterprise beans can be deployed into an EJB server, their execution environment must be configured.

In CICS, this is achieved by installing a CORBASERVER resource definition. A CORBASERVER defines an execution environment for enterprise beans and CORBA stateless objects. For convenience, we shall refer to the execution environment defined by a CORBASERVER definition as a **CorbaServer**.

Note that:

- A CICS EJB server may contain more than one CorbaServer.
- Any number of enterprise beans can be deployed into the same CorbaServer.
- A specific enterprise bean can be deployed multiple times into the same CICS EJB server, but not into the same CorbaServer. (In other words, to install a specific enterprise bean multiple times into the same CICS EJB server you must install it into different CorbaServer execution environments. One reason for doing this might be to make the bean available with different deployment properties—see "Enterprise beans—the deployment descriptor" on page 218.) Each deployment results in the creation of a distinct home object (see "Enterprise beans—the home and component interfaces").

Enterprise beans—the home and component interfaces

Client applications do not interact with an enterprise bean directly.

Instead, the client interacts with the enterprise bean through two intermediate objects that are created by the container from classes generated by a deployment tool—one of which classes implements the EJB **home interface** and the other the EJB **component interface**. As the client invokes operations using these intermediate objects, the container intercepts each method call and inserts the management services.

The home and component interfaces are implemented as Java RMI remote objects, which allows the ORB to support them as distributed objects.

The home interface
The home interface is the mechanism by which the client identifies the enterprise bean it wants. It allows a client to create, remove, and (for entity beans, not supported by CICS) find existing instances of, enterprise beans. *Note that the "client" might not be a program running on a network workstation; it might, for example, be a servlet running on a Web server; or an enterprise bean, program, or object on the local EJB server, or on another EJB server.*

When a bean is deployed in an EJB server, the container registers the home interface in a namespace that is accessible remotely. Using the Java Naming and Directory Interface (JNDI) API, any client with access to the namespace can locate the home interface by name. (To be precise, the client locates, by name, an object that implements the home interface. The home interface extends the EJBHome interface.)

The component interface

The component interface allows a client to access the business methods of the enterprise bean. It intercepts all business method calls from the client and inserts whatever transaction, state management, persistence, and security services were specified when the bean was deployed.

When a client creates or finds an instance of an enterprise bean, the container returns a component interface object (one per instance). (To be precise, the container returns a reference to an instance of a class that implements the component interface. The component interface extends the EJBObject interface.)

Enterprise beans—the deployment descriptor

The rules governing an enterprise bean's life cycle, transaction management, security, and persistence are defined in an associated XML document called a **deployment descriptor**.

See "Deploying enterprise beans—overview" on page 225.

Re-usable components may be customizable through a set of external property values, so that they can be modified to suit the requirements of a particular application without changing the source code. An enterprise bean developer can provide (within the deployment descriptor) a set of **environment properties** to allow the application developer to customize the bean. For example, a property might be used to specify the location of a database or to specify a default national language. At run time, an environment object is created which contains the customized property values set during the application assembly process or the bean deployment process.

The EJB server: summary

This topic summarizes the information about EJB servers presented in the previous topics.

The following figure shows enterprise bean objects in a CICS EJB server.

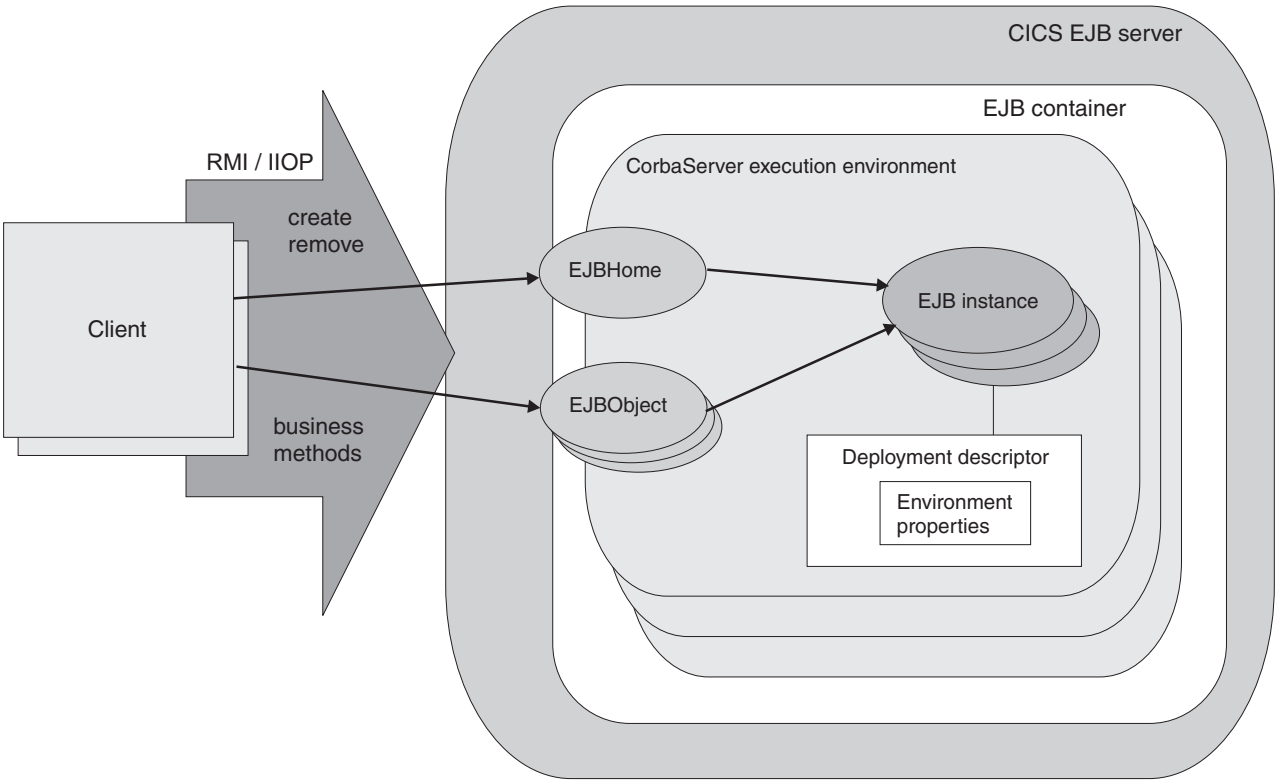

Figure 16. Enterprise bean objects in a CICS EJB server. The EJB container manages and provides services to the enterprise beans contained within it. When a bean is deployed, the deployment tool generates the EJB home and component interface classes.

The home interface is accessible through JNDI and implements lifecycle services for the bean. The client uses it to create, remove, and (for entity beans, not directly supported by CICS) find instances of enterprise beans.

The container creates an EJB component interface object for each instance of the bean. The component interface provides access to the business methods within the bean. It intercepts all business method calls from the client and implements transaction, state management, persistence, and security services for the bean, based on the settings of the bean's deployment descriptor.

Types of enterprise bean

This section discusses two types of enterprise bean—**session beans** and **entity beans**.

Session beans

A session bean:

- Is created by a client and represents a single conversation, or session, with that client.
- Typically, persists only for the life of the conversation with the client. In this sense, it can be likened to a pseudoconversational transaction.

  If the bean developer chooses to save information beyond the life of a session, he or she must implement persistence operations—for example, JDBC or SQL calls—directly in the bean class methods.
- Typically, performs operations on business data on behalf of the client, such as accessing a database or performing calculations.

- May or may not be transactional. If it's transactional, it can manage its own Object Transaction Service (OTS) transactions, or use container-managed OTS transactions. For an explanation of the relationship between OTS transactions and CICS units of work, see "Enterprise beans—managing transactions" on page 221.
- Is not recoverable—if the EJB server crashes, it may be destroyed.
- Has two flavours: **stateful** and **stateless**.

Stateful session beans

A stateful session bean has a *client-specific* conversational state, which it maintains across methods and transactions; for example, a "shopping cart" object would maintain a list of the items selected for purchase by the user.

A stateful session bean that manages its own transactions can begin an OTS transaction in one method and commit or roll it back in a subsequent method.

Stateless session beans

A stateless session bean has no client-specific (nor any other kind of) non-transient state; for example, a "stock quotation" object might return current share prices.

A stateless session bean that manages its own transactions and begins a transaction must commit (or roll back) the transaction in the same method in which it started it.

Entity beans

CICS does not support entity beans directly. That is, entity beans cannot run in a CICS EJB server. However, a session bean or program running in a CICS EJB server can be a client of an entity bean running in a non-CICS EJB server.

Important

An entity bean:

- Is typically an object representation of business data, such as a customer order. Typically, the data:
  - Are maintained in a permanent data store, such as a database.
  - Need to persist beyond the life of a client instance. Therefore, an entity bean is relatively long-lived, compared to a session bean.
- Object can be accessed by more than one client at the same time. This is possible because each instance of an entity bean is identified by a **primary key**, which can be used to find it via the home interface.
- Can manage its own persistence (**bean-managed persistence**), or delegate the task to its container (**container-managed persistence**).

  If the bean manages its own persistence, the bean developer must implement persistence operations—for example, JDBC or SQL calls—directly in the bean.

  If the entity bean delegates persistence to the container, the latter manages the persistent state transparently; the bean developer doesn't need to code any persistence operations within the bean.
- May or may not be transactional. If it's transactional, all transaction functions are performed implicitly by the EJB container and server. There are no transaction demarcation statements within the bean code. Unlike session beans, an entity bean is not permitted to manage its own OTS transactions. See "Enterprise beans—managing transactions" on page 221.
- Is recoverable—it survives a server crash.

Session beans and entity beans compared

This is a summary of the differences between entity and session beans.

Table 9. Comparison of session and entity beans

Session bean	Entity bean
Represents a single conversation with a client. Typically, encapsulates an action or actions to be taken on business data.	Typically, encapsulates persistent business data—for example, a row in a database.
Is relatively short-lived.	Is relatively long-lived.
Is created and used by a single client.	May be shared by multiple clients.
Has no primary key.	Has a primary key, which enables an instance to be found and shared by more than one client.
Typically, persists only for the life of the conversation with the client. (However, may choose to save information.)	Persists beyond the life of a client instance. Persistence can be container-managed or bean-managed.
Is not recoverable—if the EJB server fails, it may be destroyed.	Is recoverable—it survives failures of the EJB server.
May be stateful (that is, have a client-specific state) or stateless (have no non-transient state).	Is typically stateful.
May or may not be transactional. If transactional, can manage its own OTS transactions, or use container-managed transactions. A stateful session bean that manages its own transactions can begin an OTS transaction in one method and commit or roll it back in a subsequent method. A stateless session bean that manages its own transactions and begins an OTS transaction must commit (or roll back) the transaction in the same method in which it was started. The state of a transactional, stateful session bean is not automatically rolled back on transaction rollback. In some cases, the bean can use session synchronization to react to syncpoint.	May or may not be transactional. Must use the container-managed transaction model. If transactional, its state is automatically rolled back on transaction rollback.
Is not re-entrant.	May be re-entrant.

Enterprise beans—managing transactions

Clients can begin, commit, and roll back ACID transactions using an implementation of the Java Transaction Service (JTS) or the CORBA Object Transaction Service (OTS).

These ACID transactions [1]are analogous to CICS distributed units of work. We use the term **OTS transaction** to differentiate these transactions from CICS transaction definitions (the ones with 4-character transaction identifiers) and CICS transaction instances (which are sometimes loosely called “tasks”).

When a client calls an enterprise bean in the scope of an OTS transaction, information about the transaction flows to the EJB server in an IIOP “service context”, which is like an extra (hidden) parameter on the method request. The EJB server uses this information if it needs to participate in the transaction. Whether the method of an enterprise bean needs to run under a client's OTS transaction (if there is one) is determined by the setting of the **transaction attribute** specified in the bean's deployment descriptor. The method may run under the client's OTS transaction, under a separate OTS transaction which is created for the duration of the method, or under no OTS transaction.

Entity beans must use **container–managed OTS transactions**. All transaction functions are performed implicitly by the EJB container and server. There are no transaction demarcation statements within the bean code.

Session beans can use either container-managed OTS transactions or **bean–managed OTS transactions**. A session bean that uses bean–managed transactions uses methods of the javax.transaction.UserTransaction interface to demarcate transactions. A stateful session bean that manages its own transactions can begin an OTS transaction in one method and commit or roll it back in a subsequent method. A stateless session bean that manages its own transactions and begins an OTS transaction must commit (or roll back) the transaction in the same method.

At runtime, the EJB container implements transaction services according to the setting of the transaction attribute specified in the bean's deployment descriptor. The possible settings of the transaction attribute are:

Mandatory
Indicates that the bean must always execute within the context of the caller's OTS transaction. If the caller does not have a transaction when it calls the bean, the container throws a javax.transaction.TransactionRequiredException exception and the request fails.

Never
Indicates that the bean must not be invoked within the context of an OTS transaction. If a caller has an OTS transaction when it calls the bean, the container throws a java.rmi.RemoteException exception and the request fails.

NotSupported
Indicates that the bean cannot execute within the context of an OTS transaction. If a caller has an OTS transaction when it calls the bean, the container suspends the transaction for the duration of the method call. It resumes the suspended transaction when the method has completed. The suspended transaction context of the client is not passed to resource managers or enterprise bean objects that are invoked from the method.

Required
Indicates that the bean must execute within the context of an OTS transaction. If a caller has an OTS transaction when it calls the bean, the method

1. Transactions possessing atomicity, consistency, isolation, and durability. Jim Gray and Andreas Reuter, *Transaction Processing: Concepts and Techniques*, 1993.

participates in the caller's transaction. If the caller does not have an OTS transaction, the container starts a new OTS transaction for the method.

RequiresNew
Indicates that the bean must execute within the context of a new OTS transaction. The container always starts a new OTS transaction for the method. If the caller has an OTS transaction when it calls the bean, the container suspends the caller's transaction for the duration of the method call. The suspended transaction context of the client is not passed to resource managers or enterprise bean objects that are invoked from the method.

Supports
Indicates that the bean can run with or without a transaction context. If a caller has an OTS transaction when it calls the bean, the method participates in the caller's transaction. If the caller does not have an OTS transaction, the method runs without one.

Note: Enterprise bean methods always execute in a CICS task, under a CICS unit of work. Even if an enterprise bean method executes under no OTS transaction, any updates that the method makes to recoverable resources are committed only at normal termination of the CICS task, and backed out if there is a need to roll back.

The setting of a method's transaction attribute determines whether or not the CICS task under which the method executes makes its unit of work part of a wider, distributed OTS transaction.

A single CICS task cannot contain more than one enterprise bean, because CICS treats an execution of an enterprise bean method as the start of a new task. You can create an application that includes more than one enterprise bean, but the application will not operate as a single CICS task.

Enterprise beans—security overview

EJB security is concerned with authentication, access control, and the Java 2 security policy mechanism.

Authentication

Authentication of EJB clients uses the TCP/IP secure sockets layer (SSL) protocol.

See Support for security protocols, in the *CICS RACF Security Guide*, for information about configuring CICS to use SSL.

Access control

Access to enterprise bean methods is based on the concept of security roles. You can use CICS transaction security and resource security with EJB resources.

Security roles

Access to enterprise bean methods is based on the concept of **security roles**. A security role represents a type of user of an application in terms of the permissions that the user must have to successfully use the application.

The roles that are permitted to execute a particular enterprise bean or particular methods of a bean are specified in the bean's deployment descriptor, and the mapping of security roles to individual users is done in the external security manager.

For more information about security roles, see “Security roles” on page 353.

CICS transaction and resource security

You can use CICS transaction security and resource security with EJB resources.

CICS transaction security applies to the CICS transactions associated with enterprise bean methods—that is, the transactions named on EJB REQUESTMODEL definitions.

CICS resource security applies to the CICS resources accessed by enterprise beans (by means of, for example, JCICS).

The Java security manager

The security of the enterprise beans container environment is protected by the Java security policy mechanism and is independent of CICS security. The security policy mechanism is one of the components that make up the Java security model.

The security policy mechanism is used to enforce the restrictions in the EJB specification concerning Java functions that may not be issued by enterprise beans. CICS provides a policy file that enforces this behavior.

To use JDBC or SQLJ from enterprise beans with a Java security policy mechanism active, you must use the JDBC 2.0 driver provided by DB2. The JDBC 1.2 driver provided by DB2 does not support Java security, and will fail with a security exception unless you disable the mechanism.

Enterprise beans—user tasks

The roles involved in the development and deployment of applications that use enterprise beans are, a bean provider, an application assembler, a deployer, and a system administrator.

Note: In smaller organizations, one person may be responsible for more than one of these roles.

The bean provider

The bean provider develops reusable enterprise beans that typically implement business tasks or business entities.

The bean provider's output is an **ejb-jar file** that contains one or more enterprise beans. The bean provider is responsible for:

- The Java classes that implement an enterprise bean's business methods.
- The definition of the bean's component and home interfaces.
- The bean's deployment descriptor.

  The deployment descriptor includes the structural information—for example, the name of the enterprise bean class—of the enterprise bean and declares all the bean's external dependencies—for example, the names and types of the resource managers that the enterprise bean uses.

The application assembler

The application assembler creates applications that use enterprise beans. He combines enterprise beans and hand-written client code into a client/server application. Although he must be familiar with the functionality provided by the

enterprise beans' component and home interfaces, he does not need to have any knowledge of the enterprise beans' implementation.

The input to the application assembler is one or more ejb-jar files produced by the bean provider. His output is one or more ejb-jar files that contain the enterprise beans, along with their application assembly instructions and customized environment settings. He has inserted the application assembly instructions, security roles, and environment values into the deployment descriptors.

The application assembler may also combine enterprise beans with other types of application components—for example, JavaBeans—when assembling an application.

Typically, the application assembly step occurs before the deployment of the enterprise beans. However, sometimes assembly may be performed after the deployment of all or some of the enterprise beans.

The deployer

The deployer takes one or more ejb-jar files produced by the application assembler and deploys the enterprise beans contained in the ejb-jar files into a specific CorbaServer in an EJB server.

The deployer must:

- Resolve all the external dependencies declared by the bean provider. For example, he must ensure that all resource manager connection factories used by the enterprise beans are present in the operational environment, and bind them to the resource manager connection factory references declared in the deployment descriptor.
- Follow the application assembly instructions defined by the application assembler. For example, the deployer is responsible for mapping the security roles defined by the application assembler to CICS user groups and external security manager profiles.

The deployment process is semi-automated. To perform his role, the deployer uses a **deployment tool**. Deployment tools are provided by CICS.

The deployer's output are enterprise beans that have been customized for the target operational environment, and deployed in one or more CorbaServers.

The system administrator

The system administrator is responsible for configuring and administering the CICS regions that comprise the logical EJB server, together with their network connections. He or she is also responsible for overseeing the well-being of the deployed EJB applications at runtime.

Deploying enterprise beans—overview

A desktop Java bean is developed, installed, and run on a workstation. An enterprise bean, however, which will run on a server, requires an additional stage, **deployment**, to prepare the bean for the runtime environment and install it into the EJB server.

Enterprise beans are produced by the bean provider and customized by the application assembler. The application assembler may use a tool such as the

Assembly Toolkit (ATK) (described in The enterprise bean deployment tool, ATK, in the *CICS Operations and Utilities Guide*) to customize the ejb-jar file. The customized ejb-jar file passed to the deployer contains:

- The java classes for one or more enterprise beans.
- A single deployment descriptor, written in XML, which describes the characteristics of each of the enterprise beans, such as:
  - Transaction attributes
  - Environment properties
  - Security levels
  - Application assembly information.

Also required is CICS-specific information, such as resource definition requirements, in either resource definition online (RDO) format (for DFHCSDUP) or CICSPlex SM Business Application Services (BAS) format (for BATCHREP).

Here's an outline of the deployment process:[2]

1. A **deployment tool** (such as the Assembly Toolkit (ATK), described in The enterprise bean deployment tool, ATK, in the *CICS Operations and Utilities Guide*) is used to transform the ejb-jar file into a **deployable JAR file**, suitable for deployment. The transformed file contains the XML deployment descriptor and enterprise bean classes from the ejb-jar file, plus additional classes generated in support of the EJB container. The transformed file is stored as a **deployed JAR file** on the z/OS UNIX file system.

   It is recommended that you store the deployed JAR file in the CorbaServer's **deployed JAR file directory** (specified by the DJARDIR option of the CORBASERVER definition). The deployed JAR file directory is also known as the **"pickup" directory**. When CICS scans the pickup directory, it automatically creates and installs a definition of each new or updated deployed JAR file that it finds there. CICS scans the pickup directory:
   - Automatically, when the CORBASERVER definition is installed, *or*
   - When instructed to by means of an explicit **EXEC CICS** or **CEMT PERFORM CORBASERVER SCAN** command, *or*
   - When instructed to by the resource manager for enterprise beans (otherwise known as the RM for enterprise beans), which issues a **PERFORM CORBASERVER SCAN** command on your behalf. (The resource manager for enterprise beans is described in The Resource Manager for Enterprise Beans, in the *CICS Operations and Utilities Guide*.)
2. CICS resource definitions are required for:
   - The CorbaServer execution environment (CORBASERVER). (The same CORBASERVER definition will be installed on each CICS AOR in the logical EJB server.)
   - TCP/IP services (for IIOP). One or more TCPIPSERVICE definitions will be installed on each CICS region in the logical EJB server.
   - Request models, to associate client IIOP requests with CICS TRANSIDs (and thus to associate bean methods with sets of execution characteristics, covering such things as security, priority, and monitoring). Request models are only required if the default TRANSID, CIRP, is unsuitable. (You may want to segregate your IIOP workload by transaction ID, for example.)

2. This simplified description of the deployment process assumes that you're using RDO rather than BAS.

Note: You can use the CREA CICS-supplied transaction to display the transaction IDs associated with particular beans and bean-methods in the CorbaServer. You can change the transaction IDs, apply the changes, and save the changes to new REQUESTMODEL definitions.

- Deployed JAR files (DJARs), each of which includes the z/OS UNIX filename of a deployed JAR file. If you store your deployed JAR files in the CorbaServer's "pickup" directory, DJAR definitions are created and installed automatically when the CorbaServer is installed (or when a subsequent scan takes place).

Note: "Setting up a logical EJB server" on page 231 contains more information about these RDO definitions.

3. Security definitions are added to the external security manager. These specify which roles can execute particular beans and methods, and which user IDs are associated with each role.
4. The resource definitions are installed in CICS. Installing a DJAR definition causes CICS to:
   - Copy the deployed JAR file (and the classes it contains) to a "shelf" directory on z/OS UNIX. The **shelf directory** is where CICS keeps copies of installed deployed JAR files.
   - Read the deployed JAR from the shelf, parse its XML deployment descriptor, and store the information it contains.

   Note: If you store your deployed JAR files in the CorbaServer's "pickup" directory, DJAR definitions are installed automatically when the CorbaServer is installed (or when a subsequent scan takes place).
5. A reference to the home interface class of each deployed bean is published in an external namespace. The namespace is accessible to clients through JNDI.

   If you specify AUTOPUBLISH(YES) on the CORBASERVER definition, the contents of a deployed JAR file are automatically published to the namespace when the DJAR definition is successfully installed into the CorbaServer. Alternatively, you can issue a **PERFORM CORBASERVER PUBLISH** or **PERFORM DJAR PUBLISH** command.

Figure 17 on page 228 shows the deployment process.

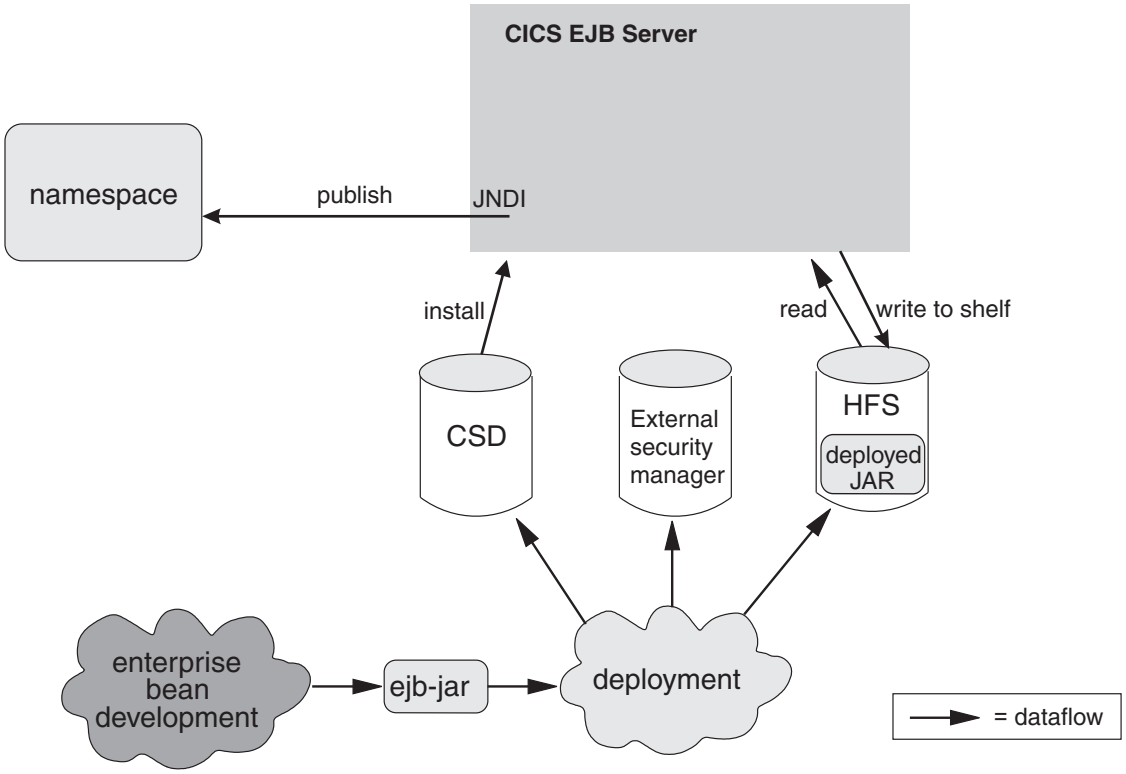

Figure 17. Deploying enterprise beans into a CICS EJB server. A deployment tool is used to perform code generation on the ejb-jar file containing the bean classes. The transformed file is stored as a deployed JAR file on z/OS UNIX. An RDO definition of the deployed JAR file is created and installed in CICS, together with other definitions for TCP/IP services, request models, and the CorbaServer execution environment. Security definitions are created on the external security manager.

Configuring CICS as an EJB server—overview

A CICS EJB server contains these basic components.

The listener
The job of the listener is to listen for (and respond to) incoming TCP/IP connection requests. An IIOP listener is configured by a **TCPIPSERVICE** resource definition to listen on a specific TCP/IP port and to attach an IIOP **request receiver** to handle each connection.

Once an IIOP connection has been established between a client program and a particular request receiver, all subsequent requests from the client program over that connection flow to the same request receiver.

The request receiver
The request receiver analyzes the structured IIOP data. It passes the incoming request to a **request processor** by means of a **request stream**, which is an internal CICS routing mechanism. The object key in the request determines whether the request must be sent to a new or an existing request processor.

If the request must be sent to a new request processor, a CICS TRANSID is determined by comparing the request data with templates defined in **REQUESTMODEL** resource definitions. (If no matching REQUESTMODEL definition can be found, the default TRANSID, CIRP, is used.) The TRANSID defines execution parameters that are used by the request processor.

The request processor

The request processor is a transaction instance that manages the execution of the IIOP request. It:

- Locates the object identified by the request
- For an enterprise bean request, calls the container to process the bean method
- For a request for a stateless CORBA object, the ORB typically processes the request itself (although the transaction service may also be involved).

For comprehensive information about listeners, request receivers, and request processors, see Chapter 15, "The IIOP request flow," on page 173.

Figure 18 shows a CICS logical EJB server. In this example, the listener regions and AORs are in separate groups, connection optimization is used to balance client connections across the listener regions, and distributed routing is used to balance OTS transactions across the AORs.

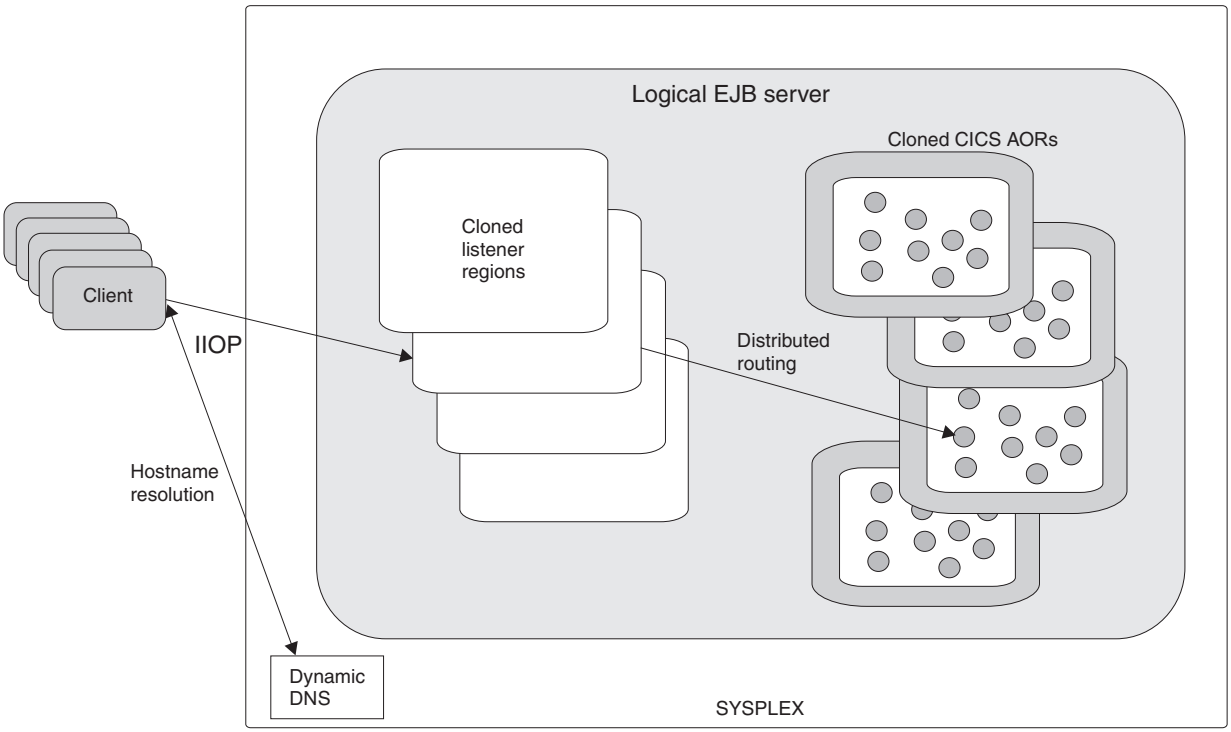

Figure 18. A CICS logical EJB server. The logical server consists of a set of cloned listener regions and a set of cloned AORs. In this example, connection optimization by means of dynamic DNS registration is used to balance client connections across the listener regions. Distributed routing is used to balance OTS transactions across the AORs.

Logical servers—enterprise beans in a sysplex

You can implement a CICS EJB server in a single CICS region.

However, in a sysplex it's likely that you'll want to create a server consisting of multiple regions. Using multiple regions makes failure of a single region less critical and enables you to use workload balancing. A **CICS logical EJB server** consists of one or more CICS regions configured to behave like a single EJB server.

Typically, a CICS logical EJB server consists of:

- A set of cloned **listener regions** defined by identical TCPIPSERVICE definitions to listen for incoming IIOP requests.
- A set of cloned application-owning regions (AORs), each of which supports an identical set of enterprise bean classes in an identically-defined CorbaServer.

Note: The listener regions and AORs may be separate or combined into listener/AORs.

Workload balancing in a sysplex

Workload balancing is implemented at two levels:

1. To balance client connections across the listener regions, you can use any of the following methods:
   - Connection optimization by means of dynamic Domain Name System (DNS) registration.
   - IP routing.
   - A combination of connection optimization and IP routing.

   With connection optimization by means of dynamic DNS registration, for example, multiple CICS regions are started to listen for IIOP requests on the same port (using virtual IP addresses). Each client IIOP connection request contains a generic host name and port number. The generic host name in each connection request is resolved to a real IP address by MVS DNS and Workload Management (WLM) services.
2. To balance OTS transactions across the AORs, you can use either of the following:
   - CICSPlex SM
   - A customized version of the CICS distributed routing program, DFHDSRP.

   Important:

   a. It is convenient to talk of balancing (or dynamically routing) OTS transactions across AORs. Strictly speaking, however, what are dynamically routed are *method requests* for enterprise beans and CORBA stateless objects. There is a correlation between routing method requests dynamically and routing OTS transactions dynamically: CICS invokes the routing program for requests for methods that will run under a *new* OTS transaction, but not for requests for methods that will run under an *existing* OTS transaction—these it directs automatically to the AOR in which the existing OTS transaction runs. However, because requests for methods that will run under *no OTS transaction* can also be dynamically routed, the correlation is not exact.

   b. We must be clear about what we mean by "new" and "existing" OTS transactions. For the purposes of this chapter:

      1) By a **"new" OTS transaction** we mean an OTS transaction *in which the target logical server is not already participating*, before the current method call; *not* necessarily an OTS transaction that was started immediately before the method call.

      2) By an **"existing" OTS transaction** we mean an OTS transaction *in which the target logical server is already participating*, before the current method call; *not* an OTS transaction that was started some time ago.

   c. For example, if a client starts an OTS transaction, does some work, and then calls a method on an enterprise bean with the **Supports** transaction attribute, so far as the CICS EJB server is concerned this is a "new" OTS transaction, because the server has not been called within this transaction's scope before. If the client then makes a second and third method call to the

same target object, before committing its OTS transaction, these second and third calls occur within the scope of the existing OTS transaction.

Setting up a logical EJB server

You must follow a number of steps to set up a CICS logical EJB server to support enterprise beans.

Before setting up a logical EJB server, make sure that the regions in a logical EJB server, both listeners and AORs, are at the same level of CICS.

Follow these steps to set up a CICS logical EJB server to support enterprise beans:

1. Create a set of cloned CICS Transaction Server for z/OS, Version 4 Release 1 listener regions. Each listener region must have the IIOPLISTENER system initialization parameter set to YES.
2. Create a set of cloned CICS Transaction Server for z/OS, Version 4 Release 1 AORs. Each of the AORs must meet these criteria:
   - Be set up to use JNDI
   - Use the same JNDI initial context as the other AORs
   - Be connected to all of the listener regions by MRO (not ISC)
   - Have the IIOPLISTENER system initialization parameter set to NO.
3. Create a shelf root directory on z/OS UNIX. For example, you might create a directory called `/var/cicsts/`. To do so, you need a z/OS UNIX user ID with write authority to the directory path to be used by CICS. Having created the shelf directory, you must give the user IDs of the AOR full access read, write, and run access to the directory.
4. Create a deployed JAR file (pickup) directory on z/OS UNIX. For example, you might create a directory called `/var/cicsts/pickup`. The AORs must have at least read access to it.

   If your AORs are to contain more than one CorbaServer runtime environment:
   - You must create a separate pickup directory for each CorbaServer.
   - Assign different sets of transaction IDs to the objects supported by each CorbaServer. That is, each CorbaServer in an AOR supports a different set of transaction IDs. To assign transaction IDs to bean methods, use REQUESTMODEL definitions; see step 5.
5. Create the following resource definitions. You can create them on a CSD that is shared by all the regions in the logical server, copy them to all the CSDs used by the regions, or add them to a CICSPlex SM Resource Description that applies to all the regions. Optionally, you can use the CICS scanning mechanism, the Resource Manager for enterprise beans, and the CICS-supplied transaction, CREA, to create some of these definitions, as described below.
   - A TCPIPSERVICE.
     - On the PROTOCOL option, specify IIOP.
     - On the SSL option, specify NO.
     - On the AUTHENTICATE option, specify NO. With this specification, the service on this port accepts unauthenticated inbound IIOP requests.
   - Some REQUESTMODEL definitions. In a single-region EJB server, the definitions are only required if the default TRANSID, CIRP, is unsuitable. In a multiregion logical server, however, the definitions are required if you want to route method requests across several AORs. The TRANSACTION

definition for CIRP specifies DYNAMIC(NO). Definitions are also required if, for example, you want to segregate your IIOP workload by transaction ID.

Note:

a. The BEANNAME attribute of each REQUESTMODEL definition must "match" (in a pattern-matching sense) the name of an enterprise bean in the deployment descriptor in a deployed JAR file on z/OS UNIX. The value of the CORBASERVER attribute must match, either literally or in a pattern-matching sense, the name of the CorbaServer on the CORBASERVER definition.

b. Copy the transaction definition for the TRANSID named on your REQUESTMODEL from that of CIRP. Set the DYNAMIC attribute to YES. You can change any of the other attributes, but the program name must be that of a JVM program with acJVMClass of com.ibm.cics.iiop.RequestProcessor.

c. When the CorbaServer is operational, you can use the CREA CICS-supplied transaction to display the transaction IDs associated with particular beans and bean methods in the CorbaServer. You can change the transaction IDs, apply the changes, and save the changes to new REQUESTMODEL definitions.

- A CORBASERVER definition.

  The value of the HOST option of the CORBASERVER definition must match that of the HOST or IPADDRESS option of the TCPIPSERVICE definition. However, if the TCPIPSERVICE specifies a value for DNSGROUP, the HOST option of the CORBASERVER definition must specify a matching generic host name.

  On the UNAUTH option, specify the name of the TCPIPSERVICE definition. You must always specify a value for the UNAUTH attribute when you define a CorbaServer, even if you intend that all inbound requests to the CorbaServer will be authenticated. This value is required because the port number from the TCPIPSERVICE is used to construct Interoperable Object References (IORs) that are exported from this logical server. You can, by specifying the name of other TCPIPSERVICE definitions on one or both of the CLIENTCERT or SSLUNAUTH options, cause your listener regions to listen on other ports for different types of authenticated inbound IIOP requests. For more information, see the documentation of the CORBASERVER and TCPIPSERVICE resource definitions in the *CICS Resource Definition Guide*.

  On the SHELF option, specify the fully qualified name of the z/OS UNIX shelf directory that you created in step 3. Because the CORBASERVER definition is installed on all the AORs in the logical server, this "high-level" shelf directory is shared by all of them. Each AOR automatically creates its own subdirectory beneath the shelf directory and a subdirectory for the CorbaServer beneath that.

  On the DJARDIR option, specify the fully qualified name of the z/OS UNIX deployed JAR file directory (pickup directory) that you created in step 4. Like the shelf directory, the pickup directory (or directories, if your AORs contain multiple CorbaServers) is shared by all the AORs in the logical server. On each AOR, when a CORBASERVER definition is installed, CICS scans the CorbaServer pickup directory and installs any deployed JAR files that it finds there. It copies them to its shelf subdirectory and dynamically creates and installs DJAR definitions for them.

Specify AUTOPUBLISH(YES) to cause CICS to publish beans to the namespace automatically, when a DJAR definition is successfully installed.

On the STATUS option, specify Enabled.

- FILE definitions for the following files required by CICS:

  The EJB directory, DFHEJDIR
  Is a file containing a request streams directory, which must be shared by all the regions, listeners and AORs, in the logical EJB server. Request streams are used in the distributed routing of method requests for enterprise beans and CORBA stateless objects. You must define DFHEJDIR as recoverable.

  The EJB object store, DFHEJOS
  Is a file of stateful session beans that have been passivated. It must be shared by all the AORs in the logical EJB server. You must define it as nonrecoverable.

  To share DFHEJDIR and DFHEJOS across multiple regions, you can, for instance, use any of the following methods:

  – Define them as remote files in a file-owning region (FOR)
  – Define them as coupling facility data tables
  – Use VSAM RLS

  Sample FILE definitions are in these groups:

  – For DFHEJDIR and DFHEJOS are in the CICS-supplied RDO group, DFHEJVS
  – For DFHEJDIR and DFHEJOS are in the CICS-supplied RDO group, DFHEJCF

  Sample VSAM RLS FILE definitions for DFHEJDIR and DFHEJOS are in the CICS-supplied RDO group, DFHEJVR. DFHEJVS, DFHEJCF, and DFHEJVR are not included in the default CICS startup group list, DFHLIST.

Note: These steps assume that the logical server has only one CorbaServer. To create another CorbaServer, create a second CORBASERVER definition and another TCPIPSERVICE definition.

6. Define the underlying VSAM data sets for DFHEJDIR and DFHEJOS. CICS supplies sample JCL to help you, in the DFHDEFDS member of the SDFHINST library.
7. Using a deployment tool such as the Assembly Toolkit (ATK), take one or more ejb-jar files and perform code generation on them to produce deployed JAR files on z/OS UNIX. Store the deployed JAR files in the pickup directory of the CorbaServer.
8. Start all the CICS regions. On each of the listener regions, the definitions to be installed from the CSD are as follows:
   - The TCPIPSERVICE definition
   - The REQUESTMODEL definitions
   - The file definition for DFHEJDIR

   On each of the AORs, the definitions to be installed from the CSD are as follows:
   - The TCPIPSERVICE definition.
   - The REQUESTMODEL definitions.

     The REQUESTMODEL definitions in the AORs are required for outbound requests to local objects. If a CORBA stateless object or enterprise bean makes a call to another object, and that object is available on the local AOR,

CICS does not send the request to a listener region. Instead, it either runs the called method in the current task ("tight loopback") or starts another request processor in the local AOR ("normal loopback"). When normal loopback is used, it is preferable that the new request processor task uses the same REQUESTMODEL as that used for the call to the first object; otherwise, unpredictable results might occur. If your CORBA stateless objects and enterprise beans make no outbound calls, the REQUESTMODELs on the AOR are not strictly required.

- The CORBASERVER definition.
- The file definitions for DFHEJDIR and DFHEJOS.

If you put your deployed JAR files in the shared pickup directory, DJAR definitions are created and installed on the AORs automatically when the CorbaServer is installed, or when a subsequent scan takes place. Create static (CSD-installed) DJAR definitions only for deployed JAR files that you place in other z/OS UNIX directories.

9. On each AOR, when the CORBASERVER definition is installed, CICS scans the pickup directory and installs any deployed JAR files it finds there. It copies them to its shelf directory and dynamically creates and installs DJAR definitions for them.

   You can put deployed Jis installed. If you do so, you can force CICS to perform another scan by issuing a **CORBASERVER PERFORM SCAN** command. Issue this command using **EXEC CICS**, the CEMT master terminal transaction, or the Web-based resource manager for enterprise beans, otherwise known as the RM for enterprise beans.

10. Because you specified AUTOPUBLISH(YES) on the CORBASERVER definition, when the DJAR definitions are successfully installed the homes of the enterprise beans are automatically bound into the JNDI namespace.

    If you specify AUTOPUBLISH(NO), you must issue a **PERFORM CORBASERVER(CorbaServer_name) PUBLISH** command on at least one of the AORs. You must issue this command using **EXEC CICS**, the CEMT master terminal transaction, the RM for enterprise beans, or from a CICSPlex SM WUI view.

11. On the DSRTPGM system initialization parameter for the listener regions, specify the name of the distributed routing program to be used. If you are using CICSPlex SM, specify the name of the CICSPlex SM routing program, EYU9XLOP. Otherwise, specify the name of your customized routing program. For information about the DSRTPGM system initialization parameter, see the *CICS System Definition Guide*.

Figure 19 on page 235 shows the RDO definitions required to define a CICS logical EJB server. It shows which definitions are required in the listener regions, which in the AORs, and which in both.

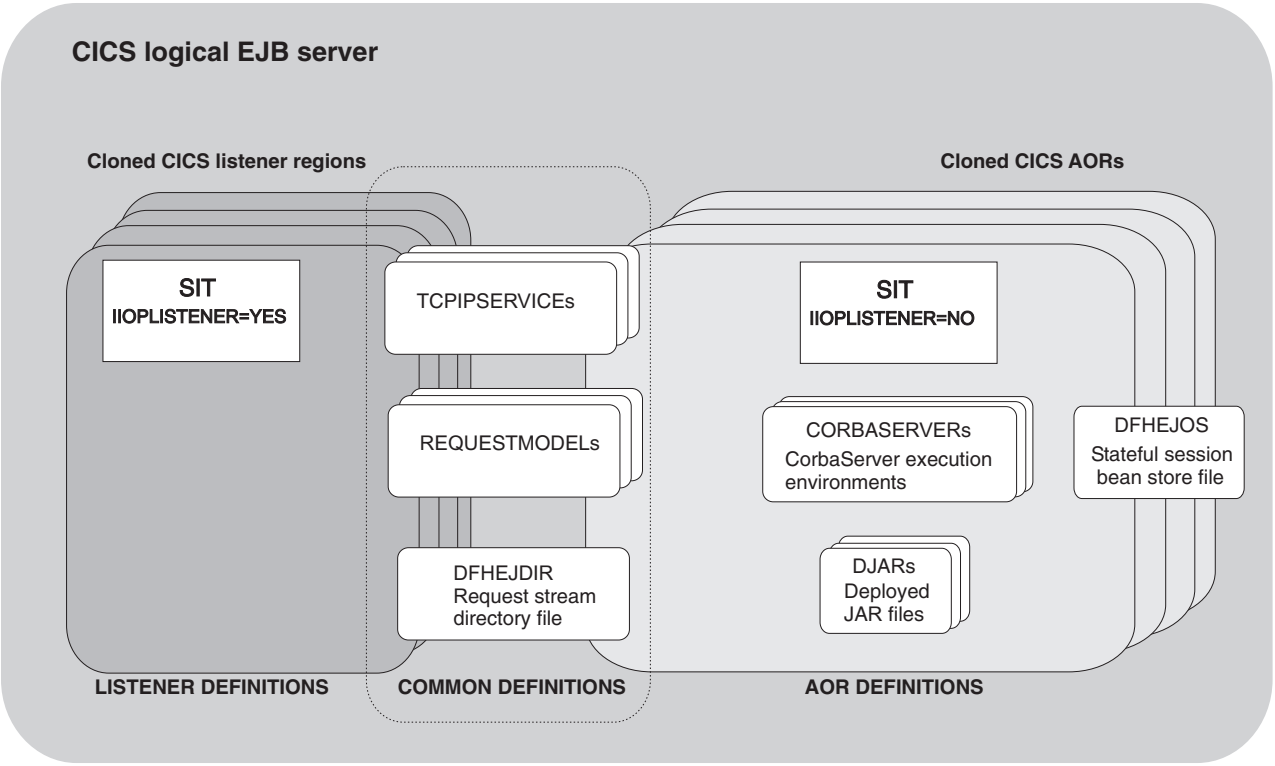

Figure 19. Resource definitions in a CICS logical EJB server

Enterprise beans—what can a client do with a bean?

This section contains example code fragments that illustrate how a client program can use an enterprise bean.

Get a reference to the bean's home

In order to do anything with the bean, the client must obtain a reference to the bean's home interface.

To do this, it looks up a well-known name via JNDI:

```
// Obtain a JNDI initial context
Context initContext = new InitialContext();

// Look up the home interface of the bean
Object accountBeanHome = initContext.lookup("JNDI_prefix/AccountBean");
// where:
// 'JNDI_prefix/' is the JNDI prefix on the CORBASERVER definition
// 'AccountBean' is the name of the bean in the XML deployment descriptor

// Convert to the correct type
AccountHome accountHome = (AccountHome)
            PortableRemoteObject.narrow(accountBeanHome,AccountHome.class);
```

Use the home interface

The client can use the bean's home interface to create a new instance of the bean, and delete an instance of the bean.

```
// Create two bean instances
Account anAccount = accountHome.create();
Account anotherAccount = accountHome.create("12345");

// Remove a bean instance
accountHome.remove("12345");
```

Use the component interface

The client can use the bean's component interface to invoke the bean's methods, and delete the bean.

```
// Use the bean
anAccount.deposit(1000000);
// Remove it
anAccount.remove();
```

Enterprise beans—what can a bean do?

An enterprise bean benefits from many services—such as lifecycle management and security—that are provided implicitly by the EJB container, based on settings in the deployment descriptor.

This leaves the bean provider free to concentrate on the bean's business logic. This section looks at some of the things a bean can do.

Look up JNDI entries
A bean can use JNDI calls to retrieve:

- References to resources
- Environment variables
- References to other beans.

Access resource managers
A bean can:

- Obtain a connection to a resource manager
- Use the resources of the resource manager
- Close the connection.

Link to CICS programs
A bean can use JCICS or the CCI Connector for CICS TS to link to a CICS program, that may be written in any of the CICS-supported languages and be either local or remote. The bean provider can use the CCI Connector for CICS TS to build beans that make use of the power of existing (non-Java) CICS programs.

The CCI Connector for CICS TS is described in Chapter 25, “The CCI Connector for CICS TS,” on page 319.

Access files
A bean can use JCICS to read and write to files.

Call other beans
A bean can:

- Obtain references to the home and component interfaces of other bean objects
- Invoke the methods of another bean object
- Be called from another bean object.

A bean can act as the client of another bean object, as the server of another bean object, or as both.

Bear in mind that a single CICS task (one instance of a transaction) cannot contain more than one enterprise bean, because CICS treats an execution of an enterprise bean as the start of a new task. You can create an application that includes more than one enterprise bean, but the application will not operate as a single CICS task.

Manage transactions
Optionally, a session bean can manage its own OTS transactions, rather than use container-managed transactions. Alternatively, it may have its transaction managed by its caller.

Benefits of EJB technology

This lists some benefits of using enterprise beans.

Component portability
The EJB architecture provides a simple, elegant component container model. Java server components can be developed once and deployed in any EJB-compliant server.

Architecture independence
The EJB architecture is independent of any specific platform, proprietary protocol, or middleware infrastructure. Applications developed for one platform can be redeployed on other platforms.

Developer productivity
The EJB architecture improves the productivity of application developers by standardizing and automating the use of complex infrastructure services such as transaction management and security checking. Developers can create complex applications by focusing on business logic rather than environmental and transactional issues.

Customization
Enterprise bean applications can be customized without access to the source code. Application behavior and runtime settings are defined through attributes that can be changed when the enterprise bean is deployed.

Multi-tier technology
The EJB architecture overlays existing infrastructure services.

Versatility and scalability
The EJB architecture can be used for small-scale or large-scale business transactions. As processing requirements grow, the enterprise beans can be migrated to more powerful operating environments.

In addition to these general benefits of using EJB technology, there are specific benefits of using enterprise beans with CICS. For example:

Superior workload management
You can balance client connections across a set of cloned listener regions.

You can use CICSPlex SM or the CICS distributed routing program to balance OTS transactions across a set of cloned AORs.

Superior transaction management
Enterprise beans in a CICS EJB server benefit from CICS transaction management services—for example:

- Shunting
- System log management
- Performance optimizations

- Runaway detection
- Deadlock detection
- TCLASS management
- Monitoring and statistics

Access to CICS resources
You can, for example, use JCICS or the CCI Connector for CICS TS to build enterprise beans that make use of the power of existing (non-Java) CICS programs. The developer of a Java client application can use your server components to access CICS—without needing to know anything about CICS programming. See Chapter 25, "The CCI Connector for CICS TS," on page 319.

Requirements for EJB support

This lists any hardware and software requirements to support Enterprise JavaBeans.

Hardware

There are no specific hardware requirements for enterprise beans, over and above those for CICS Transaction Server for z/OS, Version 4 Release 1 itself.

Software

These are the software requirements for enterprise beans.

- IBM SDK for z/OS, Java Technology Edition. CICS TS 4.1 supports Version 6 of the SDK.

  Note: 31-bit and 64-bit versions of the IBM SDK for z/OS, Java Technology Edition are available. CICS TS 4.1 supports only the 31-bit version.
- A name server that supports the Java Naming and Directory Interface (JNDI) Version 1.2. (The JNDI API provides directory and naming functions for Java applications. It enables a client to locate an enterprise bean. The JNDI is mapped to an external name server.) You can use either of the following:

  A Lightweight Directory Access Protocol (LDAP) name server
  such as IBM SecureWay™ Directory, which is shipped with the IBM SecureWay Security Server, an optional feature of z/OS.

  A distributed version of IBM SecureWay Directory is also available.

  A Corba Object Services (COS) Naming Directory Service
  such as that provided with IBM WebSphere Application Server Version 6. This provides a transient CosNaming Service implementation. Being transient means that its contents are lost when it is stopped or restarted; as such, it is likely to be used only on a test system.

  Any industry-standard COS Naming Server that supports JNDI Version 1.2 can be used. For example, CICS also supports the COS Naming Server supplied with IBM WebSphere Application Server Advanced Edition for AIX®, Version 3.5 and later.

  WebSphere Application Server Version 5.0, or later
  The required component is the Assembly Toolkit (ATK) for Windows, which is used to deploy enterprise beans. (The Application Assembly Tool (AAT), provided with WebSphere Application Server Version 4 and early copies of WebSphere Application Server Version 5.0, can still be used but is not supported).

Chapter 19. Setting up an EJB server

This chapter tells you how to set up and test an EJB server.

Setting up a single-region EJB server

This section tells you how to set up a single-region CICS EJB server. The single-region is both a listener region and an AOR.

This minimal configuration can be used as the basis for developing a multi-region CICS EJB server, as described in "Setting up a multi-region EJB server" on page 246.

Important

- For clarity's sake, we're assuming that:
  1. You start from a basic, non-customized, CICS Transaction Server for z/OS, Version 4 Release 1 region.
  2. There will be only one CorbaServer execution environment in your EJB server.
- We recommend that, when creating your first EJB server, you use the default JVM profile, DFHJVMCD. After you've got your first EJB server up and running, you may want to customize your JVM profile. How to do this is described in "After running the EJB IVP—optional steps" on page 245.
- This section doesn't tell you how to deploy enterprise beans. Deployment is a separate process that occurs after you've set up your EJB server. It's described in Chapter 23, "Deploying enterprise beans," on page 303.
- The rest of this section is split into two parts:
  - "Before running the EJB IVP" takes you as far as being able to run the EJB Installation Verification Program, which tests that you have configured CICS correctly as an EJB server and set up a name server correctly.

    Note: By default the EJB IVP uses the lightweight `tnameserv` COS Naming Server that is supplied with Java 1.3 and later. Therefore you don't need to have set up an enterprise-quality name server before running the IVP. However, after you've set up your "real" name server, you can use the IVP to test it.
  - "After running the EJB IVP—optional steps" on page 245 describes some optional ways in which you can customize your EJB server.

Before running the EJB IVP

The steps in this section enable you to run the EJB Installation Verification Program, which tests that you have configured CICS correctly as an EJB server.

The steps in this section enable you to run the EJB Installation Verification Program, which tests that you have configured CICS correctly as an EJB server. Actions are required on:

1. z/OS or Windows NT, depending on the type of name server that you use
2. z/OS UNIX
3. CICS

Actions required on z/OS or Windows NT

To run the EJB IVP, you need a name server that supports the Java Naming and Directory Interface (JNDI) Version 1.2. By default the IVP uses the lightweight `tnameserv` COS Naming Server that is supplied with Java 1.3 and later.

To start `tnameserv` on the local host, enter the following command at the z/OS UNIX System Services or Windows NT command prompt:

```
tnameserv -ORBInitialPort 2809
```

This causes the name server to listen for connections on TCP/IP port 2809. If this port is already in use on your system, you will be asked to try again with a different port.

Note: If you run firewall software, by default the firewall may block your specified port. You must ensure that your firewall policy allows CICS and any EJB client applications to communicate with the name server.

For information about choosing and setting up an enterprise-quality name server, see "Enabling JNDI references" on page 183.

Actions required on z/OS UNIX

To perform the tasks in this section, you need a z/OS UNIX userid with write authority to the directory path to be used by CICS.

About this task

Create the following directories on z/OS UNIX, if they do not already exist. (If you have previously configured CICS as an IIOP server, some of these directories may already exist.) Remember that z/OS UNIX names are case-sensitive.

1. A CICS working directory. Each CICS region needs a working directory. The name is specified by the WORK_DIR parameter of the JVM profile. You need to set the directory permissions so that the USERID the region runs under can read and write to the directory. See Giving CICS regions access to z/OS UNIX System Services for guidance.
2. A shelf root directory. You can call your shelf directory anything you like. However, it's recommended that you create it somewhere under the `/var` directory. For example, you might create a z/OS UNIX directory called `/var/cicsts/`. Having created the shelf directory, you must give the CICS region userid full access to it—read, write, and execute. How to do this is described in Giving CICS regions access to z/OS UNIX System Services.
3. A deployed JAR file directory (also known as a pickup directory). You can call your pickup directory anything you like. However, it's recommended that you create it somewhere under the `/var` directory. For example, you might create a z/OS UNIX directory called `/var/cicsts/pickup`. You must give the CICS region userid at least read access to it.

   Note:

   a. If you were to install multiple CorbaServer execution environments into your EJB server, you would need to create a separate pickup directory for each one.

   b. If you use the scanning mechanism (to install deployed JAR files from the pickup directory) in a production region, be aware of the security implications: specifically, the possibility of CICS command security on DJAR definitions being circumvented. To guard against this, we recommend that

user IDs given write access to the z/OS UNIX deployed JAR file directory should be restricted to those given RACF authority to create and update DJAR and CORBASERVER definitions.

Actions required on CICS

Note that if you have previously configured CICS as an IIOP server, to support method calls to CORBA stateless objects, you might already have completed some of these steps.

About this task

1. Install the IBM SDK for z/OS, Java Technology Edition. You can download this product, and find out more information about it, at http://www.ibm.com/servers/eserver/zseries/software/java/.
2. Set up CICS to support IIOP calls. (CICS uses the same RMI-over-IIOP protocol to support client method requests for both CORBA stateless objects and enterprise beans.) How to do this is described in "Setting up CICS for IIOP" on page 195.

   Bear in mind when reading "Setting up CICS for IIOP" on page 195 that:

   - Because our single-region EJB server is a combined listener/AOR, you must specify 'YES' on the IIOPLISTENER system initialization parameter.
   - CICS loads JVM profiles from the z/OS UNIX directory that is specified by the **JVMPROFILEDIR** system initialization parameter. Make sure this value specifies the directory containing the JVM profiles used by your CICS region.
   - If you want to use your single-region server as the basis of a multi-region server, you should ensure that the request streams directory file, DFHEJDIR, and the EJB object store file, DFHEJOS, can be shared across multiple regions. For this reason, it is recommended that you define them in one of the following ways:
     - As remote files in a file-owning region (FOR)
     - As coupling facility data tables
     - Using VSAM RLS.
   - PROGRAM definitions are not required for enterprise beans as such. The only PROGRAM definitions required are those for the request receiver and request processor programs. The default request processor program—named by the default CIRP transaction on REQUESTMODEL definitions—is DFJIIRP. CIRP and DFJIIRP are defined in the supplied resource definition group DFHIIOP, as are CIRR and DFHIIRRS, the request receiver transaction and program. DFHIIOP is included in the default CICS startup group list.

     If you are using a JVM profile other than the default DFHJVMCD, you must specify the name of your profile on the JVMPROFILE option of the PROGRAM definition for the request processor program. (It is possible to use a CEMT SET PROGRAM JVMPROFILE command to change the JVM profile from that specified on the installed PROGRAM definition. However, if you create your own JVM profile you are recommended to create new TRANSACTION and PROGRAM definitions for the request processor program, rather than change the default definitions.)
   - You must specify the location of your name server on the **-Dcom.ibm.cics.ejs.nameserver** system property in the profiles that are used by CORBA applications or enterprise beans, including the profiles that CICS uses to publish deployed JAR files.

     For detailed information about defining the location of your name server, see "JVM system properties" on page 119.

- You don't need to install REQUESTMODEL or DJAR definitions at this stage, because:
  - The EJB IVP and EJB sample applications use the default REQUESTMODEL transaction ID, CIRP.
  - REQUESTMODEL definitions are most easily created by using the CREA transaction after you have deployed your enterprise beans into CICS. Deployment is a separate process that occurs after you have set up your EJB server. It is described in Chapter 23, "Deploying enterprise beans," on page 303.
  - DJAR definitions are typically created and installed by the CICS scanning mechanism during deployment.

3. Create the following CICS resource definitions:
   - A TCPIPSERVICE
   - A CORBASERVER

   The CICS-supplied sample group, DFH$EJB, contains TCPIPSERVICE and CORBASERVER definitions suitable for running the EJB IVP. You must change some of the attributes of these resource definitions to suit your own environment. To do this, use the CEDA transaction or the DFHCSDUP utility.

   a. Copy the sample group to a group of your own choosing. For example:
      ```
      CEDA COPY GROUP(DFH$EJB) TO(mygroup)
      ```
   b. Display group `mygroup` and change the following attributes appropriately:
      - On the TCPIPSERVICE resource definition, modify the PORTNUMBER as necessary to a suitable TCP/IP port on your installation. The port number that you specify must be authorized by your network administrator.

        Note:

        1) Note that, on the supplied TCPIPSERVICE definition:
           - The PROTOCOL option specifies IIOP. This is the required protocol for method calls to enterprise beans and CORBA stateless objects.
           - The SSL option specifies NO.
           - The AUTHENTICATE option defaults to NO. This means that the service on this port will accept unauthenticated inbound IIOP requests.
        2) If you want to use your single-region server as the basis of a multi-region server, as described in "Setting up a multi-region EJB server" on page 246, you should specify a value for the DNSGROUP option. This ensures that, in a multi-region server, you will be able to use connection optimization, by means of dynamic DNS registration, to balance client connections across the listener regions.
        3) For reference information about TCPIPSERVICE definitions, see the *CICS Resource Definition Guide*.
      - On the CORBASERVER resource definition:
        1) Modify the SHELF option so that it specifies the fully-qualified name of the z/OS UNIX shelf directory that you created in step 2 of "Actions required on z/OS UNIX" on page 240.

           Note: In a multi-region EJB server, because the CORBASERVER definition will be installed on all the AORs this "high-level" shelf directory will be shared by all of them. Each AOR will automatically create its own sub-directory beneath the shelf directory, and a sub-directory for the CorbaServer beneath that.

2) Modify the DJARDIR option so that it specifies the fully-qualified name of the z/OS UNIX deployed JAR file directory (pickup directory) that you created in step 3 of "Actions required on z/OS UNIX" on page 240.

Note: In a multi-region EJB server, the pickup directory (or directories, if the AORs contain multiple CorbaServers), like the shelf directory, will be shared by all the AORs in the logical server.

3) Set the HOST to your TCP/IP hostname.

Note:

1) Note that, on the supplied CORBASERVER definition:
   - The UNAUTH option specifies the name of the TCPIPSERVICE definition.

     You must always specify a value for the UNAUTH attribute when you define a CorbaServer, even if you intend that all inbound requests to the CorbaServer should be authenticated. This is because the port number from the TCPIPSERVICE is used to construct Interoperable Object References (IORs) that are exported from this logical server. You can, by specifying the name of other TCPIPSERVICE definitions on one or both of the CLIENTCERT or SSLUNAUTH options, cause your listener regions to listen on other ports for different types of authenticated inbound IIOP requests. For more information, see the *CICS Resource Definition Guide*.
   - The AUTOPUBLISH option specifies YES. This causes CICS to publish beans to the namespace automatically, when a DJAR definition is successfully installed.
   - The STATUS option specifies Enabled.
2) The value of the HOST option of the CORBASERVER definition must be compatible with that of the HOST or IPADDRESS options for the associated TCPIPSERVICE resources. In a multi-region server, if dynamic DNS registration is used to balance client connections across the listener regions, the value of the HOST option must match the generic host name specified on the DNSGROUP option of the TCPIPSERVICE definition.
3) For reference information about CORBASERVER definitions, see the *CICS Resource Definition Guide*.

c. Install group `mygroup` to make these definitions known to CICS.

   When the CORBASERVER definition is installed, CICS:

   1) Scans the pickup directory that you specified on the DJARDIR option
   2) Copies any deployed JAR files that it finds in the pickup directory to its shelf directory
   3) Dynamically creates and installs DJAR definitions for the deployed JAR files (if any) that it found in the pickup directory
   4) Because the CORBASERVER definition specifies AUTOPUBLISH(YES), publishes any enterprise beans contained in the DJARs to the JNDI namespace.

d. Set the status of the TCPIPSERVICE to OPEN:

```
CEMT SET TCPIPSERVICE(EJBTCP1) OPEN
```

   On the CICS Console, you should see, among others, messages similar to the following:

```
DFHEJ0701 CorbaServer EJB1 has been created.
DFHEJ5024 Scan commencing for CorbaServer EJB1, directory being scanned is
          DJARDIR_name.
DFHEJ5025 Scan completed for CorbaServer EJB1, 0 DJars created, 0 DJars
          updated.
DFHEJ1520 CorbaServer EJB1 is now accessible.
DFHSO0107 TCPIPSERVICE EJBTCP1 has been opened on port port_number at IP
          address xxx.xxx.xxx.xxx
```

where:

- **DJARDIR_name** is the name of your CorbaServer's deployed JAR file ("pickup") directory.
- **port_number** is the number of the TCP/IP port used by your CorbaServer.
- **xxx.xxx.xxx.xxx** is your CorbaServer's IP address.

4. Set up CICS to use JNDI. To enable Java code running under CICS to issue JNDI API calls, and CICS to publish references to the home interfaces of enterprise beans, you must specify the location of the name server. (For an LDAP name server there is additional information to be specified.) Specify the URL and port number of your name server on the **-Dcom.ibm.cics.ejs.nameserver** system property.

   For example, to use `tnameserv`, the lightweight COS Naming Directory Server supplied with Java 1.3 and later, specify:

   ```
   -Dcom.ibm.cics.ejs.nameserver=iiop://tnameserv.yourcompany.com:2809
   ```

   where `tnameserv.yourcompany.com` is the address of the host on which you started the `tnameserv` name server and `2809` is the port you selected.

   If you are using an enterprise-quality LDAP server you might specify:

   ```
   -Dcom.ibm.cics.ejs.nameserver=ldap://demojndi.yourcompany.com:389
   ```

   For the other properties that are required, and the way to set up your LDAP name server, see "Setting up an LDAP server" on page 184.

   If you are using a standard COS Naming Directory Server you might specify:

   ```
   -Dcom.ibm.cics.ejs.nameserver=iiop://demojndi.yourcompany.com:900
   ```

   If you are using the COS Naming Directory Server supplied with WebSphere Application Server Version 5 or later, you should specify:

```
-Dcom.ibm.cics.ejs.nameserver=iiop://demojndi.yourcompany.com:2809/domain/legacyRoot
```

   Important: For detailed information about defining the location of the name server, see the description of the **-Dcom.ibm.cics.ejs.nameserver** property in "JVM system properties" on page 119.

   The JVM profile for the default request processor program is DFHJVMCD. If you have followed the previous steps in this section, the profile or profiles you are using should be in the z/OS UNIX directory specified by the **JVMPROFILEDIR** system initialization parameter.

Important: These instructions have shown you how to set up a single-region EJB server that contains a single CorbaServer execution environment. In a production region that supports multiple applications, each of which uses its own set of enterprise beans, you may require multiple CorbaServers. To facilitate maintenance in a production region, you should follow the guidelines on how to allocate beans to CorbaServers and transaction IDs in Chapter 24, "Updating enterprise beans in a production region," on page 307.

Having completed the above steps, you can, if you wish, run the EJB Installation Verification Program, which tests that you have configured CICS correctly as an EJB server. For details of the EJB IVP, see Chapter 20, "Using the EJB IVP," on page 257. Alternatively, you can continue with the next section before running the IVP.

After running the EJB IVP—optional steps

Optionally, to finish the setup of your complete EJB server, you can customize one of the sample JVM profiles, or create your own JVM profiles for use with enterprise beans, rather than using the default JVM profile DFHJVMCD.

About this task

DFHJVMCD can only be customized in limited ways, because it is used for internal CICS programs, but other JVM profiles can be customized as you want.

"Setting up JVM profiles" on page 93 tells you how to select and customize a JVM profile, or if you prefer, how to create your own JVM profile based on one of the supplied sample profiles. Follow the procedures in that section to customize or create your JVM profile.

When you have customized or created your JVM profile, in order for the profile to be used by enterprise beans:

1. Specify the name of your JVM profile on the JVMPROFILE option of the PROGRAM definition for the request processor program. (The supplied PROGRAM definition for the default request processor program, DFJIIRP, specifies the default profile, DFHJVMCD.)

   You should create your own TRANSACTION and PROGRAM definitions for the request processor program, as described in "Defining CICS resources" on page 196, rather than change the default definitions. Specify the name of your TRANSACTION on REQUESTMODEL definitions for bean methods that are to run under the new profile.
2. Place your profile in the z/OS UNIX directory specified by the **JVMPROFILEDIR** system initialization parameter.

Important: You must specify the location of your name server on the **-Dcom.ibm.cics.ejs.nameserver** system property in all the JVM profiles or optional properties files that are used by CORBA applications or enterprise beans, including the profiles that CICS uses to publish deployed JAR files. For detailed information about defining the location of your name server, see "JVM system properties" on page 119.

Testing your EJB server

This section tells you how to check that your single-region CICS EJB server is configured correctly.

Running the EJB IVP

The easiest way to test your CICS EJB configuration, including that of your name server, is to run the EJB Installation Verification Program (IVP) supplied with CICS.

The IVP consists of:

- A line-mode client program that runs in UNIX System Services (USS) on z/OS
- An enterprise bean running on the CICS EJB server

To run the IVP, you must have completed all the steps in “Before running the EJB IVP” on page 239. You may or may not have completed the steps in “After running the EJB IVP—optional steps” on page 245. Running the IVP successfully confirms that external programs are able to invoke enterprise beans on your CICS EJB server.

For details of the EJB IVP, see Chapter 20, “Using the EJB IVP,” on page 257.

Using the EJB “Hello World” sample

“Hello World” is a simple application consisting of an HTML form, a Java servlet and Java Server Pages running on a Web server, and a CICS enterprise bean.

It requests input from the user, uses the enterprise bean to append the user's input to a standard message, and then displays the resulting string.

To run the EJB “Hello World” sample, you must have completed all the steps in “Before running the EJB IVP” on page 239. You may or may not have completed the steps in “After running the EJB IVP—optional steps” on page 245.

For details of the EJB “Hello World” application, and instructions on how to install it, see “The EJB “Hello World” sample application” on page 263.

Using the EJB Bank Account sample

After you've run the Hello World” sample successfully, you might want to try something more ambitious.

The EJB Bank Account sample demonstrates how you can use an enterprise bean to make CICS-controlled information available to Web users. It extracts customer information from data tables and returns it to the user.

The sample consists of an HTML form, a Java servlet and Java Server Pages running on a Web server, a CICS enterprise bean, two CICS COBOL server programs, and some DB2 data tables. The enterprise bean uses the CCI Connector for CICS TS to link to the CICS server programs, which access the DB2 data tables.

To run the EJB Bank Account sample, you must have completed all the steps in “Before running the EJB IVP” on page 239. You may or may not have completed the steps in “After running the EJB IVP—optional steps” on page 245.

For details of the EJB Bank Account application, and instructions on how to install it, see “The EJB Bank Account sample application” on page 270.

Using your own enterprise beans

After you've run the sample applications and established that your CICS EJB server is working correctly, you'll probably want to deploy your own enterprise beans into CICS.

For details of how to do this, see Chapter 23, “Deploying enterprise beans,” on page 303.

Setting up a multi-region EJB server

This section tells you how to set up a CICS logical EJB server consisting of multiple listener regions and multiple AORs.

About this task

It assumes that you have already created a single-region EJB server, as described in "Setting up a single-region EJB server" on page 239.

Important: It is strongly recommended that all the regions in a multi-region EJB server—both listeners and AORs—should be at the same level of CICS.

Procedure

1. Create a set of listener regions by cloning the single-region-server CICS. (All the cloned regions share the CICS system definition file (CSD) of the single-region server.) Optionally, you can discard the following resource definitions from the listener regions, where they're not required:
   - CORBASERVER
   - DJARs
   - DFHEJOS

   Leave the value of the IIOPLISTENER system initialization parameter set to 'YES'.

   Note: If you use CICSPlex SM, you can define a CICS Group (CICSGRP) containing all of the listener regions. This has the advantage that resources can be associated (by means of a Resource Description) with the Group rather than with individual regions. When a region is added to or removed from the Group, the resources are automatically added to or removed from the region.
2. Create a set of AORs by cloning the single-region-server CICS. (All the cloned regions share the CSD of the single-region server.)

   Each of the AORs must use the same JNDI initial context as the other AORs.

   Because the AORs are not listener regions, change the value of the IIOPLISTENER system initialization parameter to 'NO'.

   Note: If you use CICSPlex SM, you can define a CICS Group (CICSGRP) containing all of the AORs. When a region is added to or removed from the Group, the resources are automatically added to or removed from the region.

   Figure 20 on page 249 shows which definitions are required in the listener regions, which in the AORs, and which in both.
3. Connect each of the AORs to all of the listener regions by MRO (not ISC). For information about how to define MRO connections between CICS regions, see the *CICS Intercommunication Guide*.

   If you use CICSPlex SM, you can significantly reduce the number of CONNECTION and SESSION definitions required (and the cost of maintaining them) by defining SYSLINKs from a single AOR to all of the listener regions. (CICSPlex SM automatically creates the reciprocal connections from the listeners to the AOR.) Use the SYSLINKs as models for the connections from the other AORs.
4. Ensure that the EJB Directory file, DFHEJDIR, is shared by all the regions in the EJB server. If you defined DFHEJDIR to the single-region EJB server in the way suggested (that is, as a remote file, a coupling facility data table, or as using VSAM RLS) the file should be shared automatically across the cloned regions of the multi-region server.

   Note: Ensure that the CICS region that owns the DFHEJDIR file is started before the other regions that access it, particularly the AORs. If you don't, attempts to install CORBASERVER and DJAR definitions on the other AORs will fail with message DFHEJ0736.

5. Ensure that the EJB Object Store file, DFHEJOS, is shared by all the AORs in the EJB server. If you defined DFHEJOS to the single-region EJB server in the way suggested, the file should be shared automatically across all the cloned regions of the multi-region server. (Optionally, you can delete the definition of DFHEJOS from the listener regions, where it's not required.)
6. To balance client connections across the listener regions, use connection optimization by means of dynamic DNS registration. How to set this up is described in "Domain Name System (DNS) connection optimization" on page 176.
7. Arrange for method requests for enterprise beans to be dynamically routed across the AORs. You can use either of the following:
   a. CICSPlex SM. How to use CICSPlex SM to route method requests for enterprise beans is described in Chapter 28, "CICSPlex SM with enterprise beans," on page 363.
   b. A customized version of the CICS distributed routing program, DFHDSRP. How to write a distributed routing program to route method requests for enterprise beans and CORBA stateless objects is described in the *CICS Customization Guide*.

   On the DSRTPGM system initialization parameter for the listener regions, specify the name of the distributed routing program to be used. If you're using CICSPlex SM, specify the name of the CICSPlex SM routing program, EYU9XLOP. Otherwise, specify the name of your customized routing program. For information about the DSRTPGM system initialization parameter, see the *CICS System Definition Guide*.

   Remember:

   a. To route method requests for enterprise beans dynamically, the TRANSACTION definition for the transaction named on your REQUESTMODEL definitions must specify DYNAMIC(YES). The default transaction named on REQUESTMODEL definitions, CIRP, is defined as DYNAMIC(NO). We recommend that you take a copy of the TRANSACTION definition for CIRP, change the DYNAMIC setting, and save the definition under a new name. Then name your new transaction on REQUESTMODEL definitions. (The easiest way to create REQUESTMODEL definitions is to use the CREA transaction after you have deployed your enterprise beans into CICS.)
   b. The "common" transaction definition specified on the DTRTRAN system initialization parameter, and used for terminal-initiated transaction routing requests if no TRANSACTION definition is found, is never associated with method requests for enterprise beans. If, on the listener region, there is no REQUESTMODEL definition that matches the request, the request runs under the CIRP transaction (which specifies DYNAMIC(NO).
   c. In Figure 20 on page 249, the REQUESTMODEL definitions in the AORs are required for outbound requests to local objects. If a CORBA stateless object or enterprise bean makes a call to another object, and that object is available on the local AOR, CICS does not send the request to a listener region. Instead, it either runs the called method in the current task ("tight loopback") or starts another request processor in the local AOR ("normal loopback"). Where normal loopback is used, it's preferable that the new request processor task should use the same REQUESTMODEL as that used for the call to the first object—otherwise, unpredictable results may occur. If your CORBA stateless objects and enterprise beans make no outbound calls, the REQUESTMODELs on the AOR are not strictly required.

Results

Important: These instructions have shown you how to set up a multi-region EJB server in which each region contains a single CorbaServer execution environment. In production regions that support multiple applications, each of which uses its own set of enterprise beans, you may require multiple CorbaServers. To facilitate maintenance in production regions, you should follow the guidelines on how to allocate beans to CorbaServers and transaction IDs in Chapter 24, "Updating enterprise beans in a production region," on page 307.

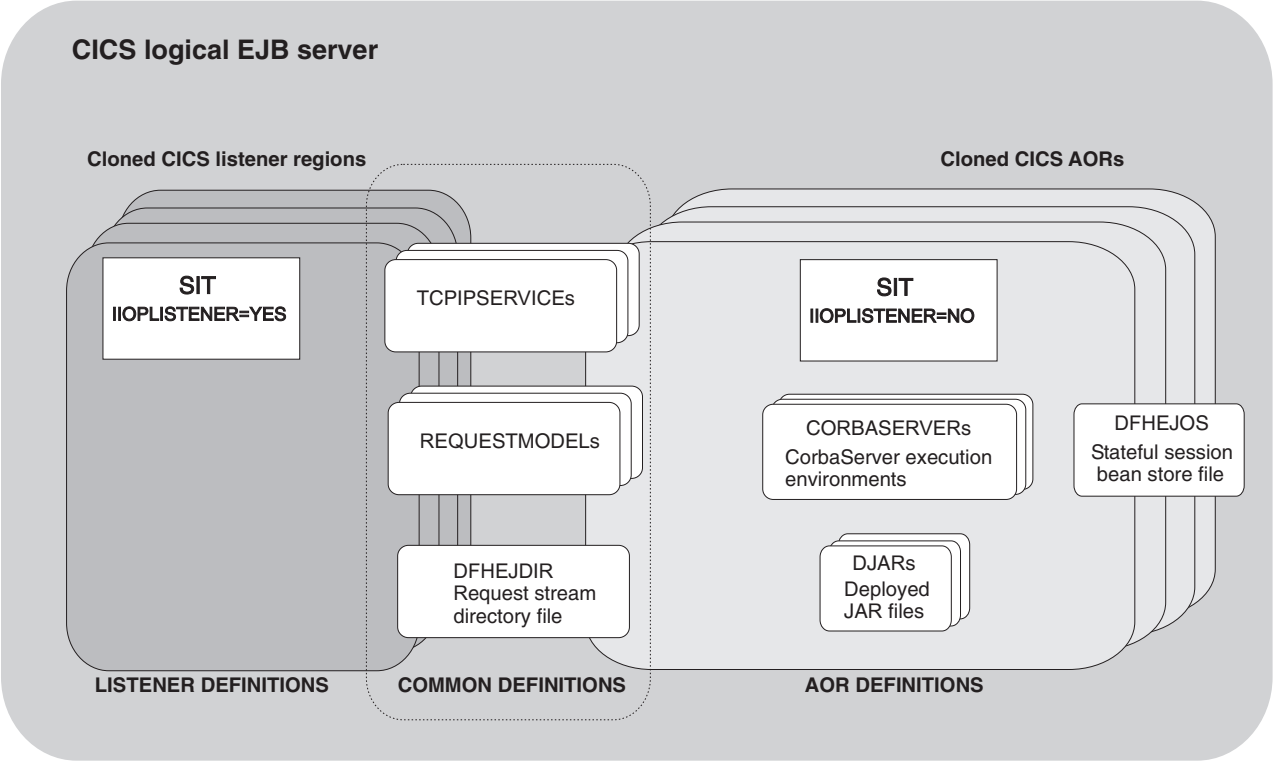

Figure 20. Resource definitions in a multi-region CICS EJB server. The picture shows which definitions are required in the listener regions, which in the AORs, and which in both.

Upgrading an EJB server to CICS Transaction Server for z/OS, Version 4 Release 1

This section tells you how to upgrade a back-level EJB server to CICS TS for z/OS, Version 4.1.

Upgrading a single-region CICS EJB/CORBA server

Perform these steps to upgrade a single-region CICS EJB/CORBA server to CICS Transaction Server for z/OS, Version 4 Release 1.

Procedure

1. Quiesce the workload.
2. Shut down the region.
3. Upgrade the region to CICS Transaction Server for z/OS, Version 4 Release 1, following the standard upgrade procedures described in the Upgrading information set for the release from which you are upgrading.

4. Review "Upgrade tips" on page 254, which describes some of the changes in EJB/CORBA support between different releases of CICS. You can also refer to "Setting up a single-region EJB server" on page 239, which describes in detail how to set up a single-region EJB server in CICS TS for z/OS, Version 4.1.
5. Restart the region.
6. Republish the Interoperable Object References (IORs) for all the enterprise beans and stateless CORBA objects processed by the server by issuing a **PERFORM CORBASERVER**(*CorbaServer_name*) PUBLISH command. You can issue this command using **EXEC CICS**, CEMT, the Resource Manager for enterprise beans, or from a CICSPlex SM WUI view. Remember to issue a separate command for each CorbaServer in the region.

Upgrading a multi-region CICS EJB/CORBA server

To upgrade a multi-region CICS EJB/CORBA server to CICS Transaction Server for z/OS, Version 4 Release 1, you can use any of these methods.

About this task

1. **Shut down the server, upgrade all the regions, and restart the server.**

   This approach is very similar to that described in "Upgrading a single-region CICS EJB/CORBA server" on page 249, except that:

   a. You must upgrade all the regions to CICS Transaction Server for z/OS, Version 4 Release 1 before restarting the server. Again, follow the standard upgrade procedures described in the Upgrading information set for the release from which you are upgrading.
   b. You should refer to "Setting up a multi-region EJB server" on page 246, which describes in detail how to set up a multi-region EJB server in CICS TS for z/OS, Version 4.1.
   c. To republish the IORs of enterprise beans and stateless CORBA objects, issue a PERFORM CORBASERVER(*CorbaServer_name*) PUBLISH command on at least one of the AORs. Remember to issue a separate command for each CorbaServer in the AOR.

   The advantage of this approach is its relative simplicity, compared to solutions 2 and 3. Its main disadvantage is that the server's applications are unavailable during the upgrade process.
2. **Create a separate, CICS TS for z/OS, Version 4.1, logical server and gradually move applications from the old, back-level, server to the new one.**

   The advantages of this approach are:

   a. Applications are kept available throughout the upgrade process.
   b. You can start with a minimal CICS TS for z/OS, Version 4.1 server, perhaps consisting of just two regions—one listener and one AOR. As more applications are moved, you can expand the CICS TS for z/OS, Version 4.1 server and simultaneously reduce the number of regions in the back-level server, thereby conserving resources.
   c. It is probably easier to implement than solution 3.

   To set up a new CICS TS for z/OS, Version 4.1 multi-region EJB server, follow all the steps in "Setting up a single-region EJB server" on page 239 and "Setting up a multi-region EJB server" on page 246.
3. **Perform a "rolling upgrade"**.

   In a "rolling upgrade", one region at a time is upgraded from the previous to the current level of CICS, while keeping the server operational.

   The advantages of this approach are:

a. Applications are kept available throughout the upgrade process.
b. Unlike solution 2, at no stage is it necessary to set up additional CICS regions.

This method is described in detail in "Performing a "rolling upgrade"."

Performing a "rolling upgrade"

The mixed level of operation described in this section, in which different CICS regions in the same logical server are at different levels of CICS, is intended to be used only for rolling upgrades.

Important

It should not be used permanently, because it increases the risk of failure in some interoperability scenarios. The normal, recommended, mode of operation is that all the regions in a logical sever should be at the same level of CICS and Java.

This section describes how to perform a "rolling upgrade" of a multi-region CICS EJB/CORBA server to CICS Transaction Server for z/OS, Version 4 Release 1. The process consists of the following steps:

1. Checking that your logical server meets the criteria for a "rolling upgrade". See "Requirement."
2. "Preliminary steps"
3. "Upgrading the listener regions" on page 252
4. "Upgrading the AORs" on page 252
5. "Tidying up" on page 254

Requirement:

Your server must consist of separate listener and application-owning regions. This is because the upgrade process requires all of the listener regions to be updated before any of the application-owning regions (AORs).

If you run composite listener-AORs, which act both as request receivers and request processors, this cannot be done. And if you don't upgrade all the listeners before any of the AORs, your IIOP client applications may receive transient failures during the migration window, depending on the CICS version of the listener region that receives the request.

Preliminary steps:
About this task

1. Review "Upgrade tips" on page 254.
2. If you are upgrading from CICS TS **2.2**, ensure that APAR PQ 79565 is installed in all your CICS TS 2.2 regions. This APAR improves CICS TS 2.2 diagnostics, should CICS TS for z/OS, Version 4.1 workload arrive at a CICS TS 2.2 region. It also allows a CICS TS 2.2 request processor (AOR) to receive work from a CICS TS for z/OS, Version 4.1 request receiver (listener).
3. Set the AUTOPUBLISH option on all your CORBASERVER definitions to `NO`. Setting a CorbaServer to autopublish IORs into the JNDI namespaces could disrupt the upgrade process.
4. If you use a distributed routing program to balance method requests for enterprise beans and CORBA stateless objects across the AORs of your logical server, customize your routing program to use the DYRLEVEL parameter. DYRLEVEL is an aid to upgrade. It contains the level of CICS required in the target AOR to successfully process the routed request. (Note that this is the

specific, *not* the minimum, level of CICS required to process the request successfully.) In a mixed-level logical server, when your routing program is invoked for route selection (or route selection error), it can use the value of DYRLEVEL to determine whether to route the request to a back-level or CICS TS for z/OS, Version 4.1 AOR.

For details of how to use DYRLEVEL, and definitive information about writing a distributed routing program, see the *CICS Customization Guide*.

Install your customized program on *all* the regions (both listeners and AORs) of the EJB server.

If you use CICSPlex SM to workload-balance method requests you can skip this step. The CICSPlex SM routing program supplied with CICS Transaction Server for z/OS, Version 4 Release 1 checks the DYRLEVEL field and routes requests accordingly.

Upgrading the listener regions:

Perform these steps to upgrade a listener region.

About this task

1. Quiesce a listener region and bring it down.
2. Upgrade this single listener region to CICS Transaction Server for z/OS, Version 4 Release 1, following the standard upgrade procedures described in the Upgrading information set for the release from which you are upgrading.

   Important:

   a. If you upgrade a CSD from CICS TS **2.2** to CICS TS for z/OS, Version 4.1 level, if it is shared by any CICS TS 2.2 regions other than that being upgraded, include the DFHCOMPA resource group (supplied with CICS TS for z/OS, Version 4.1) in the startup group list of these regions. DFHCOMPA is a compatibility group that provides a definition of DFJIIRP, the default request processor program, that can be used by a CICS TS 2.2 region when sharing a CICS TS for z/OS, Version 4.1 CSD.

      This step is necessary because, in CICS TS for z/OS, Version 4.1, the JVM profile used by DFJIIRP is DFHJVMCD. In CICS TS 2.2, it is DFHJVMPR.

   b. At this stage, don't enable any new, CICS TS for z/OS, Version 4.1-specific, options on resource definitions, because they won't be understood by the back-level AORs. Use of these new features must wait until the whole logical server—both listener regions and AORs—has been upgraded.

   For definitive information about setting up a listener region in CICS TS for z/OS, Version 4.1, refer to Chapter 16, “Configuring CICS for IIOP,” on page 181.
3. Bring the listener back up. This region is now at the newer version of CICS but may continue to participate as part of the back-level logical server.
4. Repeat steps 1 through 3 for all of the listener regions in the logical server.

Upgrading the AORs:

To upgrade an AOR for enterprise beans, perform these steps.

About this task

1. Quiesce an AOR and bring it down.

2. Upgrade this single AOR to CICS Transaction Server for z/OS, Version 4 Release 1, following the standard upgrade procedures described in the Upgrading information set for the release from which you are upgrading.

   If you are upgrading from CICS TS 2.2, part of this will involve updating the JVM profile used by the CorbaServers. Note the changes to JVM profiles and property files that were introduced in CICS TS 2.3, as described in "Upgrade tips" on page 254.

   Important:

   a. If you upgrade a CSD from CICS TS **2.2** to CICS TS for z/OS, Version 4.1 level, if it is shared by any CICS TS 2.2 regions other than that being upgraded, include the DFHCOMPA resource group (supplied with CICS TS for z/OS, Version 4.1) in the startup group list of these regions.
   b. At this stage, don't enable any new, CICS TS for z/OS, Version 4.1-specific, options on resource definitions.
3. Bring the AOR back up again.
4. Ensure that all TCPIPSERVICEs are open both in this AOR and in the listener regions.
5. Use the CEMT PERFORM DJAR PUBLISH command to re-publish the IORs of one or more enterprise beans in CICS TS for z/OS, Version 4.1 format. For each CorbaServer, select one or more deployed JAR files to re-publish. When choosing deployed JAR files to re-publish, bear the following in mind:
   - Try to pick DJARs whose entire workload can be processed by a single region.
   - Wherever possible, all the beans used by an application should be upgraded at the same time. For example, if bean A is known to call bean B the two beans should be upgraded together. If this is not possible, bean A should be upgraded first.

     This is particularly important if you are upgrading from CICS TS 2.2 and the beans are installed in the same CorbaServer but in different AORs that are at different levels of CICS. This is because a CICS TS 2.2 region cannot do a JNDI look up of an object in a CICS TS for z/OS, Version 4.1 region if both objects are in the same CorbaServer. For example, bean A in CorbaServer EJB1 in a CICS TS 2.2 AOR cannot look up bean B in CorbaServer EJB1 in a CICS TS for z/OS, Version 4.1 AOR.

     Note: If A and B are installed in different CorbaServers, or in AORs that are at the same level of CICS, they can be upgraded separately.

   Re-publish the selected DJARs to the JNDI namespace, in the same location as that used by the back-level AORs.

   At this point :
   - This AOR is ready to accept workload.
   - The logical server contains a pool of back-level AORs and a pool (currently containing only one region) of CICS TS for z/OS, Version 4.1 AORs.
   - Any clients that look up the IOR of a re-published bean in the namespace get the new IOR in CICS TS for z/OS, Version 4.1 format. Your customized routing program or CICSPlex SM directs such requests to the CICS TS for z/OS, Version 4.1 AOR.
   - Any clients that have a stale, cached, IOR for a bean that's been re-published are still able to use the bean. Your customized routing program or CICSPlex SM directs such old-format requests to one of the back-level AORs.

Note: Many application servers cache the results of JNDI lookups locally to increase performance, so you may find that these caches have to be purged before the new IORs are used. Over a period of time, requests for re-published enterprise beans should move gradually from the pool of back-level AORs to the pool of CICS TS for z/OS, Version 4.1 AORs.

6. Repeat steps 1 through 5 for all of the AORs in the logical server. As each AOR is upgraded:
   - Re-publish a different set of enterprise beans, so that gradually more and more beans are supported by the pool of CICS TS for z/OS, Version 4.1 regions.
   - It becomes less important, when selecting deployed JAR files to re-publish, to choose those whose entire workload can be processed by a single region—because there are more AORs in the CICS TS for z/OS, Version 4.1 pool.

   Eventually, all the AORs will be running CICS TS for z/OS, Version 4.1 and processing 100% of the workload.

Tidying up:

To complete rolling upgrade, you must perform these final tasks.

About this task

Procedure

1. If required, reset the AUTOPUBLISH option on your CORBASERVER definitions to `YES`.
2. Enable any CICS TS for z/OS, Version 4.1-specific resource definition options that you want to use.

Results

Upgrade tips

This section briefly lists some general tips, as a reminder of things to be aware of when upgrading an EJB server to CICS TS for z/OS, Version 4.1.

All these changes are described in detail in Chapter 11, “Setting up Java support,” on page 65.

1. JVM profiles are stored in the z/OS UNIX directory pointed to by the **JVMPROFILEDIR** system initialization parameter.
2. The default JVM profile used by CorbaServers is DFHJVMCD.
3. Don't enable any new, CICS TS for z/OS, Version 4.1-specific, attributes on resource definitions during the “rolling upgrade” process. Use of these new features must wait until the whole logical server, both listener regions and AORs, has been upgraded.
4. From a CICS TS for z/OS, Version 4.1 AOR, you can re-publish a deployed JAR file that has previously been published from an earlier release of CICS without first retracting it. The IORs of the beans are updated to the format for the new release. **However, you cannot do the reverse**. From an earlier release of CICS, before re-publishing a deployed JAR file that has previously been published from a CICS TS for z/OS, Version 4.1 AOR you must first retract it; furthermore, because earlier CICS releases do not understand the format of CICS TS for z/OS, Version 4.1 IORs, *you must retract it from a CICS TS for z/OS, Version 4.1 AOR*.

Bear this in mind if, for any reason, you need to back out the upgrade of one or more AORs. If you ever need to revert the IORs of enterprise beans that have been published from a CICS TS for z/OS, Version 4.1 AOR to an earlier level of CICS (so that they can be routed to a back-level AOR once more) you must:

a. Retract the deployed JAR file from a CICS TS for z/OS, Version 4.1 AOR
b. Publish the deployed JAR file from a back-level AOR

Trying to re-publish the beans without retracting them first, or trying to retract them from the wrong level of CICS, results in an `InvalidUserKeyException: Bad version number` exception.

Potential problems

1. After the EJB server has been upgraded to CICS TS for z/OS, Version 4.1, some clients may have stale, cached, IORs that point to the old server. This is because some application servers cache the results of JNDI lookups locally to increase performance. You may find that these caches have to be purged before the new IORs are used.
2. CICS TS 2.3 and later, including CICS TS for z/OS, Version 4.1, support GIOP 1.2, whereas CICS TS 2.2 supports only GIOP 1.1. If a GIOP 1.2 message is received in a CICS TS 2.2 region it will be rejected. Under normal conditions this should never happen, because the maximum version of GIOP supported by CICS is stored in the IORs that CICS publishes. If a client knows that a given server only supports GIOP 1.1, it will never attempt to use anything more recent when communicating with that server. This means that CICS TS for z/OS, Version 4.1 can send GIOP messages to CICS TS 2.2.

   The problem will only occur if the client thinks it is talking to CICS TS for z/OS, Version 4.1 (or CICS TS 3.1 or CICS TS 2.3) but its message is routed to a CICS TS 2.2 region. This will only happen if CICS TS 2.2 and CICS TS for z/OS, Version 4.1 regions are set up as sibling request processors (AORs) in the same logical server. (This is one reason why mixed-level logical servers are not recommended in CICS.) During a “rolling upgrade”, the logical server does, of course, contain mixed-level request processors. However, if you follow the steps in “Performing a “rolling upgrade”” on page 251, the problem (of a GIOP 1.2 message being received in a CICS TS 2.2 region) will not occur.
3. CICS TS 2.3 and later, including CICS TS for z/OS, Version 4.1, use a different format of IOR from CICS TS 2.2. If a GIOP 1.1 message intended for CICS TS for z/OS, Version 4.1 is routed to a CICS TS 2.2 region, the CICS TS 2.2 region will reject the request due to a unknown IOR format being in use. If all the regions in an EJB/CORBA server are at the same level of CICS and Java, this error cannot occur.

   During a “rolling upgrade”, the logical server does, of course, contain mixed-level regions. However, if you follow the steps in “Performing a “rolling upgrade”” on page 251, this problem will not occur.

Chapter 20. Using the EJB IVP

The EJB Installation Verification Program (IVP) is a small application that CICS installers can use to verify the CICS EJB environment.

The EJB IVP uses a client program that does not require the use of a Web server. The IVP consists of:

- A line-mode client program that runs in UNIX System Services on z/OS
- A stateless session enterprise bean running on the CICS EJB server

The IVP tests:

- The CICS JVM (including its reusability).
- Optionally, your "real", enterprise-level, name server. (By default, the IVP uses the lightweight `tnameserv` COS Naming Server supplied with Java.)
- The EJB server's ability to run a basic enterprise bean.
- z/OS UNIX settings (including file access permissions).

Once configured, the client:

1. Performs a JNDI lookup to find the published reference to a specific enterprise bean in the JNDI namespace
2. Creates a new instance of the enterprise bean in CICS
3. Calls a remote method on the bean-instance

Prerequisites for the EJB IVP

Before running the EJB IVP, you will need these resources.

- A UNIX System Services userid and file editor.
- A CICS EJB server. The way to set one up is described in "Setting up a single-region EJB server" on page 239.
- A name server that supports the Java Naming and Directory Interface (JNDI) Version 1.2 or later. The way to set up an enterprise-quality name server is described in "Enabling JNDI references" on page 183. Alternatively, you can use the lightweight `tnameserv` COS Naming Server supplied with Java.

Note:

1. These prerequisites assume you are testing a single-region CICS EJB server.
2. To run the IVP, you need only complete the steps in "Before running the EJB IVP" on page 239. You may or may not have completed the steps in "After running the EJB IVP—optional steps" on page 245.
3. Before starting, check that the storage size for your TSO/E session is at least 6000KB. To increase the storage size, at the standard TSO/E logon screen change the value in the SIZE field.

Installing the EJB IVP

To install the EJB you must set up z/OS UNIX, and CICS. On z/OS UNIX System Services you must configure the client.

z/OS UNIX setup for the EJB IVP

The IVP uses the same CICS enterprise bean as the EJB "Hello World" sample application.

The sample is described in "The EJB "Hello World" sample application" on page 263. Thus, on z/OS UNIX, you must copy the `HelloWorldEJB.jar` deployed JAR file from the EJB samples directory to the deployed JAR file ("pickup") directory that you created in "Before running the EJB IVP" on page 239.

Note: Both the source and executable code of the enterprise bean is in the `HelloWorldEJB.jar` file.

The samples directory is: `/usr/lpp/cicsts/cicsts41/samples/ejb/helloworld`, where `/usr/lpp/cicsts/cicsts41` is the install directory for CICS files on z/OS UNIX.

Remember that z/OS UNIX names are case-sensitive.

CICS setup

Before running the EJB IVP, you must perform these CICS setup tasks.

About this task

1. If EJB role-based security is active in your CICS region, you must turn it off before running the IVP. That is, if both the SEC and XEJB system initialization parameters currently specify 'YES', you must set XEJB to 'NO' and restart CICS.
2. The CICS-supplied sample resource group, DFH$EJB, contains TCPIPSERVICE and CORBASERVER definitions suitable for running the IVP. You must change some of the attributes of these resource definitions to suit your own environment, and install the changed definitions into CICS. You should already have done this, as part of the task of setting up your EJB server. If you have not, follow the step-by-step instructions in "Actions required on CICS" on page 241.
3. Issue a `CEMT PERFORM CORBASERVER(EJB1) SCAN` command.

   CICS:

   a. Scans the pickup directory that you specified on the DJARDIR option of the CORBASERVER definition
   b. Copies the `HelloWorldEJB.jar` deployed JAR file that it finds in the pickup directory to its shelf directory
   c. Dynamically creates and installs a DJAR definition for `HelloWorldEJB.jar`
   d. Because the CORBASERVER definition specifies AUTOPUBLISH(YES), publishes the enterprise bean contained in `HelloWorldEJB.jar` to the JNDI namespace.
4. If you have not already done so while setting up your CorbaServer, set the status of the TCPIPSERVICE to OPEN:

   ```
   CEMT SET TCPIPSERVICE(EJBTCP1) OPEN
   ```

   On the CICS Console, you should see, among others, messages similar to the following:

   ```
   DFHEJ5024 Scan commencing for CorbaServer EJB1, directory being scanned is
             DJARDIR_name.
   DFHEJ5030 New DJar HelloWorldEJB is being created during a scan against
             CorbaServer EJB1.
   DFHEJ0901 DJar HelloWorldEJB within CorbaServer EJB1 has been created.
   DFHEJ5025 Scan completed for CorbaServer EJB1, 1 DJars created, 0 DJars updated.
   DFHEJ5032 DJar HelloWorldEJB is having its contents automatically published to
             the namespace.
   DFHEJ5009 Published bean HelloWorld to JNDI server
             iiop://nameserver.location.company.com:2809 at location samples.
   DFHEJ1540 DJar HelloWorldEJB and the Beans it contains are now accessible.
   ```

where:

- **DJARDIR_name** is the name of your CorbaServer's deployed JAR file ("pickup") directory.
- **iiop://nameserver.location.company.com:2809** is the URL and port number of your name server. In this example, a COS Naming Server is used.

Configuring the client

The source code of the client application is in the `HelloWorldCLI.jar` file.

About this task

On z/OS UNIX System Services, you must:

1. Copy the `runEJBIVP` script to a working directory. The original `runEJBIVP` script is located, with the IVP sample, in the following directory:

   ```
   /usr/lpp/cicsts/cicsts41/samples/ejb/helloworld
   ```

   where `cicsts41` is the install directory for CICS files on z/OS UNIX.

2. Edit your copy of `runEJBIVP` script as follows. This is necessary so that the client can locate the published enterprise bean in the JNDI namespace. (A typical client will not have access to the CICS JVM profile.)

   a. Modify the JAVA_HOME variable to your IBM SDK 6.0 installation directory, as indicated by the comments in the script. The line to be changed is:

      ```
      JAVA_HOME=/usr/lpp/<Java SDK java installation directory>/J6.0
      ```

   b. Modify the CICS_HOME variable to your install directory for CICS files on z/OS UNIX, as indicated by the comments in the script. The line to be changed is:

      ```
      CICS_HOME=/usr/lpp/cicsts/<CICS installation directory>
      ```

   c. Modify the JNDI_PROVIDER_URL variable to the URL and port number of your name server, as indicated by the comments in the script. The line to be changed is:

      ```
      JNDI_PROVIDER_URL=iiop://nameserver.location.company.com:2809
      ```

      The above line assumes that you are using a COS name server, such as `tnameserv`, the lightweight COS Naming Directory Server supplied with Java 1.3 and later, and that it is configured to listen on port 2809.

      If, for example, you are using a COS name server configured to listen on port 900, you might specify:

      ```
      JNDI_PROVIDER_URL=iiop://nameserver.location.company.com:900
      ```

      If you are using the `tnameserv` name server, configured to listen on port 2809, on a workstation named `myworkstation.acme.com` you should specify:

      ```
      JNDI_PROVIDER_URL=iiop://myworkstation.acme.com:2809
      ```

      To start the `tnameserv` program, type the following command at the workstation command prompt:

      ```
      tnameserv -ORBInitialPort 2809
      ```

      If you are using the COS Naming Directory Server supplied with WebSphere Application Server Version 5 or later, configured to listen on port 2809, you should specify:

      ```
      JNDI_PROVIDER_URL=iiop://nameserver.location.company.com:2809/domain/legacyRoot
      ```

      If you are using an LDAP name server, the protocol should be `ldap` rather than `iiop`; the port number should be 389. For example:

```
JNDI_PROVIDER_URL=ldap://nameserver.location.company.com:389
```

d. If you are using an LDAP name server, modify the LDAP_CONTAINERDN and LDAP_NODEROOTDN variables, as indicated by the comments in the script.

   If you are using a COS naming server, these properties are ignored.

e. If necessary, modify the INITIAL_CONTEXT_FACTORY variable as indicated by the comments in the script. Usually, you can leave this property to default. However, some JNDI service providers cannot be accessed using the default initial context factory. For example, if you are using WebSphere Application Server as your JNDI provider you should set this variable to `com.ibm.websphere.naming.WsnInitialContextFactory`.

f. If you have set up your CorbaServer and installed the IVP in the way suggested, the CORBASERVER_JNDI_PREFIX and BEAN_NAME variables will already be set to the correct values. See the comments in the script.

Running the EJB IVP

To run the EJB Installation Verification Program, you must perform these steps.

About this task

Procedure

1. Check that the name server is running.

   a. To start `tnameserv` on the local host, enter the following command at the UNIX System Services or Windows NT command prompt:

      ```
      tnameserv -ORBInitialPort 2809
      ```

      This causes `tnameserv` to listen for connections on TCP/IP port 2809.

2. Run the IVP client program from your UNIX System Services working directory by typing `./runEJBIVP`. On your UNIX System Services terminal, you should see messages similar to the following:

   ```
   CICS EJB IVP: Querying the Java SDK level
   java version "1.6.0"
   Java(TM) 2 Runtime Environment, Standard Edition (build build pmz31dev-20080224 (SR7))
   IBM J9 VM (build 2.3, J2RE 1.5.0 IBM J9 2.3 z/OS s390-31 j9vmmz3123-20080223 (JIT enabled)
   J9VM - 20080222_17484_bHdSMr
   JIT  - 20080130_0718ifx1_r8
   GC   - 200802_08)
   JCL  - 20080224
   CICS EJB IVP: Starting the EJB client program
   HelloWorld client program started
   Performing JNDI lookup using CosNaming
   Testing the following location: samples/HelloWorld
   Located home interface for HelloWorld bean
   You said: Hello from CICS EJB IVP client
   HelloWorld client program ended
   CICS EJB IVP: Completed successfully
   ```

   Note:

   a. In this example, a COS Naming Server has been used. If you use an LDAP name server, similar messages are produced.

   b. If you get a `javax.naming.CommunicationException`, it may be because the MVS hostname is incorrect in your `tcpip.data` file. You may be able to fix the problem by adding an entry for the MVS system to your `/etc/hosts` file. For guidance, see the MVS manuals.

   In your JVM stdout file, you should see the following message:

```
CICS EJB hello world sample called with string: Hello from CICS EJB IVP client
```

If you re-run the client, you will probably notice a performance improvement. This is because the JVM should be reused.

3. After running the IVP, you must perform the following steps.
   a. Discard the resource definitions that you created in `mygroup`.
   b. If you turned off EJB role-based security before running the IVP, turn it back on. To do this, restart CICS with the XEJB system initialization parameter set to 'YES'.

Chapter 21. Running the sample EJB applications

The sample EJB applications require a CICS EJB server.

Important

You must configure CICS, as described in Chapter 19, “Setting up an EJB server,” on page 239, before attempting to install the samples.

CICS supplies the following sample EJB applications:

The EJB Installation Verification Program (IVP)
A simple application that you can use to test your CICS EJB environment and name server. A Web server is not required. See Chapter 20, “Using the EJB IVP,” on page 257.

The EJB "Hello World" sample
A simple application that you can use to test your EJB environment, including CICS, your name server, and your Web server. See “The EJB “Hello World” sample application.”

The EJB Bank Account sample
A more complex application that demonstrates how you can use enterprise beans to make existing, CICS-controlled, information available to Web users. See “The EJB Bank Account sample application” on page 270.

The EJB “Hello World” sample application

“Hello World” is a simple application that you can use to test your EJB environment, including CICS, your name server, and your Web server.

What the EJB “Hello World” sample does

The sample application requests input, appends the input to a standard message, and displays the resulting string.

The sample consists of:
- An HTML form.
- A Java servlet, plus JavaServer Pages (JSPs), running in a J2EE-compliant Web application server.
- An enterprise bean running on a CICS EJB server.

The sample works like this:
1. The user starts the application from a Web browser. A form is displayed.
2. The form asks the user to input a phrase. When the user presses the SUBMIT button, the servlet is invoked.
3. The servlet:
   a. Looks up a reference to the enterprise bean in the JNDI namespace
   b. Creates a new remote instance of the enterprise bean in CICS
   c. Invokes a method on the bean-instance, passing as input the phrase input by the user
4. The enterprise bean appends the user's phrase to the string “You said ” and returns the result to the servlet.

5. The servlet uses a JavaServer Page to display the result on the user's browser.

Figure 21 shows the components of the sample application.

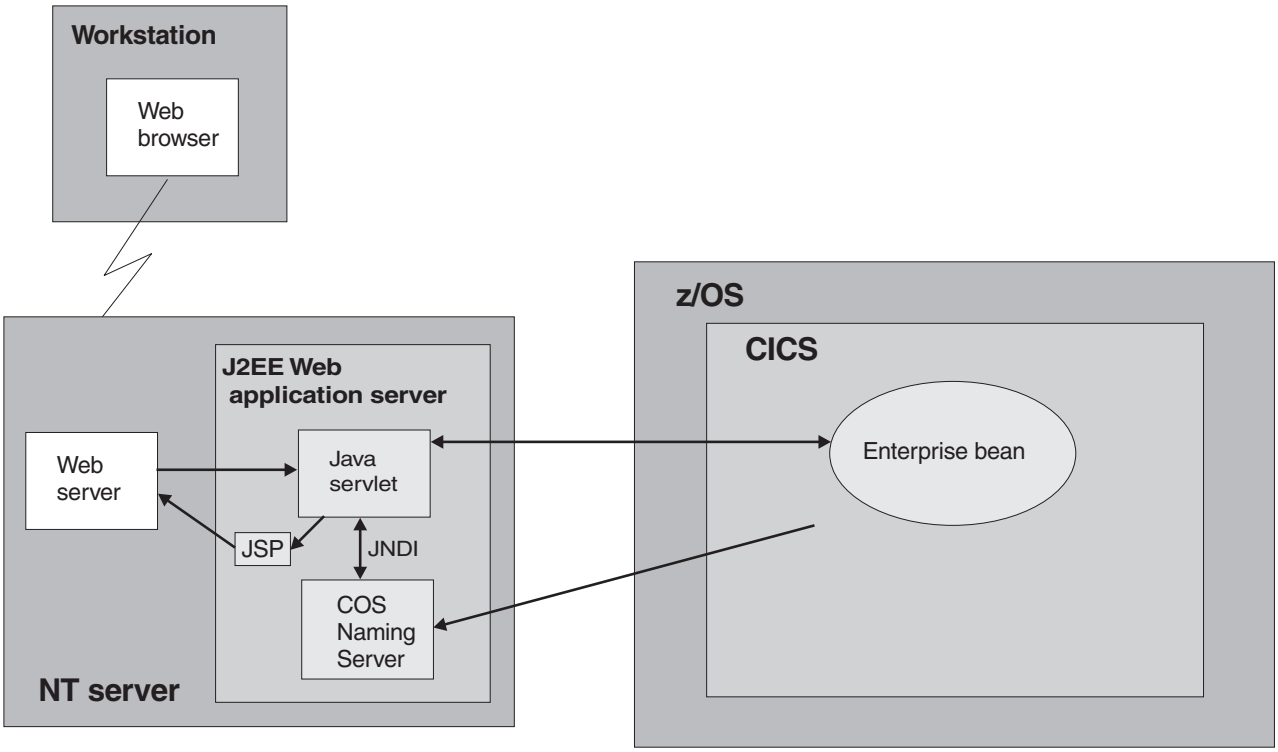

Figure 21. Overview of the EJB "Hello World" sample application. The main elements of the sample are a Java servlet and an enterprise bean. In this example, the servlet is running in a Web application server on a Windows NT server; a COS Naming Server is used. Other configurations are possible. For example, an LDAP name server could have been used; or the COS Naming Server might not have been hosted in the same application server as the servlet.

Prerequisites for the EJB "Hello World" sample

You need these resources to run the EJB "Hello World" sample.

- A CICS EJB server. The way to set one up is described in Chapter 19, "Setting up an EJB server," on page 239.
- A Web application server that supports J2EE Version 1.2.1 or later. If you are using WebSphere Application Server, note that the sample requires WebSphere Application Server Version 4 or later.
- A name server that supports the Java Naming and Directory Interface (JNDI) Version 1.2 or later. The way to set one up is described in "Actions required on z/OS or Windows NT" on page 240.

Supplied components of the EJB "Hello World" sample

These files are supplied with the EJB "Hello World" sample.

Table 10. Supplied components of the EJB "Hello World" sample

Filename	Type	Default location	Comments
`CICSHelloWorld.ear`	EAR file	z/OS UNIXsamples directory: see Note.	The Web components of the sample application—Java servlet classes and source files; HTML and JSPs.
DFH$EJB	Resource definition group	CSD	Contains the CICS resource definitions required by the sample application.

Table 10. Supplied components of the EJB "Hello World" sample (continued)

Filename	Type	Default location	Comments
`HelloWorldCLI.jar`	JAR file	z/OS UNIX samples directory: see Note.	Client EJB stubs required by the servlet.
`HelloWorldEJB.jar`	Deployed JAR file	z/OS UNIX samples directory: see Note.	Java classes, source files, deployment descriptor, plus supporting classes for the CICS enterprise bean. Doesn't need to be unpacked unless you want to modify the source code.
`readme.txt`	Text file	z/OS UNIX samples directory: see Note.	Contains: 1. Step-by-step instructions for installing the Web components of the EJB "Hello World" sample on WebSphere Application Server. 2. Hints, tips, and debugging information.
Note: The default z/OS UNIX samples directory is `/usr/lpp/cicsts/cicsts41/samples/ejb/helloworld` where `/usr/lpp/cicsts/cicsts41` is the install directory for CICS files on z/OS UNIX.			

Installing the EJB "Hello World" sample

You must set up these resources to install the EJB "Hello World" sample.

1. z/OS UNIX. If you've previously run the EJB IVP, you will have performed this action already.
2. CICS. If you've previously run the EJB IVP, you will have performed these actions already.
3. The Web application server.

z/OS UNIX setup for EJB "Hello World" sample

If necessary, on z/OS UNIX copy the `HelloWorldEJB.jar` deployed JAR file from the EJB samples directory to your CorbaServer's deployed JAR file ("pickup") directory.

Note:

1. You need to do this only if you haven't already installed the `HelloWorldEJB.jar` deployed JAR file while running the EJB IVP.
2. The deployed JAR file directory is the directory that you created in "Before running the EJB IVP" on page 239 and specified on the DJARDIR option of the CORBASERVER definition.
3. The samples directory is: `/usr/lpp/cicsts/cicsts41/samples/ejb/helloworld`, where `/usr/lpp/cicsts/cicsts41` is the install directory for CICS files on z/OS UNIX.
4. Remember that z/OS UNIX names are case-sensitive.
5. The `HelloWorldEJB.jar` file contains both the source and executable code for the enterprise bean.

CICS setup

About this task

1. If EJB role-based security is active in your CICS region, you must turn it off before running the EJB "Hello World" sample. That is, if both the SEC and XEJB system initialization parameters currently specify 'YES', you must set XEJB to 'NO' and restart CICS.

2. The CICS-supplied sample group, DFH$EJB, contains TCPIPSERVICE and CORBASERVER definitions suitable for running the EJB "HelloWorld" sample. You must change some of the attributes of these resource definitions to suit your own environment, and install the changed definitions into CICS. You should already have done this, as part of the task of setting up your EJB server. If you haven't, follow the step-by-step instructions in "Actions required on CICS" on page 241.

   Note: Group DFH$EJB does not contain a REQUESTMODEL definition, because it's not necessary to install one. The sample uses the default transaction ID, CIRP.

   a. If necessary, issue a `CEMT PERFORM CORBASERVER(EJB1) SCAN` command. (You need to do this only if you haven't already installed the `HelloWorldEJB.jar` deployed JAR file while running the EJB IVP.) CICS:
      1) Scans the pickup directory
      2) Copies the `HelloWorldEJB.jar` deployed JAR file that it finds in the pickup directory to its shelf directory
      3) Dynamically creates and installs a DJAR definition for `HelloWorldEJB.jar`
      4) Because the CORBASERVER definition specifies AUTOPUBLISH(YES), publishes the enterprise bean contained in `HelloWorldEJB.jar` to the JNDI namespace.
3. If you have not already done so, set the status of the TCPIPSERVICE to OPEN:

   ```
   CEMT SET TCPIPSERVICE(EJBTCP1) OPEN
   ```

   If you issued the `CEMT PERFORM CORBASERVER(EJB1) SCAN` command, on the CICS Console you should see, among others, messages similar to the following:

   ```
   DFHEJ5024 Scan commencing for CorbaServer EJB1, directory being scanned is
             DJARDIR_name.
   DFHEJ5030 New DJar HelloWorldEJB is being created during a scan against
             CorbaServer EJB1.
   DFHEJ0901 DJar HelloWorldEJB within CorbaServer EJB1 has been created.
   DFHEJ5025 Scan completed for CorbaServer EJB1, 1 DJars created, 0 DJars updated.
   DFHEJ5032 DJar HelloWorldEJB is having its contents automatically published to
             the namespace.
   DFHEJ5009 Published bean HelloWorld to JNDI server
             iiop://nameserver.location.company.com:900 at location samples.
   DFHEJ1540 DJar HelloWorldEJB and the Beans it contains are now accessible.
   ```

   where:
   - **DJARDIR_name** is the name of your CorbaServer's deployed JAR file ("pickup") directory.
   - **iiop://nameserver.location.company.com:900** is the URL and port number of your name server. In this example, a COS Naming Server is used.

Web application server setup

On the Web application server, you must install the Web components of the EJB "Hello World" sample application.

About this task

From the z/OS UNIX EJB samples directory, you need:

- `CICSHelloWorld.ear`. A J2EE enterprise archive (EAR) file, containing the Web components of the sample and the source code of the servlet and JSPs.
- `readme.txt`. A text file, containing:

1. Step-by-step instructions for installing the Web components of the sample on WebSphere Application Server.
2. Hints, tips, and debugging information.

Note: The default samples directory is
`/usr/lpp/cicsts/cicsts41/samples/ejb/helloworld`

where `/usr/lpp/cicsts/cicsts41` is the install directory for CICS files on z/OS UNIX.

Important: The rest of this section contains generic instructions for installing the Web components of the sample on a J2EE-compliant Web application server (which may or may not be WebSphere). It is suitable for experienced users. If your Web application server is WebSphere Application Server Version 4 or later *and* you are a novice user of that product, we recommend that you follow instead the detailed, WebSphere-specific instructions in the `readme.txt` file.

1. Install the Web components of the EJB "Hello World" sample (contained in `CICSHelloWorld.ear`) in your J2EE Web application server, following the vendor's guidelines for installing applications. In WebSphere Application Sever, for example, this involves using the administration console to:
   a. Install a new application
   b. Generate the updated Web server plugin
   c. Save the configuration

   Note: `CICSHelloWorld.ear` includes a default configuration for the EJB "Hello World" sample. To run the sample, it is not necessary to edit or add any configuration information.
2. Start the application using your Web application server's standard procedure.

Testing the EJB "Hello World" sample

You must perform these steps to test the application.

About this task

1. Ensure that all the following are running:
   - The Web server
   - The Web application server and the sample application
   - The name server
   - The CICS region
2. Start a Web browser and point it at the URL of your Web server, followed by "cicshello". For example:
   `http://myServer.ibm.com/cicshello`

   The opening screen shown in Figure 22 on page 268 appears.

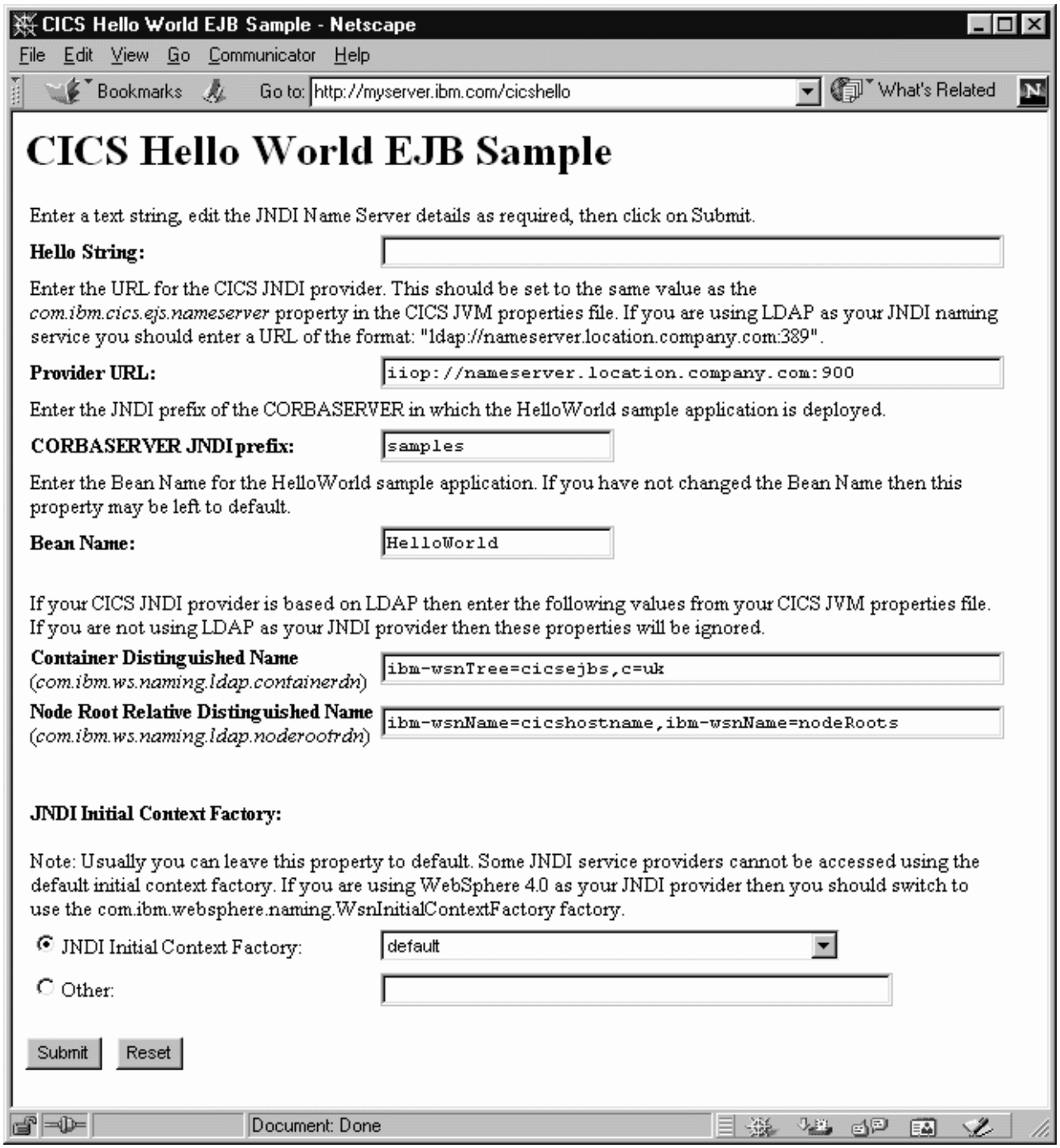

Figure 22. Opening screen of the EJB "Hello World" sample application

3. Enter a phrase in the `Hello String:` field.
4. Check that the `Provider URL:`, `CORBASERVER JNDI prefix:`, `Bean Name:`, `Container Distinguished Name:`, `Node Root Relative Distinguished Name:`, and `JNDI Initial Context Factory:` fields contain values that are valid for your installation. If they do not, overtype them as follows:

   `Provider URL:`
   Enter the URL and port number of the name server where the enterprise

bean is published. (These are specified by the **-Dcom.ibm.cics.ejs.nameserver** property in your JVM properties file.) For example:

- If you are using an LDAP name server with a URL of `myldapns.ibm.com` and a port number of 389, specify `"ldap://myldapns.ibm.com:389"`.
- If you are using a standard COS Naming Server with a URL of `mycosns.ibm.com` and a port number of 900, specify `"iiop://mycosns.ibm.com:900"`.
- If you are using the COS Naming Directory Server supplied with WebSphere Application Server Version 5 or later, with a URL of `mycosns.ibm.com` and a port number of 2809, specify:

  ```
  -Dcom.ibm.cics.ejs.nameserver=iiop://mycosns.ibm.com:2809/domain/legacyRoot
  ```

For detailed information about how to specify the location of the name server, see the description of the **-Dcom.ibm.cics.ejs.nameserver** property in "JVM system properties" on page 119.

CORBASERVER JNDI prefix:
Enter the JNDI prefix of your CorbaServer. If you are using the CORBASERVER definition supplied in DFH$EJB, you do not need to change the default value of "`samples`".

Bean name:
Enter the name of the enterprise bean used by the sample, as defined in the deployment descriptor in the supplied `HelloWorldEJB.jar` file. *Unless you have renamed the bean, you do not need to change the default value of "HelloWorld".*

Container Distinguished Name:
If you are using an LDAP name server, enter the distinguished name of the LDAP system namespace root, as supplied by your LDAP administrator. (The distinguished name of the LDAP system namespace root is specified by the **-Dcom.ibm.ws.naming.ldap.containerdn** property in your JVM properties file.) *If you are using a COS Naming Server, the value of this field is ignored.*

Node Root Relative Distinguished Name:
If you are using an LDAP name server, enter the distinguished name of the LDAP node root, as supplied by your LDAP administrator. (The distinguished name of the LDAP node root is specified by the **-Dcom.ibm.ws.naming.ldap.noderootrdn** property in your JVM properties file.) *If you are using a COS Naming Server, the value of this field is ignored.*

JNDI Initial Context Factory:
Select the appropriate JNDI initial context factory from the drop-down list. If your Web application server is WebSphere, the factory to use depends on:

- The version of WebSphere you're using
- The location of WebSphere—that is, whether it's on a distributed platform such as Windows NT or a host platform such as z/OS or OS/390
- The type of name server you're using—COS naming or LDAP

Table 11 on page 270 shows the correct initial context factory to specify, if your Web application server is WebSphere.

Table 11. Setting the initial context factory, according to the version and location of WebSphere and the type of name server

WebSphere Version	Location of Web application server	Name server type	Initial context factory to use
3.5	Distributed	COS	`com.ibm.ejs.ns.jndi.CNInitialContextFactory`
3.5	Distributed	LDAP	`com.ibm.jndi.LDAPCtxFactory`
3.5	z/OS or OS/390	COS	`com.sun.jndi.cosnaming.CNCtxFactory`
3.5	z/OS or OS/390	LDAP	`com.sun.jndi.ldap.LdapCtxFactory`
4 or later	Distributed	COS or LDAP	`com.ibm.websphere.naming.WsnInitialContextFactory`
4 or later	z/OS or OS/390	COS	`com.sun.jndi.cosnaming.CNCtxFactory`
4 or later	z/OS or OS/390	LDAP	`com.sun.jndi.ldap.LdapCtxFactory`

If your Web application server is not WebSphere, choose the appropriate value from the drop-down list.

Note: The drop-down list contains several initial context factory classes, plus a "default" list item. The sample application assigns the value of the default list item as follows:

a. If the `com.ibm.websphere.naming.WsnInitialContextFactory` class is found in the Java classpath, the sample makes it the default item. This class is a "wrapper" class that wraps both `com.ibm.ejs.ns.jndi.CNInitialContextFactory` and `com.ibm.jndi.LDAPCtxFactory`. The sample determines the correct base class to use by examining the type of name server that you've specified in the **Provider URL** field: if the specified protocol is "iiop", the sample uses `com.ibm.ejs.ns.jndi.CNInitialContextFactory`; if it's "ldap", the sample uses `com.ibm.jndi.LDAPCtxFactory`.

b. If the `com.ibm.websphere.naming.WsnInitialContextFactory` class is *not* found in the Java classpath, the sample determines the correct class to use by examining the type of name server that you've specified in the **Provider URL** field: if the specified protocol is "iiop", the sample uses `com.ibm.ejs.ns.jndi.CNInitialContextFactory`; if it's "ldap", the sample uses `com.ibm.jndi.LDAPCtxFactory`.

If none of the values in the drop-down list are valid for your installation, select the `Other` radio button and enter the correct value in the lower text field.

5. Press the SUBMIT button. This invokes the servlet and runs the application.

   If the application is configured correctly and the input values are valid, the HelloWorldResults JSP displays the message "You said *your phrase*" in the browser (where *your phrase* is the phrase you entered in step 3).

   If the application is not configured correctly, or one or more of the input values is invalid, the HelloWorldError JSP displays an error message in the browser. The `readme.txt` file contains hints and tips that may help you debug a failed application.

The EJB Bank Account sample application

The EJB Bank Account sample demonstrates how you can use enterprise beans and DB2 to make existing, CICS-controlled, information available to Web users.

What the EJB Bank Account sample does

The sample application extracts customer information from data tables and returns it to the user.

The sample consists of:

- An HTML form.
- A Java servlet, plus JavaServer Pages, running in a J2EE-compliant Web application server.
- An enterprise bean running on a CICS EJB server.
- Two DB2 data tables containing customer information. One contains account information such as current balance; the other contains name and address details.
- Two CICS server programs, written in COBOL. The DFH0ACTD program retrieves information from the accounts data table. The DFH0CSTD program retrieves information from the name and address data table.

The sample works like this:

1. The user starts the application from a Web browser. A form is displayed.
2. The form requests a customer number from the user. When the user has entered a customer number and pressed the SUBMIT button, the servlet is invoked.
3. The servlet:
   a. Looks up a reference to the enterprise bean in the JNDI namespace
   b. Creates a new remote instance of the enterprise bean in CICS
   c. Invokes a method on the bean-instance, passing as input the customer number input by the user
4. The enterprise bean uses the Common Connector Interface (CCI) of the CCI Connector for CICS TS to link to the CICS COBOL server programs, passing the customer number.

   The CCI Connector for CICS TS is described in Chapter 25, "The CCI Connector for CICS TS," on page 319.
5. The server programs use the specified number as the key to the DB2 records for this customer. They retrieve the customer's details from the DB2 data tables and return the account number, balance, and address to the enterprise bean.
6. The enterprise bean returns the customer's details to the servlet, which uses a JavaServer Page to display them on the user's browser. If the customer number is not valid, the browser displays an error page.

Design note: An alternative design would be to replace the connector code with a JCICS LINK call. The advantage of using a CCI-compliant connector such as the CCI Connector for CICS TS is that it makes it easier to port the application between application servers such as WebSphere and CICS. If portability is not required, a JCICS call would be sufficient.

Figure 23 on page 272 shows the components of the sample application.

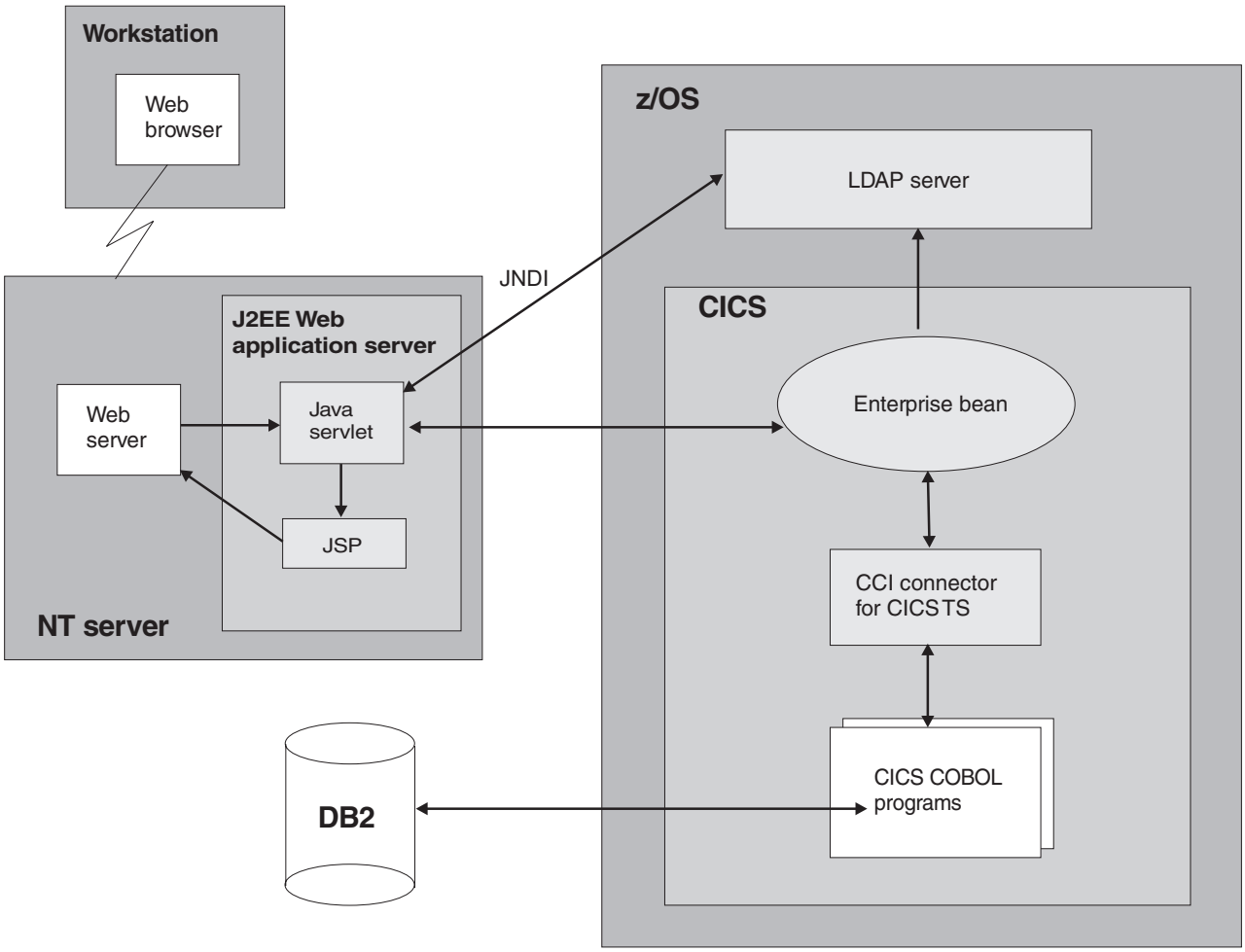

Figure 23. Overview of the EJB Bank Account sample application. The main elements of the sample are a Java servlet, an enterprise bean, two CICS server programs, and two DB2 data tables. The sample extracts customer details from the data tables and returns them to the user. In this example, the servlet is running in a Web application server on a Windows NT server; an LDAP name server is used. Other configurations are possible. For example, a COS Naming Server could have been used.

Prerequisites for the EJB Bank Account sample

You will need these resources to run the EJB Bank Account sample.

- A CICS EJB server. The way to set one up is described in Chapter 19, "Setting up an EJB server," on page 239.
- DB2 Version 7 or later.
- A Web application server that supports J2EE Version 1.2.1 or later. If you are using WebSphere Application Server, note that the sample requires WebSphere Application Server Version 4 or later.
- A name server that supports JNDI Version 1.2 or later. The way to set one up is described in "Actions required on z/OS or Windows NT" on page 240.

Supplied components of the EJB Bank Account sample

These files are supplied with the EJB Bank Account sample.

Table 12. Supplied components of the EJB Bank Account sample

Filename	Type	Default location	Comments
DFH$EDB2	Text deck	SDFHSAMP	DB2 data definition language (DDL) statements to define the DB2 data tables used by the sample and to populate them with data.
DFH$ESQL	Text deck	SDFHSAMP	DB2 data manipulation language (DML) statements to bind the DB2 data tables to the COBOL server programs.
DFH$EJB2	Resource definition group	CSD	Contains the CICS resource definitions required by the sample application.
DFH0ACTD	COBOL source code	SDFHSAMP	Source code of the DFH0ACTD server program.
DFH0CSTD	COBOL source code	SDFHSAMP	Source code of the DFH0CSTD server program.
DFHEBURM	Sample user replaceable program	SDFHSAMP	Changes the user ID under which the sample runs.
`CicsSample.ear`	EAR file	z/OS UNIX samples directory: see Note.	The Web components of the sample application—Java servlet classes and source files; HTML and JSPs.
`readme.txt`	Text file	z/OS UNIX samples directory: see Note.	Contains: 1. Step-by-step instructions for installing the Web components of the EJB sample on WebSphere Application Server. 2. Hints, tips, and debugging information.
`SampleCLI.jar`	JAR file	z/OS UNIX samples directory: see Note.	Client EJB stubs required by the servlet.
`SampleEJB.jar`	Deployed JAR file	z/OS UNIX samples directory: see Note.	Java classes, source files, deployment descriptor, plus supporting classes for the CICS enterprise bean. Doesn't need to be unpacked unless you want to modify the source code.
Note: The default z/OS UNIX samples directory is `/usr/lpp/cicsts/cicsts41/samples/ejb/bankaccount` where `cicsts41` is the install directory for CICS files on z/OS UNIX.			

Security of the EJB Bank Account sample

It is recommended that you run the Bank Account sample in a secure environment. However, in order to simplify the installation process, you may choose not to do so at first.

If you don't want to activate the secure environment immediately, set the XEJB system initialization parameter to 'NO' and skip the rest of this section. To activate the secure environment at a later date, follow the instructions in the rest of this section.

You can implement security for the sample in a number of ways. For example, you can use any of the following alternatives:

- Allow all users to run the sample under the default user ID.
- Allow all users to run the sample under a user ID specified by the security exit program for IIOP.
- Use an SSL server-side certificate to encrypt the data sent between the Web-tier and CICS, allowing all users to run the sample over a secure transport, under the default user ID.
- Use an SSL server-side certificate to encrypt the data sent between the Web-tier and CICS, allowing all users to run the sample over a secure transport, under a user ID specified by the security exit program for IIOP.
- Use SSL client certification to automatically authenticate the Web-tier application server to CICS, allowing all users to run the sample over a secure transport, under a user ID assigned to the Web-tier application server.
- Use asserted identity authentication to allow Web-tier client applications running in WebSphere Application Server for z/OS to propagate their existing user IDs to CICS over a secure transport.

Note:

1. By default, the Bank Account application does not require the user to be authenticated at the Web-tier. You can choose to activate authentication in the Web container by following your application server's instructions. If you do authenticate in the Web tier, the security principle is not propagated to CICS, so in terms of CICS security it has no effect. However, early authentication in the Web-tier could be used to create a “protection domain” under which CICS trusts the Web-tier not to allow unauthenticated users to invoke business methods on CICS enterprise beans.
2. In order to use SSL encryption or authentication, you require a J2EE-compliant Web application server that fully supports SSL. Consult your vendor's documentation for further details.
3. For more information about SSL authentication, see SSL authentication, in the *CICS RACF Security Guide*.

Whichever authentication method you choose, you need (among other things) to:

1. Provide authorization information in the deployment descriptor of the enterprise bean in CICS. This authorization information consists of:

   A "security role" element
   Identifies a class of user who is allowed to perform a given action or use a given resource.

   A "method permission" element
   Identifies specific methods of the enterprise bean that members of the specified security role are authorized to use.

2. Update your CICS external security manager (ESM) to map the specified security role to a number of real user IDs. The following step-by-step instructions for implementing security assume that your ESM of choice is RACF. If you use a different ESM, consult your ESM vendor for guidance.

Implementing role-based security for the Bank Account sample

You can implement role-based security for the Bank Account sample using the Assembly Toolkit (ATK, which is a component of the Application Server Toolkit, ASTK).

About this task

This tool is shipped as part of WebSphere Application Server Version 5.1 and later. You can use the graphical user interface of ATK to (among other things) edit the contents of an enterprise bean's deployment descriptor.

Before you start, ensure that you have ATK installed on your workstation. Once installed, the tool can be launched from an icon which is added to your Start menu in Windows.

ATK is used for the first stage of implementing role-based security, which involves editing the deployment descriptor for the enterprise bean. When you have completed that stage, follow the instructions for the second stage of implementing role-based security, which involves configuring other software.

Stage 1. Using ATK to edit the deployment descriptor:

At this point, in order to familiarise yourself with ATK, you can browse through the contents of the JAR file.

1. Copy the `SampleEJB.jar` file from the z/OS UNIX samples directory to your workstation. You can do this using FTP in binary mode, or any other method of your choice. The z/OS UNIX samples directory is `/usr/lpp/cicsts/cicsts41/samples/ejb/bankaccount`. For ATK, you also need to perform the same process for the `dfjcci.jar` file, which is in the `/usr/lpp/cicsts/cicsts41/lib` directory. You do not need to edit that JAR file, but ATK needs it to rebuild the JAR file for the EJB bank account sample correctly after editing.
2. Import the JAR file into ATK as an EJB project.
   a. Start ATK, and go to the J2EE perspective by selecting **Window > Open Perspective > J2EE**.
   b. Select the **Import** option from the **File** menu. Select **EJB JAR file** as the import source. Select **Browse** and find the `SampleEJB.jar` file. Enter a suitable name for the project. Select **Next** and choose to import all enterprise beans, which is the default. Select **Finish** to create the EJB project.
   c. When the project is created, you should see some errors appear in the Tasks list. To correct these errors, you need to add the `dfjcci.jar` file to the build path for the EJB project. In the left-hand navigation pane (using the J2EE hierarchy view), expand the EJB Modules item to see your EJB project. Right-click on the project name and select **Properties**. Select **Java Build Path**. Go to the **Libraries** tab and select the **Add External JARs** button. Navigate to the `dfjcci.jar` file and select **Open**. Select **OK**. ATK rebuilds the EJB project and the errors should disappear.

   For more information about the EJB deployment descriptor, see “Enterprise beans—the deployment descriptor” on page 218.

3. Add security roles to the deployment descriptor. In ATK, in the left-hand navigation pane (using the J2EE hierarchy view), expand the EJB Modules item to see your EJB project. Double-click on the project name to open the project. Select the **Assembly Descriptor** tab at the bottom of the pane. Under **Security Roles**, select the Add button to add a new security role.

   If your organisation has already set up security roles for use with other applications, you may want to reuse an existing role. If so, supply the name of the role that you want to use in the field provided. If you don't have an existing security role that you want to reuse, enter a new role name, such as "All_users". You can also provide an optional description of the role to act as a memory aid in the future. Select **Finish** to return to the main window.

   Note: If you reuse an existing security role which is already defined to your ESM, you must remove the `Display Name` element from the JAR file's deployment descriptor. This element is used by CICS to provide an application name which is prefixed to all security role names when performing a security check at runtime, thus providing support for security roles scoped at the application level rather than enterprise-wide. In ATK, you can remove this element by selecting the **Overview** tab at the bottom of the pane. Select the text in the `Display Name` field and delete it.
4. Now define a method permission and associate it with a security role. In ATK, select the **Assembly Descriptor** tab again. Under **Method Permissions**, select the **Add** button. The wizard presents a list of the security roles that you have defined. For the Bank Account sample, it's appropriate to run all the methods under the same security role. Select the security role that you want to associate with the method permission, and select **Next**. Select the CICSSample bean, and select **Next**. Check the box for CICSSample to select all the method elements for the bean. Select **Finish**. You are returned to the previous screen.
5. Save the updated deployment descriptor by selecting the **Save** option from the **File** menu.
6. Export the project from ATK back into a JAR file on your workstation. To do this, select the **Export** option from the **File** menu. Select **EJB JAR file** as the export destination, and select **Next**. Select your EJB project from the drop-down list. Select **Browse** and locate the `SampleEJB.jar` file to be used as the destination. (This overwrites your original version of the file. You might want to keep a backup of the original version of the file on your workstation under a different name.) Select the checkbox for **Export source files** to keep the source files with the JAR file. Select **Finish**. Exit ATK.
7. Copy the updated `SampleEJB.jar` file back to z/OS UNIX. You can use either FTP in binary mode or your preferred file transfer process. Save the `SampleEJB.jar` file to the pickup directory of your CorbaServer.

Stage 2. Configuring other security settings:

The CICS user ID (or IDs) that you choose to associate with the security role defined in the enterprise bean's deployment descriptor should be chosen according to which security implementation you opted for at the start of this section.

1. Ensure that both the SEC and the XEJB CICS system initialization parameters specify 'YES'. (If either specifies 'NO', EJB role-based security is turned off.)
2. If you reused an existing security role that had already been set up in your installation, you can skip this step, which is to update RACF to associate the EJB security role with a set of CICS user IDs.

Note: If your ESM is not RACF, you must seek advice from your ESM vendor as to how to perform this step.

For example:

- If you want to allow all anonymous users to run the sample (whether using SSL or not), you should associate the CICSUSER default user ID with the security role.
- If you want to run the sample under a user ID (or IDs) selected by the security exit program for IIOP (whether using SSL or not), you should associate that user ID (or IDs) with the security role.
- If you want to use full SSL client certification, you should associate the user ID of the Web-tier application server's certificate with the security role.

To set up the necessary EJB security role-to-CICS user ID mapping:

a. Run the RACF EJBROLE generator utility against the updated `SampleEJB.jar` file. (The RACF EJBROLE generator utility is a Java program that extracts security role information from deployment descriptors, and generates a REXX program which defines security roles to RACF. For information on how to use the generator utility, see "Using the RACF EJBROLE generator utility" on page 359.)
b. Ask your RACF administrator to run the REXX program generated by the RACF EJBROLE generator utility.

3. If you don't want to use the the security exit program for IIOP to alter the user ID that the sample runs under (from the default CICS user ID to another ID of your choice), you can skip this step.

   CICS supplies a sample security exit program, DFHEBURM, that alters the user ID under which the Bank Account sample runs from the default CICS user ID to "SAMPLE". You can use this version of the user-replaceable program, or alter it to suit your needs. If you already have a customized security exit program for IIOP, you can update your version to perform a similar function.

   You must specify the name of your security exit program on the URM option of the TCPIPSERVICE definition under which the sample is to be run.

   For guidance information about the security exit program for IIOP, see"Using the IIOP user-replaceable security program" on page 203.

   For information about writing a security exit program for IIOP, see the *CICS Customization Guide*. Also, study the source of the supplied sample program, which contains comments and tips.

   For information about compiling and installing user-replaceable programs, see Assembling and link-editing user-replaceable programs, in the *CICS Customization Guide*.

   For information about coding TCPIPSERVICE definitions, see the *CICS Resource Definition Guide*.

4. If you are using SSL encryption or authentication, you must:
   - Configure your J2EE-compliant Web application server to use SSL. Refer to your Web server's documentation for guidance.
   - Have a server certificate available for use.
   - Alter the definitions of the CORBASERVER and TCPIPSERVICE resources under which the sample is to be run. That is:
     – If you are using SSL client-side authentication, the CLIENTCERT option of the CORBASERVER definition must specify the name of a TCPIPSERVICE that defines the port to be used for inbound IIOP requests with SSL client certification. Also, the Web application server's SSL certificate must be:
       - Included in the list of certificates trusted by CICS, in RACF

- Mapped to a RACF userid

– If you are using SSL server-side authentication, the SSLUNAUTH option of the CORBASERVER definition must specify the name of a TCPIPSERVICE that defines the port to be used for inbound IIOP requests with SSL but no client certification.

For information about coding CORBASERVER resource definitions and TCPIPSERVICE resource definitions, see the the *CICS Resource Definition Guide*.

- If you are using asserted identity authentication for encryption, authentication, and identity propagation, you must:
  – Configure WebSphere Application Server for z/OS to authenticate users.
  – If you are using WebSphere Application Server for z/OS Version 6.1 or later, to enable a suitable authentication protocol, specify the system property **-Dcom.ibm.cics.iiop.CSIv2Enabled=true** in all of the JVM properties files used in the CICS region. (Release 6.1.0.13 or later of WebSphere Application Server for z/OS is required to support this function.)
  – Enable SSL client certification in WebSphere.
  – Have a server SSL certificate available for use in CICS.
  – Include the server certificate associated with WebSphere Application Server in the RACF's list of certificates trusted by CICS. Additionally, the userid associated with the RACF certificate must be granted permission to assert the identity of other users.
  – Alter the definitions of the CORBASERVER and TCPIPSERVICE resources under which the sample is to run. The ASSERTED option of the CORBASERVER definition must specify the name of a TCPIPSERVICE that defines the port to be used for inbound IIOP requests with asserted identity authentication.

Installing the EJB Bank Account sample

Installing the EJB Bank Account sample requires actions on:
1. z/OS (DB2 and CICS)
2. The Web application server

z/OS setup

On z/OS you must perform these steps to install the sample.

About this task

1. Compile and link-edit the CICS COBOL DB2 server programs, using your organization's normal procedures. The DFH0ACTD and DFH0CSTD members of the SDFHSAMP library contain the source code of the server programs.

   Store the load modules in an application load library that is included in the CICS DD DFHRPL concatenation. (For information about storing load modules in application load libraries, see the *CICS System Definition Guide*.)
2. Define the DB2 data tables used by the sample, and populate the tables with data. The DFH$EDB2 text deck contains the necessary DB2 DDL statements and the supplied data.

   Before using DFH$EDB2, you must modify the following line to suit your system:

   ```
   CREATE STOGROUP EBSAMPSG VOLUMES(SYSDA,SYSDB) VCAT DSNxxxxx;
   ```

Change DSNxxxxx to the name of your high-level integrated catalog facility (ICF) catalog identifier for user-defined VSAM data sets.

Authority required: DB2 authority to create a database, storage group, tablespace, tables, and indices.

3. Bind the DB2 tables to the COBOL server programs. The DFH$ESQL text deck contains the necessary DB2 DML statements.

   Authority required: DB2 authority to perform a BIND for this database.

   Note:

   a. This step statically binds the SQL statements in the server programs to DB2, so that they don't have to be dynamically bound at execution time, thus improving runtime performance.

   b. If you recompile one of the server programs subsequently and intend it to access DB2, each time you recompile you must:

      1) Re-bind the DB2 tables to the COBOL server programs.

      2) Refresh the copy of the server program on CICS by executing the following CICS command in the CICS region:

         ```
         CEMT SET PROG(program_name) NEW
         ```

         For example, if you change the DFH0CSTD program and recompile it, use CEMT SET PROG(DFH0CSTD) NEW. (DFH0CSTD is defined to the CICS region in the DFH$EJB2 resource definition group—see step 5.)

4. Grant authority to the CICS user ID to access the DB2 plan, using your organization's normal procedures (for example, SPUFI). For information about granting authority to access a DB2 plan, see Controlling users' access to plans, in the *CICS DB2 Guide*.

5. Define the programs and DB2 connections used by the sample to CICS. The CICS-supplied sample group, DFH$EJB2, contains resource definitions for the EJB "Bank Account" sample. You must change some of the attributes of these resource definitions to suit your own environment. To do this, use the CEDA transaction or the DFHCSDUP utility.

   a. Copy the sample group to a group of your own choosing. For example:

      ```
      CEDA COPY GROUP(DFH$EJB2) TO(mygroup)
      ```

   b. Display group mygroup and change the attributes of the following definitions as shown:

      - On the DB2CONN definition, change the value of DB2ID to the ID of the DB2 subsystem on which you created the DB2 tables used by the sample.
      - The PROGRAM definitions do not need to be modified.

   c. Discard the definitions that you don't need from group mygroup.

      As well as DB2CONN and PROGRAM definitions, DFH$EJB2 also contains a CORBASERVER and a TCPIPSERVICE definition. However, these are for reference only. It is strongly recommended that you set up your EJB server, as described in Chapter 19, "Setting up an EJB server," on page 239, *before* attempting to install the sample programs. If you do this, you don't need the CORBASERVER and TCPIPSERVICE definitions in DFH$EJB2 because you will already have created your own based on those supplied in resource group DFH$EJB. Discard them from group mygroup.

      If you *do* decide to use the CORBASERVER and TCPIPSERVICE definitions in DFH$EJB2, you must modify them as described in "Actions required on CICS" on page 241.

If your CICS region uses program autoinstall, you don't need the PROGRAM definitions. Discard them from group `mygroup`.

Note: There is no supplied REQUESTMODEL definition, because it's not necessary to install one. The sample uses the default transaction ID, CIRP.

d. Add the resource group containing the modified resource definitions to the CICS CSD, and to the CICS startup group list. To do this, it is recommended that you use the CICS system definition utility program, DFHCSDUP. For information about using DFHCSDUP, see System definition file utility program (DFHCSDUP), in the *CICS Operations and Utilities Guide*.

Authority required: RACF authority to install resource definitions into the CICS region.

6. If you have not already done so while setting up security, put the supplied `SampleEJB.jar` deployed JAR file into your CorbaServer's "pickup" directory.
7. Ensure that the name server has been started. If CICS has not been started, start it now.
8. Issue the following command at the CICS region console:

   ```
   CEMT PERFORM CORBASERVER(corbaserver_name) SCAN
   ```

   CICS scans the pickup directory, copies the `SampleEJB.jar` deployed JAR file to its shelf directory, and creates and installs a DJAR definition for it.

   Note: If you had to start CICS in step 7, this step is not necessary, because CICS will have scanned the pickup directory on startup.

   Authority required: RACF authority to create a DJAR and update access to the CORBASERVER.
9. Publish the enterprise bean to the JNDI namespace. If your CORBASERVER definition specifies AUTOPUBLISH(YES), this will have happened automatically when the `SampleEJB.jar` deployed JAR file was installed. If your CORBASERVER definition specifies AUTOPUBLISH(NO), issue the following command at the CICS region console:

   ```
   CEMT PERFORM DJAR(SampleEJB) PUBLISH
   ```

   Authority required: RACF authority to update a DJAR.
10. Use the `CICSConnectionFactoryPublish` sample program to create a ConnectionFactory object for use by the CCI Connector for CICS TS, and to publish it to the name server. For instructions on how to use the `CICSConnectionFactoryPublish` program, see "Using the sample utility programs to manage and acquire a connection factory" on page 327.
11. Ensure that the DB2 connection status is CONNECTED by issuing the following command at the CICS system console:

    ```
    CEMT SET DB2CONN CONNECTED
    ```

Web application server setup

On the Web application server, you must install the Web components of the EJB Bank Account sample application.

About this task

From the z/OS UNIX EJB samples directory, you need:

- `CicsSample.ear`. A J2EE enterprise archive (EAR) file containing the Web components of the sample.
- `readme.txt`. A text file containing:

1. Step-by-step instructions for installing the Web components of the sample on WebSphere Application Server.
2. Hints, tips, and debugging information.

Note: The default samples directory is
`/usr/lpp/cicsts/cicsts41/samples/ejb/bankaccount`

where `/usr/lpp/cicsts/cicsts41` is the install directory for CICS files on z/OS UNIX.

Important: The rest of this section contains generic instructions for installing the Web components of the sample on a J2EE-compliant Web application server (which may or may not be WebSphere). It is suitable for experienced users. If your Web application server is WebSphere Application Server Version 4 or later *and* you are a novice user of that product, we recommend that you follow instead the detailed, WebSphere-specific instructions in the `readme.txt` file.

Procedure

1. Install the Web components of the EJB Bank Account sample (contained in `CicsSample.ear`) in your J2EE Web application server, following the vendor's guidelines for installing applications. In WebSphere Application Sever, for example, this involves using the administration console to:
   a. Install a new application
   b. Generate the updated Web server plugin
   c. Save the configuration

   Note: `CicsSample.ear` includes a default configuration for the EJB Bank Account sample. To run the sample, it is not necessary to edit or add any configuration information.
2. Start the application using your Web application server's standard procedure.

Results

Testing the EJB Bank Account sample

You must perform these steps to test the application.

About this task

1. Ensure that all the following are running:
   - The Web server
   - The Web application server and the sample application
   - The name server
   - The CICS region
   - The DB2 subsystem
2. Start a Web browser and point it at the URL of your Web server, followed by "cicssample". For example:
   `http://myServer.ibm.com/cicssample`

   The opening screen shown in Figure 24 on page 282 appears.

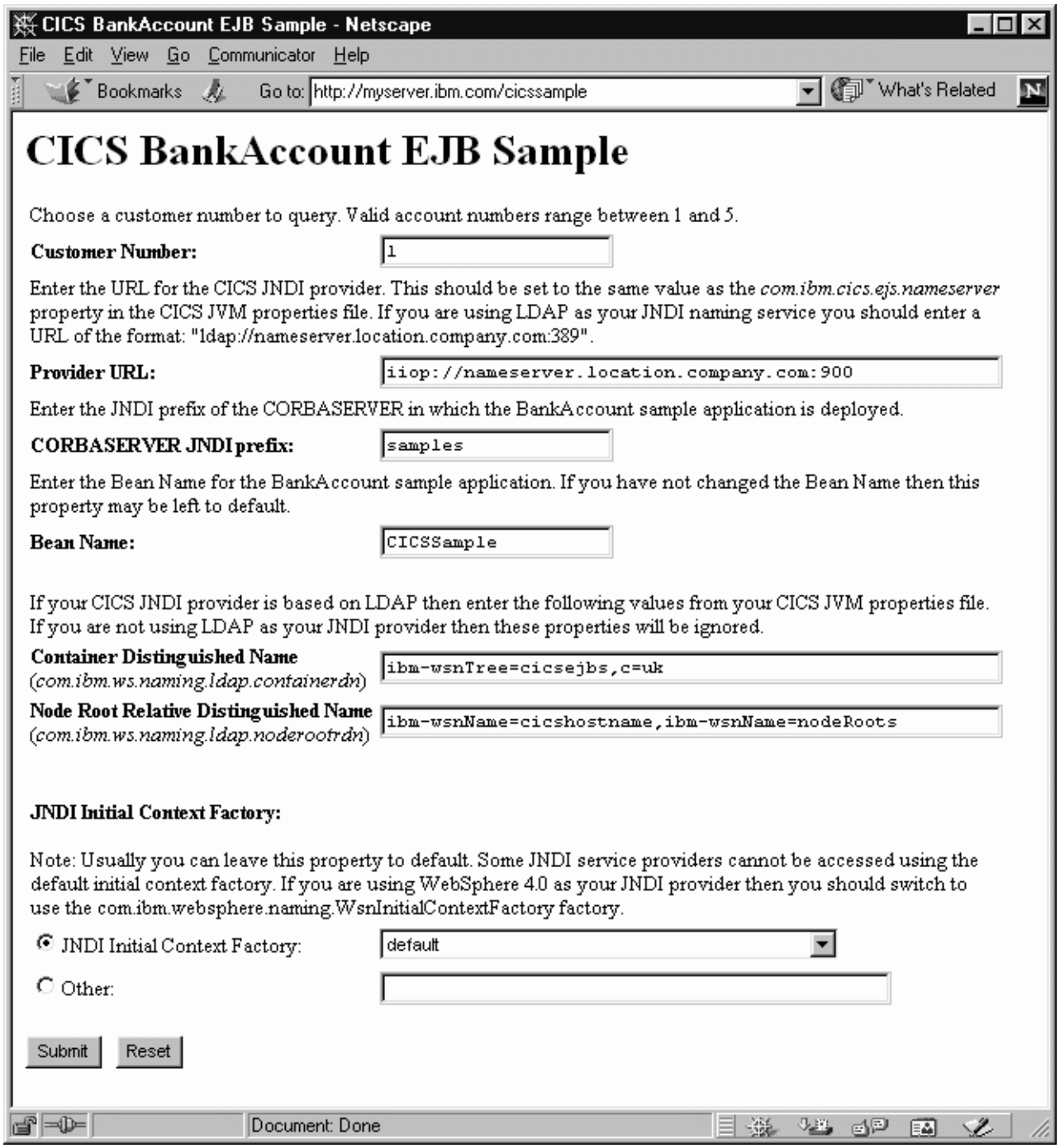

Figure 24. Opening screen of the EJB Bank Account sample application

3. Enter a customer number. (Using the supplied DB2 data, valid customer numbers are 1 through 5).
4. Check that the `Provider URL:`, `CORBASERVER JNDI prefix:`, `Bean Name:`, `Container Distinguished Name:`, `Node Root Relative Distinguished Name:`, and `JNDI Initial Context Factory` fields contain values that are valid for your installation. If they do not, overtype them as follows:

   `Provider URL:`
   Enter the URL and port number of the name server where the enterprise

bean is published. (These are specified by the **-Dcom.ibm.cics.ejs.nameserver** property in your JVM properties file.) For example:

- If you are using a COS Naming Server with a URL of `mycosns.ibm.com` and a port number of 900, specify `"iiop://mycosns.ibm.com:900"`.
- If you are using an LDAP name server with a URL of `myldapns.ibm.com` and a port number of 389, specify `"ldap://myldapns.ibm.com:389"`.
- If you are using the COS Naming Directory Server supplied with WebSphere Application Server Version 5 or later, with a URL of `mycosns.ibm.com` and a port number of 2809, specify:

  ```
  -Dcom.ibm.cics.ejs.nameserver=iiop://mycosns.ibm.com:2809/domain/legacyRoot
  ```

For detailed information about how to specify the location of the name server, see the description of the **-Dcom.ibm.cics.ejs.nameserver** property in "JVM system properties" on page 119.

CORBASERVER JNDI prefix:
Enter the JNDI prefix of your CorbaServer. If you are using the CORBASERVER definition supplied in DFH$EJB, you do not need to change the default value of "`samples`".

Bean name:
Enter the name of the enterprise bean used by the sample, as defined in the deployment descriptor in the supplied `SampleEJB.jar` file. *Unless you have renamed the bean, you do not need to change the default value of "CICSSample".*

Container Distinguished Name:
If you are using an LDAP name server, enter the distinguished name of the LDAP system namespace root, as supplied by your LDAP administrator. (The distinguished name of the LDAP system namespace root is specified by the **-Dcom.ibm.ws.naming.ldap.containerdn** system property.) *If you are using a COS Naming Server, the value of this field is ignored.*

Node Root Relative Distinguished Name:
If you are using an LDAP name server, enter the distinguished name of the LDAP node root, as supplied by your LDAP administrator. (The distinguished name of the LDAP node root is specified by the **-Dcom.ibm.ws.naming.ldap.noderootrdn** property.) *If you are using a COS Naming Server, the value of this field is ignored.*

JNDI Initial Context Factory:
Select the appropriate JNDI initial context factory from the drop-down list. If your Web application server is WebSphere, the factory to use depends on:

- The version of WebSphere you're using
- The location of WebSphere—that is, whether it's on a distributed platform such as Windows NT or a host platform such as z/OS or OS/390
- The type of name server you're using—COS naming or LDAP

Table 13 shows the correct initial context factory to specify, if your Web application server is WebSphere.

Table 13. Setting the initial context factory, according to the version and location of WebSphere and the type of name server

WebSphere Version	Location of Web application server	Name server type	Initial context factory to use
3.5	Distributed	COS	`com.ibm.ejs.ns.jndi.CNInitialContextFactory`

Table 13. Setting the initial context factory, according to the version and location of WebSphere and the type of name server (continued)

WebSphere Version	Location of Web application server	Name server type	Initial context factory to use
3.5	Distributed	LDAP	`com.ibm.jndi.LDAPCtxFactory`
3.5	z/OS or OS/390	COS	`com.sun.jndi.cosnaming.CNCtxFactory`
3.5	z/OS or OS/390	LDAP	`com.sun.jndi.ldap.LdapCtxFactory`
4 or later	Distributed	COS or LDAP	`com.ibm.websphere.naming.WsnInitialContextFactory`
4 or later	z/OS or OS/390	COS	`com.sun.jndi.cosnaming.CNCtxFactory`
4 or later	z/OS or OS/390	LDAP	`com.sun.jndi.ldap.LdapCtxFactory`

If your Web application server is not WebSphere, choose the appropriate value from the drop-down list.

Note: The drop-down list contains several initial context factory classes, plus a "default" list item. The sample application assigns the value of the default list item as follows:

a. If the `com.ibm.websphere.naming.WsnInitialContextFactory` class is found in the Java classpath, the sample makes it the default item. This class is a "wrapper" class that wraps both `com.ibm.ejs.ns.jndi.CNInitialContextFactory` and `com.ibm.jndi.LDAPCtxFactory`. The sample determines the correct base class to use by examining the type of name server that you've specified in the **Provider URL** field: if the specified protocol is "iiop", the sample uses `com.ibm.ejs.ns.jndi.CNInitialContextFactory`; if it's "ldap", the sample uses `com.ibm.jndi.LDAPCtxFactory`.

b. If the `com.ibm.websphere.naming.WsnInitialContextFactory` class is *not* found in the Java classpath, the sample determines the correct class to use by examining the type of name server that you've specified in the **Provider URL** field: if the specified protocol is "iiop", the sample uses `com.ibm.ejs.ns.jndi.CNInitialContextFactory`; if it's "ldap", the sample uses `com.ibm.jndi.LDAPCtxFactory`.

If none of the values in the drop-down list are valid for your installation, select the `Other` radio button and enter the correct value in the lower text field.

5. Press the SUBMIT button. This invokes the servlet and runs the application.

   If the application is configured correctly and the input values are valid, the `SampleResults` JSP displays the customer's details in the browser. Figure 25 on page 285 shows the result of a successful inquiry.

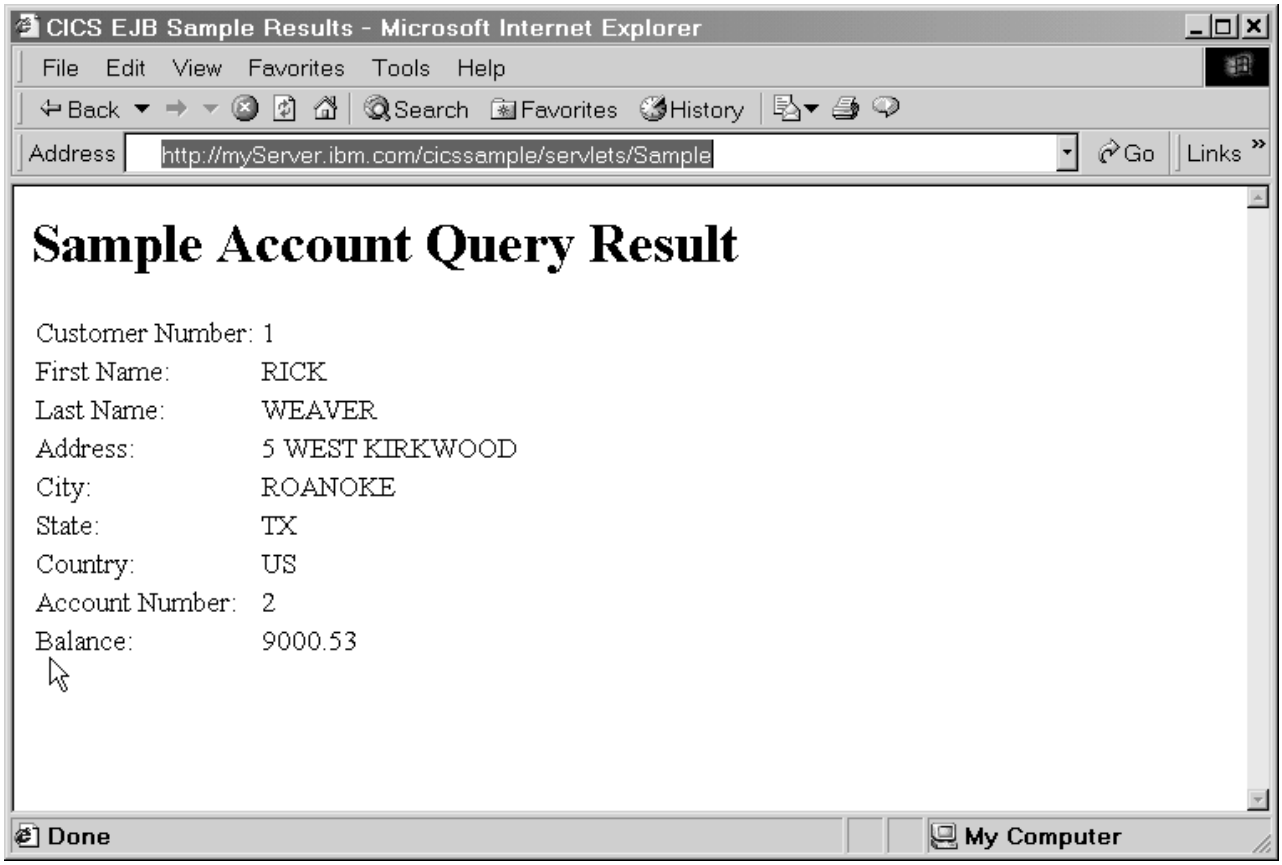

Figure 25. Results screen of the EJB Bank Account sample application

If the application is not configured correctly, or one or more of the input values is invalid, the `SampleError` JSP displays an error message in the browser. The `readme.txt` file contains hints and tips that may help you debug a failed application.

A note about distributed transactions

A number of protocols exist to support distributed transactions.

The CICS enterprise Java environment supports only the CORBA Object Transaction Service (OTS) protocol. However, some J2EE-compliant web application servers (such as WebSphere) either do not use this protocol, or do not use this protocol by default. WebSphere can be configured to use pure OTS distributed transactions; for detailed instructions on how to set up WebSphere to use the OTS, see the `readme.txt` file supplied with the Bank Account sample.

If objects on your web application server call CICS enterprise beans within the scope of existing transaction contexts, you must set up your web application server to use the CORBA OTS.

Changing the sample to use distributed transactions

You can try this exercise to test whether or not your J2EE web application server is fully compatible with CICS.

About this task

By default, the EJB Bank Account sample is not configured to use distributed transactions. However, you can change this. The `SampleServlet` servlet contains sample code, which has been commented-out, to turn on client-demarcated transactions. (The `SampleServlet.java` source file is in the `CicsSample.ear` file.)

To turn on client-demarcated transactions:
1. Uncomment the transaction-related code in `SampleServlet.java`.
2. Recompile the `SampleServlet` servlet.
3. Install the updated copy of the servlet into your web application server.

If you set up the sample to use client-demarcated transactions but your J2EE web application server does not support (or is not configured to use) pure OTS transactions, when you run the sample CICS throws an `org.omg.CORBA.INVALID_TRANSACTION` exception. This is because a transaction context was sent but CICS could not use it.

Changing the enterprise bean's transaction attribute

You may also want to change the enterprise bean's transaction attribute (in the deployment descriptor) from 'Supports' to 'Mandatory'.

If you do this, CICS allows the remote method of the bean to be invoked only if an existing OTS transaction context is passed from the client's environment on the call.

If, on the other hand, you leave the enterprise bean's transaction attribute set to 'Supports', CICS binds the method invocation to the client's transaction context if such a context exists; otherwise the method runs in an atomic transaction and does not propagate a new transaction context when calling other beans.

To change the transaction attribute, you can use the Assembly Toolkit (ATK), which is described in the *CICS Operations and Utilities Guide*. Having changed the transaction attribute, to make the change effective you must:
1. Store the updated `SampleEJB.jar` file in your pickup directory (overwriting the previous version).
2. Issue a `CEMT CORBASERVER(corbaserver_name) PERFORM SCAN` command.

If you set the transaction attribute to 'Mandatory' but don't update the servlet to use client-demarcated transactions, when you run the sample CICS throws a `javax.transaction.TransactionRequiredException`. This is because no transaction context has been sent.

A note about data conversion

To represent text data, Java programs always use the Unicode character set, while CICS TS programs use EBCDIC.

When a Java program or enterprise bean calls a CICS TS server program, any text values in the communications area of the server program must be converted from Unicode to EBCDIC on input, and from EBCDIC to Unicode on output. The enterprise bean in the EJB Bank Account sample uses the CCI Connector for CICS TS, which handles this data conversion automatically—see “Data conversion and the CCI Connector for CICS TS” on page 327.

Note: Only the text data returned by COBOL program DFH0CSTD is converted from EBCDIC to Unicode . (No conversion is necessary for server program DFH0ACTD, nor on input to DFH0CSTD, because there are no text values in the communications areas.)

Chapter 22. Writing enterprise beans

You can write session beans. The interfaces used by these beans are mapped to CICS services and resources and the beans are portable to any other EJB-compliant server.

Session beans use the interfaces defined by Sun Microsystem's *Enterprise JavaBeans Specification, Version 1.1*, which is described at `http://www.javasoft.com/products/ejb`.

You can also write session beans that use the JCICS classes to access CICS services and resources directly. These beans are portable only to other CICS EJB servers.

CICS does not support entity beans—that is, you cannot run entity beans in a CICS EJB server. (A session bean or program running in a CICS EJB server can communicate with an entity bean running in a non-CICS EJB server.)

You can write your beans on a workstation using any integrated development environment (IDE) that supports the *Enterprise JavaBeans Specification, Version 1.1*.

When developing new Java enterprise beans and programs for CICS , you should use an application development environment that supports Java 2 at the SDK 5.0 level. You should **not**:

- Use any API calls that are supported only by a newer version of the Java SDK than that supported by CICS.
- Use features supported only by a later version of Sun's *Enterprise JavaBeans Specification* than that supported by CICS. (Currently, CICS supports the *Enterprise JavaBeans Specification, Version 1.1*.)

Any enterprise beans developed to the EJB 1.0 specification must be migrated to the EJB 1.1 specification level using the supplied development tools—see "The deployment tools for enterprise beans in a CICS system" on page 303.

"Coding a session bean" on page 290 gives an example of the steps involved in writing a session bean without using an IDE.

You can use the CCI Connector for CICS TS to build enterprise beans that make use of existing CICS programs. See Chapter 25, "The CCI Connector for CICS TS," on page 319 for a description of the CCI Connector for CICS TS , and how to use it.

Preparing beans for execution

The process of installing and preparing an enterprise bean for execution is known as **deployment**.

CICS provides workstation based tools to manage the deployment of enterprise beans into the host CICS environment.

The workstation and WebSphere components of the deployment tools are supplied as a set of InstallShield packages. You can download these packages from your z/OS system or run them from the supplied CD on the target workstation.

See Chapter 23, "Deploying enterprise beans," on page 303 for a description of the deployment process, and "Using CICS deployment tools for enterprise beans" on page 304 for guidance on using the tools.

Coding a session bean

This section describes how to code a very simple session bean.

When you have completed the steps in this section, you will have a JAR file that is ready for deployment. See Chapter 23, "Deploying enterprise beans," on page 303 for a description of the deployment process and the tools available to help you.

The example bean shown here simulates a roulette wheel in a casino. The roulette wheel is a stateful session bean, containing two stateful fields. The first field is the current number that the wheel is on; the second field is the amount of credit the gambler still has for betting. The client creates a roulette wheel, optionally specifying the amount of money to gamble (defaulting to 100 dollars if the amount is not supplied). The client can place bets on the color that will come up and then the wheel spins and tells the caller if he has won or not. The client may then collect the winnings or continue betting.

There are three elements that you must code:

1. "Coding the home interface."
2. "Coding the remote interface."
3. "Coding the bean implementation" on page 291.

Then you need to compile and package your program:

1. "Compiling the code" on page 293
2. "Packaging the code" on page 293

Coding the home interface

The home interface for a bean extends the javax.ejb.EJBHome interface. It defines one or more create methods that the client program may call to create a bean instance.

For stateless session beans there must be exactly one create method taking no parameters. Stateful session beans may overload the create method with different variants taking different combinations of parameters. The RouletteWheel bean is a stateful session bean. We overload create so that we can specify the amount of credit we have on a roulette wheel instance when it is created:

```
package casino;

     public interface RouletteWheelHome extends javax.ejb.EJBHome {

       public RouletteWheel create()
         throws javax.ejb.CreateException, javax.ejb.EJBException;

       public RouletteWheel create(int dollars)
         throws javax.ejb.CreateException, javax.ejb.EJBException;
     }
```

Coding the remote interface

The remote interface for a bean extends the javax.ejb.SessionBean interface. The remote interface defines the actual business methods a client program may call on an individual bean instance.

```
package casino;

     public interface RouletteWheel extends javax.ejb.EJBObject {

       // Place a bet on either "red" or "black" of the given amount,
       // the return value indicates to the caller whether the bet was
       // successful or not.
       public String bet(String bet,int amount) throws javax.ejb.EJBException;

       // Check the current status of the wheel.
       public String getCurrentStatus() throws javax.ejb.EJBException;

       // Collect winnings from the wheel (if any!)
       public int collectWinnings() throws javax.ejb.EJBException;

     }
```

Coding the bean implementation

This class implements the business methods defined in the bean remote interface.

It also defines some standard methods that are declared abstract on SessionBean and so these methods should be implemented for our bean implementation to be complete. Finally, because we overloaded the create method on the home interface, we must provide matching ejbCreate methods in the bean implementation that accept the same sets of parameters. This is because the bean implementation class is the only place that you put your bean code. The implementation of the home interface that we defined in "Coding the home interface" on page 290 is generated by the tooling, so if we need to implement an overloaded create method, we have to do it here:

```
package casino;

     import java.util.Random;
     import javax.ejb.*;

     public class RouletteWheelBean implements SessionBean {

       // Necessary code to fulfill SessionBean interface definition.

       private SessionContext ctx = null;

       public void ejbActivate() throws javax.ejb.EJBException {}
       public void ejbPassivate() throws javax.ejb.EJBException {}
       public void ejbRemove() throws javax.ejb.EJBException {}
       public SessionContext getSessionContext() { return ctx; }
       public void setSessionContext(SessionContext ctx) throws
         javax.ejb.EJBException { this.ctx = ctx;
       }

       ////////////////////////////
       // The bean state information
       private int wheelValue;

       private int currentCredit;

       ////////////////////
       // Our create methods

       public void ejbCreate() throws javax.ejb.EJBException, CreateException {
         currentCredit = 100;
         wheelValue = ((int)System.currentTimeMillis())%37;
       }

       public void ejbCreate(int credit) throws javax.ejb.EJBException,
         CreateException { currentCredit = credit;
```

```
  wheelValue = ((int)System.currentTimeMillis())%37;
}

//////////////////////////////////////////////////////////////////////
// Implementations of the remote methods the client may call on an instance

//
// Place a bet, either "red" or "black" for the specified amount.
// Then simulate the wheel spinning and construct a response string
// indicating the outcome to the caller.
//
public String bet(String color,int amount) throws javax.ejb.EJBException {

  if (!color.equalsIgnoreCase("red") && !color.equalsIgnoreCase("black"))
    return new String("You can only bet on red or black");

  if (amount > currentCredit)
    return new String("You only have $"+currentCredit+" !");

  // Use the current wheel value as the random number seed
  Random randomizer = new Random((long)wheelValue);

  // Spin the wheel
  wheelValue = Math.abs(randomizer.nextInt()) % 37;

  // Construct a reply
  StringBuffer result =
    new StringBuffer("Number: "+wheelValue+" Color: "+color(wheelValue)+"\n");

  // Did the caller win?
  if (color(wheelValue).equalsIgnoreCase(color)) {
    currentCredit+=(amount*2);
    result.append("Well Done! You won $");
    result.append((amount*2));
  } else {
    currentCredit -= amount;
    result.append("Bad Luck! You lost $");
    result.append(amount);
  }
  result.append(", you now have $");
  result.append(currentCredit);
  return result.toString();
}


//
// Return the current status of this roulette wheel instance.
// The number and color
// it is currently on and the amount of credit the client still has to gamble.
//
public String getCurrentStatus() throws javax.ejb.EJBException {
  return new String("Number:"+wheelValue+" Color:"+color(wheelValue)+"
  You have $"+currentCredit);
}


//
// Allow the client to collect his winnings, then zero the credit so
// they cannot collect twice!
//
public int collectWinnings()throws javax.ejb.EJBException {
  int winnings = currentCredit;
  currentCredit = 0;
  return winnings;
}
```

```
  //
  // Convert a number on the wheel into a color
  //
  private String color(int value) {
    if (value == 0) return "Green";
    if (value % 2 == 0) return "Black";
    return "Red";
  }

}
```

Compiling the code

All that you need in addition to the base SDK is the JAR file containing the `javax.ejb` interfaces.

This is available as `ejb11.jar` in the `standard/ejb/1_1` directory of the java installation. If you add `ejb11.jar` to your CLASSPATH, you should be able to compile the classes and interfaces described.

Packaging the code

The compiled classes must be packaged in a JAR file ready for deployment.

Assuming the class files are in the sub directory casino, the following jar command can be used:

```
jar -cvf casino.jar casino\*.class
```

Writing the client program

A client program is any program that calls an enterprise bean.

It can be:

1. Another enterprise bean, JavaBean, Java program, or object executing in the same CICS
2. An enterprise bean, JavaBean, Java program, or object executing in another CICS
3. An enterprise bean, JavaBean, Java program, or object executing on a non-CICS system or workstation

The client obtains references to bean homes of enterprise beans that it wants to call by using the JNDI namespace it shares with the CICS server environment.

Creating object references in the namespace

To create object references, you need to publish the beans that are installed in your CICS region.

About this task

You can do this in two ways:

1. Issue PERFORM DJAR(XXXX) PUBLISH on the server CICS system. You can use any of the following methods to do this:
   - CEMT
   - CICSPlex SM
   - A CICS application

For each bean installed from the named DJAR, an object reference is published to the naming directory server. See “Defining name servers” on page 182 for information about using name servers.

2. If you have installed a number of DJARs into a single CORBASERVER, you can use the PERFORM CORBASERVER(XXXX) PUBLISH command to publish every bean currently installed under that CORBASERVER. The subcontext in the namespace where the object references for the beans will appear is determined by the JNDI prefix defined in the resource definition of the CORBASERVER into which the DJAR was installed.

Retraction is never done implicitly. The recommended way to 'unpublish' beans is to issue PERFORM DJAR(XXXX)/CORBASERVER(XXXX) RETRACT. If a DJAR or CORBASERVER is discarded, the bean object references will still exist in the namespace, although they will be unusable by a client since the actual beans no longer exist in CICS. It is possible to reinstall a DJAR and retract those references.

Using JNDI to obtain bean references

Java Naming and Directory Interface (JNDI) defines an application programming interface (API) specified in the Java programming language that provides the naming and directory function to Java programs.

It also defines a service provider interface (SPI) that allows various directory and naming service drivers to be plugged in.Figure 26 illustrates this by showing a Naming Manager interfacing with a Java application by means of the JNDI API, and with various Name servers via the JNDI SPI.

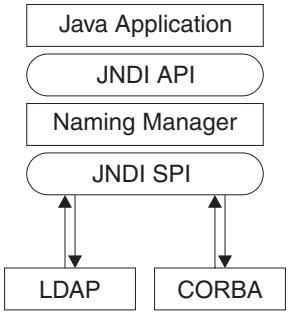

Figure 26. JNDI structure

The JNDI API and the SPI are described in documents that are available from the Sun Microsystem's Web site at http://www.javasoft.com/products/jndi/index.html . An overview is available at http://www.javasoft.com/products/jndi/tutorial/getStarted/overview/index.html .

After an enterprise bean has been registered in a name server by the administrator of the server system, using PERFORM CORBASERVER/DJAR PUBLISH, a client application can use the JNDI interface to locate its home interface.

To enable this, you must set up a suitable name server that supports the Java Naming and Directory Interface (JNDI) Version 1.2, and then define its location to CICS. This is described in “Setting up an LDAP server” on page 184 and “Setting up a COS Naming Directory Server” on page 193. For details of the JVM properties that are needed, see “JVM system properties” on page 119.

Writing a Client program to use LDAP

CICS Transaction Server supports LDAP. Some changes to your client programs might be necessary to allow a client program to find the bean homes published from a CICS region.

An LDAP client must use either the WebSphere Context Factory or the Sun LDAP Context Factory. The advantage of using the WebSphere Context Factory is that it understands automatically the system namespace (that is the structured namespace on the LDAP server into which CICS publishes your bean homes). However, this context factory has a number of dependencies and so is not the most lightweight client. The SUN context factory has no dependencies apart from the base IBM Developer Kit for the Java Platform and so is very lightweight, however it does not understand the system namespace and so it is necessary to negotiate it programmatically, but there are some utility methods provided by CICS to help with this.

These alternatives are best demonstrated by examples:

WebSphere Context Factory

This is an example of some client source code that uses the WebSphere context factory to locate the home for a HelloWorld bean.

```
import org.omg.CORBA.ORB;
import java.io.*;
import javax.naming.*;
import examples.helloworld.*;
import java.util.*;

public class WASNamingClient {
   public static void main(String[] argv) {
      try {

// Set the necessary properties
      Properties prop = new Properties();

// These four are *fixed* values, you never need to change them.

  prop.put(Context.INITIAL_CONTEXT_FACTORY,
       "com.ibm.websphere.naming.WsnInitialContextFactory");

  prop.put("com.ibm.websphere.naming.namespaceroot","bootstraphostroot");
  prop.put("com.ibm.ws.naming.ldap.config","local");
  prop.put("com.ibm.ws.naming.implementation","WsnLdap");

// These two depend on your server settings and should match your CICS region settings

  prop.put("com.ibm.ws.naming.ldap.containerdn","ibm-wsnTree=WASNaming,c=us");
  prop.put("com.ibm.ws.naming.ldap.noderootrdn",
 "ibm-wsnName=legacyroot,ibm-wsnName=PLEX2,ibm-wsnName=domainRoots");

// Finally, instead of com.ibm.cics.ejs.nameserver,
// set com.ibm.ws.naming.ldap.masterurl to your destination LDAP server

      prop.put("com.ibm.ws.naming.ldap.masterurl","ldap://wibble.example.com:389");

      InitialContext ctx = new InitialContext(prop);
      org.omg.CORBA.Object obj =
           (org.omg.CORBA.Object)ctx.lookup("samples/HelloWorld");

      HelloWorldHome hhome =
           (HelloWorldHome)javax.rmi.PortableRemoteObject.narrow
           (obj,HelloWorldHome.class);
```

```
        System.out.println("HelloWorldHome successfully found!");
        HelloWorld hello = hhome.create();
        System.out.println(hello.sayHello());

    } catch (Exception e) {
      System.err.println("Exception while looking up and calling the HelloWorld bean:");
      e.printStackTrace();
     }
  }
}
```

As noted in the comments, the first four properties are fixed, the remaining three match settings for your CICS region (Albeit the **-Dcom.ibm.cics.ejs.nameserver** property has become com.ibm.ws.naming.ldap.masterurl). However, the WebSphere Context Factory has dependencies on components of WebSphere so in order to run it from the command line you must run a script to set up your environment appropriately.

The script `DFHWAS4Setup.bat` is a command line script provided with CICS. It can be downloaded from the utils subdirectory in the z/OS UNIX area where CICS is installed. It must be run on a system that has WebSphere installed, because it relies on the environment variable WAS_HOME being set to point to the location where WebSphere has been installed, for example `c:\WebSphere\AppServer`. When the the script has been run, you should extend your CLASSPATH further to include the necessary client side code for your Enterprise Bean. For the example above this is the `HelloWorld.jar` - then the code above can be compiled and executed. (The example code assume that the home is published in a CorbaServer whose JNDI Prefix is *samples*).

In CICS we set **-Dcom.ibm.cics.ejs.nameserver =** <*hostname*> but in this client program, we set **com.ibm.ws.naming.ldap.masterurl =** <*hostname*>. CICS understands the former, WebSphere understands the latter.

SUN LDAP Context Factory

From an IBM Developer Kit for the Java Platform configuration point of view, it is much easier to use the SUN LDAP Context Factory, since it is provided in the IBM Developer Kit for the Java Platform base and has no dependencies outside of it.

However, because this context factory does not understand the namespace structure that exists on any LDAP server configured for WebSphere, it can be more demanding for the client application programmer. CICS provides some namespace helper functions that ease this added complexity. The **com.ibm.cics.portable.CICSNameSpaceHelper** class is provided in `CICSEJBClient.jar`. This JAR file is available in the `utils` subdirectory in the z/OS UNIX area where CICS is installed.

Here is an example of using this class:

```
import org.omg.CORBA.ORB;
import java.io.*;
import examples.helloworld.*;
import javax.naming.*;
import javax.naming.directory.*;
import java.util.*;
import com.ibm.cics.portable.CICSNameSpaceHelper;

public class SUNNamingClient {

public static void main(String[] argv) {
```

```
        try {
           Hashtable env = new Hashtable();

    // Set up the first two obvious properties, the Sun LDAP factory and LDAP server
           env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.ldap.LdapCtxFactory");
           env.put(Context.PROVIDER_URL,   "ldap://wibble.example.com:389");
    // These two settings match the values from the CICS system
           env.put("com.ibm.ws.naming.ldap.containerdn",  "ibm-wsnTree=WASNaming,c=us");
           env.put("com.ibm.ws.naming.ldap.noderootrdn",
                 "ibm-wsnName=legacyroot,ibm-wsnName=PLEX2,ibm-wsnName=domainRoots");

    // Use the LDAPSNSLookup helper method to negotiate the WebSphere System Name
    // Space on wibble.example.com and locate our HelloWorld bean.  "samples"
    // is the JNDI prefix on the CICS CorbaServer that published the HelloWorld Bean.
           org.omg.CORBA.Object obj =
           CICSNameSpaceHelper.LDAPSNSLookup(env,"samples/HelloWorld");

        HelloWorldHome hhome =
                (HelloWorldHome)javax.rmi.PortableRemoteObject.narrow
                (obj,HelloWorldHome.class);

        System.out.println("HelloWorld home successfully found!");
        Hello hello = hhome.create();
          System.out.println(hello.sayHello());
     } catch (Exception e) {
       System.err.println("Exception while looking up and calling the HelloWorld bean:");
      e.printStackTrace();
   }
  }
}
```

You are using the SUN LDAP code, which understands the `providerURL` property, rather than the `masterurl` property used in the WebSphere Context Factory example.

The helper class `CICSNameSpaceHelper` may also work with other context factories. Notice that the syntax of the name passed to `LDAPSNSLookup` is JNDI syntax `a/b/c/d`.

Writing a client program to use COS Naming

The following example shows a client program, `Gambler.java`, that works with the `RouletteWheel` bean developed in "Coding a session bean" on page 290. When a bean reference is obtained from a COS Naming namespace, there are a number of operations that must be performed before the client can use that reference. These operations are the same for the majority of client programs, so they are collected in the utility class `EJBUtils`. This utility class is used by the client program `Gambler`.

EJBUtils.java

This is the implementation of the utility class, `EJBUtils`.

```
import javax.naming.*;
import java.util.Hashtable;

class EJBUtils {

  public static Object jndi_lookup(String name, Class resultClass) {

    // Set up environment for creating initial context
    Hashtable env = new Hashtable(11);

    // Define the nameserver - see note 1 below
    env.put(Context.PROVIDER_URL,
      "iiop://wibble.example.com:900");
```

```
    // Define the initial context factory -see note 2
    env.put(Context.INITIAL_CONTEXT_FACTORY,
      "com.sun.jndi.cosnaming.CNCtxFactory");

    try {

      // Create the initial context
      Context ctx = new InitialContext(env);

      // Lookup the object
      Object tempObject = ctx.lookup(name);

      // Narrow that to the requested class
      return javax.rmi.PortableRemoteObject.narrow(tempObject,resultClass);

    } catch (NamingException ne) {
      System.err.println("EJBUtils.jndi_lookup() failed:");
      ne.printStackTrace();
    }
    return null;
  }

}
```

Note:

1. Here we define the nameserver that will be used to lookup beans as "iiop://wibble.example.com:900". This value should be the name of your nameserver, and must match the **-Djava.naming.provider.url** that was defined in the CICS JVM properties file, so that the client looks up the bean on the same nameserver it was published into by CICS. See "Defining name servers" on page 182 for information about using name servers.
2. Here we define the initial context factory for your client environment. you should set it to the value required by your client environment. The example shows the value you would set when using the ORB included with the IBM SDK. If your client is a java application or enterprise bean running in CICS Transaction Server for z/OS, Version 2, then you should not specify an initial context factory here, but should allow it to default to `com.ibm.websphere.naming.wsnInitialContextFactory`.

Gambler.java

This is the implementation of the example client program, `Gambler.java`.

```
import org.omg.CORBA.ORB;
import java.io.*;
import casino.*;

public class Gambler {

  public static void main(String[] argv) {

    try {

      System.out.println("Gambler\n");

      System.out.println("Looking up RouletteWheel home");
      RouletteWheelHome wheelHome =
        (RouletteWheelHome)
        EJBUtils.jndi_lookup("cics/ejbs/RouletteWheel",
                             RouletteWheelHome.class);
        //
        // See Note 1.
        //
```

```
        System.out.println("Creating a new roulette wheel");
        RouletteWheel wheel = wheelHome.create();

        System.out.println("");
        System.out.println("Gambling $50 on red !");
        System.out.println(wheel.bet("red",50));

        System.out.println("");
        System.out.println("Gambling $20 on black !");
        System.out.println(wheel.bet("black",20));

        System.out.println("");
        System.out.println("Gambling $20 on red !");
        System.out.println(wheel.bet("red",20));

        System.out.println("");
        System.out.print("Collecting winnings:$");
        System.out.println(wheel.collectWinnings());


        System.out.println("");
        System.out.print("Removing the roulette wheel");
        wheel.remove();

      } catch (Exception e) {
        System.err.println("Error while gambling:");
        e.printStackTrace();
      }

    }

  }
```

Note:

1. The client program Gambler.java looks up the RouletteWheel at "cics/ejbs" in the namespace. This means the CORBASERVER in CICS into which you have installed the RouletteWheel bean must have a JNDI prefix of cics/ejbs. Once installed and published the RouletteWheel will then be accessible by the client program.
2. There is a `remove` call at the end of this client program. The roulette wheel bean is stateful and CICS manages the state of every instance. Unless `remove` is called when you finish operating with that bean instance then CICS will continue to store it. Bean timeout can be controlled using the SESSBEANTIME parameter of the CORBASERVER resource definition. This indicates to CICS how long it should manage instance state if no requests are coming in to utilize that instance, implementing a kind of garbage collection. However, it is good programming practice to call `remove` when you have finished working with an instance so that you do not depend on this type of garbage collection.

Using the client program

When compiling the client program, your classpath must be set carefully to include the deployed JAR file you successfully processed earlier with the CICS Jar Development Tool, and also the javax.ejb interfaces for EJB 1.1 support, which are available in `ejb11.jar` in the `standard/ejb/1_1` directory of the java installation.

Once compiled, run the client with:

```
java Gambler
```

Transaction interoperability with web application servers

A number of protocols exist to support distributed transactions. The CICS enterprise Java environment supports only the standard CORBA Object Transaction Service (OTS) protocol. However, some J2EE-compliant web application servers (such as WebSphere Version 4) either do not use this protocol, or do not use this protocol by default.

If objects on your web application server call CICS enterprise beans within the scope of existing transaction contexts, you must set up your web application server to use the CORBA OTS. If this is not possible, your web application server is not fully compatible with CICS enterprise Java support. (For a way of using the EJB Bank Account sample application to test whether your web application server is fully compatible with CICS enterprise Java support, see "A note about distributed transactions" on page 285.)

If your web application server is WebSphere Application Server Version 4, be aware that, by default, it does not use the standard CORBA OTS, but can be made to do so. If you have WebSphere objects that call CICS enterprise beans within the scope of existing transaction contexts, you must set up WebSphere to use the CORBA OTS. Versions of WebSphere Application Server from Version 5 onwards are not affected by this problem.

To force WebSphere Application Server to use the CORBA OTS:

1. At the WebSphere Administration Console, select the JVM settings tab.
2. Enter the following in the System Properties section:

   ```
   com.ibm.ejs.jts.ControlSet.interoperabilityOnly=true
   com.ibm.ejs.jts.ControlSet.nativeOnly=false
   ```

   Save your changes.
3. Restart the application server.

Working with EJB Handles, HomeHandles and EJBMetaData

The Enterprise JavaBeans specification describes how a session bean supports not only the methods defined on its remote interface but some additional methods.

- There are methods defined on the EJBHome interface, they are callable by a client wishing to:
  - obtain a "storable" reference to the home (a home handle), or
  - obtain the EJBMetaData for the bean type.

  .
- There are methods defined on the EJBObject interface, they are callable by a client wishing to:
  - obtain the home for the EJB, or
  - obtain a "storable" reference to the object itself (a handle).

The purpose of handles is that they are serializable, once a handle is obtained for a bean instance it can be serialized, perhaps to a flat file. If, sometime later, a program wants make calls against that same instance, it can deserialize the handle and start calling methods again. The implementations of the handles and the meta data class are product specific.

In CICS, the implementations of the three interfaces HomeHandle, Handle and EJBMetaData are:

- com.ibm.cics.portable.CICSSessionHomeHandle
- com.ibm.cics.portable.CICSSessionHandle
- com.ibm.cics.portable.CICSEJBMetaData

These implementations are included in the `CICSEJBClient.jar` JAR file, which can be downloaded from the `utils` subdirectory in the z/OSUNIX area where CICS is installed. This JAR file should be included in the CLASSPATH of any client program calling the special methods described above, to ensure it understands the types of object returned from the server. If, for example, its CLASSPATH does not include `CICSEJBClient.jar`, a client program that calls the **getEJBMetaData** function of an enterprise bean may be returned either of the following:

1. An exception
2. Null

The precise value returned depends on the implementation of the client's object request broker (ORB).

Using EDF with enterprise beans

To use EDF to test enterprise beans, you must perform these tasks.

About this task

- Set the CEDF parameter to YES in the PROGRAM resource definition for DFJIIRP that is supplied in group DFHIIOP.
- Set MAXACTIVE to one in TRANCLASS(DFHEDFTC).
- Activate EDF by entering CEDX (*transid*) at the terminal where the transaction will be trapped. The transid is either the default transid CIRP or the transaction specified on the RequestModel definition.
- Initiate the bean.

Bean-to-bean communication

If your bean uses bean-to-bean communication with the same transaction id within the same AOR, setting MAXACTIVE to one will result in the communication not working.

This is because the execution of the second transaction will be suspended waiting for a slot in which to execute, and the original bean will then experience a “timeout” condition. The way to avoid this is to take one of the following actions:

- Use REQUESTMODELs to specify a unique transaction id for each bean.
- Allow all create methods to use CIRP (the default transaction id), and use REQUESTMODELs to define a unique transaction id for each set of business methods.

Note: When a bean is running inside a request processor, CICS will only utilize requestmodels (and therefore start a new CICS transaction under the new transaction ID) if a remote method call made by that bean cannot be satisfied in the current request processor. A method call cannot be satisfied locally in the current request processor if:

- The transaction attributes of the method being called require a different transaction context
- The bean being called is in a different CorbaServer

Chapter 23. Deploying enterprise beans

This section explains the process of deploying enterprise beans into a CICS EJB server in more detail.

The concept of deployment is introduced in “Deploying enterprise beans—overview” on page 225.

The term “deployment” used in the EJB specification describes a series of tasks that makes the enterprise beans in one or more JAR files available for use in a specific operating environment (in this case, a CICS EJB server).

The deployment tools for enterprise beans in a CICS system

CICS supplies three tools to assist you in deploying enterprise beans into a CICS EJB server.

- “The Assembly Toolkit (ATK)”
- “The resource manager for enterprise beans”
- “CREA” on page 304

The Assembly Toolkit (ATK)

The Assembly Toolkit (ATK) is a general tool used by several IBM EJB servers, including CICS, to build JAR files ready for the runtime environment.

The Assembly Toolkit for Windows is supplied with WebSphere Application Server Version 5.1 and later. (The Application Assembly Tool (AAT), provided with WebSphere Application Server Version 4 and early copies of WebSphere Application Server Version 5.0, can still be used but is not supported).

For detailed information about using ATK, see The enterprise bean deployment tool, ATK, in the *CICS Operations and Utilities Guide*.

The resource manager for enterprise beans

The resource manager for enterprise beans is a web-based tool that enables you to perform certain operations on the resources (CORBASERVERs and DJARs) installed into CICS to support the use of enterprise beans.

The tool can also be used for EJB-related problem diagnosis, because it offers the ability to view any errors associated with DJAR definitions, and indicates if the beans in a deployed JAR file have been published to the naming service.

The tool enables you to perform common tasks without having to use a CICS terminal.

For a full description of the resource manager for enterprise beans, see The Resource Manager for Enterprise Beans, in the *CICS Operations and Utilities Guide*.

CREA

CREA is a CICS-supplied transaction that enables the system programmer (usually with help from the application programmer) to create REQUESTMODEL definitions for the beans in an installed deployed JAR file.

CREA can install definitions into a running CICS system by using **EXEC CICS CREATE** commands, or can write the definitions to the CSD.

CREC is a read only version of CREA. It offers inspection facilities without giving the ability to make changes.

For full descriptions of CREA and CREC, see CREA - create REQUESTMODELs for enterprise beans, in the *CICS Supplied Transactions* manual.

CREA and CREC can be used without needing to access a 3270 terminal. For details of such access, see Connecting CICS to the Web, in the *CICS Internet Guide*.

Using CICS deployment tools for enterprise beans

To develop and deploy a bean into CICS, an application developer, working with a CICS system programmer in the later stages, has to carry out a number of steps.

About this task

Develop the bean and make it deployable
Develop the bean and package it into a JAR file. The bean can be written and tested using your choice of tooling.

Note: The JAR file may contain the Java classes for one or for several enterprise beans. Typically a JAR file used in a CICS EJB server contains several enterprise beans.
After the bean has been packaged in a JAR file, use ATK to make it deployable. For a short introduction to ATK and a reference to further information, see The enterprise bean deployment tool, ATK, in the *CICS Operations and Utilities Guide*.

Store in z/OS UNIX pickup directory
Store a copy of the deployable JAR file in the z/OS UNIX pickup directory of the CorbaServer in which you want to run the bean. You can do this using FTP, NFS, or SMB. If the z/OS UNIX directory can be mounted on your workstation, this process can be integrated into the previous JAR file creation process.

Scan the pickup directory
Using either CEMT or the resource manager for enterprise beans, initiate a scan of the pickup directory. (For a description of the resource manager for enterprise beans, see The Resource Manager for Enterprise Beans, in the *CICS Operations and Utilities Guide*.) CICS creates and installs a DJAR definition for the deployed JAR file in the pickup directory.

After the pickup directory has been scanned, you can view the state of the new DJAR definition to determine if the deployed JAR file is ready for use.

If the deployed JAR file is not ready for use, the cause of the error can be determined and in most cases corrected by an application developer without the need for a system programmer to become involved.

Publish

Publish a reference to the home interface of each bean in the deployed JAR file to an external namespace. The namespace is accessible to clients through JNDI.

If you specify AUTOPUBLISH(YES) on the CORBASERVER definition, the beans in a deployed JAR file are automatically published to the namespace when the DJAR definition is successfully installed into the CorbaServer. Alternatively, you can issue a PERFORM CORBASERVER PUBLISH or PERFORM DJAR PUBLISH command.

The Resource Manager for enterprise beans (see The Resource Manager for Enterprise Beans, in the *CICS Operations and Utilities Guide*) indicates if the "autopublish" feature is on or off.

Ensure any additional classes are on class paths

For enterprise beans, you do not need to add the deployed JAR files to the class paths in the JVM profile or JVM properties file. CICS manages the loading of the classes included in these files by means of the DJAR definitions. However, if your enterprise beans use any classes, such as classes for utilities, that are **not** included in the deployed JAR file, you do need to include these classes on the class path that will be used by the JVM for the request processor program. "Enabling applications to use a JVM" on page 142 tells you how to do this.

Unit Test

Once the beans in the deployed JAR file have been published to the naming server, the application programmer can unit test them in the CICS environment.

System Test

When the beans are ready for system testing, an application programmer can work with a system programmer to consider if any REQUESTMODEL definitions are needed. Use the CICS-supplied transaction CREA to generate REQUESTMODEL definitions. (For a description of CREA, see CREA - create REQUESTMODELs for enterprise beans, in the *CICS Supplied Transactions* manual.)

You can identify the beans and bean methods from the application. Your system programmer can associate the bean methods with transaction IDs by causing the optimum set of REQUESTMODEL definitions to be generated. Running different beans under different transaction IDs is useful, for example, for workload-management purposes, and for gathering effective monitoring and statistical information.

Install in production environment

To move from a system test to a production environment:

1. Use ATK to verify that the container bindings for resources and references that have been set in the deployment descriptor of each JAR file are appropriate for your production environment.
2. If you have set the DJARDIR parameter in your production region CORBASERVER definition to identify a pickup directory:
   a. Store the deployable JAR file in the pickup directory of the CorbaServer.
   b. Install the CORBASERVER definition.
   c. A suitable DJAR definition is produced.
3. If not:

a. Store the deployable JAR file in the z/OS UNIX directory that you intend to use in the production region.
b. Install the production CORBASERVER definition.
c. Create and install a DJAR definition equivalent to that which you had in your test region, using whatever process you would normally use in your installation.

4. If you have set the AUTOPUBLISH(YES) parameter in your production region CORBASERVER definition:
   a. The beans in the deployed JAR file is automatically published to the namespace when the DJAR definition is successfully installed into the CorbaServer.
5. If not:
   a. Publish the beans to the JNDI server that you use for production using CEMT PERFORM CORBASERVER PUBLISH or CEMT PERFORM DJAR PUBLISH.
6. Transfer REQUESTMODEL definitions from the test region CSD to the production CSD using the process that you normally use in your installation.
7. Ensure that any additional classes, such as classes for utilities, that are not included in the deployed JAR files for your enterprise beans, are present on the standard class path.

Note: If you want to update enterprise beans in a production region, see Chapter 24, “Updating enterprise beans in a production region,” on page 307.

Chapter 24. Updating enterprise beans in a production region

This section considers how best to update enterprise beans in a production region. It contains the following topics:

The problem

How do you update enterprise beans in a running CICS production region, while causing the minimum disruption to the current workflow and without recycling CICS?

It is simple enough to introduce *new* enterprise beans into a running EJB server without disrupting the current workflow. You can do either of the following:

1. Use the CICS scanning mechanism. That is, place the deployed JAR file containing the new beans into a CorbaServer's deployed JAR file ("pickup") directory and issue a **PERFORM CORBASERVER SCAN** command. Repeat on all the AORs in the logical EJB server. If the CORBASERVER definition specifies AUTOPUBLISH(NO), on one of the AORs issue a **PERFORM DJAR PUBLISH** command.

   Note: If you use the scanning mechanism in a production region, be aware of the security implications: specifically, the possibility of CICS command security on DJAR definitions being circumvented. To guard against this, we recommend that user IDs given write access to the z/OS UNIX deployed JAR file directory should be restricted to those given RACF authority to create and update DJAR and CORBASERVER definitions.
2. Use an **EXEC CICS CREATE DJAR** command to install a definition of the deployed JAR file which contains the new beans. Repeat on all the AORs in the logical EJB server. On one of the AORs, issue a **PERFORM DJAR PUBLISH** command.

Unfortunately, because of the unpredictable effects on in-flight transactions, you can't use these methods to *update* beans in an active EJB server. You would have no way of controlling which version of a bean, the old or the new, was used by successive method calls. (Because of timing differences, the problem could well be exacerbated in a multi-region EJB server.)

An alternative approach would be to quiesce and shut down CICS, then restart it with the updated DJAR definitions in place. While this is acceptable in a test environment, it is not an attractive solution for a production region. Consider Figure 27 on page 309. Imagine that you want to update `bean5` and `bean6` in CorbaServer `COR2`. If you were to close down CICS, not only would `bean5` and `bean6` be unavailable during the shutdown, but also all the beans in CorbaServer `COR1`.

What if your EJB server contains several AORs, with workload management being used to balance requests across them? Could you not then shut down and upgrade each AOR in turn, with a minimal effect on performance? Unfortunately not, because:

- During the upgrade process, different AORs would have different versions of the beans. Unless the new versions of the beans were completely backward-compatible with the old versions, this would cause unpredictable effects. ("Completely backward-compatible" means that, among other things, the home and component interfaces of the two versions must be identical, and the state of any stateful session beans must be preserved.)
- Shutting down even one AOR would inevitably degrade the performance of the EJB server to some extent. (If the upgrade is an important one, this might be acceptable. To compensate for the degraded performance you could, perhaps, add an extra AOR to your EJB server.)

The rest of this chapter discusses what you need to do on a CICS EJB *server* to update enterprise beans in production regions. Note that changes may also be required on the *client* side. In particular, if, due to an update, the home or component interface of an enterprise bean changes, before any client applications can use the updated bean they must be rewritten to use the new interface.

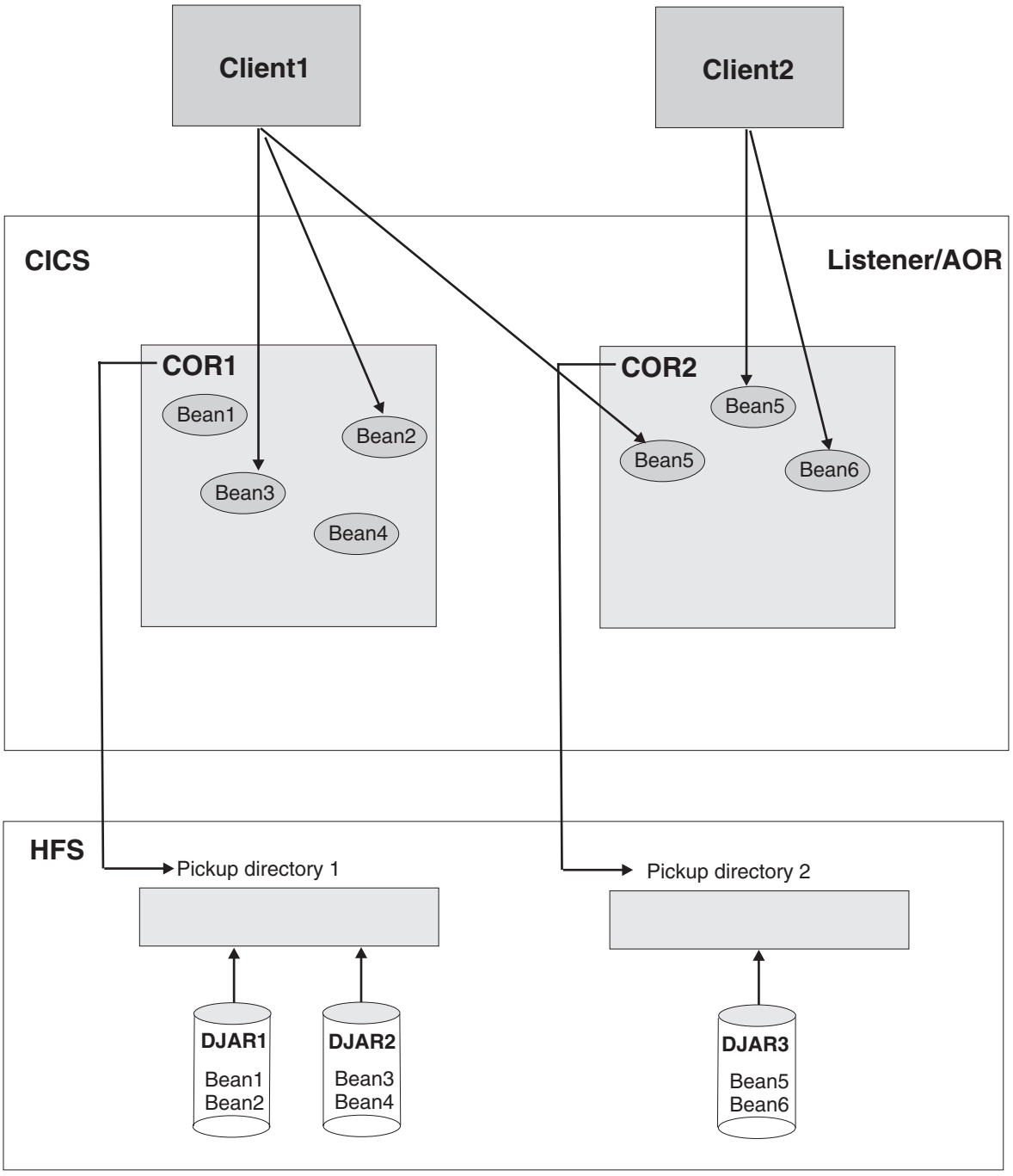

Figure 27. A CICS EJB production region. The clients are invoking bean methods in CorbaServers COR1 and COR2.

You are recommended to divide beans between CorbaServers based on the beans' maintenance and availability requirements.

Possible solutions

Here are some suggested solutions for our problem of how best to update beans in a production region. The solutions offered depend on whether your EJB server consists of a single listener/AOR or of multiple listeners and AORs.

As a general rule, upgrade solutions will be easier to implement if you:

1. Divide your enterprise beans between CorbaServers based not only on the beans' functions but also on their maintenance and availability requirements. That is, sets of beans that have distinct maintenance and availability requirements should be installed in distinct CorbaServers.
2. Allocate CICS transaction IDs to enterprise bean methods based not only on the beans' functions but also on their maintenance and availability requirements. That is, for ease of maintenance sets of beans that have distinct maintenance and availability requirements should run under distinct CICS transaction IDs.

   Important:

   a. In a multi-region EJB server, if your AORs contain multiple CorbaServers you are strongly advised to assign different sets of transaction IDs to the objects supported by each CorbaServer. That is, each CorbaServer in an AOR should support a different set of transaction IDs.

   b. This makes it easier for the distributed routing program to route around a disabled CorbaServer, while keeping available any other, enabled, CorbaServers in the region. For further information about how to code a distributed routing program to deal with a disabled CorbaServer, see the *CICS Customization Guide*.

   Note: The CICS transaction under which a bean method runs is specified on the REQUESTMODEL definition that matches the method. You can use the CREA CICS-supplied transaction to:
   - Display the transaction IDs associated with particular beans and bean methods
   - Change the transaction IDs, apply the changes, and save the changes to new REQUESTMODEL definitions

Solutions for a single listener/AOR

These solutions are valid for an EJB server consisting of a single listener/AOR.

Let us assume that, in Figure 27 on page 309, you want to update `bean5` and `bean6` in CorbaServer `COR2`. `DJAR3.jar` is the deployed JAR file containing the beans to be updated. You require:

1. CorbaServer `COR1` and its beans to remain available throughout the upgrade process.
2. If possible, the upgrade to the beans in CorbaServer `COR2` to be seamless. That is, there should be no time (or, at least, the smallest possible period of time) during which it is impossible to create a new instance of `bean5` or `bean6`.

Solution 1

About this task

The advantage of this solution is that it is relatively easy to implement. The disadvantage is that it is not seamless—that is, there is a period (while instances of the old versions of `bean5` and `bean6` are being destroyed or passivated) during which it is impossible to create a new instance of `bean5` or `bean6`.

1. Issue an **EXEC CICS SET CORBASERVER**(COR2) ENABLESTATUS(DISABLED) command or a **CEMT SET CORBASERVER**(COR2) DISABLED command. Any attempts to create new instances of bean5 or bean6, regardless of whether the clients have references to the beans' home interfaces, will fail.

   Typically, currently-executing methods on instances of bean5 and bean6 will proceed to completion.

   An instance of bean5 or bean6 that is not participating in an OTS transaction is destroyed or passivated at the end of the currently-executing method. (If there is no currently-executing method, all instances will already have been destroyed or passivated.)

   Note: *Stateless* session beans are destroyed. *Stateful* session beans are passivated.

   An instance of bean5 or bean6 that *is* participating in an OTS transaction is not destroyed or passivated until the end of the OTS transaction; typically, any future method calls against this instance (within the scope of the OTS transaction) will succeed. At the end of the OTS transaction the instance is destroyed or passivated.
2. Check when all instances of bean5 and bean6 have been destroyed or passivated by issuing **EXEC CICS** or **CEMT INQUIRE CORBASERVER**(COR2) ENABLESTATUS commands. A status of DISABLED indicates that all bean instances have been destroyed or passivated.
3. When all instances of bean5 and bean6 have been destroyed or passivated, install the updated version of the DJAR3.jar deployed JAR file, using either the CICS scanning mechanism or a static DJAR definition. (You cannot use the scanning mechanism to update a static DJAR definition.)

   Either:

   a. Put the new version of the DJAR3.jar deployed JAR file into CorbaServer COR2's pickup directory.
   b. Issue a **PERFORM CORBASERVER**(COR2) SCAN command. CICS scans COR2's pickup directory, installs the new definition of DJAR3.jar, and copies the new versions of bean5 and bean6 to COR2's shelf directory.

   or:

   a. Issue an **EXEC CICS** or **CEMT DISCARD DJAR** (DJAR3) command, to remove the current definition of DJAR3.jar from CICS.
   b. Issue a **CEDA INSTALL DJAR**(DJAR3) or an **EXEC CICS CREATE DJAR**(DJAR3) CORBASERVER(COR2) HFSFILE(new_version_of_DJAR3.jar_on_HFS) command. CICS installs the new definition of DJAR3.jar, and copies the new versions of bean5 and bean6 to COR2's shelf directory.

   Note:

   a. It is *not* necessary to re-publish the updated versions of bean5 and bean6 to the namespace, even if the home or component interfaces of the beans have changed since the previous version.
   b. If the home or component interface of bean5 or bean6 has changed since the previous version, before using the changed bean client applications must be updated to use the new signature.
   c. If you update a *stateful* session bean, depending on exactly what changes are made you may change the structure of its serialised state. If this happens, you will invalidate any passivated instances of the bean in the

object store. If this happens, any attempts to use the now invalidated bean will result in an exception. You should code your client applications to cope with this possibility.

4. Issue a **CEMT SET CORBASERVER**(COR2) ENABLED command. *From this moment, all new work will use the updated versions of* bean5 *and* bean6.

Solution 2

This solution requires CICSPlex System Manager. All CICS applications on your listener/AOR must be suitable for cloning across multiple regions.

About this task

The advantage of this solution is that, unlike solution 1, it is relatively seamless—that is, there should at worst be only a tiny period during which it is impossible to create a new instance of bean5 or bean6. The disadvantage is that it is more complicated to implement than solution 1.

1. Using CICSPlex SM:
   a. Clone your single listener/AOR.
   b. Direct all new workload to the clone—that is, quiesce the original AOR and activate the clone. For information on how to do this, see Balancing an enterprise bean workload, in the *CICSPlex System Manager Managing Workloads* manual.

      All requests for bean methods that will run under a new OTS transaction, or under no OTS transaction, whether in COR1 or COR2, are routed to the clone.

      Requests for bean methods that will run under an existing OTS transaction (whether in COR1 or COR2) are routed to the original region.

      Note:

      1) By "a *new* OTS transaction" we mean an OTS transaction in which the bean's participation begins *after* all new work is directed to the clone.
      2) By "an *existing* OTS transaction" we mean an OTS transaction in which the bean's participation began *before* all new work was directed to the clone.

      On the original region:

      - An instance of an enterprise bean that is not participating in an OTS transaction is destroyed or passivated at the end of the currently-executing method. (If there is no currently-executing method, all instances will already have been destroyed or passivated.)
      - An instance of an enterprise bean that *is* participating in an OTS transaction is not destroyed or passivated until the end of the OTS transaction; typically, any future method calls against this instance (within the scope of the OTS transaction) will succeed. At the end of the OTS transaction the instance is destroyed or passivated.

2. On the original region:
   a. Check when all instances of bean1 through bean6 have been destroyed or passivated:
      1) If you don't already know the CICS transaction ID or IDs associated with bean1 through bean6, use the CREC transaction to display this information.
      2) Use the **INQUIRE TASK** command to check whether any instances of these transactions are running.

b. When all instances of bean1 through bean6 have been destroyed or passivated, install the updated version of the DJAR3.jar deployed JAR file, using either the CICS scanning mechanism or a static DJAR definition. (You cannot use the scanning mechanism to update a static DJAR definition.)

   Either:

   1) Put the new version of the DJAR3.jar deployed JAR file into CorbaServer COR2's pickup directory.
   2) Issue a **PERFORM CORBASERVER**(COR2) SCAN command. CICS scans COR2's pickup directory, updates its definition of DJAR3.jar, and copies the new versions of bean5 and bean6 to COR2's shelf directory.

   or:

   1) Issue a **CEMT DISCARD DJAR**(DJAR3) command to delete the old definition of DJAR3.jar.
   2) Issue a **CEDA INSTALL DJAR**(DJAR3) or an **EXEC CICS CREATE DJAR**(DJAR3) CORBASERVER(COR2) HFSFILE(new_version_of_DJAR3.jar_on_HFS) command. CICS installs the new definition of DJAR3.jar, and copies the new versions of bean5 and bean6 to COR2's shelf directory.

   Note:

   1) It is *not* necessary to re-publish the updated versions of bean5 and bean6 to the namespace, even if the home or component interfaces of the beans have changed since the previous version.
   2) If the home or component interface of bean5 or bean6 has changed since the previous version, before using the changed bean client applications must be updated to use the new signature.
   3) If you update a *stateful* session bean, depending on exactly what changes are made you may change the structure of its serialised state. If this happens, you will invalidate any passivated instances of the bean in the object store. If this happens, any attempts to use the now invalidated bean will result in an exception. You should code your client applications to cope with this possibility.

3. Using CICSPlex SM, direct all new workload to the original region—that is, quiesce the clone and activate the original region.

   All requests for bean methods that will run under a new OTS transaction, or under no OTS transaction, whether in COR1 or COR2, are now routed to the original region. *From this moment, all new work will use the updated versions of bean5 and bean6*. Requests for bean methods that will run under an existing OTS transaction (whether in COR1 or COR2) continue to be routed to the clone.

   Note:

   a. By "a *new* OTS transaction" we mean an OTS transaction in which the bean's participation begins *after* all new work is redirected to the original region.
   b. By "an *existing* OTS transaction" we mean an OTS transaction in which the bean's participation began *before* all new work was redirected to the original region.

   Eventually, all instances of enterprise beans on the clone will be destroyed or passivated, as described above.

4. On the clone region, use the **INQUIRE TASK** command to check when all instances of bean1 through bean6 have been destroyed or passivated. When this has happened, you can discard the clone region.

Solutions for a multi-region EJB server

These solutions are valid for an EJB server consisting of one or more listener regions and multiple, identical, AORs.

Assume that your EJB server consists of three identical listener regions and five identical AORs. Each of the AORs is a clone of the region shown in Figure 27 on page 309 (except that it is an AOR rather than a listener/AOR). All the AORs share the same pickup directories, and the same sets of enterprise beans are deployed on each, in identical CorbaServers named `COR1` and `COR2`.

You want to update `bean5` and `bean6` in logical CorbaServer `COR2`. `DJAR3.jar` is the deployed JAR file containing the beans to be updated.

You require:

1. Logical CorbaServer `COR1` and its beans to remain available throughout the upgrade process.
2. If possible, the upgrade to the beans in logical CorbaServer `COR2` to be seamless. That is, there should be no time (or, at least, the smallest possible period of time) during which it is impossible to create a new instance of `bean5` or `bean6`.

Solution 1

This solution is a development of solution 1 for a single-region.

About this task

Its advantage is that it is relatively easy to implement. Its disadvantage is that it is not seamless—that is, there is a period (while instances of the old versions of `bean5` and `bean6` are being destroyed or passivated) during which it is impossible to create a new instance of `bean5` or `bean6`.

1. On each of the AORs, issue an **`EXEC CICS SET CORBASERVER`**`(COR2)` ENABLESTATUS(`DISABLED`) or a **`CEMT SET CORBASERVER(`**`COR2`) DISABLED command. On all the AORs:
   - Any attempts to create new instances of `bean5` or `bean6`, regardless of whether the clients have references to the beans' home interfaces, will fail.
   - Typically, currently-executing methods on instances of `bean5` and `bean6` will proceed to completion.
   - An instance of `bean5` or `bean6` that is not participating in an OTS transaction is destroyed or passivated at the end of the currently-executing method. (If there is no currently-executing method, all instances will already have been destroyed or passivated.)
   - An instance of `bean5` or `bean6` that *is* participating in an OTS transaction is not destroyed or passivated until the end of the OTS transaction; typically, any future method calls against this instance (within the scope of the OTS transaction) will succeed. At the end of the OTS transaction the instance is destroyed or passivated.
2. On each of the AORs, check when all instances of `bean5` and `bean6` have been destroyed or passivated by issuing **`EXEC CICS`** or **`CEMT INQUIRE CORBASERVER`**`(COR2)` ENABLESTATUS commands. A status of DISABLED indicates that all bean instances have been destroyed or passivated.
3. When all instances of `bean5` and `bean6`, on all the AORs, have been destroyed or passivated, install the updated version of the `DJAR3.jar` deployed JAR file, using either the CICS scanning mechanism or static DJAR definitions. (You cannot use the scanning mechanism to update static DJAR definitions.)

Either:

a. Put the new version of the DJAR3.jar deployed JAR file into CorbaServer COR2's pickup directory (which is shared by all the AORs).

b. On each of the AORs, issue a **PERFORM CORBASERVER**(COR2) SCAN command. The AOR scans COR2's pickup directory, installs the new definition of DJAR3.jar, and copies the new versions of bean5 and bean6 to COR2's shelf directory.

or, on each of the AORs:

a. Issue an **EXEC CICS** or **CEMT DISCARD DJAR**(DJAR3) command, to remove the current definition of DJAR3.jar from CICS.

b. Issue a **CEDA INSTALL DJAR**(DJAR3) or an **EXEC CICS CREATE DJAR**(DJAR3) CORBASERVER(COR2) HFSFILE(new_version_of_DJAR3.jar_on_HFS) command. CICS installs the new definition of DJAR3.jar, and copies the new versions of bean5 and bean6 to COR2's shelf directory.

Note:

a. It is *not* necessary to re-publish the updated versions of bean5 and bean6 to the namespace, even if the home or component interfaces of the beans have changed since the previous version.

b. If the home or component interface of bean5 or bean6 has changed since the previous version, before using the changed bean client applications must be updated to use the new signature.

c. If you update a *stateful* session bean, depending on exactly what changes are made you may change the structure of its serialised state. If this happens, you will invalidate any passivated instances of the bean in the object store. If this happens, any attempts to use the now invalidated bean will result in an exception. You should code your client applications to cope with this possibility.

4. On each of the AORs, issue a **CEMT SET CORBASERVER**(COR2) ENABLED command. *From this moment, all new work will use the updated versions of bean5 and bean6*.

Solution 2
About this task

This solution requires CICSPlex System Manager. It is a development of solution 2 for a single-region. Its advantage is that it is relatively seamless—that is, there should at worst be only a tiny period during which it is impossible to create a new instance of bean5 or bean6. Its disadvantage is that it is more complicated to implement than solution 1.

1. Using CICSPlex SM:

   a. Create clones of all your AORs.

   b. Direct all new workload to the clones—that is, quiesce the original AORs and activate the clones. For information on how to do this, see Balancing an enterprise bean workload, in the *CICSPlex System Manager Managing Workloads* manual.

      Each request for a bean method that will run under a new OTS transaction, or under no OTS transaction, whether in COR1 or COR2, is routed to one or other of the clones.

      Each request for a bean method that will run under an existing OTS transaction (whether in COR1 or COR2) is routed to the appropriate original AOR.

Note:

1) By "a *new* OTS transaction" we mean an OTS transaction in which the bean's participation begins *after* all new work is directed to the clones.
2) By "an *existing* OTS transaction" we mean an OTS transaction in which the bean's participation began *before* all new work was directed to the clones.
3) By "the *appropriate* original AOR" we mean the original AOR containing the request processor for the OTS transaction.

2. On each of the original AORs:

   Check when all instances of `bean1` through `bean6` have been destroyed or passivated:

   a. If you don't already know the CICS transaction ID or IDs associated with `bean1` through `bean6`, use the CREC transaction to display this information.
   b. Use the **INQUIRE TASK** command to check whether any instances of these transactions are running.

3. When all instances of `bean1` through `bean6`, on all the original AORs, have been destroyed or passivated, install the updated version of the `DJAR3.jar` deployed JAR file, using either the CICS scanning mechanism or static DJAR definitions. (You cannot use the scanning mechanism to update static DJAR definitions.)

   Either:

   a. Put the new version of the `DJAR3.jar` deployed JAR file into `COR2`'s pickup directory (which is shared by all the original AORs).
   b. On each of the original AORs, issue a **PERFORM CORBASERVER**(`COR2`) SCAN command. The AOR scans `COR2`'s pickup directory, updates its definition of `DJAR3.jar`, and copies the new versions of `bean5` and `bean6` to `COR2`'s shelf directory.

   or:

   a. On each of the original AORs, issue a **CEMT DISCARD DJAR**(`DJAR3`) command to delete the old definition of `DJAR3.jar`.
   b. On each of the original AORs, issue a **CEDA INSTALL DJAR**(`DJAR3`) or an **EXEC CICS CREATE DJAR**(`DJAR3`) CORBASERVER (`COR2`) HFSFILE(`new_version_of_DJAR3.jar_on_HFS`) command. CICS installs the new definition of `DJAR3.jar`, and copies the new versions of `bean5` and `bean6` to `COR2`'s shelf directory.

   Note:

   a. It is *not* necessary to re-publish the updated versions of `bean5` and `bean6` to the namespace, even if the home or component interfaces of the beans have changed since the previous version.
   b. If the home or component interface of `bean5` or `bean6` has changed since the previous version, before using the changed bean client applications must be updated to use the new signature.
   c. If you update a *stateful* session bean, depending on exactly what changes are made you may change the structure of its serialised state. If this happens, you will invalidate any passivated instances of the bean in the object store. If this happens, any attempts to use the now invalidated bean will result in an exception. You should code your client applications to cope with this possibility.

4. Using CICSPlex SM, direct all new workload to the original AORs—that is, quiesce the clones and activate the original AORs.

All requests for bean methods that will run under a new OTS transaction, or under no OTS transaction, whether in COR1 or COR2, are now routed to the original AORs. *From this moment, all new work will use the updated versions of bean5 and bean6*. Requests for bean methods that will run under an existing OTS transaction (whether in COR1 or COR2) continue to be routed to the clones.

Note:

a. By "a *new* OTS transaction" we mean an OTS transaction in which the bean's participation begins *after* all new work is redirected to the original AORs.

b. By "an *existing* OTS transaction" we mean an OTS transaction in which the bean's participation began *before* all new work was redirected to the original AORs.

Eventually, all instances of enterprise beans on the clones will be destroyed or passivated.

5. On each of the clones, use the **INQUIRE TASK** command to check when all instances of bean1 through bean6 have been destroyed or passivated. When this has happened, you can discard the clone.

Other possible solutions

The solutions described in "Solutions for a single listener/AOR" on page 310 and "Solutions for a multi-region EJB server" on page 314 are not the only possibilities. Another approach, for example, is to:

1. Use non-default TRANIDs for the request processors associated with the beans to be updated. (In other words, segregate your enterprise beans by CorbaServer and transaction ID in the way previously suggested.)
2. Disable the request processor transactions, or put the transactions into a transaction class and reduce the TCLASS limit to zero.
3. When all instances of the beans have been destroyed or passivated, install the updated versions of the deployed JAR files in one of the ways described for the other solutions.

Chapter 25. The CCI Connector for CICS TS

The CCI Connector for CICS TS helps you to build Enterprise JavaBean (EJB) server components that make use of existing CICS programs.

Overview of the CCI Connector for CICS TS

The CCI Connector for CICS TS helps you to build Enterprise JavaBean (EJB) server components that make use of existing CICS programs.

The background—connectors

Frequently, new Java applications can be developed more quickly and reliably by harnessing the power of existing (non-Java) CICS programs.

A **CICS connector** is a software component that allows a Java client application to invoke a CICS application. Typically, the Java client programs that use a CICS connector are servlets.

For several releases, CICS has supported CICS connectors that enable a Java client program, *running outside CICS* (on, for example, Windows, UNIX, or native z/OS), to connect to a specified program on a CICS server. The CCI Connector for CICS TS enables a Java program or enterprise bean *running on CICS Transaction Server for z/OS* to link to a CICS server program.

The CCI Connector for CICS TS implements the industry-standard **Common Client Interface (CCI)** defined by the J2EE Connector Architecture Specification, Version 1.0.

Note: The CICS Connector for CICS TS, introduced in CICS TS for z/OS, Version 2.1, is no longer supported. Unlike the CCI Connector for CICS TS, the CICS Connector for CICS TS implemented a non-standard, IBM-proprietary, client interface. For advice on upgrading existing applications that use the CICS Connector for CICS TS to use the CCI Connector for CICS TS instead, see "Upgrading from the CICS Connector for CICS TS to the CCI Connector for CICS TS" on page 334.

The Common Client Interface

This section presents an overview of the Common Client Interface. For definitive information about the interface, see the J2EE Connector Architecture Specification, Version 1.0, which you obtain from `java.sun.com/j2ee/download.html`.

The Common Client Interface (CCI) is part of the J2EE Connector architecture. The CCI provides a standard interface that allows developers to communicate with any number of Enterprise Information Systems (EISs) through their specific resource adapters, using a generic programming style. The CCI is closely modeled on the client interface used by Java Database Connectivity (JDBC), and is similar in its use of *Connections* and *Interactions*.

Within the CCI, there are two distinct types of class: for convenience, we shall call them *framework* classes and *input/output* classes.

Framework classes

Framework classes are used to request a connection to an EIS such as CICS, and execute commands on the EIS, passing input and retrieving output.

The framework classes are:

ConnectionFactory
A ConnectionFactory object is used to manufacture connections that a Java component can use to communicate with a specific EIS. Attributes of the ConnectionFactory specify the EIS for which connections can be created. A ConnectionFactory is the factory for a Connection object.

Connection
A Connection object identifies a unique connection to a specific server. It is the factory for an Interaction object.

Interaction
The execute method of an Interaction object allows you to drive an interaction with a server. In CICS TS, the execute method takes three arguments—an InteractionSpec object that specifies the type of interaction, and two Record objects that carry the input and output data.

J2EE components use the framework classes to acquire a connection to an EIS and to send and receive data. First, a J2EE component obtains a ConnectionFactory object for the particular EIS that is to be accessed—for example, CICS. (The component may manufacture the ConnectionFactory programatically or, more likely, look it up in a JNDI namespace.) It uses the ConnectionFactory to get a Connection object. Then it uses the **Connection** object to create one or more Interaction objects. It executes commands on the EIS through these Interaction objects.

Figure 28 shows the CCI framework classes being used to connect to an EIS and execute a command.

```
ConnectionFactory cf = <Lookup from JNDI namespace>
Connection conn = cf.getConnection();
Interaction int = conn.createInteraction();
int.execute(<Input output data>);
int.close();
conn.close();
```

Figure 28. Using the CCI framework classes to connect to an EIS and execute a command

Input/output classes

Using the framework classes gives a generic way of accessing an EIS by means of a J2EE resource adapter.

However, because every EIS has different input and output needs, the CCI interfaces provide a way for J2EE components to pass EIS-specific information to a J2EE resource adapter. The following types of object are used for this purpose by a J2EE component:

- ConnectionSpec objects
- InteractionSpec objects
- Record objects

ConnectionSpec
A ConnectionSpec object can be used to specify security attributes (such as userid and password) used in an interaction with a server.

Note: CICS ignores any security settings specified in a ConnectionSpec object, because it has already established a suitable security context for the connector.

The CCI Connector for CICS TS's ConnectionSpec class is called ECIConnectionSpec.

InteractionSpec
An InteractionSpec object holds essential attributes necessary for an interaction with a server—for example, the name of the target program. It is passed as a required argument on an **Interaction.execute()** method call when a particular interaction is to be carried out.

The CCI Connector for CICS TS's InteractionSpec class is called ECIInteractionSpec.

Record
Record objects are beans that hold the data exchanged with the target program—you can think of them as the equivalent of CICS communication areas (COMMAREAs). The data is accessible through Record-defined interfaces.

Figure 29 shows the CCI framework classes and input/output classes being used together to connect to an EIS, pass EIS-specific input/output parameters, and execute a command.

```
ConnectionFactory cf = <Lookup from JNDI namespace>
ECIConnectionSpec cs = new ECIConnectionSpec();
cs.setXXX();     //Set any connection specific properties

Connection conn = cf.getConnection(cs);
Interaction int = conn.createInteraction();
ECIInteractionSpec is = new ECIInteractionSpec();
is.setXXX();    //Set any interaction specific properties

RecordImpl in = new RecordImpl();
RecordImpl out = new RecordImpl();
int.execute(is,in,out);
int.close();
conn.close();
```

Figure 29. Complete CCI interaction with an EIS

The CCI Connector for CICS TS

The CICS Transaction Gateway includes an External Call Interface (ECI) resource adapter for CICS.

The **ECI resource adapter** provides standard CCI interfaces that enable J2EE components to call CICS server programs, using data areas (COMMAREAs) to pass information to and from the server. Typically, these J2EE components are servlets or enterprise beans; in all cases, they execute outside CICS.

CICS TS includes the CCI Connector for CICS TS, which provides standard CCI interfaces that enable Java programs and components (for example, enterprise beans) running *within CICS* to call CICS server programs.

A Java program or enterprise bean running on CICS TS can use the CCI Connector for CICS TS to link to a suitable CICS server program. The CICS server program:

- May be written in any of the CICS-supported languages
- Must use a suitable communications area (COMMAREA)

- Must not do any terminal input/output
- Typically, runs on a separate back-end CICS Transaction Server for z/OS region, but optionally may be on the same CICS region as the Java program or bean.

The connector uses a JCICS `Program.link()` call to access the back-end server program. Link and distributed program link (DPL) calls are supported. This scenario is shown in Figure 30. In this example, a Java client application or servlet uses RMI-IIOP to create an instance of an enterprise bean in a CICS EJB server. The enterprise bean uses the CCI Connector for CICS TS to link to a server program on a back-end CICS Transaction Server for z/OS region.

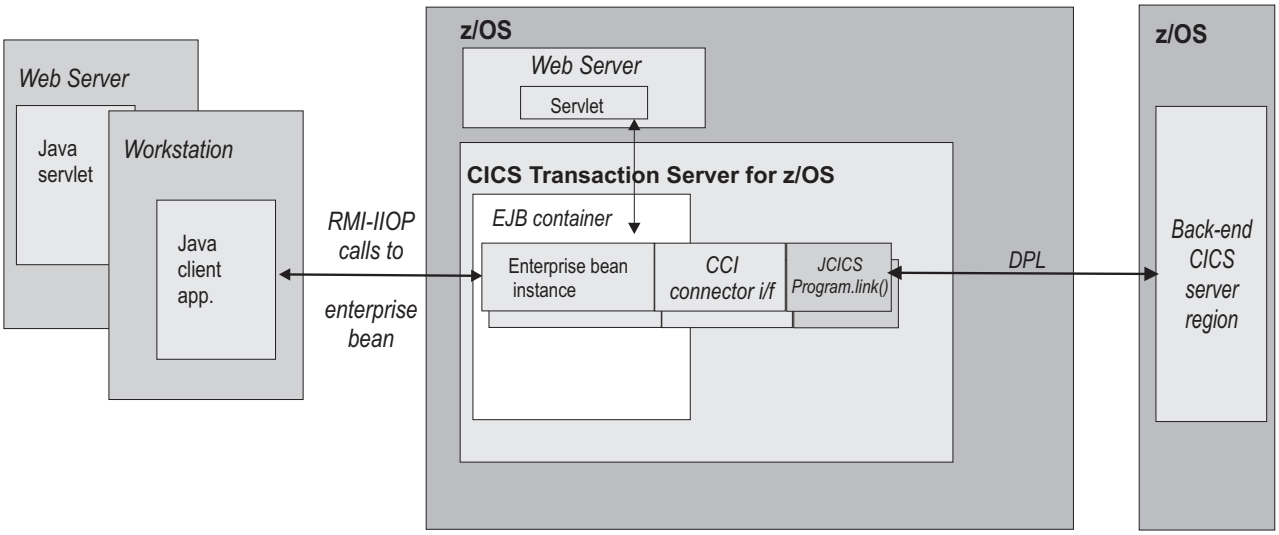

Figure 30. A CICS enterprise bean uses the CCI Connector for CICS TS to connect to a CICS server program.

A Java client application or servlet uses RMI-IIOP to create an instance of an enterprise bean, which exists in a CICS EJB container. The enterprise bean uses the CCI Connector for CICS TS to link to a server program on a back-end CICS TS for z/OS region.

To create an enterprise bean that uses the CCI Connector for CICS TS, the Java programmer requires a reasonable knowledge of CICS (although somewhat less than if he or she were using JCICS). However, the enterprise beans that are created can be used by Java programmers who have little knowledge of CICS.

The CCI Connector for CICS TS is highly optimized for execution within CICS; there is very little overhead involved in using it rather than a JCICS `Program.link()` call.

Benefits of the CCI Connector for CICS TS

There are a number of benefits in using CCI Connector for CICS TS to build powerful server components that make use of existing CICS programs.

1. CICS enterprise beans that use the connector:
   - Enable programmers of Java client applications, who typically have little or no knowledge of CICS, to add the power of CICS to their applications.
   - Can be called by Java client applications and servlets running on many platforms. The client code used to call the bean (and through it the CICS server program) is identical on all Java platforms. Thus, for example, the

client could be an enterprise bean running on WebSphere, a servlet running on a Web server, or a standalone application on a workstation.
   - If written correctly, should be portable, with little or no modification, between all EJB servers that support the Common Client Interface.
2. Because the Common Client Interface is a non-proprietary standard, the CCI code that calls the server program should be portable, with little or no modification, to and from most Java-enabled platforms.
3. Because the CCI Connector for CICS TS runs *inside* CICS, no network flows are required between the connector and CICS. Thus, the connector's performance is better than that of CCI connectors that use the ECI resource adapter to access CICS programs from outside CICS.
4. Using the connector from a CICS session bean results in a simple, two-tier deployment model: Client → CICS TS.
5. Programs written to use the ECI resource adapter can be easily adapted to use the CCI Connector for CICS TS. Thus, client programs that previously accessed CICS server programs from outside CICS can be migrated to run inside CICS.

   Note: If you port a program written to use the ECI resource adapter to use the CCI Connector for CICS TS, you must recompile the program to use the CICS TS-supplied classes in the `dfjcci.jar` JAR file, rather than the CICS Transaction Gateway classes.
6. The CCI Connector for CICS TS supports the Java 2 security policy mechanism.

Sample applications

CICS supplies two sample applications that illustrate how a CICS Java program or enterprise bean can use the CCI Connector for CICS TS to call a CICS server program.

1. The CCI Connector sample. This is a relatively simple application that shows how to code the CCI APIs directly.

   The CCI Connector sample illustrates how to:
   a. Look up a previously-published connection factory in a JNDI namespace
   b. Use the CCI Connector for CICS TS to call a CICS server program

   The CCI Connector sample is described in "The CCI Connector sample application" on page 331.
2. The EJB Bank Account sample. This is a more complex sample that illustrates how you can use enterprise beans and DB2 to make CICS-controlled information available to Web users. The sample implements a CICS enterprise bean that uses the CCI Connector for CICS TS to link to back-end CICS COBOL programs. The COBOL programs extract information from DB2 data tables.

   The EJB Bank Account sample is described in "The EJB Bank Account sample application" on page 270.

CICS also supplies two sample utility programs that show you how to:

1. Publish a connection factory to a JNDI namespace (the `CICSConnectionFactoryPublish` sample). This is described in "Publishing a connection factory using CICSConnectionFactoryPublish" on page 329.
2. Retract a previously-published connection factory from the JNDI namespace (the `CICSConnectionFactoryRetract` sample). This is described in "Retracting a connection factory using CICSConnectionFactoryRetract" on page 330.

Using the CCI Connector for CICS TS

CICS Java components that use the CCI Connector for CICS TS can be programmed in two ways.

About this task

1. Program directly to the connector's implementation of the Common Client Interface. This approach produces the best performance.
2. Use a rapid application development (RAD) tool that provides visual interfaces and high-level constructs for programming the connector's Common Client Interface.

Whichever method you choose, you need to understand how to use the CCI Connector for CICS TS from a Java component running in CICS TS.

The logic a CICS enterprise bean should use to link to a back-end CICS program is shown in Figure 29 on page 321. That is:

1. Use the CICS-supplied sample program, `CICSConnectionFactoryPublish`, to publish a ConnectionFactory object suitable for use with the CCI Connector for CICS TS to the JNDI namespace used by the local CICS region. (See "Using the sample utility programs to manage and acquire a connection factory" on page 327.)
2. Declare a ConnectionFactory object, and set it to the CICS connection factory by means of a JNDI lookup.
3. Create an ECIConnectionSpec object. Set its properties as necessary.

   Note: This step is included for completeness. However, any userid or password specified in the ECIConnectionSpec object is ignored by CICS.
4. Use the ConnectionFactory to create a Connection object. This object represents a single connection to CICS.
5. Create an Interaction object from the Connection object.
6. Create an ECIInteractionSpec object. Set its properties, including the name of the target program and the mode—synchronous or asynchronous—of the interaction. (For CICS TS, only synchronous mode is supported.)
7. Create two Record objects, to represent the input and output communications areas of the target program.
8. Run the execute method of the Interaction object, passing the ECIInteractionSpec, and the input and output Record objects, as arguments.
9. Retrieve the data returned by the target program from the output Record object.
10. Execute the close method of the Interaction object.
11. Execute the close method of the Connection object.

Note: To specify the CICS server region which owns the program to be linked to, use the local PROGRAM definition of the server program. The PROGRAM definition should specify the location of the server program (local or remote) and, if it's remote, whether or not dynamic routing should occur.

Important: We recommend that you get the Javadoc for the CCI Connector architecture API from the Sun Web site. This will help you code your CCI applications. It also provides information such as the exceptions used by CCI

implementations. Javadoc for the CICS-specific ECIConnectionSpec and ECIInteractionSpec classes is in the *CCI Connector for CICS TS: Class Reference*, in the CICS Information Center.

Which classes to use?

Which classes should you use, the standard CCI classes in the `javax.resource.cci` package or the CICS-specific classes provided by the CCI Connector for CICS TS in the `com.ibm.connector2.cics` package?

Framework classes

The CCI Connector for CICS TS provides implementations of the framework classes called **ECIConnectionFactory**, **ECIConnection**, and **ECIInteraction**.

However, the standard **ConnectionFactory**, **Connection**, and **Interaction** classes should be used, rather than the CICS-specific implementations. For guidance information about programming these classes, see the *CICS Transaction Gateway: Programming Guide*. For reference information, see the Sun Javadoc generated from the **ConnectionFactory**, **Connection**, and **Interaction** classes' source code.

Note that not all the information in the *CICS Transaction Gateway: Programming Guide* is applicable to the CCI Connector for CICS TS. The following properties of the **ConnectionFactory** class (and of the CICS-supplied **ECIManagedConnectionFactory** class) are ignored by CICS TS:

- clientSecurity
- connectionURL (in CICS TS, this is always `local:`)
- password
- portNumber
- serverName
- serverSecurity
- userName

Specifying a value for any of the above properties has no effect.

Input/output classes

The CCI Connector for CICS TS provides implementations of the input/output classes. Use these CICS-specific classes (ECIConnectionSpec and ECIInteractionSpec} rather than the standard ConnectionSpec and InteractionSpec classes.

For guidance information about programming the CICS-specific classes, see the *CICS Transaction Gateway: Programming Guide*. For reference information, see the CICS Javadoc generated from the ECIConnectionSpec and ECIInteractionSpec classes in the *CCI Connector for CICS TS: Class Reference*. Special considerations that apply to the CCI Connector for CICS TS are listed below.

Note: Specifying a property or value described as “not supported by CICS TS” results in an exception. Specifying a property or value described as “ignored by CICS TS” has no effect.

`ECIConnectionSpec`

This class allows the J2EE component to pass security credentials different from those defined for the connection factory. Properties include:

`Password`

The password for the userid specified in UserName. Ignored by CICS TS.

UserName
The userid to be used to access CICS. Ignored by CICS TS.

ECIInteractionSpec
This class holds all the interaction-relevant attributes (for example, the name of the target program and the mode of the interaction—synchronous or asynchronous) necessary for an interaction with CICS. It is a required parameter on each Interaction.execute() method call. Its properties are:

InteractionVerb
The mode of the call to CICS—synchronous or asynchronous. The CCI Connector for CICS TS supports only the following:

SYNC_SEND_RECEIVE
A synchronous call. This is used to link to a CICS program.

FunctionName
The name of the program to execute on CICS. The CCI Connector for CICS TS requires you to specify FunctionName.

Note: FunctionName can refer to either a local or a remote program. The PROGRAM definition in the local region should specify the location of the server program (local or remote) and, if it's remote, whether or not dynamic routing should occur.

ExecuteTimeout
The timeout value for interactions with CICS.

0 No timeout. This is the default value, and the only value supported by CICS TS.

A positive integer
The length of time in milliseconds. Ignored by CICS TS.

CommareaLength
The length of the communications area (COMMAREA) being passed to CICS inside your input record. If this is not supplied, the default used by the CCI Connector for CICS TS is the length of the input record data.

ReplyLength
The amount of data you want back from CICS. Where only a small amount of a large returned COMMAREA is required by your enterprise bean or Java component, you can use this setting to cut down on network bandwidth. If not supplied, the default is to receive all data in the COMMAREA.

Note: You are recommended not to set ReplyLength. Because the CCI Connector for CICS TS always runs in local mode—that is, the enterprise bean or Java component that calls the connector executes on the same CICS region as the connector itself—there is no network flow to consider and therefore no need to receive less than the whole reply.

Record
For input and output, the CCI Connector for CICS TS supports only Record classes that implement the javax.resource.cci.Streamable interface. This allows the connector to read and write the streams of bytes that make up CICS COMMAREAs directly to and from the Record objects supplied to the **execute()** method of ECIInteraction.

For further information about using the javax.resource.cci.Streamable interface to build input records and retrieve byte arrays from output records, see the *CICS Transaction Gateway: Programming Guide*.

Data conversion and the CCI Connector for CICS TS

To represent text data, Java programs always use the Unicode character set, while CICS TS programs use EBCDIC.

When a Java program or enterprise bean calls a CICS TS server program, any text values in the communications area of the server program must be converted from Unicode to EBCDIC on input, and from EBCDIC to Unicode on output. However, the CCI Connector for CICS TS handles this data conversion automatically. When converting to and from Unicode, the JCICS Program.link() call issued by the connector uses, as the alternative coding system, the coding system of the execution environment; because the connector runs on z/OS, the alternative coding system is EBCDIC.

Note: By default, the Record objects passed to the connector's Interaction.execute() method use the EBCDIC code page used by the connector's execution environment.

Installing the CCI Connector for CICS TS

Requirements for the CCI Connector for CICS TS

The hardware and software requirements for the CCI Connector for CICS TS are the same as for CICS Transaction Server generally.

Compiling CCI applications

To compile an application that uses the CCI Connector for CICS TS, you must include these CICS-supplied JAR files in your Java classpath:

`connector.jar`
The CCI APIs, required by all CCI applications

`dfjcci.jar`
The CICS TS implementations of the CCI APIs

When you install CICS, `connector.jar` is installed into the `%JAVA_HOME%/standard/jca` z/OS UNIX directory (where %JAVA_HOME% is the value of the JAVADIR parameter on the DFHISTAR CICS installation job); `dfjcci.jar` is installed into the `/usr/lpp/cicsts/cicsts41/lib` directory (where `cicsts41` is the value of the USSDIR parameter on the DFHISTAR installation job).

Running CCI applications on CICS TS

You shouldn't need to take any special steps to set up CICS to support applications that use the CCI Connector for CICS TS.

Using the sample utility programs to manage and acquire a connection factory

About this task

CICS supplies three sample programs that illustrate how to:

1. Publish a connection factory to a JNDI namespace (the `CICSConnectionFactoryPublish` sample). You can use the sample to create a **ConnectionFactory** object suitable for use with the CCI Connector for CICS TS, and to publish it to the JNDI namespace used by the local CICS region. An enterprise bean or Java program, running on CICS, can then perform a JNDI lookup to obtain a reference to the connection factory.

   This sample is described in "Publishing a connection factory using CICSConnectionFactoryPublish" on page 329.
2. Retract a previously-published connection factory from the JNDI namespace (the `CICSConnectionFactoryRetract` sample). This sample is described in "Retracting a connection factory using CICSConnectionFactoryRetract" on page 330.
3. Look up a connection factory in the JNDI namespace (the CCI Connector sample application). This sample also shows you how to use the CCI Connector for CICS TS to call a CICS server program. It is described in "The CCI Connector sample application" on page 331.

Using the `CICSConnectionFactoryPublish` and `CICSConnectionFactoryRetract` samples, you can create, publish, and manage a connection factory separately from the applications that use it.

To use the sample programs, you need a suitably configured name server. If you need to configure a name server, see "Enabling JNDI references" on page 183 and "Specifying the location of the JNDI name server" on page 183.

Installing the publish and retract sample programs

This section describes how to install the `CICSConnectionFactoryPublish` and `CICSConnectionFactoryRetract` programs.

About this task

How to install the CCI Connector application is described in "Installing the CCI Connector sample" on page 333.

The CICS-supplied JAR file `CICSCCISamples.jar` contains the object (.class) files for the sample programs. CICS installs `CICSCCISamples.jar` into the `/usr/lpp/cicsts/cicsts41/samples/cci` directory (where `/usr/lpp/cicsts/cicsts41` is the install directory for CICS files on z/OSUNIX). Also installed into the `/usr/lpp/cicsts/cicsts41/samples/cci` directory are the source (.java) files of the programs.

To install the `CICSConnectionFactoryPublish` and `CICSConnectionFactoryRetract` programs:

Procedure

1. Add the JAR file containing the programs, `/usr/lpp/cicsts/cicsts41/samples/cci/CICSCCISamples.jar`, to the CLASSPATH_SUFFIX statement in the JVM profile that the programs will use. As supplied, the sample programs use the CICS-supplied sample JVM profile DFHJVMPR, which is the default if no JVM profile is specified in the program's resource definition. CICS installs DFHJVMPR into the `/usr/lpp/cicsts/cicsts41/JVMProfiles` directory.
2. Place your edited version of DFHJVMPR in the z/OS UNIX directory specified on the **JVMPROFILEDIR** system initialization parameter. (In a default CICS installation, **JVMPROFILEDIR** specifies `/usr/lpp/cicsts/cicsts41/JVMProfiles`

3. Use CEDA to install transactions CCPB and CCRT from group DFH$CCI.
4. Use CEDA to install programs DFJ$CCPB and DFJ$CCRT from group DFH$CCI.

   Note: If your CICS region uses program autoinstall, this last step is not required.

Results

Publishing a connection factory using CICSConnectionFactoryPublish

The `CICSConnectionFactoryPublish` program performs these tasks.

1. Gets the initial JNDI context of the CICS region.
2. Checks to see if a `ConnectionFactory` subContext exists in the context structure.
3. If the `ConnectionFactory` subContext does not exist, creates it.
4. If the `ConnectionFactory/CICSConnectionFactory` connection factory has not already been published (bound) to the name server, publishes it.

The default name of the connection factory, as set by the supplied version of the `CICSConnectionFactoryPublish` program, is `CICSConnectionFactory`. The default name of the JNDI subContext in which the connection factory is published is `ConnectionFactory`. By editing the source code of the `CICSConnectionFactoryPublish` program, you can change:

- The name of the connection factory.
- The JNDI subContext.
- If the linked-to server program is remote, the name of the mirror transaction under which the program runs on the remote region. However, the recommended way to specify the mirror program is on the local PROGRAM definition of the server program.

For instructions on how to make the changes, see the comments in the source code.

If you change the name of the connection factory, or of the subContext, remember to make the same change in all three of the sample programs.

Running the program

To publish (bind) a ConnectionFactory suitable for use with the CCI Connector for CICS TS to the CICS JNDI name server, run transaction CCPB.

Unless you have changed the `CICSConnectionFactoryPublish` program, the ConnectionFactory will be named `CICSConnectionFactory`, and will be published to subContext `ConnectionFactory` in the JNDI server's namespace.

The following message appears on your screen:

```
ccpb - ConnectionFactory published to JNDI successfully.
```

Note: If a ConnectionFactory with the same name and subContext has already been published to the JNDI server (and not retracted), a different message appears:

```
ccpb - The ConnectionFactory is already published to JNDI.
```

Assuming that the connection factory is published successfully, the following output is sent to **stdout**:

```
*********************************************************************************************
**** CICSConnectionFactoryPublish: Started
**** CICSConnectionFactoryPublish: Binding ConnectionFactory ConnectionFactory/CICSConnectionFactory
**** CICSConnectionFactoryPublish: ConnectionFactory bound to JNDI
**** CICSConnectionFactoryPublish: Ended
*********************************************************************************************
```

Figure 31. Stdout output from transaction CCPB to publish a ConnectionFactory with default name and subContext

It is not recommended that you run `CICSConnectionFactoryPublish` as a PLTPI program, or link to it from a PLTPI program. This is because, if a JVM is not available, CICS startup time will be lengthened.

Looking up a connection factory

This code example shows you how to look up a previously-published connection factory in the JNDI namespace used by CICS.

```
// Declare a ConnectionFactory object
ConnectionFactory cf = null;

try{
    // Get the initial JNDI context
    javax.naming.Context ic = new javax.naming.InitialContext();

    // Do the lookup, casting the returned CICSConnectionFactory to type
    // ConnectionFactory
    cf = (ConnectionFactory)ic.lookup("ConnectionFactory/CICSConnectionFactory");

    // Use the connection factory to create a connection to CICS
    Connection eciConn = (Connection)cf.getConnection();
}
catch (Exception e){
    // Lookup failed, or specified connection factory has not been published
    // Exception processing
}
```

This is illustrated in the CCI Connector application—see "The CCI Connector sample application" on page 331.

Retracting a connection factory using CICSConnectionFactoryRetract

To retract (unbind) a connection factory that you have published, run transaction CCRT. Unless you have changed the `CICSConnectionFactoryRetract` program, the ConnectionFactory to be retracted will be `CICSConnectionFactory`, in subContext `ConnectionFactory` in the JNDI server's namespace.

The following message appears on your screen:

```
ccrt - ConnectionFactory retracted from JNDI successfully.
```

Note: If the ConnectionFactory named in the `CICSConnectionFactoryRetract` program does not exist on the JNDI server (it may, for example, have already been retracted), a different message appears:

```
ccrt - unable to locate ConnectionFactory on JNDI.
```

Assuming that the connection factory is retracted normally, the following output is sent to **stdout**:

```
********************************************************************************
**** CICSConnectionFactoryRetract: Started
**** CICSConnectionFactoryRetract: Unbinding ConnectionFactory/CICSConnectionFactory
**** CICSConnectionFactoryRetract: ConnectionFactory/CICSConnectionFactory unbound
**** CICSConnectionFactoryRetract: Ended
********************************************************************************
```

Figure 32. Stdout output from transaction CCRT to retract a connection factory with default name and subContext

It is not recommended that you run `CICSConnectionFactoryRetract` as a PLTSD program, or link to it from a PLTSD program. This is because CICS shut down time will be lengthened.

The CCI Connector sample application

The CCI Connector sample is a relatively simple application that shows how to code the CCI APIs directly.

It illustrates how to:
1. Look up a previously-published connection factory in a JNDI namespace
2. Use the CCI Connector for CICS TS to call a CICS server program

The sample consists of:
- A CICS Java program
- A custom Record that demonstrates the use of the javax.resource.cci.Streamable interface
- A CICS COBOL server program

The sample works like this:

1. A user starts the application by running the CCCI transaction from a CICS terminal.
2. The CICS Java program, `CICSCCISample` (DFJ$CCIC), is started. The Java program:
   a. Asks the user to input a sequence of random, unsorted, decimal numbers
   b. Does a JNDI lookup of the name server, to obtain a CICS connection factory
   c. If a connection factory has not been published to the name server, creates one programatically
   d. Uses the connection factory to create a connection to CICS
   e. Creates an Interaction object from the **Connection** object, and sets the properties of the interaction (including the name of the target program) by means of an ECIInteractionSpec object
   f. Uses the Interaction.execute method to link to the COBOL program, DFH$0CCIS, passing as input (in a custom Record object) the user's sequence of unsorted numbers, plus the ECIInteractionSpec object
3. The COBOL program sorts the numbers into ascending order and returns the sorted sequence in its output COMMAREA.
4. The Java program retrieves the COBOL program's output from the output **Record** object and displays the sorted list on the user's terminal.

Figure 33 on page 332 shows the components of the sample application.

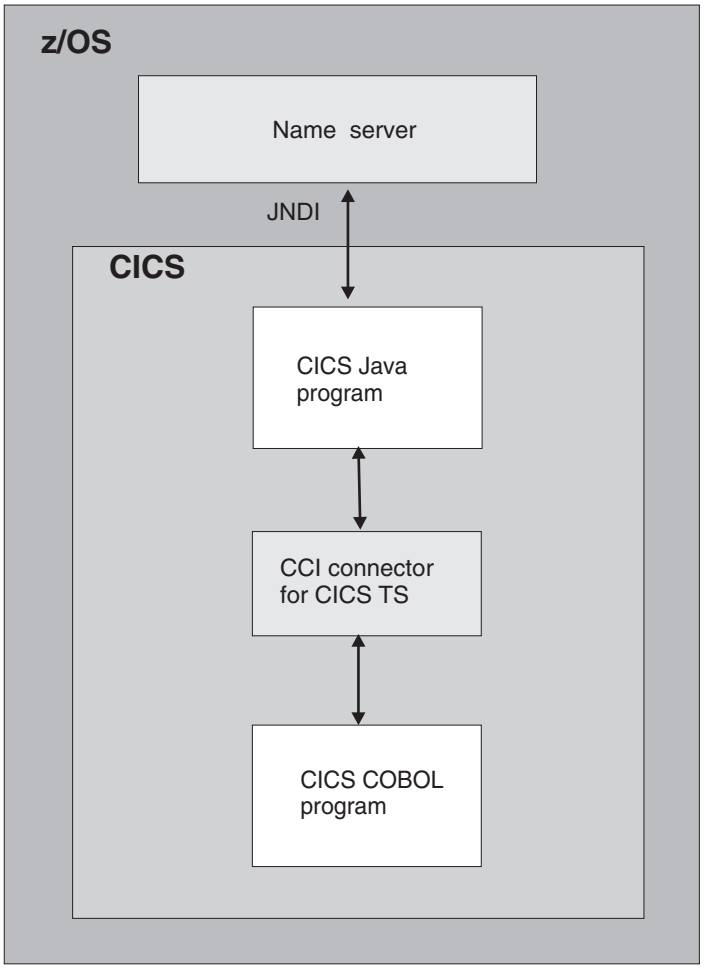

Figure 33. Overview of the CCI Connector sample application. The main elements of the sample are a CICS Java program and a CICS COBOL server program. The Java program uses the CCI Connector for CICS TS to link to the COBOL server program. The CICS connection factory can be published to either a COS Naming Server or an LDAP name server.

Requirements for the CCI Connector sample

To enable the CCI Connector sample to obtain a CICS connection factory by performing a JNDI lookup, you need a name server that supports the Java Naming and Directory Interface (JNDI), Version 1.2 or later.

The way to set one up is described in "Actions required on z/OS or Windows NT" on page 240. You can use either a COS Naming Server or an LDAP server.

However, if the sample cannot connect to the name server, or a CICS connection factory has not been published to the name server, the sample creates the connection factory programatically. Therefore, strictly speaking, a name server is not a requirement to run the sample.

Installing the CCI Connector sample

About this task

Procedure

1. If you have not already done so when running the `CICSConnectionFactoryPublish` and `CICSConnectionFactoryRetract` samples, locate the JAR file containing the sample programs, `/usr/lpp/cicsts/cicsts41/samples/cci/CICSCCISamples.jar`, where `/usr/lpp/cicsts/cicsts41` is the install directory for CICS files on z/OS UNIX. Add this JAR file to the CLASSPATH_SUFFIX statement in the JVM profile that the programs will use. As supplied, the sample programs use the CICS-supplied sample JVM profile DFHJVMPR, which is the default if no JVM profile is specified in the program's resource definition.

   CICS installs DFHJVMPR into the `/usr/lpp/cicsts/cicsts41/JVMProfiles` directory.

   Place your edited version of DFHJVMPR in the z/OS UNIX directory specified on the **JVMPROFILEDIR** system initialization parameter.
2. Ensure that the `connector.jar` and `dfjcci.jar` files are on the standard class path.

   Note: When you install CICS, `connector.jar` is installed into the `%JAVA_HOME%/standard/jca` directory and `dfjcci.jar` is installed into the `/usr/lpp/cicsts/cicsts41/lib` directory, as described in "Compiling CCI applications" on page 327. The `/usr/lpp/cicsts/cicsts41/lib` directory is on the base class path built by CICS, which is not visible in the JVM profiles, so this directory is always included.
3. Ensure that the name server is running.
4. Use the `CICSConnectionFactoryPublish` program to create a ConnectionFactory object for use by the CCI Connector for CICS TS, and to publish it to the name server. See "Publishing a connection factory using CICSConnectionFactoryPublish" on page 329.
5. Use CEDA to install transaction CCCI from group DFH$CCI.
6. Use CEDA to install definitions of the CICS Java and COBOL programs. Install programs DFJ$CCIC and DFH0CCIS from group DFH$CCI.

   Note: If your CICS region uses program autoinstall, this step is not required.

Testing the sample

About this task

To test the CCI Connector sample:

1. Start transaction CCCI at a CICS terminal.
2. The sample asks you to input some numbers. Enter at least five decimal numbers, separated by spaces, and press the Return key. (Each number should be of five digits or less, and the numbers should not be ordered by size.)
3. The sample writes the sorted list of numbers to your screen and to **stdout**. If, for example, you entered the numbers 54, 3, 77, 55, and 19, your screen would look like this:

```
CCCI - CCI sample transaction starting.

A Connection object has been instantiated.

An Interaction object has been instantiated.
```

```
Enter a series of numbers:  54 3 77 55 19

An InteractionSpec object has been instantiated.

Connecting to program DFH0CCIS by invoking execute() on Interaction object.

Commarea sent:    54    3   77   55   19*

Commarea returned:     3   19   54   55   77*

CCCI - CCI sample transaction finished.
```

Problem determination

You can use CCI Connector for CICS TS messages, and CICS trace, to diagnose problems.

CCI Connector for CICS TS messages

CICS messages related to the CCI Connector for CICS TS are described in the *CICS Messages and Codes* manual.

Tracing the CCI Connector for CICS TS

The CICS trace points related to the connector are in the range EJ 0600—EJ 06FF.

These are described in the *CICS Trace Entries* manual.

To control the output of CICS trace information from the connector, use CICS trace control in the normal way.

Upgrading from the CICS Connector for CICS TS to the CCI Connector for CICS TS

If you have existing applications that use the CICS Connector for CICS TS, you must upgrade them to use the CCI Connector for CICS TS instead.

Table 14 summarizes the upgrade choices for CICS Java components that use either the CICS Connector for CICS TS or the CCI Connector for CICS TS, and states a preferred solution for each case.

Table 14. Suggested upgrade path for CICS Java components that use the CICS CCF or CCI connectors

Connector used by current program	**Connector interface used by current program**	**Status in CICS TS 4.1**	**Suggested upgrade strategy**
CICS Connector for CICS TS	CICS Transaction Gateway API (**ECIRequest**)	Not supported	The CICS Transaction Gateway API is no longer supported. Re-engineer to use the CCI Connector for CICS TS. Program the connector either directly or by means of a rapid application development (RAD) tool that supports it.

Table 14. Suggested upgrade path for CICS Java components that use the CICS CCF or CCI connectors (continued)

Connector used by current program	**Connector interface used by current program**	**Status in CICS TS 4.1**	**Suggested upgrade strategy**
CICS Connector for CICS TS	CCF, programmed either directly or with VAJ Enterprise Access Builder or similar	Not supported	CCF is replaced by CCI. Re-engineer to use the CCI Connector for CICS TS, which performs better than the CICS Connector for CICS TS and uses an industry-standard interface. Program the connector either directly or by means of a RAD tool that supports it. **Note:** It is possible to program the CCI Connector for CICS TS using VAJ Enterprise Access Builder, but this is not recommended because VAJ/EAB is no longer supported.
CCI Connector for CICS TS	CCI, programmed directly	Supported	CCI can be used indefinitely. Programming the CCI directly gives the best performance.
CCI Connector for CICS TS	CCI, programmed with VAJ Enterprise Access Builder or similar	Supported	To continue using VAJ/EAB, changes must be made to the application.

Chapter 26. Dealing with CICS enterprise bean problems

This section contains information on guidance in dealing with problems setting up and using the CICS enterprise bean support.

See the *CICS Problem Determination Guide* for guidance on the more general aspects of CICS problem determination and diagnostics.

CICS enterprise bean set-up problems

If you have difficulties setting up the CICS EJB server, the problem could be related to your basic CICS Java set up. Try running the Java HelloWorld sample. If this also fails it points to a problem with the set up of your JVM rather than anything else.

Methods that require multiple request processors

If a single execution of an enterprise bean method requires more than one request processor, your application could experience deadlock problems.

About this task

(A method can be said to "require more than one request processor" if it calls one or more other, typically remote, methods, each of which must execute in a different request processor.) Deadlocks can be caused by all the request processors required to satisfy the method being forced to wait for a JVM when no more JVMs are permitted. This can occur for two reasons:

1. In the simple case, the maximum number of JVMs allowed to exist concurrently under CICS (**MAXJVMTCBS**) is smaller than the number of request processors required to service the method request.
2. In the complex case:
   - CICS is processing multiple requests simultaneously.
   - All the requests are waiting for another JVM.
   - All the permitted JVMs are currently in use.

Avoiding the simple case is easy; avoiding the complex case is more difficult. It is necessary to ensure there are always enough free JVMs to allow at least one method's requirement of request processor instances to be satisfied.

The maximum number of concurrent JVMs available to a bean method is set by the MAXACTIVE attribute of the TRANCLASS definition for the request processor transaction. The maximum number of concurrent JVMs available to CICS is set by the **MAXJVMTCBS** system initialization parameter.

To remove the possibility of deadlocks caused by bean methods that use multiple request processors:

1. Wherever it is consistent with your applications' requirements, try to minimize the number of request processors each method requires, preferably to one. If you can reduce the requirements of all methods, in all applications, to one request processor, you need do no more.
2. If it is not possible to reduce the requirements of all methods to one request processor, discover which is your "worst case"—that is, the bean method that requires the most request processors in order to be satisfied.
3. Create a new TRANCLASS definition. This transaction class will apply to the request processor transaction under which bean methods that require multiple request processors will run.
4. On the TRANCLASS definition, set the value of MAXACTIVE using the following formula:

   ```
   MAXACTIVE <= ((MAXJVMTCBS - n) / (n - 1)) + 1
   ```

   where `n` is the maximum number of request processors required by your "worst case" method.

   If the result of this calculation is a decimal value, round it down to the nearest (lower) whole number.
5. Create new TRANSACTION and REQUESTMODEL definitions:
   a. Create a new TRANSACTION definition for the request processor transaction under which bean methods that require multiple request processors will run. (The easiest way to do this is to copy the definition of the default CIRP request processor transaction and modify it.) On the TRANCLASS option, specify the name of your new transaction class.
   b. Create one or more REQUESTMODEL definitions. Between them, your new REQUESTMODEL definitions must cover all requests that may be received for bean methods that require multiple request processors. On the TRANSID option of the REQUESTMODEL definitions, specify the name of your new transaction.

Using EJB server runtime diagnostics

The EJB server provides runtime diagnostics to help you diagnose and resolve problems. These include error messages, JVM trace, and the Java Platform Debugger Architecture (JPDA).

CICS enterprise bean errors and messages

This is a list of places to look for error messages from CICS.

Enterprise Java domain (DFHEJnnnn) messages
CICS issues a large number of information, warning and error messages from the enterprise Java domain. Most of these are routed to the CEJL and CJRM transient data queues, others are sent to the console. See the *CICS Messages and Codes* manual for a complete listing.

CICS JVM (DFHSJnnnn) messages
These are messages issued by the CICS JVM. Most are routed to the transient data queue CSMT. See the *CICS Messages and Codes* manual for a complete listing.

CICS Development Deployment Tool (DFHADnnnn) messages
These are messages issued by this tool and routed to CICS as SYSPRINT messages. See the *CICS Messages and Codes* manual for a complete listing.

CICS abend codes

- AJMA to AJM9 are issued by the CICS JVM
- AJ01 to AJ99 are issued by Java environment setup class Wrapper

See the *CICS Messages and Codes* manual for a listing.

JVM trace

Java Virtual Machines (JVMs) have their own internal trace facility. JVM trace can aid in the diagnosis of problems in the JVM. Note that JVM trace can produce a large amount of output, so you should normally activate JVM trace for special transactions, rather than turning it on globally for all transactions.

Defining and activating tracing for JVMs explains the different ways to activate JVM trace and change the JVM trace options.

When you activate JVM trace, each JVM trace point that is generated appears as an instance of a CICS trace point in the SJ domain.

In addition to the JVM trace options, the standard trace points for the SJ (JVM) domain, at CICS trace levels 0, 1 and 2, can be used to trace the actions that CICS takes in setting up and managing JVMs and the shared class cache. The *CICS Trace Entries* manual has details of all the standard trace points in the SJ domain.

Java platform debugger architecture (JPDA)

The JVM in CICS supports the Java Platform Debugger Architecture (JPDA), which is the standard debugging mechanism provided in the Java 2 Platform.

The JVM in CICS supports the Java Platform Debugger Architecture (JPDA), which is the standard debugging mechanism provided in the Java 2 Platform. This architecture provides a set of APIs that allow the attachment of a remote debugger to a JVM. A variety of third party debuggers are available that exploit JPDA and can be used to attach to and debug a JVM that is running an enterprise bean, CORBA object or CICS Java program. Typically the debugger provides a graphical user interface that runs on a workstation and allows you to follow the application flow, setting breakpoints and stepping through the application source code, as well as examining the values of variables.

See “Debugging an application that is running in a CICS JVM” on page 160 for guidance on setting up and using a debugger with the CICS JVM.

You can find information about JPDA and JPDA-compliant applications at the Web site http://java.sun.com/javase/technologies/core/toolsapis/jpda/.

Using EJB client runtime diagnostics

Most of the error messages issued by the client are of limited use if the problem is in CICS, but you can sometimes get useful information from the client, and it is an obvious place to start.

Some of the more useful client exceptions are as follows:

`NoClassDefFoundException and ClassNotFoundException`
: If the client issues either of these, there is probably something missing or corrupt on your client-side classpath. The exception should give you a good indication of which class is missing, and from this you may be able to work out which JAR file to add to the classpath. Remember that you

need `j2ee.jar`, and the fully deployed jar in the classpath. It is unlikely that CICS will issue any useful additional information for these problems.

NoClassDefFoundError:javax/ejb/HomeHandle
This indicates that a client application does not have EJB 1.1 level classes available on the classpath. Ensure that `j2ee.jar` is available.

ObjectNotFoundException
This exception can indicate that a session bean has timed out or that an attempt has been made to use the session bean in two or more concurrent transactions.

`RemoteException`
This indicates a problem in the server application and often contains a nested exception giving more information. These include:

NoClassDefFoundError
This points to a missing JAR file on the server side. Check the CICS system console and the JVM standard error and output files for additional information.

CORBA.INTERNAL
This indicates a failure in the server side application outside the JVM (for example, in a COBOL program called by an enterprise bean). Check the CICS system console for more information.

CORBA exceptions

These exceptions can sometimes provide useful information.

The **completion status** can have one of three values:

- **No** means that the server definitely did not complete running the invoked method successfully.
- **Yes** means that the invoked operation on the server did complete.
- **Maybe** means that the client cannot determine whether or not the operation completed on the server.

If the completion status is **Yes**, you can be sure that the client found something to run on a server (however if your JNDI/IOR is incorrect, it may not have been the correct enterprise bean or on the expected CICS region). You will usually find some more useful information in the CICS output about why the method call failed.

Some of the more common CORBA exceptions received by the client are:

org.omg.CORBA.COMM_FAILURE
This can occur in one of the following situations:

- The JNDI nameserver is not running (if it is on a JNDI lookup)
- The enterprise bean has not been published to the JNDI nameserver.
- The CICS region is down
- TCPIPSERVICE is not installed or is open (for method invocations on CICS)

if either the JNDI server is not running (if it is on a JNDI lookup), if the CICS region is down, or if your TCPIPSERVICE is not installed or open (for method invocations on CICS). It can also occur

org.omg.CORBA.INTERNAL
This is usually caused by an abend or failure of the server-side application. Look in the CICS console for more information.

org.omg.CORBA.INVALID_TRANSACTION
This can occur because of transaction interoperability problems between a web application server and CICS.

A number of protocols exist to support distributed transactions. The CICS enterprise Java environment supports only the standard CORBA Object Transaction Service (OTS) protocol. However, some J2EE-compliant web application servers (such as WebSphere Version 4) either do not use this protocol, or do not use this protocol by default. (Versions of WebSphere Application Server from Version 5 onwards are not affected by this problem.)

If objects on your web application server call CICS enterprise beans within the scope of existing transaction contexts, you must set up your web application server to use the CORBA OTS. If this is not possible, your web application server is not fully compatible with CICS enterprise Java support. (For a way of using the EJB Bank Account sample application to test whether your web application server is fully compatible with CICS enterprise Java support, see "A note about distributed transactions" on page 285.)

To force WebSphere Application Server to use the CORBA OTS:
1. At the WebSphere Administration Console, select the JVM settings tab.
2. Enter the following in the System Properties section:

```
com.ibm.ejs.jts.ControlSet.interoperabilityOnly=true
com.ibm.ejs.jts.ControlSet.nativeOnly=false
```

   Save your changes.
3. Restart the application server.

org.omg.CORBA.OBJECT_NOT_EXIST
This can occur when a client finds a reference to a bean on the JNDI nameserver but the bean is no longer installed in CICS.

org.omg.CORBA.UNKNOWN
There are many reasons for this exception including errors in your code, and errors in CICS. See the CICS output for more clues about the cause of the problem

In many instances, the CORBA exception includes a CICS specific minor code to aid in problem determination. CICS currently uses the following minor codes:

Table 15. CICS specific CORBA minor codes

Code	CICS component detecting problem
1229111296	CICS IIOP request receiver
1229111297	Elsewhere in CICS II domain
1229111298	ORB component of CICS OT domain
1229111299	JTS component of CICS OT domain
1229111300	CSI component of CICS OT domain
1229111301	CSI component of CICS EJ domain

If the client receives a CORBA exception containing any of the CICS minor codes, you should examine the CICS message logs for further information about the error.

Class version issues with RMI-IIOP

Remote Method Invocation over IIOP (RMI-IIOP) is the communication protocol used, in CICS, by both enterprise beans and CORBA stateless objects. The information in this section therefore applies to both enterprise beans and CORBA stateless objects.

Java RMI is an object-by-value protocol. This means that whenever a Java object is used as a parameter on a method call what gets sent on the wire is the object state. The same is true of return types and exceptions. This state is a "serialized" Java object. The state can be de-serialized by the remote JVM to create a new copy of the original object in the remote JVM. The serialized state contains, among other things, a version number to indicate the version of the class that the state represents. In order for the serialized object to be de-serialized by the remote JVM, it is necessary for the same version of the class file to be present at each end of the IIOP connection. If the remote JVM cannot understand the object state, it will probably cause the following exception to be thrown:

```
java.rmi.MarshalException:unable to read from underlying bridge
```

(This exception may be thrown for other reasons too.)

When you create a class in Java it is possible to provide your own customised serialization mechanism. Using this mechanism, you can handle versioning of your classes explicitly, rather than rely on Java's default serialization process. Moreover, if you provide a custom serialization mechanism you can achieve significant performance savings over the default mechanism. If you want to take advantage of custom serialization, your objects must implement the `java.io.Externalizable` interface.

Often the objects that must be serialized are instances of classes from the standard Java class library. These usually do not change from one version of Java to the next, but if they do it can lead to the kind of problem described above. In order to minimize these problems, it is recommended that you use the same version of Java on the partner machines as CICS uses. For example, between Java 1.3.1 and Java 1.4 the `java.lang.Throwable` class changed significantly. This class is the super-type of all exceptions in Java and thus many exceptions serialized by Java 1.4.1 and later cannot be de-serialized by older versions of Java.

There is a mechanism in CORBA that is used by many ORBs to get around the problem of version changes in classes. Unfortunately, that mechanism does not fully work in CICS because it involves affinities between the partner ORB and the JVM in CICS. Multiple RMI-IIOP calls to the same CORBA object in CICS are likely to be processed in different JVMs. This means that affinities are not supported and that the mechanism for avoiding class versioning issues does not work in CICS. CICS applications suffer from this problem only when sending serialized objects to a remote JVM. If a remote JVM sends a serialized object to CICS, CICS can use the standard CORBA mechanism to cope with any version incompatibilities.

If you experience this kind of problem and are unable to change the version of Java in use at the partner platform, it is recommended that the application be changed to use a datatype that does not cause versioning issues.

Using EJB trace and serviceability commands

You might want to trace an EJB request when you are trying to diagnose hanging or failing requests, or when you need to be able to uniquely identify all transactions associated with a single request in order to monitor that activity or perhaps for accounting purposes.

The main problems when trying to diagnose hanging or failing requests when an EJB logical server comprises multiple CICS regions are that you have to determine:

- The region where the request originated (the request receiver)
- The target (a CICS region or other server) that the request has been routed to.

The system programming interface (SPI) commands **INQUIRE WORKREQUEST** and **SET WORKREQUEST** enable you to:

- determine which transactions are associated with a single request
- correlate all transactions associated with a single request
- purge selected work requests

Each request shows:

- the local task number and transaction id
- the type of request, the first type supported is IIOP
- a unique (printable) string that can be entered on the command as a filter e.g.
  - Worktype
  - ClientIPAddress
  - Target VTAM applid or TCPIP address

For more information about these commands, see the *CICS System Programming Reference* and the *CICS Supplied Transactions* manuals.

The INQUIRE and SET WORKREQUEST commands are only available for IIOP tasks.

WorkRequests associated with RequestReceivers are not included, they are very lightweight and all this information is available in the RequestProcessor. A RequestReceiver may process more that one request per instance and may have left the system long before the request has completed.

When you interrogate a logical server using the CPSM WUI, you have a single screen displaying all WorkRequests in the server

You are able with these commands to purge a RequestProcessor in a manner similar to purging a task from the CEMT INQ TASK list.

Chapter 27. Managing security for enterprise beans

The security mechanisms, Java2 security, Secure Sockets Layer (SSL) security, MRO security, and Security Roles can be used with enterprise beans.

You can implement any combination of these.

Java security
This form of security control is implemented by the Java Virtual Machine (JVM) and can be used with any Java program that executes under JVM control. See "Protecting Java applications in CICS by using the Java security policy mechanism" for guidance on using this type of security control.

Secure Sockets Layer (SSL) security
The Secure Sockets Layer (SSL) is a security protocol that provides privacy and authentication between clients and servers communicating using TCP/IP. For more information about SSL, see Support for security protocols, in the *CICS RACF Security Guide*. For information about using SSL with enterprise beans see Authentication of IIOP requests.

MRO security
After the request receiver has established a CICS USERID to be associated with the request, it may need to be routed to an application-owning-region (AOR). If the routing mechanism uses a multiple region operation (MRO) connection, the transmission of the userid is subject to MRO security rules. See Link security with MRO, inthe *CICS RACF Security Guide*.

Security roles
A security role represents a type of user of an application in terms of the permissions that the user must have to successfully use the application. See "Security roles" on page 353.

Protecting Java applications in CICS by using the Java security policy mechanism

The security of the enterprise beans container environment is protected by the Java security policy mechanism and is independent of CICS security. The security policy mechanism is one of the components that make up the Java security model.

The security policy mechanism is used to enforce the restrictions in the EJB specification concerning Java functions that may not be issued by enterprise beans. By default, Java applications have no security restrictions placed on activities requested of the Java API; the Java API does whatever it is asked. To use Java security to protect a Java application or enterprise bean from performing potentially unsafe actions, you enable a security manager for the Java Virtual Machine (JVM) in which the application or enterprise bean runs. If no security manager is enabled, the JVM runs without Java security by default. A default security manager is supplied with the Java platform. To prevent unauthorized access to system resources by enterprise beans, you are recommended to enable the default security manager.

The security manager enforces a security policy, which is a set of permissions (system access privileges) that are assigned to code sources. Every time the JVM executes code in a class, the JVM determines the code source for the class and

consults the security policy before granting the class the appropriate permissions. Thus, if a piece of code requests access to a particular system resource while a security manager is active, the JVM grants the code access to that resource only if such an access is a privilege associated with that class.

When a JVM starts up, its security manager determines the security policy for the JVM by looking at one or more *policy files* that you have specified. The policy files contain details of the permissions that are granted to particular code sources. A default policy file is supplied with the Java platform. If you enable the default security manager for a JVM, but do not specify any policy files, the security manager determines a security policy using the permissions given in the default policy file. You can specify one or more additional policy files containing permissions that you want to grant, and the security manager adds these permissions to the security policy. So, although only one security policy is in effect for the JVM at any given time, this security policy can be the result of processing one or more policy files.

To enable Java applications and enterprise beans to run successfully in CICS when Java security is active, you specify, as a minimum, an additional policy file that gives CICS the permissions it needs to run the enterprise beans container and gives applications the permissions outlined in the *Enterprise JavaBeans* specification, Version 1. The CICS-supplied enterprise beans policy file, `dfjejbpl.policy`, contains the permissions for this purpose. You specify this additional policy file for each kind of JVM that has a security manager enabled.

You enable the security manager for a JVM, and specify additional policy files, using the security system properties for the JVM.

If you need more information about Java security than is provided here, refer to the Java documentation.

Java security with JDBC or SQLJ

To use JDBC or SQLJ from enterprise beans that run in a JVM with a Java security policy mechanism active, you must use the JDBC 2.0 driver provided by DB2. The JDBC 1.2 driver provided by DB2 does not support Java security and will fail with a security exception unless you disable the mechanism (by deactivating the security manager for the JVM). You must also modify your additional policy file to grant permissions to the JDBC driver.

Enabling a Java security manager and specifying policy files for a JVM

To enable a Java security manager for a JVM and specify additional policy files that you want the security manager to use, you need to customize the security system properties for the JVM.

For each JVM profile that your Java applications and enterprise beans request, if you want JVMs with that profile to run with Java security, you need to enable the default security manager and specify a suitable policy file. For each JVM profile that you want to use Java security, customize the following system properties in either the JVM profile or a JVM properties file:

`-Djava.security.manager`
This system property indicates the Java security manager to be enabled for the JVM. To enable the default Java security manager, include this system property in one of the following formats:

```
-Djava.security.manager=default
```

or

```
-Djava.security.manager=""
```

or

```
-Djava.security.manager=
```

All these statements have the effect of enabling the default security manager. If you do not include the **-Djava.security.manager** system property, then the JVM runs without Java security enabled. If you need to disable Java security for a JVM, comment out this system property.

-Djava.security.policy

This system property describes the location of additional policy files that you want the security manager to use to determine the security policy for the JVM. A default policy file is provided with the JVM in `/usr/lpp/java/J6.0/lib/security/java.policy`, where `java/J6.0` is your install location for the IBM SDK for z/OS, Java Technology Edition. The default security manager always uses this default policy file to determine the security policy for the JVM, and you can use the **-Djava.security.policy** system property to specify any additional policy files that you want the security manager to take into account as well as the default policy file.

To enable CICS Java applications and enterprise beans to run successfully when Java security is active, you need to specify, as a minimum, an additional policy file that gives CICS the permissions it needs to run the enterprise beans container, and gives applications the permissions outlined in the *Enterprise JavaBeans* specification, Version 1. If you do not provide these permissions, then the container code may become inaccessible, preventing CorbaServers from being initialized. The CICS-supplied enterprise beans policy file, `dfjejbpl.policy`, contains the permissions that you need. To specify this policy file, include the system property:

```
-Djava.security.policy=/usr/lpp/cicsts/cicsts41/lib/security/dfjejbpl.policy
```

where *cicsts41* is your chosen value for the USSDIR installation parameter that you defined when you installed CICS TS. "The CICS-supplied enterprise beans policy file" on page 349 has more information about `dfjejbpl.policy`.

If you need to give any of your applications further permissions, you can modify the CICS-supplied enterprise beans policy file, or create and specify your own additional policy file. Policy files are stored in text format, so you can display or modify them using any standard text editing tool. In particular, if you want to use JDBC or SQLJ from enterprise beans, you need to modify the enterprise beans policy file that you have specified, to grant permissions to the JDBC driver. Requirements to support Java programs in the CICS DB2 environment, in the *CICS DB2 Guide*, tells you how to do this.

It is recommended that policy files are made secure, with update authority restricted to system administrators.

When you specify a policy file in the JVM properties file, the policy file is used for JVMs that are built using JVM profiles which reference that JVM properties file. As an alternative, you can specify a policy file to be used for **all** the JVMs in your system for which you have enabled a Java security manager, whatever JVM properties file they might have. For example, you could specify the CICS-supplied enterprise beans policy file, `dfjejbpl.policy`, to be used for all your JVMs. To do this, instead of including the **-Djava.security.policy** system property, use the

alternative method described in "Specifying policy files to apply to all JVMs." If you specify a policy file to be used for all JVMs, remember that to activate Java security for your JVMs, you still need to add the **-Djava.security.manager** system property to your JVM profile or optional JVM properties files to enable a Java security manager.

Specifying policy files to apply to all JVMs

As an alternative to using the **-Djava.security.policy** system property to specify additional policy files, you can name the additional policy files in the JVM default **security properties file**, which applies to all JVMs.

This file is where the default Java security manager looks for the name of the default policy file, which it always uses to determine the security policy for a JVM.

The default security properties file is called `java.security` and is provided in:

```
/usr/lpp/java/J6.0/lib/security/java.security
```

where the `java/J6.0` subdirectory names are your install location for the IBM SDK for z/OS, Java Technology Edition on z/OS UNIX.

The default security properties file already includes the name of the default policy file, `/usr/lpp/java/J6.0/lib/security/java.policy`. You can add the names of additional policy files, and the security manager will then use these files, as well as the default policy file, to determine the security policy for **all** JVMs. The security manager will also refer to any policy files that you have specified in the JVM profile or optional JVM properties file for a particular type of JVM.

In the default security properties file `java.security`, policy files are specified in the form:

```
policy.url.n=URL
```

where `n` represents the precedence number for the order in which the policies should be loaded. The location of a policy file is specified as a URL, so policy files do not need to be stored in the local file system.

Note that the precedence numbers must be serial and continuous. For example, if `policy.url.1` and `policy.url.3`, are present, but `policy.url.2` is missing, then `policy.url.3` is ignored and only `policy.url.1` is considered.

The default security properties file `java.security` contains these two entries:

```
policy.url.1=file:${java.home}/lib/security/java.policy
policy.url.2=file:${user.home}/.java.policy
```

To specify the CICS-supplied enterprise beans policy file, `dfjejbpl.policy`, as an additional policy file to be used for all JVMs, add the entry:

```
policy.url.3=file:/usr/lpp/cicsts/cicsts41/lib/security/dfejbpl.policy
```

where *cicsts41* is your chosen value for the USSDIR installation parameter that you defined when you installed CICS TS. It is specified as `policy.url.3` because two other policy files are already specified. You can substitute the path to your own policy file in place of `dfjejbpl.policy`, or add further entries to specify additional policy files.

It is possible to bypass the default security properties file `java.security` for a JVM. You can do this by specifying your own policy file on the **-Djava.security.policy**

system property for the JVM, and inserting a double equals sign (= =). For example, if you include the system property:

```
-Djava.security.policy==/usr/lpp/cicsts/cicsts41/lib/security/dfejbpl.policy
```

then the security manager ignores any policy files that are specified in the `java.security` file, and uses only `dfjejbpl.policy` to determine the security policy for the JVM. However, you should bear in mind that if you bypass the default security properties file, the security manager will not grant any permissions that are specified in that file; it will only grant the permissions that are specified in your own policy file.

The CICS-supplied enterprise beans policy file

The CICS-supplied enterprise beans policy file, `dfjejbpl.policy`, is based on the Java security policy mechanism.

The Java security policy mechanism is described in the Sun Microsystems *Enterprise JavaBeans Specification, Version 1.1*, which is available at `http://www.javasoft.com/products/ejb`. The sample policy file is shown in Figure 34 on page 350.

In Java, the security policy is defined in terms of protection domains which map permissions to code sources. A protection domain contains a code source with a set of associated permissions.

The CICS-supplied enterprise beans policy file defines two protection domains, which do the following:

1. Grants the required permissions to the CICS enterprise beans Container code source for execution. See the 'grant codeBase' block in Figure 34 on page 350.
2. Grants any code source only the permissions outlined in the *Enterprise JavaBeans* specification, Version 1. See the default 'grant' block in Figure 34 on page 350:
   - To allow anyone to initiate a print job request.
   - To allow outbound connection on any TCP/IP ports.
   - To allow all system properties to be read.

Remember that if you want to use JDBC or SQLJ from enterprise beans, you need to amend the CICS-supplied enterprise beans policy file to grant permissions to the JDBC driver. Requirements to support Java programs in the CICS DB2 environment, in the *CICS DB2 Guide*, tells you how to do this.

```
// permissions granted to CICS enterprise beans Container codesource protection
//domain
   grant codeBase "file:usr/lpp/cicsts/cicsts41//-" {
     permission java.security.AllPermission;
   };

// default EJB 1.1 permissions granted to all protection domains
   grant {
     // allows anyone to initiate a print job request
     permission java.lang.RuntimePermission "queuePrintJob";

     // allows outbound connection on any TCP/IP ports
     permission java.net.SocketPermission "*:0-65535", "connect";

     // allows anyone to read properties
     permission java.util.PropertyPermission "*", "read";
   };
```

Figure 34. Sample CICS enterprise beans security policy

Using enterprise bean security

The EJB 1.1 specification defines the following security APIs to allow enterprise beans to make application decisions based on their callers' security details.

java.security.Principal getCallerPrincipal()
: This method is used to determine who invoked the current bean method. The getCallerPrincipal method is fully supported in CICS. Details of the way that the identity of the current caller is determined are shown in "Deriving distinguished names" on page 352.

boolean isCallerInRole(String SecurityRoleReference)
: This method is used to test whether the current caller is assigned to a security role that is linked to the security role reference specified on the method call.

CICS will throw a runtime exception (which conforms to the EJB 1.1 specification) if the following deprecated EJB 1.0 security APIs are used.

- java.security.Identity getCallerIdentity()
- boolean isCallerInRole(java.security.Identity role)

Note: Note that enterprise beans developed to the Enterprise JavaBeans (EJB) 1.0 specification need to be upgraded to the Enterprise JavaBeans 1.1 specification level, using the supplied development tools.

- See "The deployment tools for enterprise beans in a CICS system" on page 303 for information about deployment tools.
- See Chapter 22, "Writing enterprise beans," on page 289 for information about writing enterprise beans.
- See "The deployment tools for enterprise beans in a CICS system" on page 303 for information about deployment tools.
- See Chapter 22, "Writing enterprise beans," on page 289 for information about writing enterprise beans.

Defining file access permissions for enterprise beans

To successfully run enterprise beans in CICS, the CICS region userid must be permitted to access the files used by the enterprise logic.

These file permissions are required to run enterprise beans, regardless of the level of security implemented. See also the *CICS Transaction Server for z/OS Installation Guide*.

Access to z/OS UNIX files used by enterprise beans

These file permissions are required to run enterprise beans.

Table 16. File access permissions required for CICS enterprise beans

File/Directory structure	Minimum permission	Comments
CORBASERVER Shelf directory (for example, `/var/cicsts/`)	Read, write and execute	The shelf is accessed during CORBASERVER and DJAR installation, and each CICS needs to create unique subdirectories (see note 1).
`/usr/lpp/cicsts/cicsts41` directory structure and classes	Read and execute	Contains the CICS-supplied Java code (see note 2).
`/usr/lpp/java/J6.0/bin` and `/usr/lpp/java/J6.0/bin/classic` directories	Read and execute	Contain the IBM JVM code (see note 3).
CICS working directory	Read, write and execute	Used to create `stdin` files (see note 4).
Deployed jar file	Read	Used during DJAR installation by the deployment process.
Security policy file (if required)	Read	Required if the **`-Djava.security.policy`** property is specified in the JVM system properties file.
System properties file	Read	Optional when creating a JVM (see note 5).

Note:

1. `/var/cicsts/` is the default SHELF directory name when you define a CORBASERVER resource definition. Each CICS region creates a unique subdirectory in this shelf when it installs the resource definition
2. *cicsts41* is your chosen value for the USSDIR installation parameter that you defined when you installed CICS TS.
3. `java/J6.0` is your install location for the IBM SDK for z/OS, Java Technology Edition.
4. The CICS working directory is defined by the WORK_DIR parameter in the JVM profile.
5. The optional system properties directory and file name are named on the JVMPROPS option in the JVM profile.

File ownership and permissions may be defined using the **chmod** and **chown** commands. For more information, see *z/OS UNIX System Services Command Reference*.

Access to data sets used by enterprise beans

Before CORBASERVERs can be installed in a CICS region, the following two data sets must be created with UPDATE access, defined to CICS and installed. These files can be VSAM data sets or coupling facility data tables.

Figure 35 on page 352 shows an example of RACF commands to access data sets with the necessary authorization.

Note: These files are used internally by CICS, so no users should be given resource level security access to them. This will prevent VSAM applications from accessing the data in these files.

DFHEJDIR
This data set contains a request streams directory which is shared by the listener regions and AORs comprising a CICS IIOP server. The file must be recoverable.

DFHEJOS
DFHEJOS is a data set containing passivated stateful session beans. It is shared by all the AORs comprising a CICS IIOP server. This file must not be recoverable.

```
ADDSD  'CICSTS41.CICS.CICS.DFHEJDIR' NOTIFY(cics_sys_admin_id) UACC(NONE)
PERMIT 'CICSTS41.CICS.CICS.DFHEJDIR' ID(cics_id1,...,cics_group1,..,cics_groupn)
        ACCESS(UPDATE)
ADDSD  'CICSTS41.CICS.CICS.DFHEJOS'  NOTIFY(cics_sys_admin_id) UACC(NONE)
PERMIT 'CICSTS41.CICS.CICS.DFHEJOS'  ID(cics_id1,...,cics_group1,..,cics_groupn)
        ACCESS(UPDATE)
```

Figure 35. An example of RACF commands used to authorize access to CICS data sets

See Authorizing access to CICS data sets, in the *CICS RACF Security Guide*, for more information about authorizing access to CICS data sets.

Deriving distinguished names

Enterprise beans can identify their end-user, or client, by means of a *Principal* object.

The getCallerPrincipal method returns a Principal object representing the client, and that Principal object contains methods that can be invoked to return information about the client. In particular, the getName method of the Principal object returns a String that contains the "distinguished name" of the client. The distinguished name, or DN, is a sequence of keyword and value pairs, known as relative distinguished names, or RDNs, and forms part of the X.500 recommendation (Standard ISO/IEC 9594). The string representation of a distinguished name is suggested by RFC2253, *LDAP V3: UTF-8 String Representation of Distinguished Names.*

Note: CICS Transaction Server for z/OS, Version 4 Release 1 does not verify that a stateful session bean instance is used only by the same principal that created it. Therefore the principal's userid and distinguished name may be different after a bean instance has been reactivated.

If the bean's client has been identified and authenticated by means of a client certificate using the secure sockets layer protocol, the distinguished name is always obtained from that certificate. However, if the bean's client has not provided a certificate, the distinguished name is obtained by invoking the DFHEJDNX user-replaceable module. The inputs to the DFHEJDNX module are the title, organizational unit, organization, locality, state, and country, obtained from the server certificate whose label is specified in the CERTIFICATE option of the CORBASERVER definition, and the userid and common name associated with the user ID of the user executing the bean, but if SEC=NO is specified, the CICS region userid is used. The common name is derived by transforming the username for that user to a mixed-case string.) The certificate label specifies a certificate within the key ring identified by the KEYRING system initialization parameter. If the

CERTIFICATE option is omitted, information is obtained from the default certificate in the key ring. If the KEYRING parameter is omitted, no certificate information is passed to DFHEJDNX, and only the common name RDN is available.

The CICS-supplied version of DFHEJDNX accepts the inputs derived from the CORBASERVER certificate and the username, and formats them into a distinguished name in the following style:

T=CICS EJB Container,CN=Louise Peters,OU=CICS/390 Development, O=IBM,L=Hursley,ST=Hampshire,C=GB

CICS-supplied samples of DFHEJDNX are located in the SDFHSAMP library, *CICSTS41.CICS.CICS.SDFHSAMP*, as:

- DFHEJDN1 for Assembler language
- DFHEJDN2 for C language

Security roles

Access to enterprise bean methods is based on the concept of **security roles**.

A security role represents a type of user of an application in terms of the permissions that the user must have to successfully use the application. For example, in a payroll application:

- A manager role could represent users who are permitted to use all parts of the application
- A team_leader role could represent users who are permitted to use the administration functions of the application
- A data_entry role could represent users who are permitted to use the data entry functions of the application

The security roles for an application are defined by the application assembler, and are specified in the bean's deployment descriptor. For more information, see “Security roles in the deployment descriptor” on page 357

The security roles that are permitted to execute a bean method are also specified in the bean's deployment descriptor, again by the application assembler. In the example, methods which update the hours worked by employees each week might be assigned to the data_entry role, while methods which delete an employee from the payroll might be assigned to the team_leader role.

To distinguish similarly named security roles in different applications, or in different systems, the security roles specified in the bean's deployment descriptor can be given a one- or two-part qualifier when the bean is deployed in a CICS system. For example:

- Security role with no qualifiers:

  ```
  team_leader
  ```

- Security role with one qualifier:

  ```
  payroll.team_leader
  ```

- Security role with two qualifiers:

  ```
  test.payroll.team_leader
  ```

A security role with its qualifiers is known as a **deployed security role**. For more information, see “Deployed security roles” on page 354.

The mapping of security roles to individual users is done in the external security manager. The mapping is not necessarily one-to-one. For example, several users might be assigned to the data_entry role, while a some users might be assigned to both the team_leader role and the data_entry role. For more information, see "Implementing security roles" on page 359.

The security role and display name in the deployment descriptor can contain any ASCII or Unicode character. This is not so for names used in RACF, which are restricted to characters in EBCDIC code page 037. In addition, some characters — the asterisk (*) for example — have special meaning when used in RACF commands. Therefore, when CICS constructs the deployed security role from its components, some characters are replaced with a different character, and others are replaced with an escape sequence. For details, see "Character substitution in deployed security roles" on page 356.

Deployed security roles

A direct mapping between the security roles specified in a bean's deployment descriptor and individual users may not adequately control access to bean methods.

For example

- Two applications, provided by different suppliers, might use similar names for security roles. In your enterprise, the users of each application might be different.
- A bean could be used in more than one application. A user may be entitled to use a particular method in one application, but not in the other.
- An application could be deployed in a test system and a production system. Members of the test department may be permitted to use all bean methods in the test system, but not in the production system.

To provide the degree of control that is needed in these and other cases, you can qualify the security roles at the application level and the system level. A security role with its qualifiers is known as a **deployed security role**. Here is an example of a role name which is qualified at both levels:

```
test.payroll.team_leader
```

- `payroll` qualifies the security role at the application level, and is used to distinguish between the `team_leader` role in the payroll application and the `team_leader` role in other applications.
- `test` qualifies the security role at the system level, and is used to distinguish between the payroll.team_leader role in the test system and the payroll.team_leader role in other systems.

At the application level, security roles are qualified by the **display name**, if one is specified in the deployment descriptor. If a display name is not specified, the security roles are not qualified at the application level. If an application level qualifier is used, a period (`.`) is used as the delimiter; if no qualifier is used, there is no delimiter.

At the system level, security roles are optionally qualified with a prefix which is specified in the EJBROLEPRFX system initialization parameter. If EJBROLEPRFX is not specified, the security roles are not qualified at the system level. If a system level qualifier is used, a period (`.`) is used as the delimiter; if no qualifier is used, there is no delimiter.

This example shows how security roles defined in a bean's deployment descriptor can be qualified:

- A bean contains three security roles: manager, team_leader, and data_entry
- The bean is used in a payroll application, with a display name of `payroll`. The bean is also part of a test application, which does not have a display name.
- The payroll application is used on two production systems: the first does not specify a prefix, while the second specifies a prefix of `executive`.
- The test application is used on a test system with a prefix of `test1`.

When the two levels of qualification are applied to the security roles specified in the deployment descriptor, the deployed security roles are:

```
payroll.manager        executive.payroll.manager        test1.manager
payroll.team_leader    executive.payroll.team_leader    test1.team_leader
payroll.data_entry     executive.payroll.data_entry     test1.data_entry
```

Each of these deployed roles can be mapped to individual users (or groups of users) to suit the security need of the enterprise.

If a security role is not qualified at the application level, or at the system level, then the deployed security role is the same as the security role defined in the deployment descriptor. For example, if the bean in the previous example is used in an application which does not have a display name, and the application is used in a system that does not specify EJBROLEPRFX, then the deployed security roles are:

```
manager
team_leader
data_entry
```

Enabling and disabling support for security roles

By default, CICS support for security roles is enabled.

You can use the XEJB system initialization parameter to disable (or explicitly enable) support for security roles. If you disable the support:

- CICS does not perform method authorization checks: all users are permitted to use all bean methods.
- The isCallerInRole() method returns `true` for all users.

Security role references

Within an application, the isCallerInRole() method can be used to determine if the user of the application is defined to a given role.

The method takes a **security role reference** as an argument, rather than a security role. The security role references coded in the bean are defined by the bean provider, and declared in the bean's deployment descriptor.

For more information, see “Security roles in the deployment descriptor” on page 357

Each security role reference is linked to a security role by the application assembler; the linkage is declared in the deployment descriptor for the bean. For example, the security role reference of administrator used within the bean's code might be linked, in the deployment descriptor, to the team_leader role.

For more information, see “Security roles in the deployment descriptor” on page 357

Character substitution in deployed security roles

The security role and display name in the deployment descriptor can contain any ASCII or Unicode character.

The character set which can be used in deployed security roles is more restricted:

- Profile names used in RACF are restricted to characters in EBCDIC code page 037.
- Some characters — the asterisk (*) for example — have special meaning when used in RACF commands, and cannot be used in a profile name.

When Unicode characters in the security role and display name cannot be used directly in the deployed security role, they are replaced by the escape sequences shown in Table 17. Substitution occurs:

- when the EJBROLE generator utility (`dfhreg`) processes the deployment descriptor to generate RACF commands
- when CICS maps a security role to a RACF user ID

Table 17. Escape sequences used in security roles

Character	Description	ASCII/Unicode	EBCDIC code page 037	Escape sequence
ASCII and Unicode values whose equivalent EBCDIC value cannot be used in a deployed security role name are replaced with a three-character escape sequence as follows:				
	blank	X'20'	X'40'	¢
¢	cent	X'A2'	X'4A'	\A2
\	backslash	X'5C'	X'E0'	\5C
*	asterisk	X'2A'	X'5C'	\2A
&	ampersand	X'26'	X'50'	\26
%	per cent	X'25'	X'6C'	\25
,	comma	X'2C'	X'6B'	\2C
(	left parenthesis	X'28'	X'4D'	\28
)	right parenthesis	X'29'	X'5D'	\29
;	semicolon	X'3B'	X'5E'	\3B
Unicode values which do not have an equivalent in EBCDIC code page 037 are replaced with the Unicode escape sequence: a character with a Unicode representation of X'yyyy' is replaced by `\uyyyy`. For example:				
€	Euro symbol	X'20AC'	not supported	\u20AC
	Hiragana Ki	X'304D'	not supported	\u304D
α	alpha	X'03B1'	not supported	\u03B1

Here are two examples that illustrate the way that characters are substituted:

Example 1

- The EJBROLEPRFX has a value of `test`
- The display name in the deployment descriptor has a value of `year.end.processing`
- The security role in the deployment descriptor has a value of `auditor 1`

In this example, when the deployed security role is constructed:

1. Each space is replaced with ¢

2. The deployed security role is composed from the EJBROLEPRFX value, the display name, and the security role; a period is used as the delimiter.

The resulting deployed security role is:

```
test.year.end.processing.auditor¢1
```

Example 2

- The EJBROLEPRFX has a value of `test`
- The display name in the deployment descriptor has a value of `αβ32`. The Unicode encoding is X'03B1 03B2 0033 0034'.
- The security role in the deployment descriptor has a value of `auditor 1`

In this example, when the deployed security role is constructed:

1. Each Unicode character that has an equivalent in EBCDIC code page 037 is replaced accordingly: In the display name, X'0033 0034' is replaced by `34`.
2. Each Unicode character that does ***not*** have an equivalent in EBCDIC code page 037 is replaced with the corresponding escape sequence. In the display name, X'03B1 03B2' is replaced by `\u03B1\u03B2`
3. Each space is replaced with `¢`
4. The deployed security role is composed from the EJBROLEPRFX value, the display name, and the security role; a period is used as the delimiter.

The resulting deployed security role is:

```
test.\u03B1\u03B234.auditor¢1
```

Security roles in the deployment descriptor

This shows a fragment of a deployment descriptor. It includes the following security role information.

- 1 A display name of `payroll`.
- 2 The security role reference of `administrator` which is linked to the `team_leader` role.
- 3 A security role of `team_leader`.
- 4 A method permission that allows a user defined in the `team_leader` role to invoke the create() method.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC
"-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 1.1//EN"
"http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">
  <ejb-jar id="ejb-jar_ID">
    <display-name>payroll</display-name>         1
      <enterprise-beans>
        <session id="Session_1">
          .
          .
          <security-role-ref id="SecurityRoleRef_1">
            <role-name>administrator</role-name> 2
            <role-link>team_leader</role-link>
          </security-role-ref>
          .
          .
        </session>
      </enterprise-beans>
      <assembly-descriptor id="AssemblyDescriptor_1">
        <security-role id="SecurityRole_1">
          <role-name>team_leader</role-name>     3
        </security-role>
        .
        .
        <method-permission id="MethodPermission_1">
          <description>team_leader:+:</description>
          <role-name>team_leader</role-name>     4
          <method id="MethodElement_01">
            <ejb-name>Managed</ejb-name>
            <method-intf>Home</method-intf>
            <method-name>create</method-name>
            <method-params>
            </method-params>
          </method>
           .
           .
        </method-permission>
         .
         .
      </assembly-descriptor>
       .
       .
    </ejb-jar>
```

Figure 36. Example of a deployment descriptor containing security roles

If an application with this deployment descriptor is used in a CICS system with the following system initialisation parameters:

```
SEC=YES
XEJB=YES
EJBROLEPRFX='test'
```

- The deployed security role of `test.payroll.team_leader` must be defined to RACF.
- Users that have READ access to that deployed security role will be permitted to invoke the create() method.
- `isCallerInRole('administrator')` will return `true` for users defined in the deployed security role of `test.payroll.team_leader`, and `false` for other users.

For detailed information about the contents of the deployment descriptor, refer to *Enterprise JavaBeans Specification, Version 1.1*.

To view the contents of a deployment descriptor, you can use the Assembly Toolkit (ATK). For more information about ATK, see The enterprise bean deployment tool, ATK, in the *CICS Operations and Utilities Guide*.

Implementing security roles

Access to enterprise bean methods is based on the concept of **security roles**.

About this task

These are described in "Security roles" on page 353.

To implement the use of security roles in a CICS enterprise bean environment, you must:

1. Determine which security roles are defined in the application's deployment descriptor.
2. Determine the display names associated with the security roles in the application's deployment descriptor. The display name qualifies the security role at the application level.
3. Decide whether you need to qualify the security role name at the system level, and — if you do — the value of the prefix which you will use in each system where the application executes.
4. Using the information gathered in steps 1 through 3, determine the names of the deployed security roles used by the application in each system. Characters in the security role and display name that do not have a direct equivalent in EBCDIC code page 37 (and some other characters) must be replaced with a different character or an escape sequence when constructing the deployed security role. See "Character substitution in deployed security roles" on page 356 for more information.
5. Using the information gathered in steps 1 through 3, define RACF profiles for the deployed security roles. See "Defining security roles to RACF" on page 361 for more information.
6. Associate individual users or groups of users with each deployed security role in RACF. See "Defining security roles to RACF" on page 361 for more information.
7. Specify these system initialization parameters:
   - `SEC=YES`
   - `XEJB=YES`. This is the default value, so you do not need to specify it explicitly.
8. For those systems where the deployed security roles contain a system level qualifier (see step 3), specify the EJBROLEPRFXEJBROLEPRFX system initialization parameter.

Using the RACF EJBROLE generator utility

The RACF EJBROLE generator utility (`dfhreg`) is a Java application program that extracts security role information from deployment descriptors, and generates a REXX program which can be used to define security roles to RACF.

The REXX program that `dfhreg` generates contains the RACF commands that define security roles as members of a profile in the GEJBROLE class. Before you run the REXX program, you will need to modify it, in order to change the name of the profile that is defined.

The dfhreg invocation scripts for USS (dfhreg) and for Windows (dfhreg.bat) are in the CICS_HOME/lib/security directory. The implementation of dfhreg (dfhreg.jar) is also in this directory. The other JAR files required to run dfhreg (dfjcsi.jar, dfjejbdd.jar, and dfjorb.jar) are in the CICS_HOME/lib directory. CICS_HOME is the z/OS UNIX directory in which you have installed the USS components of CICS.

You can execute dfhreg on any platform that supports Java; however, you must execute the resulting REXX program against the RACF database on the z/OS system where you want to define the security roles. When you run dfhreg:

1. Your classpath must contain:

   ```
   dfhreg.jar
   dfjcsi.jar
   dfjejbdd.jar
   dfjorb.jar
   ```

2. You must be using a 1.4 or later version of the Java 2 SDK.

The REXX program which the utility generates is in the code page of the platform where the utility executes. If you run the utility on a platform that uses an ASCII code page, you must convert the REXX program to the EBCDIC code page used on the target z/OS system.

Executing the utility

About this task

To execute the utility enter the following on the command line:

```
dfhreg [options] inputfiledesc
```

The full syntax is

```
dfhreg [-secprfx secprfx]
       [-out outputfiledesc]
       [-f | -force]
       [-v | -verbose]
       [-? | -help]
       inputfiledesc
```

where

–secprfx *secprfx*
: Specifies the name used to qualify the security role name at system level. The value you specify must match the value of the EJBROLEPRFXEJBROLEPRFX system initialization parameter for the CICS system where the security roles will be used

out *outputfiledesc*
: Specifies the file which to which the utility writes its output. If you do not specify a file, output will be written to standard output.

inputfiledesc
: Specifies the input file containing the deployment descriptor. The file must be a Java archive file (file type jar).

–f | –force
: Specifies that the utility will overwrite an existing output file.

–v | –verbose
: Specifies that processing messages will be written to standard output.

–? | –help
Displays a summary of the syntax for the utility.

All options are case sensitive; the keywords (`-secprfx`, `-out`, `-force`, `-f`, `-verbose`, `-v`, `-help`) must be entered in lower case.

If the utility encounters an error, it generates one or more messages. These are described in the *CICS Messages and Codes* manual.

Defining security roles to RACF

In RACF, deployed security roles are managed as general resources. To define the deployed security roles, define profiles in the GEJBROLE or EJBROLE resource classes, with appropriate access lists.

For example, to use the following commands to define deployed security roles `deployed_security_role_1`and `deployed_securityrole_2` as members of the `securityrole_group` profile in the GEJBROLE class, and give READ access to `user1` and `user2`:

```
RDEFINE GEJBROLE securityrole_group UACC(NONE)
                ADDMEM(deployed_security_role_1, deployed_securityrole_2, ...)
                NOTIFY(sys_admin_userid)
PERMIT securityrole_group CLASS(GEJBROLE) ID(user1, user2) ACCESS(READ)
```

Alternatively, use the following commands to define deployed security roles in the EJBROLE class, and to give users READ access to each deployed security role:

```
RDEFINE EJBROLE (deployed_security_role1, deployed_security_role2, ...) UACC(NONE)
                NOTIFY(sys_admin_userid)
PERMIT deployed_security_role1 CLASS(EJBROLE) ID(user1, user2) ACCESS(READ)
PERMIT deployed_security_role2 CLASS(EJBROLE) ID(user1, user2) ACCESS(READ)
```

Note:

1. The security role you specify is the deployed security role, and not the unqualified security role which is defined in the deployment descriptor.
2. To execute a bean method, or to receive a `true` response from the isCallerInRole() method, a user requires READ access.

Chapter 28. CICSPlex SM with enterprise beans

The management of enterprise beans may be undertaken at a CICSplex wide level.

CICSPlex SM support for enterprise beans

The management of enterprise beans may be undertaken at a CICSplex wide level, by utilizing the Operator and API services of CICSPlex SM.

The function provided by CICSPlex SM for the support of Enterprise JavaBeans includes:

- Object management for CorbaServer and DJAR definitions
- Object management for installed CorbaServer and DJAR instances
- Dynamic management of enterprise bean execution

The CICSPlex SM areas that cover these facilities are:

- The application programming interface (API) - to allow the definition, inquiry and management of enterprise bean objects through the **EXEC CPSM** interface. See the *CICSPlex System Manager Application Programming Guide* for information.
- The web user interface - to allow the inquiry and management of enterprise bean objects through an http browser such as Internet Explorer and Netscape Navigator. See the *CICSPlex System Manager Web User Interface Guide* for information about the Web User Interface.

CICSPlex SM definition support for enterprise beans

Business Application Services (BAS) is the CPSM component concerned with the definition and installation of CICS resources

For more information on BAS, see the *CICSPlex System Manager Managing Business Applications* manual. The BAS objects that are specific to Enterprise JavaBeans are:

- EJCODEF—enterprise bean CorbaServer definition
- EJDJDEF—enterprise bean CICS-deployed JAR file definition

The CorbaServer definition object (EJCODEF) allows the specification of exactly the same CorbaServer characteristics as the CEDA version. EJCODEF is described in Defining CorbaServers using BAS, in the *CICSPlex System Manager Managing Business Applications* manual

The CICS-deployed JAR file definition object (EJDJDEF) allows the specification of exactly the same DJAR characteristics as the CEDA version. EJDJDEF is described in Defining a CICS-deployed JAR file using BAS, in the *CICSPlex System Manager Managing Business Applications* manual.

These resources are fully integrated into the standard BAS functionality, and they may be managed and installed automatically, or on an ad hoc basis as a user may require.

In addition to these two object types, there are some other BAS objects that are related to enterprise bean operation:

- TCPDEF—TCPIPSERVICE definition

- RQMDEF—REQUESTMODEL definition
- TRANDEF—CICS TRANSACTION definition
- PROGDEF—PROGRAM definition

Enterprise bean execution requests from clients reach the CICS listener region through a TCP/IP port. If using BAS, the number of this port must be specified through a TCPDEF object that should be installed at all listener regions expected to respond to these calls. The content of a TCPDEF should mirror that specified for the CEDA TCPIPSERVICE definition. See “Setting up TCP/IP for IIOP” on page 193 for information.

If users require the execution requests for specific enterprise beans to be recognized and managed differently to that for generic enterprise bean executions, then a request model may be used to associate it with a user specified transaction code. Within CICSPlex SM, request models are defined through RQMDEF objects, and should be installed on all listener regions where such requests need interception. Depending on the complexity of the enterprise bean, it may be necessary to additionally install the request models on the associated AORs. The contents of these RQMDEFs should mirror that specified for the CEDA REQUESTMODEL definition. See “Obtaining a CICS TRANSID” on page 204 for information.

In a distributed enterprise bean processing environment, it would be expected that certain CICS regions will act as listeners to receive the IIOP execution requests, and others will act as the AORs, to provide the actual EJB environment for execution of the required enterprise beans. The CICSPlex SM TRANDEF object is a particularly powerful tool to employ here, because a single transaction definition object may be installed both dynamically on the Listener regions, and statically on the AORs, through a single BAS resource assignment (RASGNDEF), as described in Resource assignments, in the *CICSPlex System Manager Managing Business Applications* manual.

BAS logical scope considerations

One of the benefits of using BAS to define and install user business application suites, is that users may then scope their object views to the resources pertinent to their installed application instances.

For example, if a business application comprises of a particular set of files, transactions, and programs, the LOCTRAN, LOCFILE and PROGRAM views will be isolated to instances of only the matching objects on the regions where they are installed. The facility to allow this restricted object view is known as "logical scoping". The CorbaServer and DJAR objects may participate in logical scoping in exactly the same way as other traditional BAS definitions.

Note: Enterprise beans are not defined to CICS as such. They become identified to CICS when their associated DJARs come into service after installation in a CICS region. Therefore, enterprise beans may "adopt" a logical scope through the association of their DJAR. However, the Enterprise JavaBean specification allows the enterprise beans for different applications, to be installed in a single DJAR. If you follow this practice, it will be impossible for the logical scope process to differentiate between the installed enterprise beans and the appropriate business application names. As such, if users want to exploit BAS logical scoping to

augment their CICSPlex views of enterprise bean objects, separate DJARs should be employed to contain enterprise beans discrete to the scoped business applications.

Migration of enterprise bean components

CICSPlex SM provides a toolset to assist users in migrating their RDO (resource definition online) objects from the CICS CSD to the CICSPlex SM data repository.

This toolset comprises an exit program for the CICS offline CSD utility program, and some sample JCL to execute it: see Extracting records from the CSD, in the *CICSPlex System Manager Managing Business Applications* manual.

This CICSPlex SM exit will recognise CORBASERVER and DJAR definitions in a CSD, and generate the appropriate BAS CREATE EJCODEF and CREATE EJDJDEF statements, for input via the CICSPlex SM BatchRep process. All of the normal selection rules for resource identification may be applied to these EJB resource types.

CICSPlex SM inquiry support for enterprise beans

Installed CorbaServer and DJAR instances may be managed by CICSPlex SM through any of the three interfaces described in "CICSPlex SM support for enterprise beans" on page 363. All of the interactive operator services provided through the CICS CEMT and CEOT transactions are functionally replicated in CICSPlex SM via the Web user Interface (WUI). In either case, the installed CICS objects mapped by CICSPlex SM are:

- EJCOSE—CorbaServer instances
- EJDJAR—CICS-deployed JAR file instances

Additionally, any executable enterprise beans may be listed through these objects:

- EJCOBEAN—Enterprise JavaBeans directly associated with a CorbaServer
- EJDJBEAN—enterprise beans directly associated with a DJAR

Both of these objects describe an enterprise bean structure: one is keyed through a CorbaServer name, and the other is keyed through a DJAR id. In both cases, the only enterprise bean content available for inquiry is the CorbaServer name, the DJAR name, and the enterprise bean name up to 240 characters in length. The Enterprise JavaBean specification states that enterprise bean names may be much longer, but the CICS implementation limits them to 240 bytes. An additional detail that CICSPlex SM inquiries provide over a standard CICS inquiry is a count of the available beans in any given DJAR or CorbaServer. When a new set of enterprise beans are deployed via a DJAR to a particular CorbaServer, the enterprise bean count can provide an instant confirmation as to the availability of the enterprise beans in question. The value is incremented according to the number of enterprise beans accepted through the DJAR installation process.

Other Enterprise Java associated CICS objects that are inquirable through CPSM are:

- TCPIPS - TCPIPSERVICE instances
- RQMODEL—REQUESTMODEL instances
- LOCTRAN—local transaction instances
- UOWORK—unit of work instances
- UOWLINK—unit-of-work-link (UOWLINK) instances

- PROGRAM—program instances

All of these objects include attributes which have relevance to the management and execution of enterprise beans.

Types of inquiry available for enterprise bean objects

There are several ways to inquire on the state of your EJB objects with CICSPlex SM.

The CICSPlex SM Application Programming Interface
To inquire on EJB objects using the available CICSPlex SM API commands, refer to the *CICSPlex System Manager Application Programming Reference*. Also refer to the details of the attributes and actions that are allowed against each CICSPlex SM object in the *CICSPlex System Manager Resource Tables Reference*.

The CICSPlex SM Web User Interface
To inquire on EJB objects using the WUI, refer to the *CICSPlex System Manager Web User Interface Guide*.

The Web User Interface has a starter set that comprises a set of menus and panels. This starter set includes a set of Enterprise Java component views.

Using CICSPlex SM to manage EJB workloads

One of the standard CICSPlex SM component functions is the facility for balancing and separating CICS transactions in an MRO environment, known as workload management (WLM).

This facility is well suited to the management of EJB workloads, where the enterprise beans are executed in a distributed, or logical CorbaServer, environment. In its most simple configuration, CICSPlex SM can balance an enterprise bean execution workload across a series of application owning regions (AORs), depending on performance targets and stability algorithms established by user definitions. These functions are implemented when the CICSPlex SM supplied distributed routing exit program (EYU9XLOP) is named as the DSRTPGM parameter in the system initialisation parameters of participating listeners and AORs (see Balancing an enterprise bean workload, in the *CICSPlex System Manager Managing Workloads* manual).

The algorithms used by CICSPlex SM to select suitable AORs for enterprise bean execution has been established and tuned since the inception of the product. However, users may choose to develop their own routing algorithm program, and replace the supplied CICSPlex SM version (EYU9WRAM) if they require to do so.

Workload balancing

CICSPlex SM workload balancing provides function that allows the most suitable AOR to be selected to host the execution of an enterprise bean, according to predetermined selection criteria specified by a Systems Administrator.

Note: Note that this AOR selection process evaluates all concurrent execution activity, over the regions designated as possible routing targets, and selects the most suitable region in terms of execution workload, and region stability at the point of inquiry. This is **not** the same as the cyclic selection of an AOR from all those available in a target scope for serially executed beans. It**is** the evaluation of all active transactions within the WLM scope at the time when a new transaction

(enterprise bean) is about to be executed, and the selection of the least loaded, or most stable, region to host the object execution.
The implementation of simple workload balancing for all Enterprise Java bean throughput has these prerequisites:

- The necessary TCP/IP definitions are installed on the designated listener regions
- DSRTPGM=EYU9XLOP is specified as a SIT parameter on all listeners and AORs
- MASPLTWAIT(YES) is included as an EYUPARM on all of the listener regions
- The request processor transaction (the default transaction is CIRP) has been dynamically defined to the listener regions and statically defined to the AORs
- The necessary CorbaServer and DJAR definitions are installed (either through BAS or CEDA) to establish the executable EJB environment
- The enterprise beans have been deployed and are INSERVICE

When the listed criteria have been met, the implementation of EJB workload balancing is relatively simple. A simple workload specification object (WLMSPEC) needs to be defined specifying the AORs as the target scope. The WLMSPEC object then needs to be installed on all listeners and AORs that are to join the workload. When the WLMSPEC has been installed, all regions encompassed by it will have their EJB workloads balanced after they have been restarted. A detailed example of enterprise bean workload balancing is given in Balancing an enterprise bean workload, in the *CICSPlex System Manager Managing Workloads* manual.

Workload separation

Workload separation is the WLM function that causes transactions which meet predesignated selection criteria to be routed to specific target scopes.

The target scope for a separated workload item may vary from a single AOR to a large AOR group comprising many CICS regions. If an AOR group is the target, the balancing algorithm will be applied to select the most suitable region from those defined to it. To implement a workload that includes separated enterprise beans, you must first establish the prerequisite workload balancing described in "Workload balancing" on page 366. That configuration needs to be augmented with the following additional components:

- A cloned CIRP transaction for each enterprise bean that needs to be separated (a simple copy of the existing definition to a new name)
- A request model for each enterprise bean to be separated, to associate it with one of the cloned CIRP transactions

This will allow the CICS and EJB environments to be established enabling enterprise bean separation. The WLM definitions will then need to be created to implement it. This entails identifying the cloned CIRP transactions as being objects of interest, and associating them with the required target scopes through a series WLM definitions. These WLM definitions must be associated to an overall WLM specification, via an intermediate WLM group, and then the specification must be added to the CICS group that includes all listeners and AORs that are to participate in the workload. A detailed example of enterprise bean workload separation is given in Separating enterprise beans in a workload, in the *CICSPlex System Manager Managing Workloads* manual.

CICSPlex SM resource monitoring for enterprise beans

CICSPlex SM monitoring allows the collection of performance-related data, at user-defined intervals, for named resource instances within a set of CICS systems.

Currently, no performance-related data is recorded for specific EJB objects (CorbaServers and DJARs). However, performance data for the IIOP request receiver and request processor transactions are available as normal, and so the execution performance of enterprise beans may be monitored through an associated transaction code (see the *CICSPlex System Manager Monitor Views Reference*). Users will require request models and CIRP clones for each bean that needs to be monitored, in the same way as for enterprise bean workload separation, described in "Workload separation" on page 367. However, CICSPlex SM monitoring is not integrated with BAS logical scoping, so your monitor views scope should be set to the physical CICS group that covers the regions to be monitored, rather than the BAS resource description that installed the transaction definitions. An overview of the monitoring function is given in Collecting statistics using CICSPlex SM monitoring, in the *CICSPlex System Manager Concepts and Planning* manual. Full details of the monitoring function is given in Preparing to monitor resources, in the *CICSPlex System Manager Managing Resource Usage* manual.

CICSPlex SM real-time analysis considerations for enterprise beans

The real-time analysis (RTA) function of CICSPlex SM provides the automatic and external notification of conditions in which users have expressed an interest.

Real-time analysis may be divided between several sub-components:

- System Availability Monitoring (SAM) - monitors CICS regions during their planned hours of availability, and generates notifications when no responses are received from a region that is expected to be active.
- MAS Resource Monitoring (MRM) - monitors the state of any inquirable CICS resource, and generates notifications when that state varies from a predetermined norm.
- Analysis Point Monitoring (APM) - replicates the function of MRM, except that it analyses states at a CICSplex level, rather than at a specific CICS region. APM is particularly useful in environments that use cloned AORs, where regions are identical and one notification is sufficient to alert you to a general problem.

Clearly SAM is a useful function for reporting the availability of CICS regions, regardless of whether they are designated listeners or AORs. If you are executing enterprise beans in a distributed environment, then MRM may be more useful for monitoring the state of CorbaServers and DJARs, rather than the region based functions of APM. However, be aware that you cannot monitor enterprise bean objects themselves (EJCOBEAN and EJDJBEAN) within RTA. Enterprise bean inquiries may be keyed only on their corresponding CorbaServer or DJAR names. Specific inquiries may not be made solely on the enterprise bean name. An overview of the RTA function is given in Exception reporting using real-time analysis (RTA), in the *CICSPlex System Manager Concepts and Planning* manual. Full detail of the RTA function is given in Preparing to perform real-time analysis, also in the *CICSPlex System Manager Managing Resource Usage* manual.

Part 6. Using stateless CORBA objects

This section tells you what you need to know to develop stateless IIOP applications.

Chapter 29. Stateless CORBA objects

From the client perspective, a stateless CORBA object invoked by means of the CICS ORB is just a collection of methods—that is, a stateless object.

Each remote method represents a piece of logic that may make one or more CICS API calls, including program-link calls, to existing CICS programs. CICS stateless CORBA objects execute in a CICS JVM. At the end of the remote method, the state of the object is no longer available.

As with all Java programs that execute in a continuous JVM in CICS, any static state created by a CORBA object is persisted within the JVM for subsequent retrieval in a later task. However, there is no affinity between a CORBA client and a CICS JVM, so there is no certainty that two subsequent CORBA requests that use the same socket will be processed in the same JVM (or even the same CICS region). This means that the availability of previously initialised static state cannot be relied upon.

Every remote method must therefore be passed sufficient information in its parameter list to enable it to complete its work. No information is passed to the server ORB by way of the object reference, except the object type, which is used to find the implementation class. However, the methods of the object may save state in application-managed data storage between invocations. They will need to ensure that sufficient information is passed as parameters to subsequent methods so that the saved state can be retrieved.

A CORBA object can make outbound IIOP calls, including calls to enterprise beans running under the same or under a different CorbaServer. A CORBA object can even pass a reference to itself as a parameter on a remote IIOP method. This is known as a **call back reference**. However, if the target object uses the call back reference to call the first CORBA object, this new request is processed in a new JVM; thus it has no access to any state from the original JVM.

Method invocations may participate in Object Transaction Service (OTS) **distributed transactions**. If a client calls an IIOP application in the scope of an **OTS transaction**, information about the OTS transaction flows as an extra parameter on the IIOP call. If a target stateless CORBA object implements CosTransactions::TransactionalObject, the object is treated as transactional.

Developing stateless CORBA objects

Stateless CORBA objects are Java server applications that communicate with a client application using the IIOP protocol. No state is maintained in object attributes between successive client invocations of remote methods; state is initialized at the start of each remote method call and referenced by explicit parameters.

Note: By a *remote method* we mean a method that may be called from a remote client. That is, a public method that is exposed as part of one of the object's (potentially multiple) remote interfaces, or declared in the IDL for the object; rather than an internal method that cannot be accessed from a remote client.

In the server programming model, each method is a subroutine. The parameters passed allow you to establish temporary state from any existing databases or applications, to perform business logic, to store data in the existing databases or applications, to return results when the subroutine returns, or to throw an exception. The remote methods of a stateless CORBA object—that is, those that may be called by a remote client—may call each other locally or call non-remote methods without the object's temporary state being lost. The temporary state is only discarded at the end of the client-initiated remote method request, when the response to the client's request is sent.

You can develop a stateless CORBA application using either of two different approaches:

1. Use the typical CORBA development style, whereby an application interface is defined in Interface Definition Language (IDL) and then the application is coded to that interface. This approach is described in the sections that follow.
2. Use the typical Java development style, whereby a Java Remote Method Invocation (RMI) application is developed and IDL is optionally generated later. This approach is known as RMI-IIOP. It is described in "Developing an RMI-IIOP stateless CORBA application" on page 380.

To develop a stateless CORBA object using the first (CORBA-style) approach, you need to perform the following steps:

1. Use the Interface Definition Language (IDL) to define the object's interfaces and operations.
2. Run the IDL-to-Java compiler (IDLJ) against the IDL to generate stub and skeleton classes for the object.
3. Write a client application that makes calls to the server using the generated stub class.
4. Write a server application (the stateless CORBA object) that extends the generated base skeleton class.
5. Compile and package the client and server applications.
6. Define CICS resources for the server and add the server application's JAR file to the standard class path in the JVM profile for the JVM that the application uses.

To develop a stateless CORBA object using the second (Java-style) approach, you need to perform the following steps:

1. Write a remote interface for the server application (the stateless CORBA object).
2. Write a client application that makes calls to the server using this remote interface.
3. Write a server application that implements the remote interface.
4. Compile the client and server applications.
5. Run the Java RMI compiler (RMIC) against the remote interface and server application to generate stub and tie classes for the object.
6. Package the client and server applications.
7. Define CICS resources for the server and add the server application's JAR file to the standard class path in the JVM profile for the JVM that the application uses.
8. Optionally, create IDL for the application for use by non-Java CORBA clients.

There are benefits and drawbacks to each of the two approaches. One of the main differences is that the CORBA approach requires the stateless CORBA object to

extend a generated base class. Given that Java supports only a single inheritance hierarchy, this means that you cannot make your stateless CORBA object extend a class of your choice. The RMI-IIOP approach allows you to use an inheritance hierarchy of your choice for the stateless CORBA object, because the object only has to implement a specific interface.

The CORBA interface and operation names are mapped to corresponding Java implementations. You can develop server implementations that use the CICS Java classes (JCICS) to access CICS services. See the *JCICS Class Reference* for details of the JCICS classes, and Chapter 6, "Java programming using JCICS," on page 17 for an explanation of how to develop server applications using them.

The JCICS classes are fully documented in JAVADOC html that is generated from the class definitions. This is available through the CICS Information Center, in the *JCICS Class Reference*.

Obtaining an interoperable object reference (IOR)

To locate a server object at run-time, the client application requires a reference to it.

This reference is called an **Interoperable Object Reference** (**IOR**). An IOR is a text string encoded in a specific way, such that a client ORB can decode the IOR to locate the remote server object. It contains enough information to allow:

- A request to be directed to the correct server (host, port number)
- An object to be located or created (classname, instance data)

IORs may be returned by server methods, but a factory class is needed to create an initial IOR. CICS uses the CORBA LifeCycle Services' (CosLifeCycle) GenericFactory class for this purpose. A client application can use this GenericFactory to create IORs for each stateless CORBA object needed at runtime. However, the GenericFactory is itself a stateless CORBA object and thus the client application will need *its* IOR before it can create the target object's IOR.

Use the PERFORM CORBASERVER PUBLISH command to publish a stringified IOR for the GenericFactory class. The GenericFactory IOR is then created and stored on the shelf (an z/OS UNIX directory associated with the CorbaServer), and published to the nameserver. The GenericFactory IOR can be used by the client application to create IORs for any stateless CORBA objects that exist for this CorbaServer (and only for this CorbaServer). The IOR is published with the name `genfac.ior`. How the client locates the GenericFactory IOR at runtime is an application architecture decision. The IOR could be retrieved from a well known location in a JNDI namespace, be kept locally on the client machine, or accessed by some other process.

To publish the IOR, you can use the **CEMT PERFORM CORBASERVER** command, or you can issue an **EXEC CICS PERFORM CORBASERVER** command from a CICS application.

The `genfac.ior` file is written to the CORBASERVER's shelf directory :
/shelf/applid/corbaserver/

where:

shelf is the SHELF directory name specified in the CORBASERVER resource definition, defaulting to `/var/cicsts/`

applid is the is the APPLID identifier associated with the CICS region

corbaserver
is the CORBASERVER resource name

You can download the IOR to your client workstation (in ASCII mode) from the shelf using FTP. Alternatively, your client can use the JNDI interface to obtain the IOR from the nameserver.

Due to the stateless nature of the object, there is seldom any point in a client creating more than one instance of a class. Once a client has created an instance of an object, for example `bankaccountfacilitator`, the same object can be used to access both Mr X's account and Mr Y's account; the account number is an input parameter in every method.

Note: We have called the object in this example a bankaccountfacilitator so that it can perform actions on any account. To have called it a bankaccount might imply that the instance always represented Mr X's account.

Creating the Interface Definition Language (IDL)

If you are using the CORBA development style to create a stateless CORBA object application, your must create an OMG IDL file that contains the definitions of interfaces the server implementation will support.

Note: This section assumes that you're using the CORBA development style to create a stateless CORBA object application (approach 1 in "Developing stateless CORBA objects" on page 371, rather than the RMI-IIOP approach). The RMI-IIOP approach is described in "Developing an RMI-IIOP stateless CORBA application" on page 380.

An OMG IDL file describes the data-types, operations, and objects that the client can use to make a request, and that a server must provide for an implementation of a given object.

For information about writing IDL, see the OMG publication, *Common Object Broker: Architecture and Specification*, obtainable from the OMG Web site at http://www.omg.org/

You process the IDL definitions with an IDL-to-Java compiler (sometimes called a "parser" or "generator"). You must use a compiler provided by the server environment to generate server-side skeletons and helper classes, and a compiler provided by the client environment to generate client-side stub (sometimes called "proxy") and helper classes. Skeleton classes appropriate for use with CICS can be created using the IDLJ compiler provided with any IBM Java 2 SDK. If you use a non-IBM IDLJ compiler, the resulting skeleton class may or may not be suitable for use with CICS. If in doubt, you may use the IDLJ compiler that ships with the Java SDK supplied on z/OS that is used by CICS.

The stub or proxy classes produced by the IBM IDL compiler (IDLJ) are appropriate for use with any IBM ORB. If you use a client-side ORB from a different vendor (for example, Sun MicroSystems or Borland) you should use the IDL compiler supplied with that ORB. If you use stub classes generated for one vendor's ORB with another vendor's ORB, the results are undefined—the stubs might or might not work.

The proxies and skeletons provide the object-specific information needed for an ORB to distribute a method invocation.

Figure 37 shows how the same IDL file is used to generate different classes used by the client and the server.

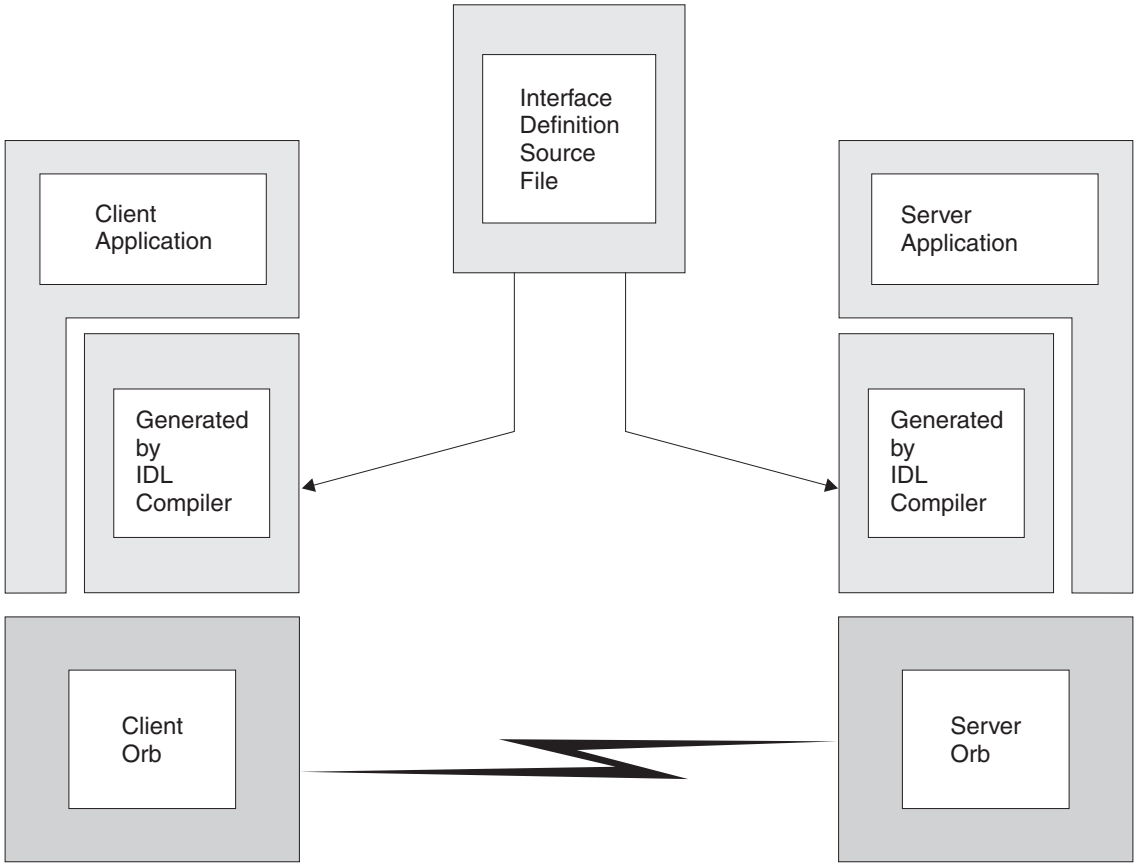

Figure 37. IDL and generated code

Developing an IIOP server program

The server program can be developed on any platform that supports Java. For example, an NT workstation, AIX or the UNIX System Services environment of z/OS.

About this task

Note: This section assumes that you're using the CORBA development style to create a stateless CORBA object application (approach 1 in "Developing stateless CORBA objects" on page 371, rather than the RMI-IIOP approach). The RMI-IIOP approach is described in "Developing an RMI-IIOP stateless CORBA application" on page 380.

The following steps are required:

Procedure

1. Write the IDL definition of the interfaces and operations that form your application.
2. Compile the IDL file to generate CORBA skeleton and helper classes, using the IDL compiler **idlj** command which is part of the Java 2 SDK.

Note:

a. You must use an IBM-supplied IDL-to-Java compiler to do this. The IDL-to-Java compiler supplied with the Sun version of the Java 2 SDK may not be 100% compatible with the IBM ORB.

b. The **idlj** command is not supplied as part of the Java Runtime Environment (JRE); you will need a full SDK installed on your machine before this will work.

The IDL compiler can be invoked as follows:

```
idlj [options] <idl file>
```

Where `<idl file>` is the name of the file containing the IDL definitions, and `[options]` is any combination of the following options, which may appear in any order. `<idl file>` is required and must appear last. At least **-f** must be specified.

For example:

```
idlj -v -fall myidl.idl
```

You must also specify the **-oldImplBase** option to ensure that a CICS-compatible implementation is generated. If you do not use this option, the generated implementation will use the Portable Object Adapter (POA), which is not supported in CICS. For example:

```
idlj -v -fall -oldImplBase myidl.idl
```

-d<symbol>
The equivalent of the following line in an IDL file: `#define <symbol>`

-emitAll
Emit all types, including those found in `#included` files.

-f<side>
Define the bindings to emit. <side> can be:

client not applicable to CICS.

server does not generate sufficient classes for normal use.

all emits all bindings.

serverTIE
not supported in CICS.

allTIE not supported in CICS

If this option is not specified, then **-fclient** is assumed. In most cases you should use **-fall**.

-i<include path>
Add another directory. By default, the current directory is scanned for included files.

-keep If a file to be generated already exists, do not overwrite it. By default it is overwritten.

-oldImplBase
This option is required. If you omit this option, IDLJ generates code which uses the Portable Object Adapter (POA). The POA is not supported under CICS.

-pkgPrefix <t> <pkg>
Make sure that wherever the type or module <t> is encountered, it resides within <pkg> in all generated files. <t> is a fully qualified Java-style name.

-v Verbose mode.

3. Write your server implementation in Java code. The idl compiler will generate an abstract class called_*interfacename***ImplBase**. Your program must extend this. If objects of this type are to be created by the Generic Factory, your implementation class must be called *_interfacename*Impl. If you do not use this naming convention, the GenericFactory will not be able to create references to your CORBA object. For example:

   ```
   public class _BankAccountImpl extends _BankAccountImplBase
   ```

   Your implementation class may make use of the JCICS API to interact with traditional CICS services.
4. Compile your program and the output from step 2, using the javac compiler or an equivalent, such as VisualAge for Java. Ensure that the location of the output files is added to the end of the CICS standard class path, by using the CLASSPATH_SUFFIX option in the JVM profile.

Example

This example describes a bank account whose contents can be queried and updated. The example has a parameter that identifies the instance of the BankAccount, to satisfy the 'stateless' restriction. The following IDL defines the interface and operations:

```
  module bank {

  // this interface is used to manage the bank accounts
  interface BankAccount {
    exception ACCOUNT_ERROR { long errcode; string message;};

    // query methods
    long querybalance(in long acnum) raises (ACCOUNT_ERROR);
    string queryname(in long acnum) raises (ACCOUNT_ERROR);
    string queryaddress(in long acnum) raises (ACCOUNT_ERROR);

    // setter methods
    void setbalance(in long acnum, in long balance) raises (ACCOUNT_ERROR);
    void setaddress(in long acnum, in string address) raises (ACCOUNT_ERROR);
};
};
```

The server implementation of the above IDL must be called `_BankAccountImpl` if objects of this type are to be created by the GenericFactory and must extend `_BankAccountImplBase`, which is generated by the IDL compiler. It is part of the Java package `bank`. You can see full details of this implementation in the stateless CORBA BankAccount sample application distributed in :

```
/usr/lpp/cicsts/<username>/samples/dfjcorb
```

where *username* is a name you can choose during CICS installation, defaulting to `cicsts41`.

To use this example, you need the following resources:

- A TCPIPSERVICE resource defined and installed to listen on a given port under CICS. This TCPIPSERVICE must be:
  - Defined to use the IIOP protocol.

  - In “open” state in order to receive requests.
- A CORBASERVER resource defined to process IIOP requests on the TCPIPSERVICE.

You may optionally choose to add a REQUESTMODEL definition, in order to force the request to be processed under a given TRANSID.

Developing the IIOP client program

you write a client application that makes calls to the server using the generated stub class.

About this task

Note: This section assumes that you're using the CORBA development style to create a stateless CORBA object application (approach 1 in “Developing stateless CORBA objects” on page 371, rather than the RMI-IIOP approach). The RMI-IIOP approach is described in “Developing an RMI-IIOP stateless CORBA application” on page 380.

Procedure

1. Process the IDL file with an IDL- to-Java compiler suitable for your client system (using the same IDL file that you used to build the server application).
2. Obtain a stringified object reference to the GenericFactory by downloading `genfac.ior` (in ASCII mode) from the CorbaServer's shelf directory, where it was created when the CORBASERVER resource was published. Alternatively, you can use JNDI, as a Generic Factory IOR for the CorbaServer is published to the namespace if you issue an **EXEC CICS PERFORM CORBASERVER PUBLISH**, or a **CEMT PERFORM CORBASERVER PUBLISH** command. If you plan to use JNDI, then you must define a nameserver, see “Defining name servers” on page 182. The IOR is bound into the context identified by the JNDI prefix in the CORBASERVER resource definition, with the name GenericFactory. For example, the pathname would be:

   `/jndiprefix/GenericFactory`

   See the *CICS Resource Definition Guide* and the *CICS Supplied Transactions* manual.
3. Write your client program, containing calls to the server. To obtain an initial object reference, use the GenericFactory as shown in “Client example.”
4. Compile the client program, and the output from step1, with javac or an equivalent compiler.

Results

Client example

The following example shows how the GenericFactory service is used by a client program to create an **account** object. The client must first create a proxy for the GenericFactory.

Java bindings for part of the CORBA CosLifeCycle and CosNaming modules are required. If they are not provided by the client ORB, you can build them using the client ORB's IDL-to-Java compiler, from the CORBA services IDL available from the OMG website (`www.omg.org`). Alternatively, you can use the precompiled Java version of the IDL provided in

`/usr/lpp/cicsts/<cicsts41>/lib/omgcos.jar`

Where *cicsts41* is your chosen value for the USSDIR installation parameter that you defined when you installed CICSTS.

The JAR file should be downloaded in binary mode and made available on the client's CLASSPATH environment entry.

The following example, and the supplied samples, require bindings that can be imported as org.omg.CosNaming and org.omg.CosLifeCycle.

In order to create an account object, the client must first create a proxy for the GenericFactory. The following example assumes that a stringified reference to the GenericFactory exists in a file available to a client, and is returned by the getFactoryIOR() method.

```
import java.io.*;
import org.omg.CORBA.*;
import org.omg.CosLifeCycle.*;
import org.omg.CosNaming.*;
public class bankLineModeClient{

//The following method reads the ior from a file and returns it in the string
 String factoryIOR = getFactoryIOR();
// Turn the stringified reference into the proxy
 org.omg.CORBA.Object genFacRef = orb.string_to_object(factoryIOR);
// narrow to correct interface
 GenericFactory fact = GenericFactoryHelper.narrow(genFacRef);
```

Now that the client has a generic factory, it can use it to create an account object.

```
// The Generic factory needs a key, which is a sequence of namecomponents
 NameComponent nc = new NameComponent("bank::BankAccount","object interface");
 NameComponent key[] = {nc};
//The  Generic factory also requires criteria (which it ignores)
 NVP mycriteria[] = {};

//Now create the object
 org.omg.CORBA.Object objRef = fact.create_object(key, mycriteria);
// and narrow to correct interface
BankAccount acctRef = BankAccountHelper.narrow(objRef);
```

Now the client has an object, it can use it:

```
int ac1 = 1234; // Tony's account
int ac2 = 3456; // Lou's account
String name;
String address;
int balance;


try {
  name=acctRef.queryname(ac1);
  System.out.println("a/c num:"+ac1+" name:"+name);
}
catch (exception e) {
  System.err.println("query error");
}
```

Note: NVP (Name Value Pair) is a datatype defined in the CORBA IDL for the Generic Factory interface.

Developing an RMI-IIOP stateless CORBA application

This section tells you how to use the RMI-IIOP development style to create a stateless CORBA object application.

This is the development style defined in approach 2 in "Developing stateless CORBA objects" on page 371, rather than the CORBA development approach described in previous sections.

The RMI-IIOP approach involves developing a standard Java Remote Method Invocation (RMI) application and deploying it to use IIOP as its transport protocol. This is the approach taken by enterprise beans.

Note: This section specifically documents how to develop a stateless CORBA application using RMI-IIOP. Enterprise beans are deployed using other tools, such as the Assembly Toolkit (ATK). For information about deploying enterprise beans, see Chapter 23, "Deploying enterprise beans," on page 303.

When using RMI-IIOP there is no need to define an interface using IDL—though, if required, the IDL can optionally be generated later. Instead, we start by defining at least one remote interface. Note that, in this context, a "remote interface" means any Java interface that extends `java.rmi.Remote`. This is not the same thing as an enterprise bean's "Remote Interface". Using the terminology just defined, both an enterprise bean's Remote Interface and its Home Interface would qualify as "remote interfaces", because they both ultimately extend `java.rmi.Remote`.

This remote interface should be coded to follow the rules of Java RMI. An example remote interface is shown below:

```
package hello;
public interface HelloWorldRMI extends java.rmi.Remote
{
   public String sayHello(String msgFromClient) throws java.rmi.RemoteException;
}
```

The above interface defines a single method called sayHello that takes a String as a parameter and returns a String. All the methods on the interface must be defined to throw java.rmi.RemoteException.

Next, you should provide a server-side implementation of this interface. An example is shown below:

```
package hello;
public class _HelloWorldRMIImpl implements HelloWorldRMI
{
 public String sayHello(String msgFromClient)
 { return "Hello: You said: " + msgFromClient;}
}
```

The implementation class implements the interface previously created. The naming convention used for the implementation class is _<*interface name*>Impl. This naming convention is required if the server object is to be located using the CORBA CosLifeCycle Generic Factory approach. If you do not use this naming convention, the Generic Factory will not be able to construct instances of your stateless CORBA object.

One of the advantages of RMI-IIOP over the more traditional IDL-based development process is that you are not forced to extend a base class. This means

that you can choose to use your own inheritance hierarchy if you want. You may also implement multiple remote interfaces with a single server object.

You should compile both of the above classes using the `javac` compiler or equivalent.

The next thing to do is to produce the server-side Tie file for this stateless CORBA object. This is done using the RMI compiler (RMIC). You must use an RMI compiler shipped with an IBM Java 2 SDK. If you use the version of RMIC supplied with a Sun MicroSystems' Java 2 SDK, the generated Tie file is not guaranteed to work with the CICS ORB.

The command to use is as follows:

```
rmic -iiop hello._HelloWorldRMIImpl
```

Note that RMIC is being run against the server-side implementation class.

Next we need the client-side stub class. This is also produced using the RMI compiler. Ensure that you use an appropriate RMI compiler for your client ORB. The command to use is as follows:

```
rmic -iiop hello.HelloWorldRMI
```

Note that RMIC is being run against the remote interface class.

Once this is complete, you should have the following classes available:

```
hello\HelloWorldRMI.class              - the remote interface
hello\_HelloWorldRMIImpl.class         - the stateless CORBA object
hello\__HelloWorldRMIImpl_Tie.class    - the RMI-IIOP server side Tie file
hello\_HelloWorldRMI_Stub.class        - the RMI-IIOP client side Stub file
```

The next thing to do is to write the client application. The client application is very similar to the client application developed using the IDL-based approach to CORBA development (described in “Developing the IIOP client program” on page 378). As before, we still need to find a reference to the stateless CORBA object using the CORBA CosLifeCycle Generic Factory. Here is part of an example RMI-IIOP client application:

```
ORB orb = ORB.init((String[]) null, (java.util.Properties) null);

// The following method reads the generic factory IOR from a file and returns
// it in the string
String factoryIOR = getFactoryIOR();

// Turn the stringified reference into the proxy
org.omg.CORBA.Object genFacRef = orb.string_to_object(factoryIOR);

// narrow to correct interface
GenericFactory fact = GenericFactoryHelper.narrow(genFacRef);

// The Generic factory needs a key, which is a sequence of namecomponents
NameComponent nc = new NameComponent("hello::HelloWorldRMI","object interface");

//Now create the object
org.omg.CORBA.Object objRef=fact.create_object(new NameComponent[]{nc},
                                               new NVP[] {});

// and narrow to correct interface using the RMI-IIOP narrow operation
HelloWorldRMI remote = (HelloWorldRMI) javax.rmi.PortableRemoteObject.narrow
                       (objRef, HelloWorldRMI.class);
```

```
// Invoke the remote method
System.out.println("Received from Server: "+remote.sayHello("Hi!")+"\n");}
```

As with the IDL-based client application, it will be necessary to have the `omgcos.jar` file from the CICS lib z/OS UNIX directory on your workstation and client machines in order to find the CosLifeCycle classes.

All that remains is to package the server- and client-side applications into JAR files and to add the server-side JAR file to the standard class path.

If you want to generate IDL, for the RMI-IIOP remote interface, that would be suitable for use with a non-Java-based CORBA client application, use the following command:

```
rmic -idl hello.HelloWorldRMI
```

Stand-alone CICS CORBA client applications

CICS CORBA support is primarily focused on supporting IIOP server-side objects—that is, enterprise beans and stateless CORBA objects. These server-side components run in a CICS EJB/CORBA server, in a CorbaServer execution environment represented by a CORBASERVER resource. Because they run in a CICS EJB/CORBA server, they have access to a rich ORB feature set.

In this section, the term "*stand-alone CICS CORBA client applications*" refers to CICS applications that:

1. Are CORBA client applications
2. Are defined to CICS as standard Java applications, by means of a PROGRAM definition on which JVM=YES specified
3. Create an ORB instance using the `new` operator
4. Do not run in a CICS CorbaServer execution environment

Stand-alone CICS CORBA client applications do not run in a CICS EJB/CORBA server, and thus do not have access to the same quality of CORBA support as server-side components. The ORB available to these client applications is a client-only ORB sometimes referred to as the "JCICS ORB". This ORB cannot listen on a socket for inbound connections; therefore any IORs published by this ORB cannot be supported. Similarly, a CICS CORBA client application cannot initiate (or participate in) a distributed OTS transaction. A CICS CORBA client application also cannot participate in asserted identity authentication.

These limitations do not extend to the CICS server ORB environment. Any server object in a CICS EJB/CORBA server can make outbound client IIOP calls that participate in an OTS transaction, providing that the ORB instance used to perform these outbound calls is the current CICS EJB/CORBA server ORB. If a new ORB instance is created by the server object using the new operator, CICS cannot automatically propagate the existing transaction context using this new ORB. An IIOP server object can programmatically get a handle to the current server ORB instance by using the following static method call:

```
com.ibm.cics.iiop.ORBFactory.getORB()
```

CORBA interoperability

The CICS implementation of the CORBA architecture provides a link between applications based on CORBA ORBs and CICS services, including enterprise beans.

An enterprise bean hosted by CICS can be made to inter-operate with objects on other CICS regions (including back-level CICS regions from CICS TS 1.3 onwards), WebSphere Application Server, and third-party J2EE application servers and ORBs. Enterprise beans are available to pure CORBA clients, and can act as clients to remote CORBA objects (potentially implemented in a different programming language and hosted on a different platform).

The CICS ORB can be used to host only client and server applications written in Java. However, it can be used to interoperate with remote ORBs which serve clients and servers written in other programming languages.

Using non-Java CORBA clients

Different programming languages require different language bindings to an ORB.

This requires a level of interoperability between the ORBs which should be taken into consideration. The CORBA architecture defines language bindings for a number of languages, including C++, Java, COBOL, Ada, PL/I, Smalltalk, and others. Note that language bindings for some programming languages might not support all IDL and IIOP features. In particular, valuetypes have been defined only for the C++ and Java language bindings. CORBA access to enterprise beans requires valuetypes, so today only C++ and Java applications can access most enterprise beans through a CORBA interface.

Writing a CORBA client to an enterprise bean

For client programming languages other than Java, such as C++, the CORBA architecture is often the only viable option for accessing enterprise beans.

About this task

For client programming languages other than Java, such as C++, the CORBA architecture is often the only viable option for accessing enterprise beans. Enterprise beans are available to CORBA clients through the CORBA programming model as follows:

- Write the enterprise bean.
- Generate IDL for the enterprise bean, using the RMI compiler with the -IDL option. (This is the reverse of the typical CORBA model, in which IDL is used to generate the object.)

  If you use only CORBA primitives as data and return types, it will be easier to access the bean from non-Java clients.
- Using an IDL compiler suitable for the client environment, compile the IDL to generate client-side stubs.
- Write the client, using the generated stub.
- Make an IOR for the enterprise bean available to the client application. The IOR contains sufficient information for any CORBA ORB to locate the enterprise bean.

Even if a session bean has been coded to use only CORBA primitives as parameter and return types, exception types are still returned as CORBA valuetypes. If your CORBA client ORB does not support valuetypes, you will be forced to work with unknown exceptions.

Note: It is not recommended to use a Java CORBA client to an enterprise bean. Use RMI-IIOP instead.

Enterprise beans as CORBA clients

Enterprise beans are Java objects operating in a sophisticated runtime environment which includes an ORB.

If the enterprise bean is to make outbound IIOP calls to remote CORBA objects (without using RMI-IIOP) it is strongly recommended that the application make use of the existing ORB instance. If the enterprise bean creates a new ORB instance using the new operator, CICS cannot propagate the existing transaction and security context under which the bean is running to method requests on this new ORB.

If you need to get a handle to the current ORB from within an enterprise bean you can use the following static method call:

```
com.ibm.cics.iiop.ORBFactory.getORB()
```

Code sets

CICS can accept GIOP char/wchar and string/wstring datatypes only if they are encoded using one of these code pages.

- UCS2—the standard Java codeset (Unicode)
- UTF-8

Chapter 30. Using the IIOP samples

These sample applications demonstrate the use of IIOP applications (stateless CORBA objects) and the CICS Java programming support (JCICS).

HelloWorld sample

This sample provides a simple test of the IIOP components. The client program:

- reads the file genfac.ior to obtain a reference to the generic factory
- uses the generic factory to create a HelloWorld object
- invokes method sayHello to send a greeting to the server (Hello from HelloWorldClient)and receive a greeting from it in reply (Hello from CICS TS)

The design of the application is described in comments in the code.

BankAccount sample

The sample consists of the following main parts:

1. A traditional CICS application that uses BMS and the **EXEC CICS** API, written in C. This application consists of two transactions:

   BNKI Initializes a file with information about a number of bank accounts. These accounts have numbers in the range 23 through 30.

   BNKQ
   Queries the information in the accounts. There is also a CICS program, DFH$IICC, which performs a credit check for an account.
2. An implementation of an IDL interface that defines a bank account object. The implementation is written in Java and runs as a stateless CORBA object. This implementation uses the bank account file to access bank account information and the DFH$IICC credit check program to obtain credit ratings.
3. A CORBA client application written in Java that displays information about bank account objects.

The design of the application is described in comments in the code.

Setting up the IIOP sample environment

To configure CICS as an IIOP server or client, you need to set up the following host software environment.

About this task

- A z/OS system with UNIX Systems Services and its file system.
- Language Environment configured and active.
- CICS.
- IBM SDK for z/OS, Java Technology Edition. You can download this product, and find out more information about it, at http://www.ibm.com/servers/eserver/zseries/software/java/.

Then follow these steps to set up the IIOP environment:

Procedure

1. Define the following JCL parameter in the start-up jobstream for a CICS region that supports IIOP:

   REGION
   1000M minimum is recommended

2. Define the following system initialization parameters in the start-up jobstream for a CICS region that supports IIOP:

   EDSALIM
   500M minimum is recommended

   MAXJVMTCBS
   Specify the number of JVMs that your CICS region can support. Managing your JVM pool for performance, in the *CICS Performance Guide*, tells you how to work out an appropriate setting for the **MAXJVMTCBS** system initialization parameter.

   TCPIP YES

3. Add the following DD statements to the start-up jobstream for a CICS region that supports IIOP, and create these files:

   DFHEJDIR
   A recoverable shared file containing the request streams directory. This can be a VSAM file or a coupling facility data table. CICS supplies sample JCL to help you create this file, in the DFHDEFDS member of the SDFHINST library.

   DFHEJOS
   A non-recoverable shared file used by CICS when CORBASERVERS are installed and to store stateful session beans that have been passivated. This can be a VSAM file or a coupling facility data table. CICS supplies sample JCL to help you create this file, in the DFHDEFDS member of the SDFHINST library.

   Sample local VSAM data set definitions for these files are provided in the CICS-supplied RDO group DFHEJVS. These data sets must be authorized with RACF for UPDATE access. See Authorizing access to CICS data sets, in the *CICS RACF Security Guide*.

4. Create a shelf directory on z/OS UNIX and give the CICS region userid full access to it. See Giving CICS regions access to z/OS UNIX System Services for guidance.

5. Choose a suitable JVM profile and ensure that CICS is able to locate them, as described in "Setting up JVM profiles" on page 93.

6. Ensure that the following environment variables are correctly defined in the JVM profile for the server side application:

   CICS_HOME
   The installation directory prefix of CICS TS:

   ```
   /usr/lpp/cicsts/cicsts41/
   ```

   where *cicsts41* is your chosen value for the USSDIR installation parameter that you defined when you installed CICS TS.

   JAVA_HOME
   Your installation directory for the IBM SDK for z/OS, Java Technology Edition. The default for Version 6 of the SDK is:

   ```
   /usr/lpp/java/J6.0/
   ```

7. Ensure that the following files are added to a suitable class path in the JVM profile:
   - The sample Java source and makefiles that are stored in the z/OS UNIX System Services file system during CICS installation, in the following directories:
     - $CICS_HOME/samples/dfjcorb/HelloWorld
     - $CICS_HOME/samples/dfjcorb/BankAccount
   - The location where you have compiled the classes for the server side applications.

   "Adding application classes to the class paths for a JVM" on page 145 tells you how to do this.
8. Ensure that the CICS-supplied resource definition groups DFHIIOP and DFH$IIOP are installed. Do this by including the groups in DFHLIST before starting CICS or by using the CEDA option INSTALL to install the resources in CICS while it is running. See CEDA - resource definition online, in the *CICS Supplied Transactions* manual, for information about using CEDA to install resource definitions.

   The supplied group DFH$IIOP contains:
   - Resource definitions required for the TCP/IP listener region (which may also be the same region that runs the sample programs):
     - SSL TCPIPSERVICE definition
     - NOSSL TCPIPSERVICE definition
   - Resource definitions required for the HelloWorld sample:
     - IIHE TRANSACTION definition
     - DFJIIRH REQUESTMODEL definition
     - IIOP CORBASERVER definition
   - Resource definitions required for the BankAccount sample:
     - DFH$IIBI PROGRAM definition
     - DFH$IIBQ PROGRAM definition
     - DFH$IICC PROGRAM definition
     - BANKINQ MAPSET definition
     - BNKI TRANSACTION definition
     - BNKQ TRANSACTION definition
     - BNKS TRANSACTION definition
     - BANKACCT FILE definition
     - DFJIIRB REQUESTMODEL definition
     - IIOP CORBASERVER definition

   The TCPIPSERVICE and IIOP CORBASERVER definitions refer to the default port numbers, 683 and 684. You may need to change these to port numbers that are available to you. Also, the IIOP definition refers to CICSHOST as the host of the CorbaServer. You will need to change this to your own host name. See TCPIPSERVICE resource definitions and CORBASERVER resource definitions, in the *CICS Resource Definition Guide*.
9. Translate and compile the following CICS C language programs and mapset and include them in a library in the CICS DFHRPL concatenation. They are stored in SDFHSAMP during CICS installation. The order of compilation is important. Both DFH$IIBI and DFH$IICC can be compiled independently, but the BMS mapset DFH$IIMA must be compiled before compiling DFH$IIBQ. See Installing application programs, in the *CICS Application Programming Guide*, for guidance on translating, compiling, and linking CICS application programs.

The file DFH$IIMA contains one mapset BANKINQ with two maps. Compile and link the mapset BANKINQ.

See Installing map sets and partition sets, in the *CICS Application Programming Guide*, for guidance on compiling and linking BMS maps.

DFH$IIBI
C program that initializes the BANKACCT file. Run by the BNKI transaction.

DFH$IIBQ
C program that queries the accounts held in BANKACCT.

DFH$IICC
C program that performs a credit check. This is called by DFH$IIBQ.

DFH$IIMA
BMS mapset BANKINQ.

Note: In the names of sample programs and files described in this book, the dollar symbol ($) is used as a national currency symbol and is assumed to be assigned the EBCDIC code point X'5B'. In some countries a different currency symbol, for example the pound symbol (£), or the yen symbol (¥), is assigned the same EBCDIC code point. In these countries, the appropriate currency symbol should be used instead of the dollar symbol.

10. To compile the IIOP HelloWorld client you require the `CosLifeCycle` and `CosNaming` runtime classes. If your client ORB environment does not provide these services ready-built you can use the `omgcos.jar` file shipped in the $CICS_HOME/lib directory. Alternatively, you may choose to build these classes from the original OMG supplied IDL. In this case a copy of the relevant IDL files is available in $CICS_HOME/samples/dfjcorb. The process of turning pure IDL into executable code is ORB dependent, but if you are using an ORB supplied with a JVM then it is likely that the following commands will work:

```
idlj -pkgprefix CosNaming org.omg -pkgprefix CosLifeCycle org.omg -fall CosLifeCycle.idl
idlj -pkgprefix CosNaming org.omg -pkgprefix CosLifeCycle org.omg -fall CosNaming.idl
javac org\omg\CosLifeCycle\*.java org\omg\CosNaming\NamingContextPackage\*.java
      org\omg\CosNaming\*.java
```

You must ensure that these classes are available on your classpath environment variable when you attempt to build any CICS stateless CORBA client application.

11. Obtain a **genfac.ior** file containing an object reference to your server's generic factory, and place it in the current directory. The `genfac.ior` file is created when you issue a PERFORM CORBASERVER PUBLISH command for the installed sample IIOP CORBASERVER resource definition. It is written to the CORBASERVER's shelf directory:

    `/var/cicsts/applid/IIOP`

    where *applid* is the APPLID identifier associated with the CICS region.

    To publish the CORBASERVER definition, you can use a **CEMT PERFORM CORBASERVER** command or an **EXEC CICS PERFORM CORBASERVER** command issued from a CICS application.

    You can download the IOR to your client workstation (in ascii mode) from the shelf using ftp.

Results

Running the IIOP HelloWorld sample

This section tells you what you need to do to run the HelloWorld sample application.

It covers the following topics:

- "Building the server side HelloWorld application"
- "Building the client side HelloWorld application"
- "Running the HelloWorld sample application" on page 390

Building the server side HelloWorld application

The makefile in `$CICS_HOME/samples/dfjcorb/HelloWorld/server` builds everything required for the server side application.

`$CICS_HOME/samples/dfjcorb/HelloWorld/server` should be added to the standard class path by using the CLASSPATH_SUFFIX option in the JVM profile, DFHJVMCD.

To build the programs, enter the command `make` from `$CICS_HOME/samples/dfjcorb/HelloWorld/server`. This command makes the HelloWorld object.

Building the client side HelloWorld application

$CICS_HOME/samples/dfjcorb/HelloWorld/client contains the CORBA client part of the application. The source of the Java client application is called **HelloWorldClient.java**. This application should run with any CORBA-compliant ORB.

About this task

The following steps are required to build the Java client application:

1. Download the following files to the client workstation (in ASCII mode):
   - `.../dfjcorb/HelloWorld/HelloWorld.idl`
   - `.../dfjcorb/HelloWorld/client/HelloWorldClient.java`
2. Compile the provided IDL with the client ORB's IDL-to-Java compiler to produce the Java client side stubs required by the sample application. These stubs will be created in a sub-directory called `hello`. Move the client application `HelloWorldClient.java` into this sub-directory.
3. Compile the client application, ensuring that the Java classes produced in the previous step are available through the CLASSPATH environment variable. To compile the client application from the current directory, enter:

   ```
   javac hello\HelloWorldClient.java
   ```

   You will also need the `CosLifeCycle` and `CosNaming` runtime classes. If your client ORB environment does not provide these services ready built then you can use the `omgcos.jar` file shipped in the `$CICS_HOME/lib` directory on z/OS UNIX. Alternatively you may choose to build these classes from the original OMG-supplied IDL. In this case a copy of the relevant IDL files is available in `$CICS.HOME/samples/dfjcorb/`.

   The process of turning pure IDL into executable code is ORB-dependent, but if you are using an ORB supplied with a JVM then it is likely that the following commands will work:

```
idlj -pkgprefix CosNaming org.omg -pkgprefix CosLifeCycle org.omg -fall CosLifeCycle.idl
idlj -pkgprefix CosNaming org.omg -pkgprefix CosLifeCycle org.omg -fall CosNaming.idl
javac org\omg\CosLifeCycle\*.java
              org\omg\CosNaming\NamingContextPackage\*.java
              org\omg\CosNaming\*.java
```

These classes must be in your classpath when you attempt to build any CICS stateless CORBA client application.

Running the HelloWorld sample application

You must use this command to run the client application.

About this task

```
java hello.HelloWorldClient
```

Running the IIOP BankAccount sample

This section tells you what you need to do to run the BankAccount sample application.

It covers the following topics:

- "Building the server side BankAccount application"
- "Building the client side BankAccount application"
- "Running the BankAccount sample application" on page 391

Creating the VSAM file

You must define the VSAM file to hold the bank account data using these IDCAMS parameters.

```
DEFINE    CLUSTER (                        -
                NAME (CICS610.BANKACCT )     -
                CYLINDERS(01)                -
                REUSE                        -
                KEYS(4 0)                    -
                RECORDSIZE(168 168))
```

Building the server side BankAccount application

The makefile in $CICS_HOME/samples/dfjcorb/BankAccount/server builds everything required for the CORBA part of the server side application.

$CICS_HOME/samples/dfjcorb/BankAccount/server should be added to the standard class path by using the CLASSPATH_SUFFIX option in the JVM profile, DFHJVMCD.

To build the programs, enter the command `make` from $CICS_HOME/samples/dfjcorb/BankAccount/server. This makes the Java server program that implements the bank account object.

Building the client side BankAccount application

$CICS_HOME/samples/dfjcorb/BankAccount/javaclient contains the CORBA client part of the application. The source of the Java client application is called **bankLineModeClient.java**. This application should run with any CORBA-compliant ORB.

About this task

The following steps are required to build the Java client application:

1. Download the following files to the client workstation (in ascii mode):
   - .../dfjcorb/BankAccount/BankAccount.idl
   - .../dfjcorb/BankAccount/javaclient/bankLineModeClient.java
2. Compile the provided IDL with the client ORB's IDL-to-Java compiler to produce the Java client side stubs required by the sample application. After compiling the IDL to create the sub-directory, **bank**, move the java file into this sub-directory. Then, this can be compiled from the current directory, as follows:

   ```
   javac bank\bankLineModeClient.java
   ```
3. Ensure that the Java classes produced in the previous step are available through the CLASSPATH environment variable.

   You will also need the CosLifeCycle and CosNaming runtime classes. If your client ORB environment does not provide these services ready built then you can obtain them in the same way as in "Building the client side HelloWorld application" on page 389.

Running the BankAccount sample application

You must perform these steps to run the sample application:

About this task

Procedure

1. Run the BNKI CICS transaction to load data into the account file.
2. Run the client application using:

   ```
   java  bank.bankLineModeClient
   ```

Part 7. Appendixes

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply in 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 publication could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Licensees of this program who want to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM United Kingdom Laboratories, MP151, Hursley Park, Winchester, Hampshire, England, SO21 2JN.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Programming License Agreement, or any equivalent agreement between us.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. A current list of IBM trademarks is available on the Web at Copyright and trademark information at www.ibm.com/legal/copytrade.shtml.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

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

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other product and service names might be trademarks of IBM or other companies.

Bibliography

CICS books for CICS Transaction Server for z/OS

General

CICS Transaction Server for z/OS Program Directory, GI13-0536
CICS Transaction Server for z/OS What's New, GC34-6994
CICS Transaction Server for z/OS Upgrading from CICS TS Version 2.3, GC34-6996
CICS Transaction Server for z/OS Upgrading from CICS TS Version 3.1, GC34-6997
CICS Transaction Server for z/OS Upgrading from CICS TS Version 3.2, GC34-6998
CICS Transaction Server for z/OS Installation Guide, GC34-6995

Access to CICS

CICS Internet Guide, SC34-7021
CICS Web Services Guide, SC34-7020

Administration

CICS System Definition Guide, SC34-6999
CICS Customization Guide, SC34-7001
CICS Resource Definition Guide, SC34-7000
CICS Operations and Utilities Guide, SC34-7002
CICS RACF Security Guide, SC34-7003
CICS Supplied Transactions, SC34-7004

Programming

CICS Application Programming Guide, SC34-7022
CICS Application Programming Reference, SC34-7023
CICS System Programming Reference, SC34-7024
CICS Front End Programming Interface User's Guide, SC34-7027
CICS C++ OO Class Libraries, SC34-7026
CICS Distributed Transaction Programming Guide, SC34-7028
CICS Business Transaction Services, SC34-7029
Java Applications in CICS, SC34-7025

Diagnosis

CICS Problem Determination Guide, GC34-7034
CICS Performance Guide, SC34-7033
CICS Messages and Codes, SC34-7035
CICS Diagnosis Reference, GC34-7038
CICS Recovery and Restart Guide, SC34-7012
CICS Data Areas, GC34-7014
CICS Trace Entries, SC34-7013
CICS Supplementary Data Areas, GC34-7015
CICS Debugging Tools Interfaces Reference, GC34-7039

Communication

CICS Intercommunication Guide, SC34-7018
CICS External Interfaces Guide, SC34-7019

Databases

CICS DB2 Guide, SC34-7011
CICS IMS Database Control Guide, SC34-7016

CICS Shared Data Tables Guide, SC34-7017

CICSPlex SM books for CICS Transaction Server for z/OS

General

CICSPlex SM Concepts and Planning, SC34-7044
CICSPlex SM Web User Interface Guide, SC34-7045

Administration and Management

CICSPlex SM Administration, SC34-7005
CICSPlex SM Operations Views Reference, SC34-7006
CICSPlex SM Monitor Views Reference, SC34-7007
CICSPlex SM Managing Workloads, SC34-7008
CICSPlex SM Managing Resource Usage, SC34-7009
CICSPlex SM Managing Business Applications, SC34-7010

Programming

CICSPlex SM Application Programming Guide, SC34-7030
CICSPlex SM Application Programming Reference, SC34-7031

Diagnosis

CICSPlex SM Resource Tables Reference, SC34-7032
CICSPlex SM Messages and Codes, GC34-7035
CICSPlex SM Problem Determination, GC34-7037

Other CICS publications

The following publications contain further information about CICS, but are not provided as part of CICS Transaction Server for z/OS, Version 4 Release 1.

Designing and Programming CICS Applications, SR23-9692

CICS Application Migration Aid Guide, SC33-0768

CICS Family: API Structure, SC33-1007

CICS Family: Client/Server Programming, SC33-1435

CICS Family: Interproduct Communication, SC34-6853

CICS Family: Communicating from CICS on System/390, SC34-6854

CICS Transaction Gateway for z/OS Administration, SC34-5528

CICS Family: General Information, GC33-0155

CICS 4.1 Sample Applications Guide, SC33-1173

CICS/ESA 3.3 XRF Guide , SC33-0661

Other IBM publications

The following publications contain information about related IBM products.

IBM Developer Kit and Runtime Environment, Java 2 Technology Edition Diagnostics Guide, SC34-6358

Persistent Reusable Java Virtual Machine User's Guide, SC34-6201

Accessibility

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use software products successfully.

You can perform most tasks required to set up, run, and maintain your CICS system in one of these ways:

- using a 3270 emulator logged on to CICS
- using a 3270 emulator logged on to TSO
- using a 3270 emulator as an MVS system console

IBM Personal Communications provides 3270 emulation with accessibility features for people with disabilities. You can use this product to provide the accessibility features you need in your CICS system.

Index

Special characters

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

W

X

Z

Readers' Comments — We'd Like to Hear from You

CICS Transaction Server for z/OS
Version 4 Release 1
Java Applications in CICS

Publication No. SC34-7025-02

We appreciate your comments about this publication. Please comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. The comments you send should pertain to only the information in this manual or product and the way in which the information is presented.

For technical questions and information about products and prices, please contact your IBM branch office, your IBM business partner, or your authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you. IBM or any other organizations will only use the personal information that you supply to contact you about the issues that you state on this form.

Comments:

Thank you for your support.

Submit your comments using one of these channels:
- Send your comments to the address on the reverse side of this form.
- Send a fax to the following number: +44 1962 816151
- Send your comments via email to: idrcf@uk.ibm.com

If you would like a response from IBM, please fill in the following information:

Name	Address
Company or Organization	
Phone No.	Email address

Readers' Comments — We'd Like to Hear from You
SC34-7025-02

Cut or Fold
Along Line

Fold and Tape | **Please do not staple** | Fold and Tape

PLACE
POSTAGE
STAMP
HERE

IBM United Kingdom Limited
User Technologies Department (MP095)
Hursley Park
Winchester
Hampshire
United Kingdom
SO21 2JN

Fold and Tape | **Please do not staple** | Fold and Tape

SC34-7025-02

Cut or Fold
Along Line

SC34-7025-02