CICS Transaction Server for z/OS

Java Applications in CICS

Version 3 Release 1

SC34-6440-07

CICS Transaction Server for z/OS

Java Applications in CICS

Version 3 Release 1

SC34-6440-07

Note!
Before using this information and the product it supports, be sure to read the general information under “Notices” on page 397.

This edition applies to Version 3 Release 1 of CICS Transaction Server for z/OS, program number 5655-M15, and to all subsequent versions, releases, and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

Contents

Preface

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.

Summary of Changes

This information is based on *Java Applications in CICS* for CICS Transaction Server for z/OS®, Version 2 Release 3, SC34-6238-00. Changes from that edition are marked by vertical bars in the left margin.

This part lists briefly the changes that have been made for the following recent releases:

Changes for CICS Transaction Server for z/OS, Version 3 Release 1

The more significant changes for this edition are:

- Various small changes have been made, throughout the manual, to document:
  - CICS support for the IBM® Software Developer Kit for z/OS, Java 2 Technology Edition, Version 1.4.2
  - CICS support for WebSphere® Application Server Version 6
- The chapter entitled "*The CICS Connector for CICS TS*" has been removed, because the CICS Connector for CICS TS is not supported in this release.
- The information about using VisualAge® for Java to create Java program objects, and the information about Java hot-pooling, has been removed, because run-time support for Java program objects and Java hot-pooling is withdrawn in this release. The chapter "*VisualAge for Java, ET/390*" and the chapter "*Java hot-pooling concepts*" have been removed. The *CICS Migration Guide* explains the process for migrating Java program objects to run in a JVM.

Changes for CICS Transaction Server for z/OS, Version 2 Release 3

The more significant changes for this edition were:

- Chapter 7, "Accessing data from CICS applications written in Java," on page 41 was a new chapter. It describes the different methods that CICS Java programs, and enterprise beans, can use to access data.
- The information about the CICS JVM was refreshed. In particular, The CICS JVM now supports the sharing of a cache of commonly-used class files that are already loaded, enabling faster JVM startup and reducing the cost of class loading. See Chapter 10, "Understanding JVMs," on page 63 and Chapter 9, "Setting up Java support," on page 53.
- CICS now supports Version 1.4.2 of the IBM Software Developer Kit for z/OS, Java 2 Technology Edition. See Chapter 10, "Understanding JVMs," on page 63.
- The CICS Object Request Broker (ORB) now supports Version 2.3 of the Common Object Request Broker Architecture (CORBA). See Chapter 12, "IIOP support in CICS," on page 151 and "Migrating an EJB server to CICS Transaction Server for z/OS, Version 3 Release 1" on page 238.
- Chapter 23, "The CCI Connector for CICS TS," on page 305 was a new chapter. It describes a new CICS connector that is compliant with the industry-standard Common Client Interface (CCI) defined by the J2EE Connector Architecture Specification. The connector helps you to build powerful Enterprise JavaBean (EJB) server components that link to existing (non-Java) CICS programs.

- It is now possible to enable and disable CorbaServer execution environments. This has led to better ways of updating beans in production regions—see Chapter 22, “Updating enterprise beans in a production region,” on page 293.
- The information about CICS support for CORBA and CORBA stateless objects was refreshed. In particular:
  - “Stand-alone CICS CORBA client applications” on page 368 was a new section.
  - “Name-mangling of the OPERATION field” on page 192 was a new section.
  - Chapter 27, “Stateless CORBA objects,” on page 357 was rewritten. Much new information was added. “Developing an RMI-IIOP stateless CORBA application” on page 365 and “CORBA interoperability” on page 368 were new sections.
- “Class version issues with RMI-IIOP” on page 326 was a new section.

Changes for CICS Transaction Server for z/OS, Version 2 Release 2

The more significant changes for this edition were:

- Parts of Chapter 14, “Configuring CICS for IIOP,” on page 165, Chapter 16, “What are enterprise beans?,” on page 199, and Chapter 17, “Setting up an EJB server,” on page 227 were rewritten to describe CICS enhanced support for enterprise beans, including an easier way to install deployed JAR files.
- Chapter 18, “Running the EJB IVP,” on page 245 was rewritten to reflect changes to the EJB Installation Verification Program (IVP).
- Chapter 19, “Running the sample EJB applications,” on page 251 was rewritten to reflect changes to the EJB sample applications.
- Chapter 21, “Deploying enterprise beans,” on page 289 and “The deployment tools for enterprise beans in a CICS system” on page 289 were updated to reflect the replacement of the EJB deployment tools.
- Support was added for Java security roles. See “Security roles” on page 337.
- Support was added for a Lightweight Directory Access Protocol (LDAP) name server. See “Setting up an LDAP server” on page 168.

Part 1. Java development roadmaps

This Part outlines the steps needed to implement different types of Java application in CICS.

Chapter 1. JCICS application roadmap

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 10, “Understanding JVMs,” on page 63 and Chapter 9, “Setting up Java support,” on page 53.

Chapter 2. CICS IIOP application roadmap

1. Set up CICS as an IIOP server. See Chapter 14, “Configuring CICS for IIOP,” on page 165.
2. Write your IIOP server application, also known as a “stateless CORBA object”. See “Developing stateless CORBA objects” on page 357, “Creating the Interface Definition Language (IDL)” on page 360, and “Developing an IIOP server program” on page 361.
3. Write your client program. See “Developing the IIOP client program” on page 364.

Chapter 3. CICS enterprise beans roadmap

1. Familiarize yourself with CICS support for enterprise beans by reading Chapter 16, “What are enterprise beans?,” on page 199.
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 214.
3. Set up a basic, single-region EJB server and name server—see “Setting up a single-region EJB server” on page 227.
4. Test your single-region EJB server by running the EJB installation verification program (IVP)—see Chapter 18, “Running the EJB IVP,” on page 245.
5. Further test your EJB server by running the EJB sample applications—see Chapter 19, “Running the sample EJB applications,” on page 251.
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 235.
7. Implement any security controls required by your system—see Chapter 25, “Managing security for enterprise beans,” on page 329.
8. Code your session bean. If you are not using an Integrated Development Environment (IDE), see “Coding a session bean” on page 276.
9. Follow the deployment process described in Chapter 21, “Deploying enterprise beans,” on page 289, using the tools as described in “Using CICS deployment tools for enterprise beans” on page 290.
10. Write the client program. See “Writing the client program” on page 279.

Part 2. Developing Java applications for CICS

This Part tells you 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 the following 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 27, "Stateless CORBA objects," on page 357.

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 16, “What are enterprise beans?,” on page 199.

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 simply 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 10, "Understanding JVMs," on page 63.

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 through 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
- 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 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

The following 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 CICSTS31.CICS.CICS.SDFHLOAD.

Threads

Only one thread (the initial thread) can access the JCICS API. You can create other threads but you must route all requests to the JCICS API through the initial thread. Additionally, you must ensure that all threads other than the original 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 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 37 for a description of the relationship between CICS conditions and Java exceptions.

Error handling and abnormal termination

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

ABEND

To initiate an ABEND from a Java program, invoke one of the 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. You should catch `AbendCancelException` only 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)

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

Channels and containers

For introductory information about channels and containers, and guidance about using channels in non-Java applications, see 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: You can use channel- and container-related JCICS commands when writing CICS enterprise beans. However, CICS doesn't 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. 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 simply:

```
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." 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

Figure 1 shows 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

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

See the *CICS Application Programming Reference* manual for information about the EXEC CICS ADDRESS command. The following support is provided for the ADDRESS options.

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. This option is not supported in JCICS.

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

See the *CICS Application Programming Reference* manual for information about the EXEC CICS ASSIGN command. The following support is provided for the ASSIGN options.

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
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

The following 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

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 the *CICS Application Programming Guide* for more information about datasets.

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).

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
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

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.

Program services

JCICS support for the CICS program control commands is described below:

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 simply to return; the application can simply 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 may not exceed 32 500 bytes if the COMMAREA is to be passed between any two CICS servers (for any combination of product/version/release).

Scheduling services

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

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

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

JCICS support for the temporary storage commands is described below.

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

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

If a task has a terminal as a principal facility, CICS automatically creates two Java `PrintWriter`s 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

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

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

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 tthe 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

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

Web and TCP/IP services

Getters in classes `HttpHeader`, `NameValueData`, and `FormField` return httpheader, name/value pairs and formfield 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
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
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 27 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 the *CICS Application Programming Guide*. For more information on the JCICS Web classes see the *JCICS Class Reference*.

Unsupported CICS services

- APPC unmapped conversations
- CICS Business Transaction Services
- DUMP services
- Journal services
- Storage services
- Timer services
- CICS Business Transaction Services

JCICS exception mapping

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

Table 3. Java exception mapping (continued)

CICS condition	Java Exception	CICS condition	Java Exception
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
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:

- Declare a reference. For example:

  ```
  TSQ tsq;
  ```

- Use the `new` operator to create an object:

  ```
  tsq = new TSQ()
  ```

- Use the `setName` method to give the object a name:

  ```
  tsq.setName("JCICSTSQ");
  ```

Using objects

The following example shows how you 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 (`final` methods) in CICS Java programs**. For an explanation of why finalizers are not recommended, see the IBM Developer Kit and Runtime Environment, Java 2 Technology Edition, Version 1.4.2 Diagnostics Guide.
- **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 23, “The CCI Connector for CICS TS,” on page 305.
- 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. 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. *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 42.
- 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 23, “The CCI Connector for CICS TS,” on page 305.

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 23, “The CCI Connector for CICS TS,” on page 305.

- 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. Using the JCICS sample programs

CICS provides sample programs that demonstrate:

- How to use the JCICS classes
- 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 HFS.

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 Tcpip charactistics 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 HFS 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. 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.

1. *PATH* is the z/OS UNIX System Services search path. Define the *PATH* environment variable by adding:

   `/usr/lpp/java142/J1.4/bin`

   where *java142*/*J1.4* is the install location that was set up when you installed the IBM Software Developer Kit for z/OS, Java 2 Technology Edition, Version 1.4.2. This is the path for the Java executables. You can use the export command to add the path as follows:

   ```
   export /usr/lpp/java142/J1.4/bin:$PATH
   ```

2. *CICS_HOME* is the installation directory prefix of CICS Transaction Server for z/OS. Define the *CICS_HOME* environment variable as follows:

   `/usr/lpp/cicsts/cicsts31`

   where *cicsts31* is defined by the USSDIR installation parameter when you installed CICS TS (`cicsts31` is the default). You can use the export command to set the directory prefix as follows:

   ```
   export CICS_HOME=/usr/lpp/cicsts/cicsts31
   ```

   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 Software Developer Kit for z/OS, Java 2 Technology Edition, Version 1.4.2 subdirectories. Define the JAVA_HOME environment variable as follows:

   `/usr/lpp/java142/J1.4/`

   where *java142*/*J1.4*/ is the install location that was set up when you installed the IBM Software Developer Kit for z/OS, Java 2 Technology Edition, Version 1.4.2

4. Install the group DFH$JVM in order to run the samples. CICS resource definitions for all the sample programs and transactions are supplied in this group.
5. Optional: If you want to run the Web sample program, which is invoked via a browser, you need to follow the instructions in the *CICS Internet Guide*. Use the web sample application DFH$WB1A 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

"The JCICS class library" on page 17

Related tasks

Chapter 8, "Using the JCICS sample programs," on page 43

"Building the Java samples"

"Running the JCICS samples" on page 46

Related reference

"JCICS command reference" on page 21

Building the Java samples

To build the Java samples, you need write permission for the HFS directory in which the samples are stored and for its subdirectories. These directories are part of the directory structure that includes the other CICS files which have been installed on HFS. 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 HFS 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 Software Developer Kit for z/OS, Java 2 Technology Edition, Version 1.4.2.

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` HFS 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 a load library in the CICS DFHRPL concatenation, and define them to CICS as described in "Defining CICS resources" on page 46.

- 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 build the JCICS samples before trying to run them. See “Building the JCICS sample programs” on page 44.

1. Add `$CICS_HOME/samples/dfjcics` to the end of the Java classpath, `ibm.jvm.shareable.application.class.path`, in the default JVM properties file, `dfjjvmpr.props`.
2. Follow the appropriate procedure to run each sample:
   - “Running the Hello World samples”
   - “Running the Program Control samples” on page 47
   - “Running the TDQ sample” on page 47
   - “Running the TSQ sample” on page 48
   - “Running the web sample” on page 48

Running the Hello World samples

There are two “Hello World” samples:

`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 message from JHE1 on `System.out`:

```
Hello from a regular Java application
```

`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 `Task.out`:

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

Running the Program Control samples

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`:

```
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`:

```
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

This sample uses the following Java class:

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

Run the JTD1 CICS transaction to execute the sample. 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

This sample uses the following Java classes:

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

Run the JTS1 CICS transaction to execute the sample. 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)`

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

Chapter 9. Setting up Java support

The following steps tell you how to verify your Java installation and set up JVMs in your CICS system using the supplied Java sample programs.

1. Verify that your Java components are installed correctly using the supplied checklist in The *CICS Transaction Server for z/OS Installation Guide*
2. Give your CICS region permission to access the resources held in the hierarchical file store (HFS).

   In order to create JVMs, CICS requires access to directories and files that z/OS UNIX System Services holds in HFS. "Giving CICS regions access to z/OS UNIX System Services and HFS directories and files" tells you how to do this.
3. Run the Java sample programs to verify that Java works in your region. "Verifying the Java installation using sample programs" on page 60 contains a task list that describes how to set up and run the supplied sample programs.

When you have run the supplied Java sample programs, read through the Chapter 10, "Understanding JVMs," on page 63 section for conceptual information on how to use JVMs in CICS. Then read the section Chapter 11, "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.

Giving CICS regions access to z/OS UNIX System Services and HFS directories and files

CICS requires access to z/OS UNIX System Services, and to directories and files in the hierarchical file store (HFS), for the purposes of:

- Creating JVMs.
- Using HFS files in connection with CICS Web support.

One possible method to achieve this is as follows:

1. Choose a RACF® group that all your CICS regions can use to access z/OS UNIX, and give a z/OS UNIX group identifier (GID) to this RACF group. Give a z/OS UNIX user identifier (UID) to each CICS region user ID, and make sure that each CICS region user ID connects to the RACF group that you chose. During this process, set up a home directory on HFS for each of your CICS regions. "Giving CICS regions a z/OS UNIX user identifier (UID) and group identifier (GID) and setting up a home directory" on page 54 tells you how to do all this.
2. Identify the files that each CICS region needs, and the HFS directories that contain the files. For each directory and file, specify the group for the directory and file as the RACF group that the CICS regions use, and give the group the appropriate permissions. "Giving CICS regions permission to access HFS directories and files" on page 56 tells you how to do this. You will need to repeat this task when you tell a CICS region to use any other files or HFS directories.

If you need more general information about RACF facilities for controlling access to z/OS UNIX System Services, see the *z/OS Security Server RACF Security Administrator's Guide*, SA22-7683. If you need more general information about the UNIX facilities that you can use to control access to HFS files and directories, see *z/OS UNIX System Services Planning*, GA22-7800.

Giving CICS regions a z/OS UNIX user identifier (UID) and group identifier (GID) and setting up a home directory

When a CICS region requests a z/OS UNIX function for the first time, RACF:

- Verifies that the user (the CICS region user ID) is defined as a z/OS UNIX user.
- Verifies that the user's current connect group is defined as a z/OS UNIX group.
- Initializes the control blocks needed for subsequent security checks.

You need to ensure that each CICS region meets these security requirements, by assigning a z/OS UNIX user identifier (UID) to the CICS region user ID, and assigning a z/OS UNIX group identifier (GID) to a RACF group to which the CICS region user ID connects. The identifiers will also be needed to give each CICS region permission to access the HFS directories and files that it needs. During this process, you also need to set up a home directory for each CICS region. This home directory can then be used, if you wish, as the work directory for Java-related activities and for output from JVMs, or as the location for HFS files used by CICS Web support.

The UID and GID are numbers that can be in the range 0 to 16 777 216. (0 is a superuser ID.) Give some thought to naming conventions, and to any existing UIDs and GIDs in your z/OS UNIX system. *z/OS UNIX System Services Planning*, GA22-7800, explains how to manage the UIDs and GIDs for your z/OS UNIX system.

To assign a z/OS UNIX UID and GID for your CICS regions and set up a home directory:

1. Choose a RACF group that can be used by all your CICS regions. For example, you could use a RACF group that is defined as the default group of your CICS region user IDs, or you could set up a RACF group to be used only for access to JVM-related directories and files or CICS Web support directories and files. If you use this RACF group for giving file access permissions, following the procedure described in “Giving CICS regions permission to access HFS directories and files” on page 56, the RACF group's z/OS UNIX group identifier (GID) will be associated with the HFS directories and files. This means that the owner of these directories and files, and anyone who is not the owner but needs to carry out operations with these files, will need to have this group as his or her group or one of their supplementary groups. “RACF group profiles” in the *CICS RACF Security Guide* explains how RACF groups work.
2. Choose a suitable z/OS UNIX group identifier (GID) for the RACF group, and assign the GID to the RACF group. To assign a GID, specify the GID value in the OMVS segment of the RACF group profile. For example, if the RACF group is CICSTSAB, and the GID you want to assign is 9, use the command:

   ```
   ALTGROUP CICSTSAB OMVS(GID(9))
   ```

3. Choose a suitable z/OS UNIX user identifier (UID) for each CICS region. Assign the UID to each of your CICS region user IDs. (“Specifying the CICS region userid” in the *CICS RACF Security Guide* explains how the region user ID under which CICS executes is specified when CICS is run as a started task, as a started job, or as a job.) To assign UIDs, specify the UID value in the OMVS segment of the RACF user profile for each CICS region user ID. Also specify the name of a home directory for each CICS region using the HOME option. The directory name should be in the format `/u/CICS region userid`. “RACF user profiles” in the *CICS RACF Security Guide* tells you how to update a RACF user profile using the ALTUSER command. For example, if the CICS region user ID is CICSHT##, and the UID you want to assign is 2001, use the command:

```
ALTUSER CICSHT## OMVS(UID(2001) HOME('/u/cicsht##'))
```

If you want to know about the other information that can be specified in an OMVS segment parameter in a user profile besides the UID and home directory, see the *z/OS Security Server RACF Command Language Reference*, SA22-7687.

Note: It is possible to assign the same UID to more than one CICS region user ID. If all your CICS regions need to use the same HFS files (for example, the supplied sample files for JVMs), you could give all the CICS regions the same UID, and then you could assign permissions to that UID, rather than to the GID. However, bear in mind that:

a. The sharing of UIDs allows each CICS region to access all of the z/OS UNIX resources that the other CICS regions with that shared user ID can access, and this might not be appropriate in your system.

b. The sharing of UIDs is not normally recommended in a z/OS UNIX system.

c. If you do choose to share UIDs, note that the z/OS UNIX System Services parameter MAXPROCUSER limits the maximum number of processes that a single user (that is, with the same UID) can have concurrently active. *z/OS UNIX System Services Planning*, GA22-7800, has more information about this parameter.

4. Set up each of the directories that you have specified as a home directory for one of your CICS regions. To do this:

   a. If you are not using an automount facility, use the `mkdir` command to create the HFS directories. For example, issuing the UNIX command

      ```
      mkdir /u/cicsht##
      ```

      creates the HFS directory `/u/cicsht##`. (If you are using the TSO command, the directory name must be enclosed in single quotes.)

   b. Whether or not you are using an automount facility, allocate an HFS data set for each directory. *z/OS UNIX System Services Planning*, GA22-7800, tells you how to do this.

   c. If you are not using an automount facility, mount the data set that you have allocated. Again, *z/OS UNIX System Services Planning*, GA22-7800, tells you how to do this.

   Note that the HFS data set that you allocate for a CICS region's home directory has a finite size, and if a particular CICS region is using the home directory extensively, you might need to increase the amount of space that the region has available.

5. Make sure that each of your CICS region user IDs connects to the RACF group to which you assigned a z/OS UNIX group identifier (GID). If your CICS region user IDs need to connect to more than one RACF group, RACF list of groups must be active in your system.

To check the UID and GID details for a user, use the `id` command in the UNIX environment. For example, issuing the `id` command for our example CICS region user ID CICSHT## would give the following result:

```
uid=2001(CICSHT##) gid=9(CICSTSAB)
```

Now that each CICS region user ID has a UID and is connected to a group with a GID, it can use z/OS UNIX functions and access z/OS UNIX files. Next, identify the files that each CICS region needs, and the HFS directories that contain the files,

and use the group name or GID to give the CICS region permission to access these directories and files. “Giving CICS regions permission to access HFS directories and files” tells you how to do this.

Giving CICS regions permission to access HFS directories and files

Because your CICS regions have a z/OS UNIX user identifier (UID), and their connect group (the RACF group) has a z/OS UNIX group identifier (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 HFS directories and files:

- 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 HFS user, so this option might not be suitable for use in your production environment.
- 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 (for JVMs, this would include the CICS-supplied JAR files and the IBM persistent reusable JVM code). As mentioned in “Giving CICS regions a z/OS UNIX user identifier (UID) and group identifier (GID) and setting up a home directory” on page 54, it is possible to assign the same UID to all your CICS regions, and then you can make that UID the owner of the directories and files. However, bear in mind the points noted in that section about the disadvantages associated with the sharing of UIDs.
- Associate the directory or file with a RACF group that has been assigned a z/OS UNIX group identifier (GID), give the RACF group the appropriate group permissions, and connect the users (the CICS regions) to this RACF group. 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.
- With z/OS Version 1 Release 3 or later, you can 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 HFS, 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.

To check the permissions for files and directories in a path, 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## CICSTS31     8192 Mar 15  2004 .
drwx------   4 CICSHT## CICSTS31     8192 Jul  4 16:14 ..
-rw-------   1 CICSHT## CICSTS31     2976 Dec  5  2004 Snap0001.trc
-rw-r--r--   1 CICSHT## CICSTS31     1626 Jul 16 11:15 dfhjvmerr
```

```
-rw-r--r--   1 CICSHT## CICSTS31        0 Mar 15  2004 dfhjvmin
-rw-r--r--   1 CICSHT## CICSTS31      458 Oct  9 14:28 dfhjvmout
-rw-r--r--   1 CICSHT## CICSTS31    64175 May 11 18:00 event.log
/u/cicsht##:>
```

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 all these examples, the owner has read and write permissions, but the group and all others have only read.

Note: The name of the file owner (CICSHT## in the example) is displayed in the list, but owner permissions are actually associated with the UID. If other CICS region user IDs have been assigned the same UID, they have the same permissions as CICSHT##. Remember that this practice is not normally recommended in a z/OS UNIX system.

You need to give each CICS region permission to access the HFS directories and files that it uses. To give your CICS regions permissions, you must be 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, if 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 all your CICS regions to access z/OS UNIX. The owner could have that RACF group as their default group (DLFTGRP) or be connected to it as one of their supplementary groups.

When you need to change the permissions for directories and files, 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

u is for user (owner) permissions, g is for group permissions,
o is for other permissions
```

HFS permissions for CICS Web support

When you use HFS files to provide static responses to requests from Web clients, a CICS region which receives those requests and provides the responses needs **read** access to the HFS files and to the directories containing them.

If you have stored all the files relevant to each CICS region in a directory structure below the home directory for the CICS region, you can make the CICS region the owner of each directory and file (with the appropriate owner permissions). If some HFS files are used by more than one CICS region, you will need to use one of the other solutions described in this topic, such as group permissions or access control lists (ACLs). In the procedure described below for Java support, the first step gives examples of how to set up group permissions for HFS directories and files used by

multiple CICS regions. The use of “other” permissions, which would give access to every HFS user, is not recommended for CICS Web support in a production environment.

HFS permissions for Java support

When you are setting up Java support in a CICS region, the required directories and files fall into three categories:

1. The directories and files that every CICS region needs to create JVMs.
2. The working directory that you have specified for input, output and messages from the JVMs in each individual CICS region. (This might be the home directory for the CICS region.)
3. Any 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. This includes any directories and files that you have *changed* from their original locations, for example, JVM profiles that you have moved to a different directory. It also includes any directories and files that you have *added* to be used with JVMs or for CORBA applications and enterprise beans, for example, your own application classes, or classes that you have added to the trusted middleware class path.

If you want to give CICS access to the files required for Java support using group permissions, you can use the following procedure to grant the appropriate permissions. If you want to give CICS access using another type of permission, or a combination of the different permissions depending on the level of security that you require for each type of directory or file, then you can use the following procedure, but substitute an alternative type of permission (“other” or owner) as appropriate for the different types of directory or file.

1. **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 Software Developer Kit for z/OS, Java 2 Technology Edition, Version 1.4.2. These directories and files are:
   - Most of the files in the `/usr/lpp/cicsts/cicsts31` directory and its subdirectories. The `cicsts31` directory name is a user-defined value that you chose for the CICS_DIRECTORY variable used by the DFHIJVMJ job during CICS installation; `cicsts31` is the default. The files in this directory and its subdirectories include the supplied sample JVM profiles and JVM properties files, the CICS-supplied JAR files such as `dfjcics.jar` and `dfjcsi.jar`, and some of the files that CICS includes on the trusted middleware class path.
   - Some of the files in the `/usr/lpp/java142/J1.4/bin` and `/usr/lpp/java142/J1.4/bin/classic` directories that contain the IBM persistent reusable JVM code. The *java142/J1.4* directory names are the default values when you install the IBM Software Developer Kit for z/OS, Java 2 Technology Edition, Version 1.4.2.

   Each CICS region requires **read and execute** access to these directories and files. To grant this access:

   a. Display the directories and files as described earlier in this topic, and check that the group permissions for the directories and files give the correct access to the group (the second set of permissions). If you need to change the permissions, use the UNIX command `chmod`, as described earlier in this topic.

b. Assign to the RACF group that all your CICS regions can use, the group permissions for the /usr/lpp/cicsts/cicsts31 directory and its subdirectories, and for the files in them. To do this, issue the UNIX command

```
chgrp -R GID /usr/lpp/cicsts/cicsts31
```

where *GID* is the numeric z/OS UNIX group identifier (GID) which you assigned to the RACF group that all your CICS regions can use. The -R in the command means that the group is changed for not only the cicsts31 directory, but also all the subdirectories, and all the files in the directory and subdirectories. Because your CICS region user IDs are connected to this group, the CICS regions now have read and execute permission for all these directories and files.

c. Assign to the same RACF group, the group permissions for the /usr/lpp/java142/J1.4/bin directory and its subdirectories, and the files in them. To do this, issue the UNIX command

```
chgrp –R GID /usr/lpp/java142/J1.4/bin
```

as you did above for the /usr/lpp/cicsts/cicsts31 directory. Your CICS regions now have read and execute permission for all these directories and files.

2. **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 while following the procedure described in “Giving CICS regions a z/OS UNIX user identifier (UID) and group identifier (GID) and setting up a home directory” on page 54. 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 HFS directories that are used by the WORK_DIR option or the Java class.

   Each CICS region requires **read, write and execute** access to the HFS 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. For each directory that is used by more than one CICS region, follow the same procedure that you carried out for the directories and files that every CICS region needs to create JVMs, ensuring that you give the group write access (w) as well as read and execute access.

3. **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 option in a JVM profile or to the `ibm.jvm.shareable.application.class.path` system property in a JVM properties file, so that the JVM will search those directories for your own application classes.
   - You add directory paths to the TMPREFIX or TMSUFFIX options on a JVM profile, so that they will be part of the trusted middleware class path.
   - You add directory paths to the LIBPATH, which contains the directories that are searched for native C dynamic link library (DLL) files that are used by the JVM, including those required to run the JVM and additional native libraries loaded by trusted code.
   - You create your own JVM profiles or JVM properties files, or move the supplied JVM profiles or JVM properties files to a directory that is not under the `/usr/lpp/cicsts/cicsts31` directory. (You can use the EXEC CICS INQUIRE JVMPROFILE command to find the HFS directory that contains a JVM profile, provided that the JVM profile has been used during the lifetime of the CICS region. The HFS 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/cicsts31` directory, or the `/usr/lpp/java142/J1.4/bin` and `/usr/lpp/java142/J1.4/bin/classic` directories.
   - 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 HFS directories and files that you have identified in this category. If you have already set up any of these items, make sure that you have set the correct permissions for the directories and files involved, and given your CICS regions permission to access them. When you set up these items later on, return to this topic and for each directory or file, follow the same procedure that you carried out for the directories and files that every CICS region needs to create JVMs.

Verifying the Java installation using sample programs

This topic describes how to run the "Hello World" and "Hello CICS World" sample programs to verify that Java has been successfully installed and set up in a CICS region.

Before you begin to run the Java sample programs, verify that the Java supplied components are correctly installed in your CICS region. Use the checklist provided in The *CICS Transaction Server for z/OS Installation Guide*.

To verify your Java installation, follow the steps below to set up and run the supplied sample programs.

1. Build the sample programs using the following steps:
   a. Define the environment variables *PATH*, *CICS_HOME* and *JAVA_HOME*. Instructions on what to define for each variable are described in “Building the JCICS sample programs” on page 44.
   b. Install the group DFH$JVM in order to run the samples.
   c. Build the Java samples, as described in “Building the Java samples” on page 45.
2. Add `$CICS_HOME/samples/dfjcics` to the end of the Java classpath, `ibm.jvm.shareable.application.class.path`, in the default JVM properties file, `dfjjvmpr.props`.
3. Run the Hello World samples using the steps outlined in “Running the Hello World samples” on page 46

If you have any problems running the Java sample programs, do the following:

Chapter 10. Understanding JVMs

CICS provides the support you need to run a Java program in a z/OS 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. You can find information about Java on the z/OS platform at `http://www.ibm.com/servers/eserver/zseries/software/java/`

CICS TS 3.1 supports the JVM provided by the IBM Software Developer Kit for z/OS, Java 2 Technology Edition, Version 1.4.2 .

Note: There are two versions of the IBM Software Developer Kit for z/OS, Java 2 Technology Edition Version 1.4, a 31-bit and a 64-bit version. CICS TS 3.1 supports only the 31-bit version, which must be at the 1.4.2 level.

This JVM features persistent reusable JVM technology and includes several optimizations designed for the execution of CICS transactions. These optimizations are:

- The ability for JVMs to share a cache of commonly-used class files that are already loaded, enabling faster JVM startup and reducing the cost of class loading. When a new JVM that shares the class cache is initialized, it can use these pre-loaded classes instead of reading them from the file system. Also, if the JVM performs just-in-time (JIT) compilation for any of the classes, it can write the results back to the shared class cache, and other JVMs can then use the compiled classes. All the heap data (objects and static variables) are owned by the individual JVMs; this maintains the isolation between the applications being processed in the JVMs.
- The serial reuse of a JVM for multiple Java programs, avoiding most of the initialization costs. Serial reuse might or might not involve resetting the state of the JVM between uses.
- An optimized garbage-collection scheme, enabled by the clean separation of short-lived application objects from long-lived classes, objects, and native state (that is, non-Java or C language state), which are reset.

“The structure of a JVM” on page 64 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 creates JVMs. This process is described in “How CICS creates JVMs” on page 71.
- CICS manages the pool of JVMs that it has created. This process is described in “How CICS manages JVMs in the JVM pool” on page 75.
- 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 79.
- Most JVMs can be reused once an application has finished using them to run a Java program. There are three levels of reusability. JVMs might be reset and reused (resettable JVMs), or they might be reused without being reset (continuous JVMs), or they might be thrown away after use (single-use JVMs). “How JVMs are reused” on page 85 explains the difference between these types of JVM.

- CICS creates 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” on page 89 describes this.

Chapter 9, “Setting up Java support,” on page 53 tells you how to set up and use JVMs in your CICS system.

Java programs that ran under CICS Transaction Server for z/OS, Version 2 Release 2 or CICS Transaction Server for z/OS, Version 2 Release 3 can also run under CICS Transaction Server for z/OS, Version 3 Release 1. CICS Transaction Server for z/OS, Version 2 supported the JVM created by the IBM Developer Kit for OS/390 Java 2 Technology Edition Version 1.3.1s, which also featured the persistent reusable JVM technology. However, the older type of JVM that was introduced in CICS Transaction Server for OS/390, Version 1 Release 3, which was not reusable, is no longer supported. Any Java programs that ran under CICS Transaction Server for OS/390, Version 1 Release 3 must be migrated to Java 2 to run under the JVM provided by the IBM Software Developer Kit for z/OS, Java 2 Technology Edition, Version 1.4.2. “Removal of support for CICS Transaction Server for OS/390, Version 1 Release 3 JVMs” on page 92 has more information about this.

The structure of a JVM

This topic summarizes what you need to know about the structure of a JVM in order to use JVMs with CICS. You can find more detailed information about the structure of a JVM in the document *Persistent Reusable Java Virtual Machine User's Guide*, SC34-6201.

This topic covers:

Classes in a JVM

There are three types of class in a JVM:

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*. They have a special status that allows the objects created from them to be associated with middleware or the application, depending on the kind of class that invokes their construction.
2. The middleware, which provides services that access resources. This includes the JCICS interfaces classes, JDBC, JNDI, and so on. These classes are known as *middleware classes*. Middleware is trusted by the JVM to manage its own state between one use of a JVM and the next, and it can therefore operate without restrictions, and is trusted to make changes to the JVM environment, even if the JVM is resettable. This enables optimizations through the caching of state (classes and native libraries, for example) to be used by multiple applications. However, middleware is responsible for resetting itself correctly at the end of a transaction and, if necessary, for reinitializing at the beginning of a new transaction, in order to isolate different applications from each other. *Persistent Reusable Java Virtual Machine User's Guide*, SC34-6201 explains how middleware should be written.

3. The classes for the user application. These classes are known as *application classes*.
   - Application classes can be shareable, meaning that when they have been loaded, they are kept across JVM reuses and resets, so that they can be used by other transactions. If the JVM is reset, they are re-initialized.
   - Alternatively, application classes can be nonshareable, if they are placed on the standard class path. In a resettable JVM, nonshareable application classes are discarded when the JVM is reset, and must be reloaded each time they are required. In a continuous JVM, however, they are not discarded, and are kept intact for subsequent reuses of the JVM.
   - When a JVM is defined as resettable, if application classes perform actions that change the JVM environment, these actions are noted and the JVM is destroyed after the application has finished using it. In a continuous JVM, this restriction does not apply, and application classes are permitted to make changes to the JVM environment. ("How JVMs are reused" on page 85 explains more about resettable and continuous JVMs.)

The classes in a JVM, and the objects that they create, are placed in different storage heaps in the JVM according to their expected lifetime. For example, nonshareable application classes (on the standard class path) are placed in the transient heap in a resettable JVM, because this heap is deleted if the JVM is reset between uses. "Storage heaps in a JVM" on page 69 explains how the classes and objects are arranged in storage heaps.

The JVM can identify the correct type for each class because of the *class path* on which the class is included. The class path determines how the class is loaded by the JVM, where it is stored, and how it is treated. So, for example, any class that is included on the shareable application class path is loaded by the shareable application class loader, stored in the application-class system heap, kept across JVM reuses and resets, and re-initialized if the JVM is reset. *Persistent Reusable Java Virtual Machine User's Guide*, SC34-6201 explains more about the process of loading classes.

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. ("How CICS creates JVMs" on page 71 explains JVM profiles and JVM properties files.) Generally speaking, when you are preparing Java applications that will run in a JVM, you need to ensure that all the middleware and application classes required by the application are included on the class paths defined by the JVM profile and JVM properties file that are requested by the application. You also need to ensure that any native C dynamic link library (DLL) files that are required for the application (which have the extension .so in z/OS UNIX) are included on the library path in the JVM profile. You do not need 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.

Note that although for convenience we refer to "including a class on a class path", the name of the class itself (including the name of the package, if the program has been built as a package) is not actually 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 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), the name of the JAR file is included on the class path as if it were the name of a directory. "Adding application classes to the class paths for a JVM" on page 128 explains more about this.

In the JVM provided by the IBM Software Developer Kit for z/OS, Java 2 Technology Edition, Version 1.4.2, which features the persistent reusable JVM technology, the four class paths on which classes or native libraries can be included are as follows:

1. The **library path** is for native C dynamic link library (DLL) files that are used by the JVM (which have the extension .so in z/OS UNIX), including those required to run the JVM and additional native libraries loaded by trusted code. This might include the DLL files needed to use the DB2 JDBC drivers, or any native code associated with a class that you are using to redirect JVM output (named on the USEROUTPUTCLASS option in the JVM profile).

   The library path is defined by the LIBPATH option in the JVM profile. “Adding application classes to the class paths for a JVM” on page 128 tells you how to include items on the library path. Note that if the JVM is to use the shared class cache (see “The shared class cache” on page 89), you will need to include the DLL files in the JVM profile for the master JVM that initializes the shared class cache, rather than in the JVM profile for the JVM where the application will run. The master and worker JVMs use the same library path to ensure that they are using the same versions of these files. However, note that the files are not loaded into the shared class cache. Unless they are shared through another z/OS facility (such as the shared library region), a copy is loaded into each worker JVM.

2. The **trusted middleware class path** is for middleware classes, that is, classes that are trusted by the JVM to manage their own state across a JVM-reset. Trusted middleware classes are permitted to change the JVM environment even if the JVM is resettable, so for this reason you should not normally place your own application classes on the trusted middleware class path. However, you might need to add classes for middleware supplied by IBM or by another vendor, which are not included in the standard JVM setup for CICS. For example, to use the DB2-supplied JDBC drivers with Java programs and enterprise beans, you need to add a DB2-supplied zip file to the trusted middleware class path.

   In CICS, the trusted middleware class path is built automatically from the paths that you specify using the CICS_DIRECTORY, TMPREFIX, and TMSUFFIX options in the JVM profile. “Options in JVM profiles” in the *CICS System Definition Guide* has more details about these options. There is a corresponding system property, `ibm.jvm.trusted.middleware.class.path`, in the JVM properties file, but you cannot use this system property for CICS. “Adding application classes to the class paths for a JVM” on page 128 tells you how to include classes on the trusted middleware class path. Note that if the JVM is to use the shared class cache (see “The shared class cache” on page 89), you will need to include the middleware classes in the JVM profile for the master JVM that initializes the shared class cache, rather than in the JVM profile for the JVM where the application will run.

3. The **shareable application class path** is for shareable application classes, that is, application classes that you want to be cached, either in the JVM or in the shared class cache. Defining the JVM as a resettable JVM subjects these classes to restrictions which mean that they cannot affect or pass state to subsequent transactions that use the JVM. When you add a class to this class path:

   - If the JVM uses the shared class cache (see “The shared class cache” on page 89), the classes are obtained from the shared class cache, rather than being loaded by each individual JVM.
   - If the JVM does not use the shared class cache but is resettable, the classes are cached in the JVM, and are reinitialized when the JVM is reset.

- If the JVM does not use the shared class cache and is a continuous JVM, the classes are cached in the JVM, and are kept across reuses, but are not reinitialized.

Adding application classes to this class path, rather than to the standard class path, produces the best performance, and it should be your normal choice for loading application classes in a production environment.

The shareable application class path is defined by a system property, `ibm.jvm.shareable.application.class.path`, in the JVM properties file."Adding application classes to the class paths for a JVM" on page 128 tells you how to include classes on the shareable application class path. Note that if the JVM is to use the shared class cache, you will need to include the shareable application classes in the JVM properties file for the master JVM that initializes the shared class cache, rather than in the JVM properties file for the JVM where the application will run.

4. The **standard class path** is for nonshareable application classes, that is, application classes that you do not want to be shared by other JVMs or across JVM resets. Like shareable application classes, defining the JVM as a resettable JVM subjects these classes to restrictions which mean that they cannot affect or pass state to subsequent transactions that use the JVM. When you add a class to this class path:
   - If the JVM uses the shared class cache (see "The shared class cache" on page 89), the standard class path is the only class path that is taken from the JVM profile for the JVM itself, rather than from the JVM profile for the master JVM that initialises the shared class cache. The classes are loaded by the individual JVM, and are not stored in the shared class cache.
   - If the JVM is resettable, classes on this class path are discarded when the JVM is reset, and reloaded from HFS files each time the JVM is reused.
   - If the JVM is a continuous JVM, nonshareable application classes are kept intact from one JVM reuse to the next.

   You should not normally place application classes on the standard class path without a good reason for doing so, as it causes a degradation in performance in a resettable JVM, and for worker JVMs (both resettable and continuous) it uses more storage than having a single copy of the classes in the master JVM. Some possible reasons for choosing this class path, instead of the shareable application class path, are:
   - In a non-production environment, you might use this class path during application development if your JVMs are resettable, because it means you do not have to phase out the JVM pool in order to update class definitions. (If your JVMs are continuous, you still need to phase out the JVM pool.)
   - If a particular class is used infrequently, you might use this class path if you prefer to incur the performance cost of reloading the class each time it is required, rather than the storage cost of keeping the class in the JVM or in the shared class cache.

   The standard class path is defined by the CLASSPATH option in the JVM profile. There is a corresponding system property, `java.class.path`, in the JVM properties file, but you cannot use this system property for CICS. "Adding application classes to the class paths for a JVM" on page 128 tells you how to include classes on the standard class path.

Enterprise beans are a special case. 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 shareable application 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 128 tells you how to do this.

Compiled classes

Java programs can execute in a JVM running on any supported platform through the ability of the JVM to interpret Java bytecode. You create Java bytecode class files using a Java compiler, such as VisualAge for Java or WebSphere Studio Application Developer, and these classes can be executed by a JVM without the need for any further translation.

This mode of executing Java classes is by interpretation, but a more efficient method in terms of performance is to convert the Java bytecode into z/OS machine code, like load modules. The JIT-compile function of the JVM provides this service. It produces JIT-compiled versions of frequently used Java methods, normally at variable times during the usage of the methods. The JIT-compiling process incurs additional CPU time and uses extra Language Environment storage, but provides more efficient executable code. The CPU cost of the Java applications reduces after the JIT-compiled code is produced.

Where a JVM is constructed

Each JVM that CICS creates 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 run-time options used by CICS for a Language Environment enclave control the initial size of, and incremental additions to, the Language Environment enclave heap storage. Within this overall allocation of storage, a JVM's storage heaps are created according to the settings in the JVM profile for the JVM. “Storage heaps in a JVM” on page 69 explains how these storage heaps are arranged.

You can tune the run-time 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.

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. (This might be the master JVM that initializes the shared class cache.) 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.

Storage heaps in a JVM

A JVM manages run-time storage in several segregated heaps. The classes described in “Classes in a JVM” on page 64, and the objects created by those classes, are grouped in these storage heaps according to their expected lifetime. The size of the storage heaps is determined by options in the JVM profile for a JVM. The level of reusability that you choose for the JVM affects the structure of the storage heaps in the JVM.

The storage heaps in a JVM are:

System heap
The main system heap contains the class definitions for all the system classes and standard extension classes, and for the middleware classes. It also contains the pooled string constant data, and it might contain some system class objects that persist for the lifetime of the JVM. For continuous JVMs and single-use JVMs, the system heap is also used for items that would be contained in the application-class system heap for a resettable JVM. (“How JVMs are reused” on page 85 explains the differences between these types of JVM.)

Application-class system heap
The application-class system heap, or ACS heap, is intended to contain objects which persist for the lifetime of the JVM (that is, they are kept across JVM reuses) and which are reinitialized if the JVM is reset. Continuous JVMs and single-use JVMs do not have an application-class system heap, because these types of JVM are not reset after each use; only resettable JVMs have an application-class system heap.

If the JVM has an application-class system heap, that heap contains the class definitions for application classes on the shareable application class path; that is, those specified by the `ibm.jvm.shareable.application.class.path` system property (see “Classes in a JVM” on page 64). It also contains class objects that represent the shareable application classes. However, it does *not* contain nonshareable application classes on the standard class path, that is, those specified by the CLASSPATH option in the JVM profile.

Nonsystem heap
This storage heap comprises two other storage heaps of variable size:

Middleware heap
The middleware heap contains objects constructed by middleware classes, and any objects constructed by system classes as a result of calls from middleware classes. It also contains static data for the middleware classes and the system classes, and other string constant data. The objects in this storage heap have a lifetime that is greater than a single transaction, so they are kept across JVM resets. For continuous JVMs and single-use JVMs, the middleware heap is also used for items that would be contained in the transient heap for a resettable JVM.

Transient heap
The objects in this storage heap are intended to have a lifetime that is the same as the transaction using the JVM. Continuous JVMs and single-use JVMs do not have a transient heap, because these types of JVM are not reset after each use; only resettable JVMs have a transient heap.

If the JVM has a transient heap, that heap contains objects constructed by shareable and nonshareable application classes, and any objects constructed by system classes as a result of calls from application classes. It also contains the class definitions and static data for any application classes on the standard class path; that is, classes that are specified by the CLASSPATH option in the JVM profile. The transient heap is completely deleted when the reset takes place. If subsequent transactions in the resettable JVM want to use the application classes that were in this heap, they must reload them from HFS files. In a continuous JVM, which does not have a transient heap, nonshareable application classes are kept intact from one JVM reuse to the next.

Figure 2 on page 71 shows how the storage heaps in a resettable JVM are allocated from the Language Environment enclave heap storage, depending on the options specified in the JVM profile for the JVM.

The system heap's initial storage allocation is set by the `Xinitsh` option in a JVM profile. The application-class system heap's initial storage allocation is set by the `Xinitacsh` option in a JVM profile. These two heaps do not have a specified maximum size; they can grow until they run out of space within the Language Environment enclave.

The nonsystem heap works differently. The nonsystem heap's maximum total size is set by the `Xmx` option in a JVM profile. From this maximum total, storage is allocated to the transient heap and to the middleware heap. The transient heap's initial storage allocation is set by the `Xinitth` option in a JVM profile, and the middleware heap's initial storage allocation is set by the `Xms` option in a JVM profile. Both heaps can grow. The middleware heap is allocated from low storage in the nonsystem heap and expands upwards; the transient heap is allocated from high storage in the nonsystem heap, and expands down towards low storage. They can expand only until the two heaps meet—their combined total size cannot exceed the maximum size set for the nonsystem heap (the `Xmx` option).

Continuous and single-use JVMs do not have an application-class system heap or a transient heap, because these types of JVM are not reset after each use. For these types of JVM, the nonsystem heap consists only of the middleware heap, and therefore the `Xmx` option only limits the maximum size of the middleware heap.

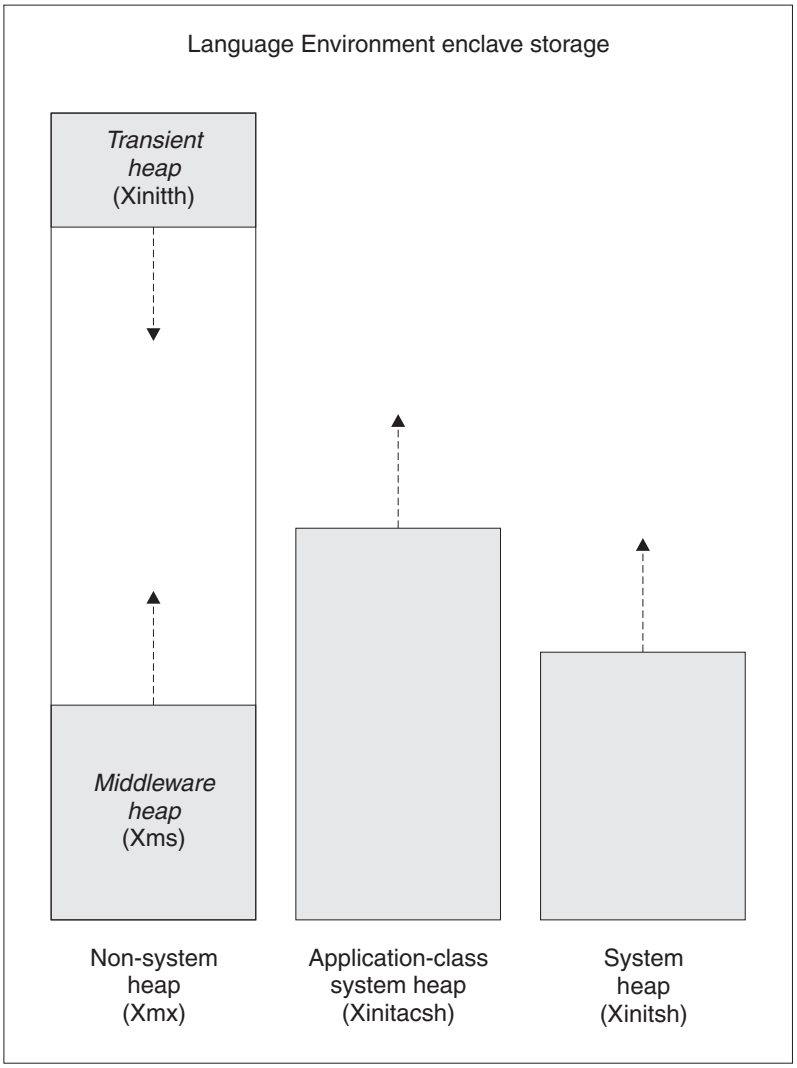

Figure 2. JVM storage heap allocations within a Language Environment enclave

Persistent Reusable Java Virtual Machine User's Guide, SC34-6201, has more detailed information about the storage heaps in a JVM.

You can tune the size of the storage heaps to achieve optimum performance for your JVMs. “Tuning storage for individual JVMs” in the *CICS Performance Guide* tells you how to do this.

How CICS creates JVMs

A JVM is created by the CICS launcher program for JVMs. *Persistent Reusable Java Virtual Machine User's Guide*, SC34-6201, explains what a launcher program does. CICS requests storage from MVS, sets up a Language Environment enclave for the JVM, and launches the JVM in the Language Environment enclave.

CICS creates JVMs in response to requests to run a Java program. JVMs are created to fit the needs of a particular Java program. You specify the program's needs using the PROGRAM resource definition, just as you would for a non-Java program. (The *CICS Resource Definition Guide* has full details about the PROGRAM resource definition.) Requests to run a Java program can be made in

various ways; “How CICS locates the PROGRAM resource definition to create a JVM” on page 74 explains how CICS finds the PROGRAM resource definition in each case.

To create a JVM for a Java program, CICS needs to obtain the following information from the PROGRAM resource definition:

- The fact that the program needs a JVM. This is specified in the JVM attribute.
- The execution key (user key or CICS key) for the Java program, which determines the execution key for the JVM. See “Execution key (EXECKEY attribute).”
- The main class in the Java program (the Java class whose `public static main` method is to be invoked). This is specified in the JVMCLASS attribute.
- The JVM profile for the JVM, which determines various characteristics of the JVM. See “JVM profiles (JVMPROFILE attribute)” on page 73.

“Enabling applications to use a JVM” on page 119 tells you how to specify all these items using the PROGRAM resource definition for the Java program. Note that CORBA stateless objects and enterprise beans do not have a PROGRAM resource definition as such. The PROGRAM resource definition that is relevant to CORBA stateless objects and enterprise beans is that for the request processor program.

Execution key (EXECKEY attribute)

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. (“How CICS manages JVMs in the JVM pool” on page 75 explains how JVMs and TCBs are related.)

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. “How CICS allocates JVMs to applications” on page 79 explains why reusing existing JVMs is more economical.

JVM profiles (JVMPROFILE attribute)

CICS can use various options when creating a JVM. You can create different sets of options, known as *JVM profiles*, that produce JVMs that are suitable for different applications. The JVM profiles contain the Java launcher options, and also reference a *JVM properties file* containing the system properties for the JVM. (System properties are key name and value pairs that contain basic information about the JVM and its environment.) Among other things, the JVM properties file determines the security properties of the JVM. CICS supplies sample JVM profiles and JVM properties files, and in many cases, you may find that you can use these unchanged.

When CICS receives a request to run a Java program, a JVM profile is named on the PROGRAM resource definition for the Java program. (“How CICS locates the PROGRAM resource definition to create a JVM” on page 74 explains how CICS locates the PROGRAM resource definition for different types of request.) CICS creates a JVM using the options given in this JVM profile, and the system properties given in the JVM properties file that the JVM profile references. Alternatively, CICS finds a free JVM that it has already created with these options and system properties.

A JVM profile specifies, among other things:

- The library path, for native C dynamic link library (DLL) files that are used by the JVM, including those required to run the JVM and additional native libraries loaded by trusted code.
- The middleware class path, for classes that are to be treated as trusted middleware classes (see “Classes in a JVM” on page 64).
- The standard class path, for nonshareable application classes that are to be discarded if the JVM is reset (see “Classes in a JVM” on page 64).
- The initial size of the storage heaps in the JVM, and how far they can expand (see “Storage heaps in a JVM” on page 69).
- The maximum size of the stacks for Java code and C code.
- The level of reusability for the JVM: whether it can be reset and reused (resettable JVMs), or reused without being reset (continuous JVMs), or thrown away after use (single-use JVMs). “How JVMs are reused” on page 85 explains more about this.
- Whether the JVM uses the shared class cache, so is a worker JVM (see “The shared class cache” on page 89).
- The destinations for messages from JVM internals and for output from Java applications running in the JVM. (“Redirecting JVM output” on page 135 tells you more about these options.)
- The level of messages that the JVM should issue about its activities.
- Whether the JVM should perform additional checks on certain activities.
- The settings for assertion checking for system classes and application classes.
- Whether the JVM should support debugging.
- The path to the JVM properties file containing the system properties for the JVM.

The *CICS System Definition Guide* has the full list of options that you can specify using a JVM profile.

Note: In some earlier versions of CICS, you could use the `-Xquickstart` option (specified using the `Xservice` option) in a JVM profile to reduce the startup time for the JVM. However, with improvements in JVM technology, the `-Xquickstart` option is now permanently enabled, and specifying `-Xquickstart` in a JVM profile has no effect.

A JVM properties file specifies, among other things:

- The shareable application class path, for application classes that are to be kept across JVM reuses and reinitialized if the JVM is reset (see "Classes in a JVM" on page 64).
- The name server to be used for JNDI references.
- Security information for access to an LDAP name server.
- 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. "Using JDBC and SQLJ to access DB2 data from Java programs and enterprise beans written for CICS" in the *CICS DB2 Guide* has more information about this.
- 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. Setting these system properties enables the Java 2 security policy mechanism for the JVM. ("Protecting Java applications in CICS by using the Java 2 security policy mechanism" on page 329 has more information about this.)
- The working directory.
- Whether event logging should be enabled.

The *CICS System Definition Guide* has the full list of options that you can specify using a JVM properties file.

"Setting up JVM profiles and JVM properties files" on page 94 tells you how to set up suitable JVM profiles and JVM properties files to meet the needs of your applications.

How CICS locates the PROGRAM resource definition to create a JVM

When an application starts a Java program, CICS obtains the information it requires to create the JVM from the CICS PROGRAM resource definition that applies to the request. 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).
- A method request for an enterprise bean or CORBA stateless object. This is matched to a request model, which specifies a transaction identifier.

  Enterprise beans and CORBA stateless objects 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. The default transaction for REQUESTMODEL definitions is CIRP, which specifies the PROGRAM resource definition for the default request processor program DFJIIRP.

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.

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, and for method requests for an enterprise bean or CORBA stateless object, CICS determines the PROGRAM resource definition by looking at the transaction identifier. CICS can then obtain the information from the PROGRAM resource definition that it needs to create the JVM: the name of the JVM profile, the main class in the Java program, and the execution key for the Java program and the JVM. Figure 3 shows this process.

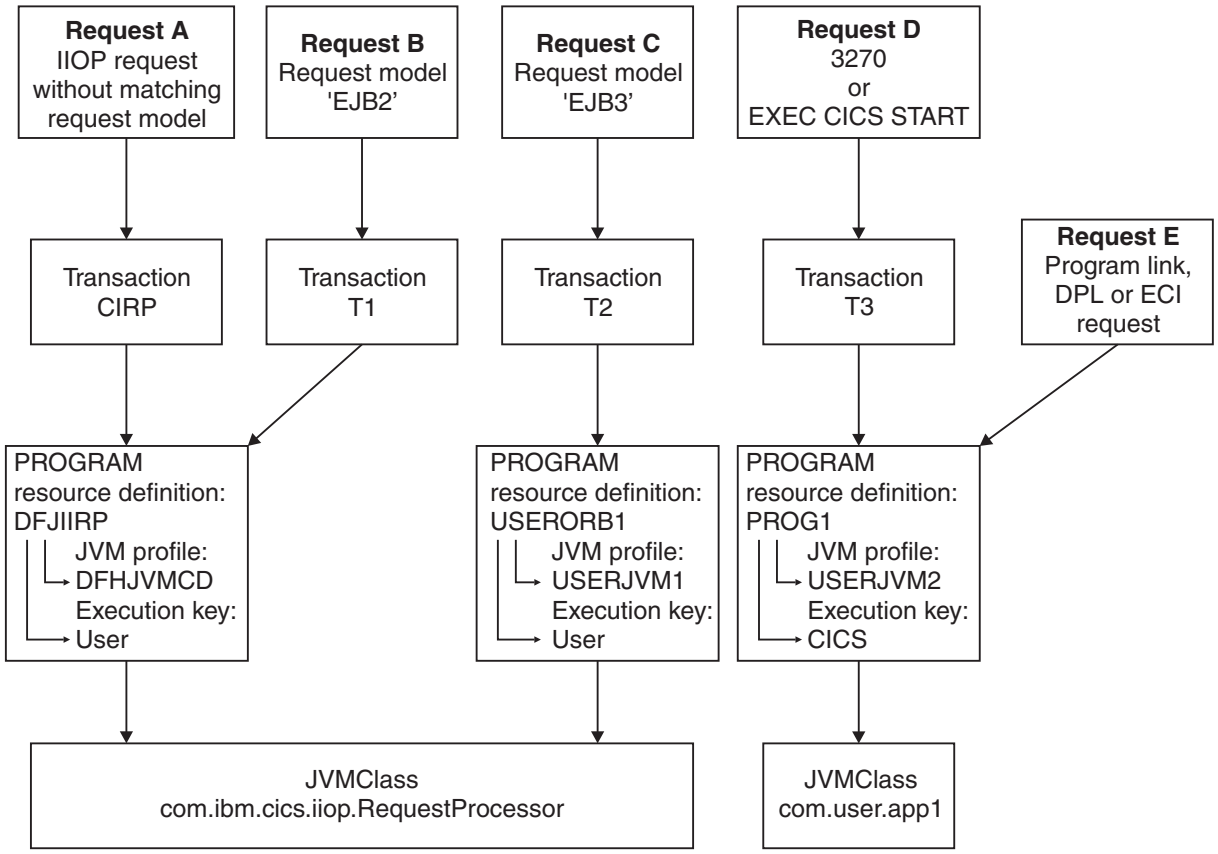

Figure 3. How CICS finds the PROGRAM resource definition

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.

When the CICS dispatcher allocates a TCB for a new JVM to run on, it associates the name of the JVM profile with the TCB, so the TCB and the JVM are linked together. However, this linkage only lasts for the lifetime of the JVM. If CICS destroys the JVM (perhaps because an unresettable event has occurred in a resettable JVM, or because CICS needs the space to create a different type of JVM), then the TCB remains in the JVM pool, and it can be reallocated for a different JVM to run on.

CICS creates JVMs and TCBs as they are needed. The CEMT INQUIRE JVMPOOL command (or the equivalent EXEC CICS command) tells you how many JVMs CICS currently has.

The total number of TCBs that CICS can create 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. (JM TCBs, used for the master JVM that initializes the shared class cache, do not count towards the MAXJVMTCBS limit. “The shared class cache” on page 89 explains more about JM TCBs.)

The JVMs that CICS creates can be in one of two execution keys: user key or CICS key. You can use the CEMT INQUIRE JVM command (or the equivalent EXEC CICS command) to find out the protection key in which a JVM has been created. JVMs that are in user key—that is, JVMs intended for programs that specify EXECKEY(USER) on their PROGRAM resource definition—need to run on a J9 TCB. JVMs that are in CICS key—that is, JVMs intended for programs that specify EXECKEY(CICS) on their PROGRAM resource definition—need to run on a J8 TCB. You cannot specify the proportions of J8 and J9 TCBs that are in the JVM pool. The MAXJVMTCBS system initialization parameter specifies the maximum total number of J8 and J9-mode TCBs in the JVM pool, and CICS decides how many of them should be J8 TCBs and how many should be J9 TCBs, according to the number of requests that specify each execution key. Statistics are collected separately for each of the modes, so you can see what proportions of each mode are in the JVM pool.

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. 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.

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. The JVM is either reset (if it is defined as a resettable JVM), or simply kept in the pool without a reset (if it is defined as a continuous JVM). “How JVMs are reused” on page 85 explains the difference between these levels of reusability. 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.

Figure 4 on page 78 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.

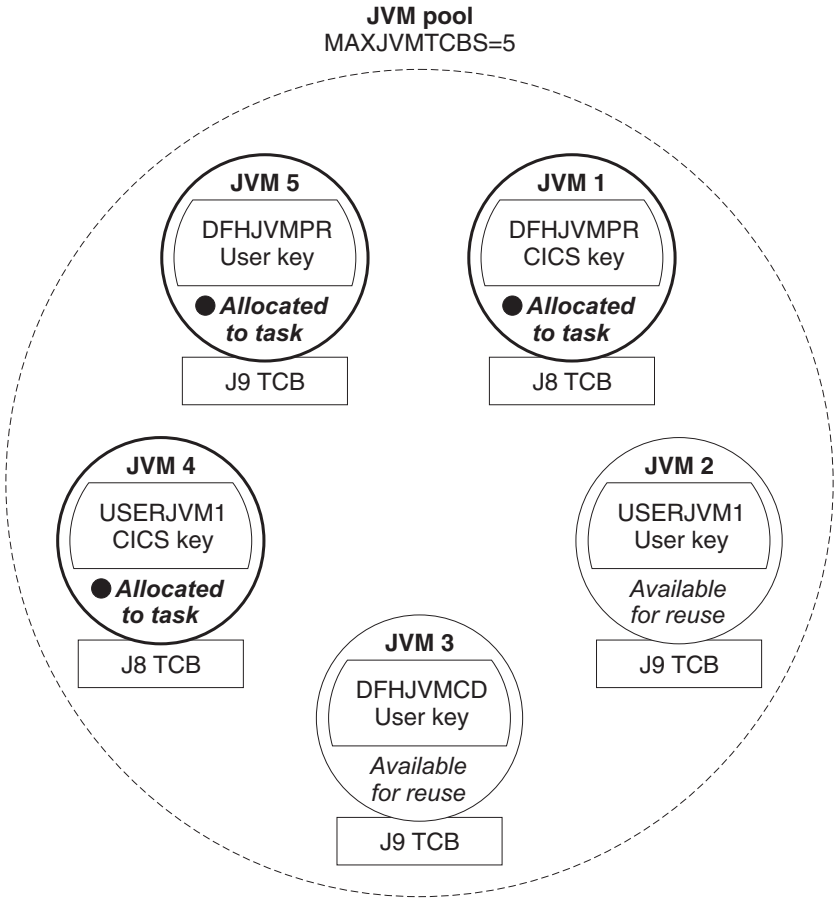

Figure 4. An example JVM pool

CICS reduces the number of active JVMs automatically if the workload does not require them. If a JVM is inactive for 30 minutes, it is discarded. You can also terminate all the JVMs in the JVM pool (by using the CEMT SET JVMPOOL Phaseout, Purge or Forcepurge command, or the equivalent EXEC CICS command), or disable the JVM pool so that it cannot service new requests (by using the CEMT SET JVMPOOL DISABLED command, or the equivalent EXEC CICS command).

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. 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 CEMT PERFORM CLASSCACHE PHASEOUT, PURGE or FORCEPURGE command (or the equivalent EXEC CICS commands).

"Managing your JVMs" on page 132 tells you more about managing the JVM pool.

When an application requests execution of a Java program, CICS first sees if the Java program can reuse one of the existing JVMs in the JVM pool that is not currently allocated to a task. If the application can reuse an existing JVM, CICS has saved the cost of creating a new JVM. If a suitable JVM is not available, and the limit set by the MAXJVMTCBS system initialization parameter has not yet been reached, CICS allocates a new open TCB in the correct mode (J8 or J9), and creates a new JVM. When the limit set by the MAXJVMTCBS system initialization parameter has been reached, and no more JVMs can be created, CICS decides how best to allocate the JVMs in the pool to the applications that request them. "How CICS allocates JVMs to applications" explains how CICS decides whether an application can reuse an available JVM, and how it allocates JVMs to applications when it cannot create any more JVMs.

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. 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 80 shows this process happening. Our example JVM pool is in the state shown above in Figure 4 on page 78, with a MAXJVMTCBS limit of 5, and 5 JVMs in the pool. CICS receives two of the requests described above in Figure 3 on page 75.

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.

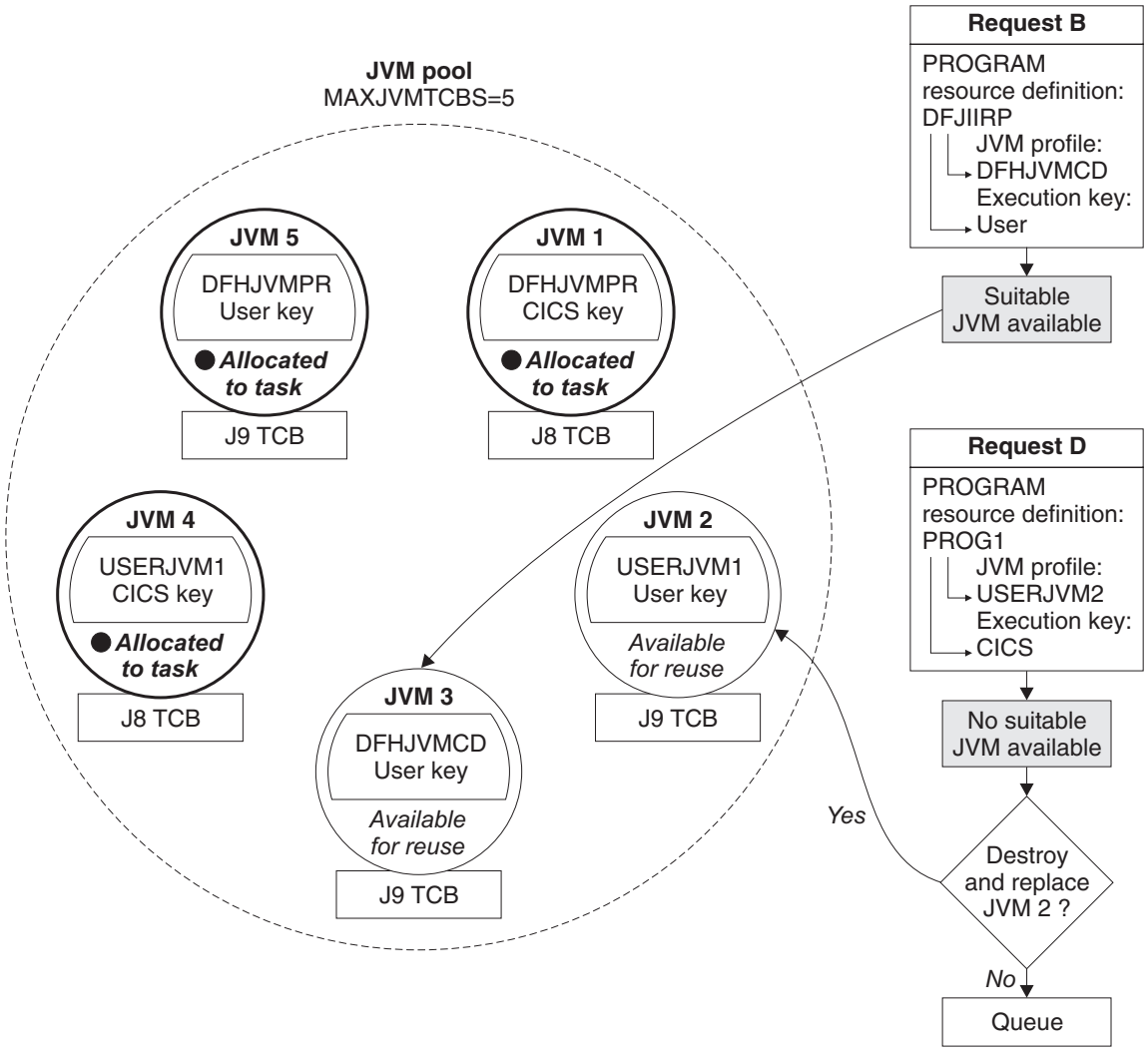

Figure 5. Dealing with requests for JVMs: example

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 in Figure 6:

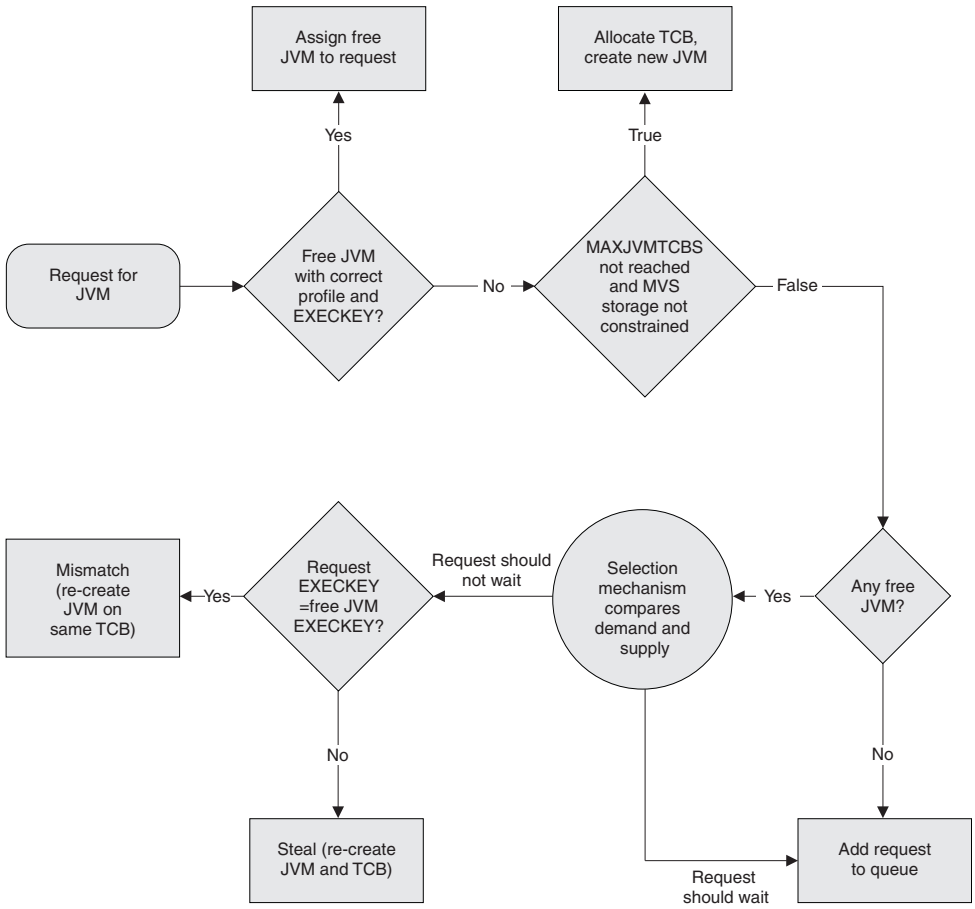

Figure 6. Dealing with incoming requests for JVMs

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 the actions summarized in Figure 7 on page 83:

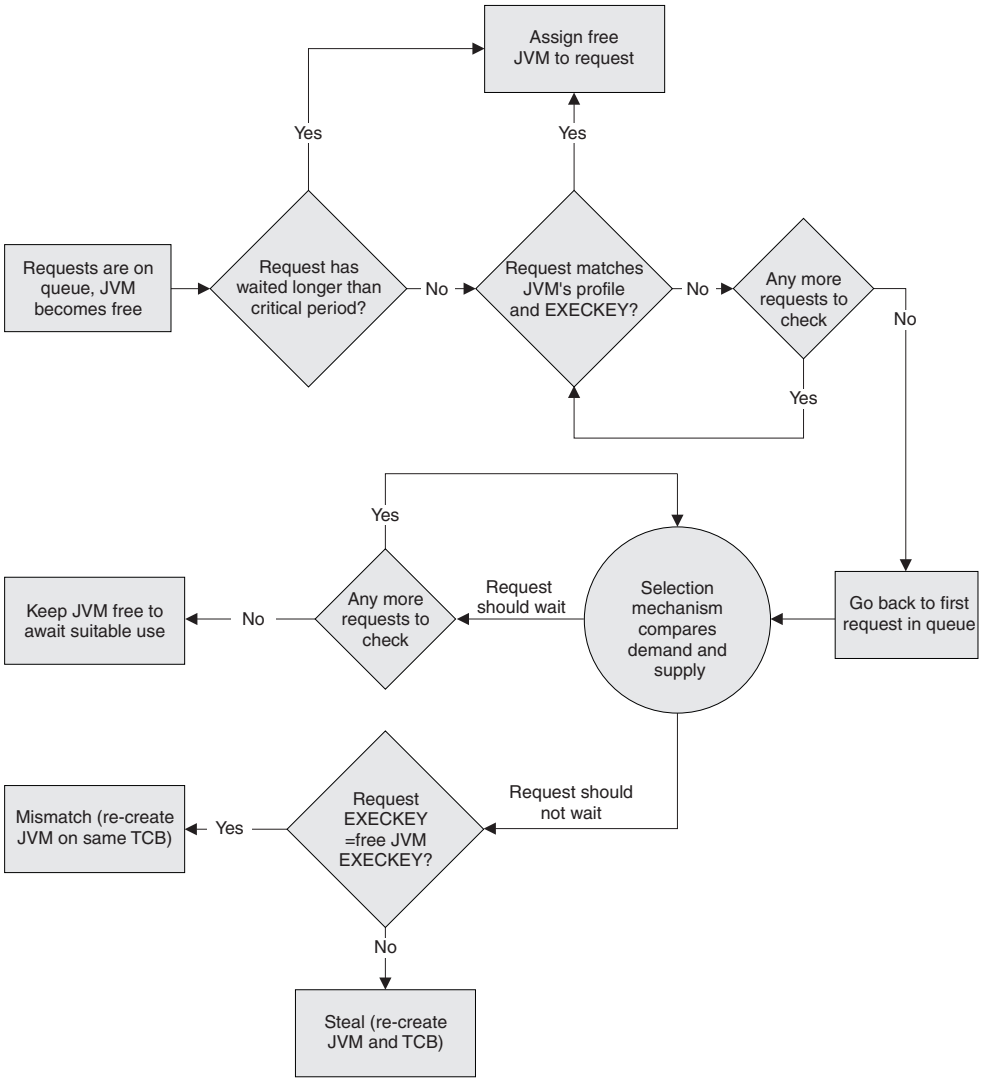

Figure 7. Dealing with a queue of requests waiting for a JVM

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

Let's look at how the selection mechanism works. As we saw, the 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 JVM provided by the IBM Software Developer Kit for z/OS, Java 2 Technology Edition, Version 1.4.2, which features the persistent reusable JVM technology, 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. The earlier JVM supported by CICS in CICS TS 1.3, which was provided by the IBM Developer Kit for the Java Platform 1.1.8, was a single-use JVM, which had to be initialized afresh for every application. This model typically had high startup overheads. JVM reuse is the preferred method for running all Java applications in CICS, and provides the only way to run Java applications comprising enterprise beans or which are started by IIOP requests.

CICS provides three levels of reusability for JVMs, which you can select depending on the needs of your applications. The level of reusability for a JVM is controlled by the REUSE option in the JVM profile for the JVM. The characteristics of the three levels of reusability can be summarized as follows:

Table 4. Reuse and reset characteristics of JVM types

JVM type (and action between JVM uses)	REUSE option in profile	Compatible with the shared class cache?	Are program invocations allowed to pass state to subsequent invocations?	Are programs allowed to change characteristics of the JVM?	Performance
Continuous (JVM reused without reset)	REUSE= YES	Yes[1]	Yes	Yes	Highest (JVM not initialized or reset for each use)
Resettable (JVM reused and reset)	REUSE= RESET	Yes[1]	No (JVM storage cleaned up after each use)	No (JVM destroyed if this occurs)	Medium (JVM reset, but not initialized for each use)
Single-use (JVM destroyed)	REUSE= NO	No	No (JVM destroyed)	Yes	Lowest (JVM initialized for each use)

Note:

1. The worker JVMs in a CICS region all have the same level of reusability as the master JVM in that region, so you cannot mix resettable worker JVMs and continuous worker JVMs in a CICS region.

The following sections discuss each of these types of JVM in more detail:

- "Continuous JVMs (REUSE=YES)" on page 86
- "Resettable JVMs (REUSE=RESET)" on page 87
- "Single-use JVMs (REUSE=NO)" on page 88

Continuous JVMs (REUSE=YES)

The continuous JVM is kept in the JVM pool for reuse. It is initialized once, and is reused many times, but it is not reset after each Java program has completed. A continuous JVM has the option REUSE=YES in its JVM profile.

Compared to the resettable JVM, the continuous JVM has a greater transaction throughput and lower CPU usage, because it is not performing a reset. The behavior of the continuous JVM is also 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, because there is no JVM reset after each Java program, the application code that runs in the next Java program or transaction is not isolated from the actions of the previous program invocation. Because of this, you can create persistent items that might be of use to future executions of the same application in the same JVM. (In a resettable JVM, this is not possible.) You do need to ensure that programs do not change the state of a continuous JVM in undesirable ways, or leave any unwanted state in the JVM.

Both middleware classes and application classes are permitted to perform actions in a continuous JVM which would cause a resettable JVM to be marked unresettable and destroyed. The application classes are trusted to reset themselves as required between transactions, and the JVM is not destroyed after use if these events take place. “Resettable JVMs (REUSE=RESET)” on page 87 explains how a resettable JVM deals with unresettable actions.

A continuous JVM maintains the content of its storage heaps between one program invocation and the next. “Storage heaps in a JVM” on page 69 explains the storage heaps that the JVM uses for different categories of objects. 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. Shareable application classes are not reinitialized, and nonshareable application classes are kept intact, instead of being discarded and reloaded. The application can choose to clean up any unwanted items and retain any desirable items. Also, a continuous JVM does not invoke the `ibmJvmTidyUp` method to request the middleware classes to perform cleanup; this cleanup will only take place if the middleware classes perform it in the course of their normal actions. (The CICS-supplied middleware does perform cleanup without a request from the JVM.)

A continuous JVM can use the shared class cache (that is, it can be a worker JVM). JVMs that use the shared class cache start up more quickly, and have lower storage requirements, than JVMs that do not. The worker JVMs in a CICS region all have the same level of reusability as the master JVM in that region, so you cannot mix resettable worker JVMs and continuous worker JVMs in a CICS region; you need to choose one level of reusability for your worker JVMs. “Setting up the shared class cache” on page 106 has more information about this.

“Programming considerations for continuous JVMs” on page 120 explains the programming considerations for applications that run in a continuous JVM.

Resettable JVMs (REUSE=RESET)

The resettable JVM is kept in the JVM pool for reuse. The JVM is initialized once, and is reused many times. It can be reset to a known state between uses. A resettable JVM has the option REUSE=RESET in its JVM profile (or the older option `Xresettable=YES`).

The resettable JVM is normally reset between transactions; that is, after the application code has terminated for one transaction and before the application code starts for the next transaction. If more than one Java program is used in a transaction, the resettable JVM is reset after each Java program has completed. This level of reusability is equivalent to specifying the `-Xresettable` option for a JVM.

The JVM reset isolates invocations of Java programs from changes made by previous invocations of programs in the same JVM. This means that programs cannot create persistent items that might be of use to future executions of the same program, but it also means that programs cannot leave unwanted state in the JVM, or change the state of the JVM. However, the time and CPU usage required for a JVM reset reduce the performance of a resettable JVM compared to the performance of a continuous JVM. An application that has been coded with attention to the state of the JVM and to the items in static storage can operate safely without the JVM reset, so it can run in a continuous JVM to achieve performance enhancements.

There are two stages in the process of resetting a JVM:

1. The resettable JVM checks whether there have been any unresettable events since the last reset of the JVM. A frequent cause of an unresettable event is that the Java program that just ran in the JVM has performed an unresettable action. An unresettable action is when a program uses Java interfaces that modify the state of a JVM in a way that cannot be properly reset, such as changing system properties or loading a native library. The *Persistent Reusable Java Virtual Machine User's Guide*, SC34-6201, has more information about unresettable actions. Another possible cause of an unresettable event is if a cross-heap reference in the JVM has been found, in the course of a trace-for-unresettability check, to be still in scope (rather than in garbage). Unresettable events can also occur if there is an error in the JVM code.

   If one or more unresettable events are detected during the execution of a user's Java program, the JVM is marked unresettable, and CICS destroys the JVM when the Java program has finished using it. The storage used by the JVM is recovered, and a new JVM is initialized. The events that prevent reuse are logged, provided that the appropriate event logging system properties are specified in the JVM properties file for the JVM.

   Middleware classes, that is, classes specified on the trusted middleware class path for the JVM, are permitted to perform unresettable actions without the JVM being marked unresettable. This is because they are trusted by the JVM to perform the actions necessary to reset themselves between transactions. You should use the application class paths for your own application classes, to ensure that if they do perform an unresettable action in a resettable JVM, the JVM is destroyed. “Classes in a JVM” on page 64 explains more about the different classes and class paths in a JVM.
2. A resettable JVM cleans up its storage heaps after each use, meaning that state cannot persist from one program invocation to the next. “Storage heaps in a JVM” on page 69 explains the storage heaps that the JVM uses for different categories of objects. In a resettable JVM, the transient heap (which contains

objects constructed by the user's application classes) is completely deleted during the reset, and the user's shareable application classes that are kept in the application-class system heap are reinitialized during the reset. The middleware heap, which contains objects constructed by the middleware classes, is not cleaned up by the JVM during the reset. Instead, the JVM requests the middleware classes to perform their own cleanup, and the middleware classes are trusted to reset the objects they have constructed.

A resettable JVM can use the shared class cache (that is, it can be a worker JVM). JVMs that use the shared class cache start up more quickly, and have lower storage requirements, than JVMs that do not.

"Programming considerations for resettable JVMs" on page 122 explains the programming considerations for applications that run in a resettable 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 (or the older option `Xresettable=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 "Removal of support for CICS Transaction Server for OS/390, Version 1 Release 3 JVMs" on page 92). 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 or for a resettable JVM.

The single-use JVM has the lowest performance of any of the JVM types in terms of transaction throughput, because the JVM must be initialized for each use. Some time is saved by the absence of a reset, but this is less than the time used to initialize the JVM.

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 unresettable actions are eliminated, and the programs can run in a continuous JVM or a resettable JVM.

A single-use JVM cannot use the shared class cache (that is, it cannot be a worker JVM). Because it cannot use the shared class cache, a single-use JVM has a longer startup time and higher storage requirements than a resettable or continuous JVM that is using the shared class cache, as well as incurring the startup costs each time the JVM is used.

"Programming considerations for single-use JVMs" on page 125 explains the programming considerations for applications that run in a single-use JVM.

The shared class cache

CICS includes a shared class cache facility for the JVM. The shared class cache is created using the JVM's `-Xjvmset` option. Multiple JVMs can share a single cache of class files that have already been loaded, including some that have been optimized by compilation. The shared class cache replaces the system heap and the application-class system heap for those JVMs, and it can contain middleware and application classes. JVMs that use the shared class cache start up more quickly, and have lower storage requirements, than JVMs that don't.

The shared class cache is initialized by a JVM referred to as the master JVM. The master JVM cannot be used to run Java applications; it exists only to initialize and own the shared class cache. The master JVM obtains shared memory in which its system heap is allocated. The system heap contains class files (including those that have been optimized by compilation) which can be shared by all the worker JVMs, and the rest of the shared memory contains other information that is common to the master and worker JVMs, such as the class loading paths needed to load classes into the shared class cache. The master JVM can be defined as a resettable JVM, with the option REUSE=RESET or the older option `Xresettable=YES` in its JVM profile, or as a continuous JVM, with the option REUSE=YES in its JVM profile. If none of these options is included, CICS assumes that the master JVM is resettable. The master JVM runs on its own open TCB, the JM TCB. JM TCBs are not used for any other purpose. They do not count towards the MAXJVMTCBS limit, and they cannot be reused like the J8 and J9 TCBs in the JVM pool.

The JVMs that share the class cache are referred to as worker JVMs, and they can be used to run Java applications. The worker JVMs use the classes loaded in the shared class cache, instead of having to load these classes from the file system. Although the worker JVMs share the class cache, each worker JVM owns all the working data (objects and static variables) for the applications that run in it. This helps to maintain the isolation between the Java applications being processed in the system.

The worker JVMs in a CICS region all have the same level of reusability as the master JVM. “How JVMs are reused” on page 85 explains the levels of reusability for JVMs. If the master JVM is a resettable JVM, the worker JVMs are also resettable, and if the master JVM is a continuous JVM, the worker JVMs are also continuous. (Single-use JVMs cannot use the shared class cache.) If the REUSE or `Xresettable` options are included in the JVM profile for a worker JVM, they are ignored.

CICS supports one active shared class cache in each region. (A region might also contain old shared class caches that are being phased out.) The shared class cache can support the majority of the JVMs in each region. Some of the JVMs in the region might not be suited to sharing the class cache, because they are debug JVMs used for problem diagnosis, or because they have an inappropriate level of reusability. These JVMs can still run as standalone JVMs, and have their own cache of classes in their storage heaps.

The shared class cache contains:

- The IBM-supplied middleware that you need to run enterprise beans and Java applications, and any other middleware classes that you have specified (on the trusted middleware class path).

- Any application classes that are loaded by shared application class loaders, including classes on the shareable application class path, and classes that are loaded from a DJAR.

The master and worker JVMs use the same library path, which is the path for native C dynamic link library (DLL) files that are used by the JVM, to ensure that they are using the same versions of these files. However, these files are not loaded into the shared class cache. Unless they are shared through another z/OS facility (such as the shared library region), a copy is loaded into each worker JVM.

The library path and trusted middleware class path for the shared class cache are defined in the JVM profile for the master JVM, and the shareable application class path for the shared class cache is defined in the JVM properties file for the master JVM. For a worker JVM, CICS ignores these class paths if they are specified in the worker's own JVM profile and JVM properties file, and instead uses the values specified for these class paths in the JVM profile and JVM properties file for the master JVM.

This means that for a worker JVM, items on the library path, middleware classes, and shareable application classes must be included in the class paths in the JVM profile and JVM properties file for the master JVM that initializes the shared class cache, rather than in the JVM profile and JVM properties file for the JVM where the application will run. The library path is defined by the LIBPATH option in the JVM profile, and the trusted middleware class path is defined by the CICS_DIRECTORY, TMPREFIX, and TMSUFFIX options in the JVM profile. The shareable application class path is defined by the `ibm.jvm.shareable.application.class.path` system property in the JVM properties file.

The standard class path (defined by the CLASSPATH option in the JVM profile) is the only class path that is taken from the profile for the worker JVM itself, rather than from the profile for the master JVM. Classes on this class path are loaded into the individual worker JVMs, and are not cached in the shared class cache. Adding classes to this class path is detrimental to performance for a resettable worker JVM, because the classes are reloaded every time the JVM is reset. For a continuous worker JVM, these classes are kept intact from one JVM reuse to the next, so there is no need to reload them, but having the classes in every JVM uses more storage than having a single copy in the master JVM. For these reasons, you should avoid using the standard class path for worker JVMs in a production environment.

Any worker JVM can modify the shared class cache. When worker JVMs perform just-in-time (JIT) compilation of classes that are in the shared class cache, they write the results of the compilation to the shared class cache, so that other worker JVMs can use the compiled classes. The master JVM that initializes the shared class cache is invoked in user key, so that worker JVMs that were invoked in user key can read and write to the shared class cache. Even if all the worker JVMs that share the class cache are invoked in CICS key, the master JVM and the shared class cache are still in user key.

Figure 8 on page 91 shows an example JVM pool when a shared class cache has been introduced for the CICS region. The JVM pool contains:

- Two worker JVMs (JVMs 6 and 7) created with the JVM profile DFHJVMPC, in user key (so running on a J9 TCB). DFHJVMPC is the CICS-supplied sample JVM profile for a worker JVM.

- A worker JVM (JVM 9) created with the JVM profile USERJVM1, in CICS key (so running on a J8 TCB). When the shared class cache was introduced, the JVM profile USERJVM1 was changed to state that JVMs with that profile use the shared class cache.
- A standalone JVM (JVM 8) created with the JVM profile DFHJVMPR, in user key (so running on a J9 TCB). As the JVM was created with DFHJVMPR, it does not use the shared class cache.
- A standalone JVM (JVM 10) created with the JVM profile USERJVM2, in CICS key. The JVM profile USERJVM2 was not changed when the shared class cache was introduced, and JVMs with that profile do not use the shared class cache.

The shared class cache, shown on the right of the diagram, is initialized by a master JVM created with the JVM profile DFHJVMCC, which is the CICS-supplied default JVM profile for a master JVM, and the execution key USER. The master JVM runs on a JM TCB. The worker JVMs (6, 7 and 9) are using the shared class cache, but the standalone JVMs (8 and 10) are not.

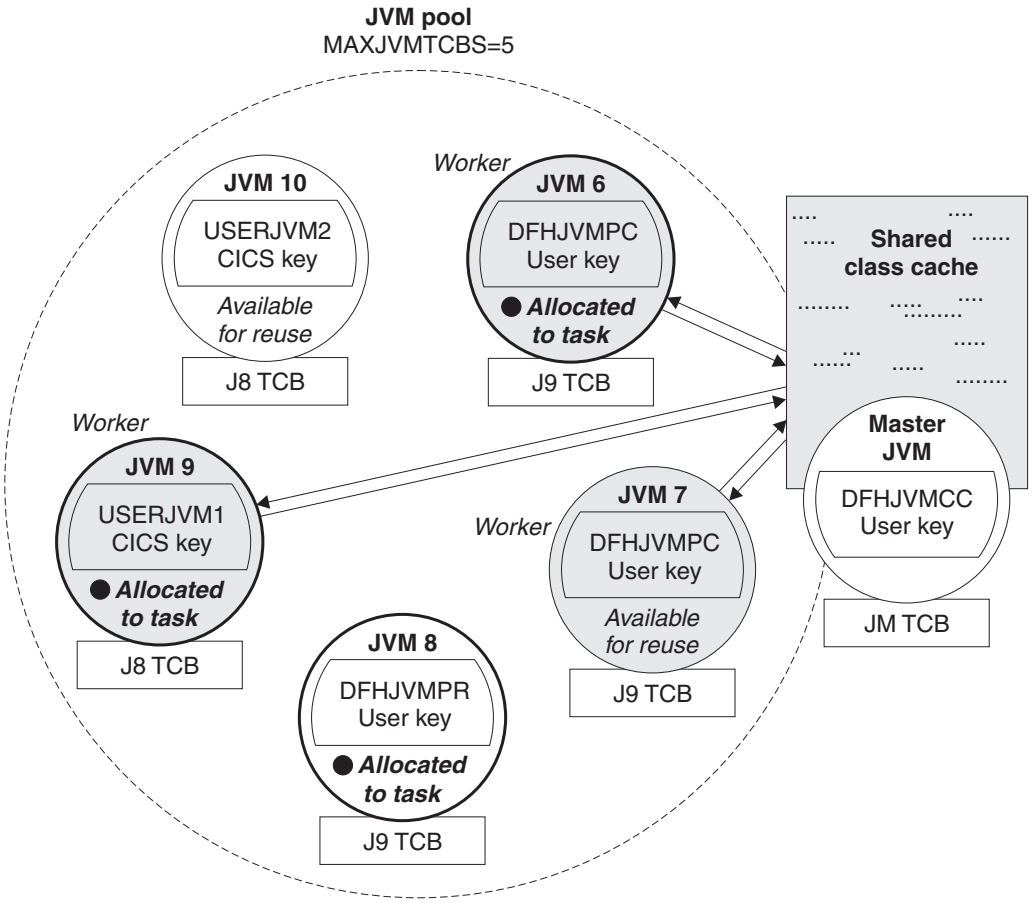

Figure 8. Example JVM pool with a shared class cache

"Setting up the shared class cache" on page 106 tells you how to set up a shared class cache in a CICS region, and how to enable JVMs to use it.

You can manage the shared class cache using CICS commands. You can prevent the shared class cache from starting automatically, start it manually, adjust its size, update the classes or JAR files that it contains, or terminate it. You can also monitor its status. "Managing the shared class cache" on page 110 tells you how to operate the shared class cache.

Removal of support for CICS Transaction Server for OS/390, Version 1 Release 3 JVMs

The JVM introduced in CICS TS 1.3 is not supported. Any Java programs that ran under CICS TS 1.3, and were not previously migrated for CICS Transaction Server for z/OS, Version 2 Release 2, should be migrated to Java 2 to run under the JVM provided by the IBM Software Developer Kit for z/OS, Java 2 Technology Edition, Version 1.4.2 or later, which features the persistent reusable JVM technology. Application migration issues are discussed at:

```
http://java.sun.com/j2se/1.4.2/compatibility.dital
http://java.sun.com/j2se/1.4/compatibility.dital
http://java.sun.com/products/jdk/1.3/compatibility.dital#incompatibilities1.3
and
http://java.sun.com/products/jdk/1.2/compatibility.dital
```

Support for the JVM provided by the IBM Software Developer Kit for z/OS, Java 2 Technology Edition, Version 1.4.2 or later, completely replaces the JVM support provided in CICS TS 1.3. However, you can modify a JVM to run as a single-use JVM and not attempt serial reuse. A single-use JVM is initialized, is used to run a single Java program, and then is automatically destroyed without attempting a JVM reset. The single-use JVM is like the earlier JVM that was supported by CICS in CICS TS 1.3. New Java applications should not be developed in such a way that they can only run in a single-use JVM.

You can modify a JVM to be a single-use JVM by specifying either REUSE=NO, or the older option `Xresettable=NO`, in the JVM profile. This might be necessary to run programs that use Java interfaces, such as multi-threading, that make JVMs unresettable. “Single-use JVMs (REUSE=NO)” on page 88 has more information about the appropriate use of single-use JVMs.

For a single-use JVM, you can, if you want, invoke the user-replaceable program DFHJVMAT to change JVM options. The *CICS Customization Guide* tells you how to use DFHJVMAT. DFHJVMAT cannot be used with any type of JVM other than the single-use JVM. If you need to change JVM options for other types of JVM, do so by customizing the JVM profile and JVM properties file for the JVM. “Setting up JVM profiles and JVM properties files” on page 94 explains how to do this.

Chapter 11. Using JVMs

This section tells you how to customize JVM profiles and properties files; manage your JVMs and shared class cache; and explains how to idenitfy problems with your Java applications and JVMs.

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

1. Set up a JVM profile and JVM properties file to create a JVM for your Java application.

   JVM profiles allow you to specify options that produce different JVMs depending on your application requirements. Setting up JVM profiles and JVM properties files tells you how to choose suitable options for your Java applications, how to use the supplied sample files, and how to customize these samples or set up your own files.
2. Set up and customize a shared class cache for your CICS region, so that the JVMs can start up faster.
   a. Setting up the shared class cache tells you how to set up a shared class cache, and how to enable JVMs to use it. Most JVMs can use the shared class cache, but if you do not want certain JVMs to use it, you can set them to run independently as standalone JVMs.
   b. Managing the shared class cache tells you how to alter the shared class cache in your CICS region while CICS is running. You can customize the shared class cache to prevent it from starting automatically, adjust its size, update the classes or JAR files that it contains, or terminate it.

   Your CICS region is now ready to create JVMs and run Java applications in them.
3. Enable your application to use a JVM.
   a. Set the appropriate Java attributes on the PROGRAM resource definition for the Java program.
   b. Add the classes for the application to the class paths for the JVM, which are set by using the options in the JVM profiles and JVM properties file for the JVM.

   Enabling applications to use a JVM tells you how to perform both of these steps.
4. You can monitor the JVMs in your JVM pool, and make tuning adjustments to achieve optimum performance. “Managing your JVMs” on page 132 tells you how to monitor your JVMs, how to redirect the output from the JVMs, and how to tune your JVM pool.
5. If you have any problems with your JVMs or Java applications, there are a number of facilities you can use to identify the cause.
   a. “Problem determination for JVMs” on page 138 gives an overview of the facilities that you can use to identify any problems with your JVMs, and “Controlling tracing for JVMs” on page 140 tells you how to control tracing for your JVMs
   b. If a Java application is causing problems, or if you are developing new Java applications, you can use debugging tools to examine and debug an application while it is running in a JVM. “Debugging an application that is running in a CICS JVM” on page 142 tells you how to set up a JVM for debugging, and how you can use debugging tools and plugins with a JVM.

Note that the older type of JVM that was introduced in CICS Transaction Server for OS/390, Version 1 Release 3 is no longer supported. Any Java programs that ran under CICS Transaction Server for OS/390, Version 1 Release 3, and were not previously migrated for CICS Transaction Server for z/OS, Version 2, must be migrated to Java 2 to run under the persistent reusable JVM. “Removal of support for CICS Transaction Server for OS/390, Version 1 Release 3 JVMs” on page 92 has more information about this.

Setting up JVM profiles and JVM properties files

The JVM is started by the CICS Java launcher, which uses a set of options known as a JVM profile. A JVM profile determines the characteristics of a JVM, and applications specify the JVM profile that they want their assigned JVM to have. In the JVM profiles used by CICS, you can specify standard options that are supported in the persistent reusable JVM runtime environment, and also some non-standard options that are subject to change in future releases of the Java language specification. You can set up several JVM profiles that use different options to cater for the needs of different applications.

JVM profiles are text files stored on HFS, and they list the Java launcher options. Each JVM profile references a JVM properties file, which is another text file containing the system properties for the JVM. (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.) Among other things, the JVM properties file determines the security properties of the JVM. You can edit JVM profiles and JVM properties files using any standard text editor. CICS supplies sample JVM profiles and JVM properties files to help you get started.

“How CICS creates JVMs” on page 71 explains how CICS uses JVM profiles, and gives an overview of the options that you can specify using JVM profiles and their associated JVM properties files.

To set up JVM profiles and JVM properties files suitable for your applications, follow the instructions in:

- “Enabling CICS to locate the JVM profiles and JVM properties files”
- “Choosing a JVM profile and JVM properties file” on page 96
- “Customizing or creating JVM profiles and JVM properties files” on page 102

As well as determining the characteristics of a JVM, the JVM profiles and JVM properties files are used to specify the class paths, that is, the directories that the JVM searches for the application classes and resources that are needed for your applications. When you have set up your JVM profiles and JVM properties files, you will need to add classes to the class paths for each application that uses the JVM profiles and JVM properties files. “Enabling applications to use a JVM” on page 119 tells you how to do this.

Enabling CICS to locate the JVM profiles and JVM properties files

When an application requests a JVM, CICS needs to find the JVM profile for that JVM, and the JVM properties file that it references, on HFS. If you alter the location or the name of either of these items, you need to let CICS know. This section tells you how to do this.

As JVM profiles and JVM properties files are HFS files, case is important. When you use the name of a JVM profile or JVM properties file anywhere in CICS, you must enter it using the same combination of upper and lower case characters that

is present in the HFS file name. The CEDA panels accept mixed case input for a JVM profile name irrespective of your terminal's UCTRAN setting. However, this does not apply when the name of a JVM profile is entered on the CEDA command line, or in 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 any other CICS transaction, ensure that the terminal you use is correctly configured, with upper case translation suppressed.

Locating the JVM profiles

When an application requests a JVM and names a particular JVM profile for CICS to use, CICS looks in the HFS directory that is specified by the JVMPROFILEDIR system initialization parameter, and loads the JVM profile from that directory.

When you install CICS, the CICS-supplied sample JVM profiles are placed in the directory `/usr/lpp/cicsts/cicsts31/JVMProfiles`, where `cicsts31` is the value that you chose for the CICS_DIRECTORY variable used by the DFHIJVMJ job during CICS installation. The default value of JVMPROFILEDIR is set as `/usr/lpp/cicsts/cicsts31/JVMProfiles`, so the supplied setting for JVMPROFILEDIR points to the **default** directory for the sample JVM profiles. If you chose a different name during CICS installation for the directory containing the sample JVM profiles (that is, if you chose a non-default value for the CICS_DIRECTORY variable used by the DFHIJVMJ job), and you plan to use the CICS-supplied sample JVM profiles, change the value of JVMPROFILEDIR to specify the correct directory name.

If you are using the CICS-supplied sample JVM profiles, and only changing them by adding your own classes to the class paths, then you can leave JVMPROFILEDIR to point to the directory containing the sample JVM profiles. However, if

- you create customized versions of the sample JVM profiles and change their behaviour, but want to keep the original versions for reference
- you create your own JVM profiles

then you might want to keep these JVM profiles in a directory other than the samples directory, and tell CICS to load the JVM profiles from the directory that you have used.

If you want CICS to load the JVM profiles from a directory other than the `/usr/lpp/cicsts/cicsts31/JVMProfiles` directory, you need to do one of the following:

- Change the value of the JVMPROFILEDIR system initialization parameter to specify your preferred directory. (The value that you specify can be up to 240 characters long.)
- Link to your JVM profiles from the directory specified by JVMPROFILEDIR, by means of UNIX soft links. (This method enables you to store your JVM profiles in any place in the HFS file system.)

You also need to ensure that CICS has read and execute access on HFS for your JVM profiles and the directory containing them. “Giving CICS regions permission to access HFS directories and files” on page 56 tells you how to do this.

Note that the JVM profiles DFHJVMPR and DFHJVMCD, and their associated JVM properties files, must always be available to CICS. 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). Both these JVM profiles must therefore either be present in the directory that is specified by JVMPROFILEDIR, or linked to by means of UNIX soft links from that directory.

If you need to locate a particular JVM profile in HFS, you can use the EXEC CICS INQUIRE JVMPROFILE command to find the full path name of the HFS file for the JVM profile, provided that the JVM profile has been used during the lifetime of the CICS region. (Note that there is no CEMT equivalent for this command.)

Locating the JVM properties files

When you install CICS, the CICS-supplied sample JVM properties files are placed in the directory `/usr/lpp/cicsts/cicsts31/props/`, where `cicsts31` is the value that you chose for the CICS_DIRECTORY variable used by the DFHIJVMJ job during CICS installation.

The JVMPROPS option on a JVM profile references a JVM properties file by using its full path name. The CICS-supplied sample JVM profiles reference the sample JVM properties files as follows:

```
JVMPROPS=/usr/lpp/cicsts/cicsts31/props/dfjjvmpx.props
```

where `dfjjvmpx.props` is the name of the sample JVM properties file that matches with the sample JVM profile.

If you are using the CICS-supplied sample JVM properties files, and only changing them by adding classes to the class paths, then you can leave this reference as it is. However, if you change the name or location of a JVM properties file, or create your own JVM properties file, you need to change the JVMPROPS option to specify the correct path name in all the JVM profiles that reference that JVM properties file. You also need to ensure that CICS has read and execute access on HFS for the JVM properties file and the directory containing it. “Giving CICS regions permission to access HFS directories and files” on page 56 tells you how to do this.

Choosing a JVM profile and JVM properties file

To help you get started, CICS supplies several sample JVM profiles and JVM properties files. Table 5 on page 97 describes these files.

Table 5. CICS-supplied sample JVM profiles and JVM properties files

JVM profile	Associated JVM properties file	Comments
DFHJVMPR	`dfjjvmpr. props`	Profile DFHJVMPR is the default if no JVM profile is specified in a Java program's resource definition. It specifies REUSE=RESET, which causes CICS to reset the JVM and make it available for reuse for another task, after the JVM finishes running each Java program. JVMs created with the profile DFHJVMPR **do not** use the shared class cache (the profile specifies CLASSCACHE=NO). So JVMs created with DFHJVMPR are resettable standalone JVMs. You can specify this profile for JVMs that are to be used by your own applications. DFHJVMPR and DFHJVMPC are the recommended profiles for defining your own JVMs that are to be used by enterprise beans. DFHJVMPR is the default if no other JVM profile is specified, and it is used for sample programs, so make sure that it is set up correctly for your CICS region.
DFHJVMPC	`dfjjvmpc.props`	DFHJVMPC is similar to the default JVM profile, DFHJVMPR, except that it specifies CLASSCACHE=YES, and omits the options that are not required when CLASSCACHE=YES is specified. JVMs with this profile **do** use the shared class cache, so they are resettable worker JVMs. This JVM profile is compatible with the shared class cache defined by DFHJVMCC. You can specify this profile for JVMs that are to be used by your own applications. DFHJVMPR and DFHJVMPC are the recommended profiles for defining your own JVMs that are to be used by enterprise beans. Single-use JVMs, and JVMs that are configured for debug, cannot use the shared class cache.
DFHJVMPS	`dfjjvmps. props`	DFHJVMPS specifies REUSE=NO, which causes CICS to make each JVM available for use by a single Java program only— it is a single-use JVM. JVMs created with the profile DFHJVMPS **do not** use the shared class cache. You can specify this profile for JVMs that are to be used by your own applications. However, this profile is not recommended for JVMs that are to be used by enterprise beans. DFHJVMPS 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. “How JVMs are reused” on page 85 has more information about this.

Table 5. CICS-supplied sample JVM profiles and JVM properties files (continued)

JVM profile	Associated JVM properties file	Comments
DFHJVMCC	`dfjjvmcc.props`	DFHJVMCC is the default profile used to configure the master JVM that initializes the shared class cache. It defines a shared class cache suitable for use by JVMs in which enterprise beans can be executed. This JVM profile is the default for the JVMCCPROFILE system initialization parameter.“Setting up the shared class cache” on page 106 has more information about this. Do not specify this profile for JVMs that are to be used by your own applications.
DFHJVMCD (reserved for the use of CICS)	`dfjjvmcd.props`	CICS-supplied system programs have their own JVM profile, DFHJVMCD, to make them independent of any changes 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. DFHJVMCD also specifies REUSE=YES, which gives a continuous JVM, but you can change that as well. Do not specify this profile in PROGRAM resource definitions that you set up for your own applications. However, because DFHJVMCD is used by CICS-supplied system programs, you do need to make sure that it is set up correctly for your CICS region. Only make the changes to DFHJVMCD that are necessary to run applications, as described in “Customizing or creating JVM profiles and JVM properties files” on page 102.

The sample files are defined with JVMPROPS, LIBPATH, CLASSPATH, and WORK_DIR parameters that use the symbols &CICS_DIRECTORY, &JAVA_HOME, and &APPLID. As part of the CICS installation process, you will have 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.

If you are following a procedure to set up IIOP support or support for enterprise beans, and you want to use the default request processor transaction CIRP and the default request processor program DFJIIRP to process requests for CORBA stateless objects or enterprise beans, then you will be using the JVM profile DFHJVMCD. When you have specific CORBA stateless objects or enterprise beans to run, you will need to add classes required by your CORBA stateless objects or enterprise beans to the appropriate class path for DFHJVMCD, as described in “Enabling applications to use a JVM” on page 119. If you do not want to customize this JVM profile at this point, and you are sure that the settings in the profile are suitable for your system, you can return to the procedure “Setting up the host system for IIOP” on page 165 or Chapter 17, “Setting up an EJB server,” on page 227. If you think that you might want to customize DFHJVMCD, or if you want to

select a JVM profile for an alternative request processor program definition that you plan to set up, carry on reading this section.

If you are setting up standard Java programs or your own request processor program definition, in many cases you may find that you can use the sample JVM profiles and JVM properties files with most of the options that are already set in them, and just add your own application classes to the class paths. Simply select the appropriate JVM profile for your application's needs by using the information in Table 5 on page 97. The JVM profile that you select references the relevant sample JVM properties file. The two main questions to ask are:

1. Whether you want the JVM to use the shared class cache (DFHJVMPC) or to run independently as a standalone JVM (DFHJVMPR). "The shared class cache" on page 89 explains the advantages for JVMs in using the shared class cache, and what the implications are. Single-use JVMs and JVMs that are configured for debug cannot use the shared class cache. Because DFHJVMPR (where JVMs do not use the shared class cache) is the default, if you **do** want JVMs to use the shared class cache, ensure that you specify DFHJVMPC as the JVM profile for those JVMs.
2. Whether you want CICS to attempt to reset the JVM after it finishes running each Java program (a resettable JVM), or to make it available for reuse without resetting it (a continuous JVM), or to destroy the JVM without attempting to reset it (a single-use JVM)."Setting a level of reusability" explains how this can be specified.

In some cases, you might find that the options in the sample JVM profiles and JVM properties files need to be changed to fit the needs of a particular application, or of your CICS region. "Changes that you could make" on page 100 has information about the circumstances in which you might want to make these changes.

Setting a level of reusability

The level of reusability for a JVM is specified by the REUSE option in the JVM profile.

The levels of reusability for a JVM are:

- Continuous (option REUSE=YES)
- Resettable (option REUSE=RESET)
- Single-use (option REUSE=NO)

"How JVMs are reused" on page 85 explains the three levels of reusability, the situations for which each level of reusability is appropriate, and the relative performance of each level of reusability.

JVMs that use the shared class cache, known as worker JVMs, inherit their level of reusability from the REUSE option that you specify in the JVM profile for the master JVM. If you include the REUSE option in the profile for a worker JVM, the option is ignored. "Defining the shared class cache" on page 107 explains what to consider when choosing a level of reusability for the master and worker JVMs.

For standalone JVMs that do not use the shared class cache, the REUSE option in the JVM profile determines the level of reusability. REUSE=RESET, which produces a resettable JVM, is the default if no REUSE option is specified.

The older option `Xresettable` is also accepted for migration purposes. If this option is present in the JVM profile and specified as `Xresettable=YES`, the JVM is resettable. If `Xresettable=NO` is specified, the JVM is single-use. The `Xresettable` option cannot be used to specify a continuous JVM. If the `Xresettable` option and

the REUSE option are both present in the JVM profile and they conflict, the REUSE option overrides the `Xresettable` option, and an information message is issued. It is advisable to remove the `Xresettable` option if both the options are present.

The CICS-supplied sample JVM profiles have the following levels of reusability:

- DFHJVMPR specifies REUSE=RESET, which gives a resettable JVM, but you can change this to REUSE=YES to make a continuous JVM. “Programming for different types of JVM” on page 120 explains the considerations for application design and development for Java programs that will run in each type of JVM. Bear in mind that DFHJVMPR is the default if no JVM profile is specified in a PROGRAM resource definition.
- DFHJVMPS is a profile for a single-use JVM, specifying REUSE=NO. Its use 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 continuous JVM or a resettable JVM. You should not attempt to convert any of the other CICS-supplied sample JVM profiles for this purpose.
- DFHJVMCC, the default profile for the master JVM that initializes the shared class cache, specifies REUSE=RESET. This means that all the worker JVMs are resettable. You can change this to REUSE=YES to make all the worker JVMs continuous. (Note that you cannot mix resettable worker JVMs and continuous worker JVMs in a CICS region.)
- DFHJVMPC does not contain a REUSE option, because worker JVMs inherit their level of reusability from the master JVM, so you can change this by changing the setting in DFHJVMCC.
- DFHJVMCD specifies REUSE=YES, which gives a continuous JVM.

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.)

Changes that you could make

In some cases, you might find that the options in the sample JVM profiles and JVM properties files need to be changed to fit the needs of a particular application, or of your CICS region.

“JVM profiles (JVMPROFILE attribute)” on page 73 gives an overview of the options that are available for you to change in the JVM profiles and JVM properties files. Note that if any changes are required to fit with the setup of your CICS region (for example, if you are required to enable Java 2 security), you need to make the same changes to the supplied sample JVM profiles DFHJVMPR and DFHJVMCD and their associated JVM properties files. 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). Both these JVM profiles therefore need to be configured so that they can be used in your CICS region.

Among other things, you might want to make the following changes:

- Enable Java 2 security for the JVM. The Java 2 security policy mechanism protects Java applications running in a JVM, and particularly enterprise beans,

from performing a potentially unsafe action. You can enable Java 2 security by changing the JVM properties file to name a security manager (using the **java.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 **java.security.policy** system property). The CICS-supplied sample JVM properties files **do not** enable Java 2 security. "Protecting Java applications in CICS by using the Java 2 security policy mechanism" on page 329 tells you what changes you need to make to the sample JVM properties files to enable Java 2 security, 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.

- Change the amount of storage available for the application's use, by changing the size of the middleware and transient storage heaps in the JVM (using the **Xmx=** option in the JVM profile). The value specified in the supplied sample JVM profiles is usually 32M, which should be adequate for most purposes. If you have large Java applications, you might want to increase this value. The *CICS Performance Guide* has more information about the storage-related JVM options, and how to determine suitable values for them.
- Change the destination for messages from JVM internals and for output from Java applications running in the JVM (using the **USEROUTPUTCLASS=** option in the JVM profile). "Redirecting JVM output" on page 135 tells you more about this option.
- Change your work directory (using the **WORK_DIR=** option in the JVM profile). This HFS directory is used for the `stdin`, `stdout` and `stderr` files for JVMs. The default is the user directory of the CICS region user ID. If you are not using the **USEROUTPUTCLASS=** option to redirect the output from your JVMs elsewhere, you might want to change the work directory to a location that is more convenient for you.
- Set up the JDBC drivers supplied by DB2, and also the DataSource interface, so that your Java applications can access DB2 data. "Using JDBC and SQLJ to access DB2 data from Java programs and enterprise beans written for CICS" in the *CICS DB2 Guide* explains how you can do this.You need to use various options in the JVM profile and JVM properties file, which are described in that topic.
- Enable or disable assertion checking at runtime. An assertion is a statement in the Java programming language that enables you to test your assumptions about your program. Using the **ENABLEASSERTIONS**, **DISABLEASSERTIONS** and **SYSTEMASSERTIONS** options in the JVM profile, you can enable or disable assertion checking for system classes, all application classes, a package, or an individual class. You can find more information about programming with assertions at http://java.sun.com/j2se/1.4.2/docs/guide/lang/assert.dital.
- 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 **com.ibm.cics.ejs.nameserver** system property), and further information that is necessary if you are using an LDAP name server. The procedures described in Chapter 14, "Configuring CICS for IIOP," on page 165 tell you how to do this.

Note: In some previous versions of CICS, you could use the `-Xquickstart` option (specified using the `Xservice` option) in a JVM profile to reduce the startup time for the JVM. However, with improvements in JVM technology, the `-Xquickstart` option is now permanently enabled, and specifying `-Xquickstart` in a JVM profile has no effect.

For further information, the *CICS System Definition Guide* has the full lists of options that you can specify using JVM profiles and JVM properties files, and "The sample JVM profiles and JVM properties files" in the *CICS System Definition Guide* gives the full text of the CICS-supplied sample files.

If you want to change any of the options in the JVM profiles or JVM properties files, you can either customize the CICS-supplied sample files, or create your own JVM profiles or JVM properties files. "Customizing or creating JVM profiles and JVM properties files" tells you how to do this.

If you do not want to change any of the options specified in the JVM profiles or JVM properties files, and you have specific applications (standard Java programs, CORBA stateless objects or enterprise beans) to run, "Enabling applications to use a JVM" on page 119 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 return to the procedure "Setting up the host system for IIOP" on page 165 or Chapter 17, "Setting up an EJB server," on page 227.

Customizing or creating JVM profiles and JVM properties files

The DFHIJVMJ job places the sample JVM profiles in the HFS directory
`/usr/lpp/cicsts/cicsts31/JVMProfiles`

where `cicsts31` is your chosen value for the CICS_DIRECTORY variable used by the DFHIJVMJ job during CICS installation.

The sample JVM properties files are in the HFS directory
`/usr/lpp/cicsts/cicsts31/props`

If you want to change any of the options specified in the JVM profiles or JVM properties files, you can either customize the CICS-supplied sample files, or create your own JVM profiles or JVM properties files. "JVM profiles (JVMPROFILE attribute)" on page 73 gives an overview of the options that are available for you to change in the JVM profiles and JVM properties files.

Security caution:

1. You should ensure that the JVM properties files are secure, with update authority restricted to system administrators. This is because the JVM properties files are typically used to define sensitive JVM configuration options, such as the security policy file and the trusted middleware class path.
2. In particular, if you specify that a secure LDAP server is to be used, by coding `java.naming.security.authentication` in the JVM properties files, you also need to specify `java.naming.security.principal` and `java.naming.security.credentials`. These properties hold the UserID and password that CICS requires to access the secure LDAP service, so you need to 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.

The full list of options that you can specify in JVM profiles and JVM properties files, and their possible values, are documented in the *CICS System Definition Guide*.

Also, if you want to enable Java 2 security, “Protecting Java applications in CICS by using the Java 2 security policy mechanism” on page 329 tells you what options you need to specify to achieve this. Some options in JVM profiles and JVM properties files are ignored for JVMs that share the class cache (those with CLASSCACHE=YES in their JVM profile, known as worker JVMs), or for the master JVM that initializes the shared class cache, and some options might or might not be relevant depending on whether the JVM is a resettable JVM, a continuous JVM, or a single-use JVM (REUSE=RESET, YES or NO). The information in the full list of options tells you where these exclusions apply.

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. Normally, a JVM profile provides sufficient flexibility to configure a JVM as required. If you find that you need to make unusual modifications, the *CICS Customization Guide* has more information about using DFHJVMAT. Resettable and continuous JVMs are more economical than single-use JVMs, so it is generally best 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) and its CICS-key equivalent DFJIIRQ, to make them independent of any changes you make to the default JVM profile DFHJVMPR. DFHJVMCD has an associated JVM properties file, `dfjjvmcd.props`. Do not specify this profile in PROGRAM resource definitions that you set up for your own applications.

You need to make sure that DFHJVMCD is set up correctly for your CICS region, but you should customize it only where necessary. Only make the changes to DFHJVMCD and `dfjjvmcd.props` that are necessary to run applications, or that are required for the correct operation of these JVMs in your system. “Options in JVM profiles” and “System properties for JVMs” in the *CICS System Definition Guide* tell you the circumstances in which these changes might be necessary. The comments in the HFS file DFHJVMCD tell you which options can and cannot be changed, and `dfjjvmcd.props` includes only those system properties which you might need to change. Follow the instructions in “Customizing the supplied sample JVM profiles and JVM properties files” to make these changes. Do not make any other changes to DFHJVMCD and `dfjjvmcd.props`.

Customizing the supplied sample JVM profiles and JVM properties files

Follow this procedure if you want to keep the existing name for the JVM profile or JVM properties file that you are customizing. When you keep the existing name for the file, applications that are already set up to use that JVM profile or JVM properties file will use your customized file right away. If you want to change the name of the file, follow the procedure in “Creating your own JVM profiles and JVM properties files” on page 105; if you do this, applications will not use your new JVM profile or JVM properties file unless you make further changes to inform the applications of the new file name. If you are customizing DFHJVMPR, bear in mind that DFHJVMPR is the default if no JVM profile is specified in a PROGRAM resource definition, and it is used by sample programs. Either make sure that all your Java programs which specify DFHJVMPR, or no JVM profile, in their

PROGRAM resource definitions are suited to the changes that you are making, or copy DFHJVMPR and change its name before carrying out any customization.

To customize the supplied sample files, keeping the file names the same, follow this procedure:

1. Open the JVM profile or JVM properties file in a standard text editor, and change the options that you want to change, using the lists of options in the *CICS System Definition Guide* 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 the *CICS System Definition Guide*.
2. If you want to enable Java 2 security, you need to specify some options in the JVM properties file, 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 2 security policy mechanism" on page 329 tells you what options you need to specify in the JVM properties file, 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. For JVM profiles, store the customized JVM profile in the HFS directory that is specified by the JVMPROFILEDIR system initialization parameter. CICS loads the JVM profiles from this directory. "Enabling CICS to locate the JVM profiles and JVM properties files" on page 94 explains how to identify and change this directory. If this directory is set to be the directory containing the supplied sample JVM profiles, you can simply store your customized profiles in the samples directory, replacing the supplied samples. (If you do this, keep a copy of the original supplied sample JVM profiles in another folder for future reference.) Ensure that CICS has read and execute access on HFS for your JVM profile and the directory containing it. "Giving CICS regions permission to access HFS directories and files" on page 56 tells you how to do this.
4. For JVM properties files, it is simplest to store the customized JVM properties file in the HFS directory `/usr/lpp/cicsts/cicsts31/props`, where the supplied sample JVM properties files were placed at installation. (Keep a copy of the original supplied sample JVM properties files in another folder.) You need to specify the full path name for the JVM properties file, using the JVMPROPS option, in all the JVM profiles that reference that JVM properties file. For example, a JVM profile that states `JVMPROPS=/usr/lpp/cicsts/cicsts31/props/dfjjvmpr.props` references the JVM properties file `dfjjvmpr.props` in the directory that contains the supplied sample JVM properties files. If you place the customized JVM properties file back in its original directory, the correct path name will already be specified in the JVM profiles. If you prefer to store your customized JVM properties file in a different directory, change the JVMPROPS option on all the relevant JVM profiles to state the new path name for the file. Also ensure that CICS has read and execute access on HFS for your JVM properties file and the directory containing it. "Giving CICS regions permission to access HFS directories and files" on page 56 tells you how to do this.

Now that you have customized the JVM profiles or JVM properties files, if you have specific applications (standard Java programs, CORBA stateless objects or enterprise beans) to run, "Enabling applications to use a JVM" on page 119 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 return to the procedure "Setting up the host system for IIOP" on page 165 or Chapter 17, "Setting up an EJB server," on page 227.

Creating your own JVM profiles and JVM properties files

Follow this procedure if you want to create a JVM profile or JVM properties file with a different name to the supplied sample files. When you create a file with a new name:

- For JVM profiles, you will need to specify the profile name in the PROGRAM resource definition for any applications that you want to use your new JVM profile.
- For JVM properties files, you will need to specify the file name in any JVM profiles that you want to reference your new JVM properties file.

To minimize administration, if you want to set up JVM profiles and JVM properties files 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 the supplied sample JVM profiles and JVM properties files" on page 103. However, if you want to set up a JVM profile or JVM properties file 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.

To create your own JVM profiles and JVM properties files, follow this procedure:

1. Base your JVM profile or JVM properties file on one of the supplied sample JVM profiles or JVM properties files. "Choosing a JVM profile and JVM properties file" on page 96 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 execute, do not base it on DFHJVMPS.
2. Create the JVM profile or JVM properties file in a standard text editor, using the lists of options in the *CICS System Definition Guide* 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 the *CICS System Definition Guide*.
3. If you want to enable Java 2 security, you need to include some system properties in the JVM properties file, 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 2 security policy mechanism" on page 329 tells you what system properties you need to include in the JVM properties file, 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 or JVM properties file a suitable name. 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, choose either the name of the JVM profile that references it, or another short name.

   The name of a JVM profile or JVM properties file can include the following characters:

   `A-Z a-z 0-9 @ # . - _ % & ¢ ? ! : v " = , ; < >`

   When creating your own JVM profile or JVM properties file, do not give it a name beginning with DFH, because these characters are reserved for use by CICS.

   As JVM profiles and JVM properties files are HFS files, case is important. Remember that when you use the name of a JVM profile or JVM properties file anywhere in CICS, you need to enter it using the same combination of upper and lower case characters that is present in the HFS file name. Although the CEDA panels accept mixed case input for a JVM profile name irrespective of

your terminal's UCTRAN setting, this does not apply when the name of a JVM profile is entered on the CEDA command line, or in another CICS transaction such as CEMT or CECI. Bear this in mind when choosing a name for your JVM profile or JVM properties file.

5. For JVM profiles:
   a. Store your JVM profile in the HFS directory that is specified by the JVMPROFILEDIR system initialization parameter. CICS loads the JVM profiles from this directory. “Enabling CICS to locate the JVM profiles and JVM properties files” on page 94 explains how to identify and change this directory. Ensure that CICS has read and execute access on HFS for your JVM profile and the directory containing it. “Giving CICS regions permission to access HFS directories and files” on page 56 tells you how to do this.
   b. 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. (“Enabling applications to use a JVM” on page 119 tells you more about doing this.) 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. However you specify the JVM profile, ensure that you use the same combination of upper and lower case characters that is present in the HFS file name of the JVM profile.
6. For JVM properties files:
   a. Store your JVM properties file in any HFS directory. Ensure that CICS has read and execute access on HFS for your JVM properties file and the directory containing it. “Giving CICS regions permission to access HFS directories and files” on page 56 tells you how to do this.
   b. 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=/usr/lpp/cicsts/cicsts31/myprops/myjvm.props` references the JVM properties file `myjvm.props`, in the directory `/usr/lpp/cicsts/cicsts31/myprops`. Ensure that you use the same combination of upper and lower case characters that is present in the HFS file name of the JVM properties file.

Now that you have created your own JVM profiles or JVM properties files, if you have specific applications (standard Java programs, CORBA stateless objects or enterprise beans) to run, “Enabling applications to use a JVM” on page 119 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 return to the procedure “Setting up the host system for IIOP” on page 165 or Chapter 17, “Setting up an EJB server,” on page 227.

Setting up the shared class cache

“The shared class cache” on page 89 explains how the shared class cache works, and how JVMs benefit from using it.

CICS supports one active shared class cache in each region. This enables you to support the majority of the JVMs in each region. Some of the JVMs in the region might not be suited to sharing the class cache, because they have an inappropriate level of reusability, or because they are debug JVMs used for problem diagnosis. These JVMs can still run as standalone JVMs, and have their own cache of classes in their storage heaps.

Before setting up the shared class cache, you need to check the options for semaphores that you have set in the BPXPRMxx members of SYS1.PARMLIB. The master JVM that initializes the shared class cache uses a single semaphore ID, and requests a set of 32 semaphores, so you need to:

- Ensure that the MNIDS value is enough for the maximum number of semaphore IDs that are in use at one time, including the shared class cache. Depending on the frequency with which you expect to reload the shared class cache, you might want to allow two or possibly three semaphore IDs for the shared class cache. One semaphore ID would be used by the master JVM that controls the active shared class cache, and the remainder would be used by a master JVM that controls a shared class cache that is being phased out, or by a new master JVM that controls a shared class cache that is being loaded. It is unlikely that you would need more than two semaphore IDs for the shared class cache, except in a CICS region that is being heavily used for development and testing. ("Managing your JVM pool for performance" in the *CICS Performance Guide* has more information about the usage that could be expected in a production system or in a development system.) If you need to change the MNIDS value, you can do this by using the IPCSEMNIDS parameter that is in the BPXPRMxx members of SYS1.PARMLIB.
- Ensure that the MNSEMS value is enough for the maximum number of semaphores that the master JVM requests in a semaphore set—the value must be 32 or greater. If you need to change the MNIDS value, you can do this by using the IPCSEMNSEMS parameter that is in the BPXPRMxx members of SYS1.PARMLIB.

See *z/OS UNIX System Services Planning*, GA22-7800, in the topic "Customizing the BPXPRMxx parmlib members", and *z/OS MVS Initialization and Tuning Reference*, SA22-7592, in the topic "BPXPRMxx (z/OS UNIX System Services parameters)", for more information about adjusting these parameters. The *CICS Transaction Server for z/OS Installation Guide* has information about other parameters in the BPXPRMxx members of SYS1.PARMLIB that need to be changed to use JVMs in a CICS environment.

Now that you have set up a shared class cache in your CICS region, "Managing the shared class cache" on page 110 tells you how to manage it.

Defining the shared class cache

Use the JVMCCSIZE system initialization parameter to specify the initial size of the shared class cache. The size of the shared class cache determines the number of classes that it can contain. The default size is 24MB. You can change the size of the shared class cache while CICS is running; "Adjusting the size of the shared class cache" on page 112 tells you how.

Besides JVMCCSIZE, the shared class cache is mainly defined through the JVM profile that is used for the master JVM that initializes the shared class cache.

The JVM profile for a master JVM is similar to the JVM profile for any other JVM. The CLASSCACHE_MSGLOG option can be specified to name the file for messages from the master JVM (the default is `dfhjvmccmsg.log`). Some options (for example, the `Xdebug` option) are not appropriate for a master JVM, and if they are specified in the JVM profile that is used for the master JVM, CICS ignores them. The *CICS System Definition Guide* has information about the options that are not appropriate in the JVM profile for a master JVM. As for any other JVM profile, you need to ensure that the settings in the profile are suitable for your system.

The JVM properties file for a master JVM omits most of the system properties that would be specified for a normal JVM, because the master JVM is not used to run Java applications. The only system property that needs to be specified is `ibm.jvm.shareable.application.class.path`, which you should use to specify the shareable application classes for all the applications that will run in worker JVMs that use the shared class cache. The *CICS System Definition Guide* has more information about other system properties that you might also want to specify in the JVM properties file for a master JVM.

One important decision to make about the master JVM is whether to define it as a resettable JVM, or as a continuous JVM. (It cannot be defined as a single-use JVM.) "How JVMs are reused" on page 85 explains the levels of reusability for JVMs.

The worker JVMs in a CICS region all inherit their level of reusability from the REUSE option specified in the JVM profile for the master JVM in that region. (If you include the REUSE option in the profile for a worker JVM, the option is ignored.) If you specify the option REUSE=RESET or the older option `Xresettable=YES` in the JVM profile for the master JVM, the master JVM and all the worker JVMs are resettable. If you specify the option REUSE=YES in the JVM profile for the master JVM, the master JVM and all the worker JVMs are continuous. If none of these options is included, CICS assumes that the master JVM is resettable.

If your worker JVMs are continuous JVMs, they have a greater transaction throughput and lower CPU usage than if they are resettable JVMs. If you choose to make your master JVM and worker JVMs into continuous JVMs, you need to note the considerations for programming and for application design which are described in "Programming for different types of JVM" on page 120.

You cannot mix resettable worker JVMs and continuous worker JVMs in a CICS region; you need to choose one level of reusability for your worker JVMs. If you have some applications that need to run in a resettable JVM and some that need to run in a continuous JVM, and you want both types to use the shared class cache, then you could set up a master JVM and worker JVMs with either level of reusability in separate CICS regions. If you require both resettable and continuous JVMs in a single CICS region that has a shared class cache, you need to choose which type should be able to use the shared class cache, and which type should be standalone. Single-use JVMs are always standalone JVMs.

By default, the supplied sample JVM profile DFHJVMCC is used for the master JVM that initializes the shared class cache. DFHJVMCC specifies the option REUSE=RESET, so the master JVM and worker JVMs are resettable. You can modify DFHJVMCC to change this setting or other settings in the JVM profile, or you can substitute your own JVM profile. "Customizing or creating JVM profiles and JVM properties files" on page 102 tells you how to change a JVM profile or create your own.

If you modify DFHJVMCC, CICS uses the new version of the JVM profile for the master JVM the next time the shared class cache is started. If the shared class cache is already started, and you want to switch to the new version of the JVM profile right away, use the CEMT PERFORM CLASSCACHE RELOAD command (or the equivalent EXEC CICS command) to create a new shared class cache. The master JVM that initializes the new shared class cache will use the new version of the JVM profile.

If you create your own JVM profile to use instead of DFHJVMCC, there are two ways that you can specify a different JVM profile to be used for the master JVM:

1. Name the JVM profile you want to use on the JVMCCPROFILE system initialization parameter. Using the system initialization parameter ensures that this JVM profile is used for the master JVM following an initial or cold start of CICS. The supplied sample JVM profile DFHJVMCC is the default value for this system initialization parameter.
2. Use either the CEMT PERFORM CLASSCACHE RELOAD command (or the equivalent EXEC CICS command) if the shared class cache is started, or the CEMT PERFORM CLASSCACHE START command (or the equivalent EXEC CICS command) if the shared class cache is stopped, to create a new shared class cache. Use the PROFILE option on the command to specify the JVM profile to be used for the master JVM that initializes the new shared class cache. The new JVM profile that you specified is then used for each subsequent initialization of the shared class cache. The new setting is remembered across a warm or emergency start, unless the JVMCCPROFILE system initialization parameter is specified as an override at startup, in which case the value from the JVMCCPROFILE system initialization parameter is used. On an initial or cold start of CICS, CICS uses the JVM profile named on the JVMCCPROFILE system initialization parameter.

Remember that when you specify the JVM profile, whether by using JVMCCPROFILE, or by using a CEMT PERFORM CLASSCACHE START or RELOAD command, or by using the equivalent EXEC CICS commands, you must enter it using the same combination of upper and lower case characters that is present in the HFS file name. If you use the CEMT transaction, and the name of the JVM profile is in mixed case or lower case, ensure that the terminal you use is correctly configured, with upper case translation suppressed. If you use an EXEC CICS command, the value is always accepted in mixed case.

Use the CEMT INQUIRE CLASSCACHE command (or the equivalent EXEC CICS command) to find out what JVM profile currently applies to the master JVM that initializes the shared class cache.

Enabling JVMs to use the shared class cache

To enable a JVM to be a worker JVM and use the shared class cache, you need to ensure that the JVM profile used by that JVM states **CLASSCACHE=YES**. If any JVMs in the region cannot share the class cache, because they have an inappropriate level of reusability or because they are being used for problem diagnosis, they need to use a JVM profile that states **CLASSCACHE=NO**. The EXEC CICS INQUIRE JVMPROFILE command tells you whether a particular JVM profile states CLASSCACHE=YES or CLASSCACHE=NO. (There is no CEMT equivalent for this command.) “Setting up JVM profiles and JVM properties files” on page 94 tells you how to select and modify JVM profiles. “Enabling applications to use a JVM” on page 119 tells you how to specify the JVM profile that an application requests.

The default JVM profile, DFHJVMPR, states **CLASSCACHE=NO**. If you use this JVM profile, the JVM does not use the shared class cache, and runs independently as a standalone JVM.

The supplied sample JVM profile DFHJVMPC states **CLASSCACHE=YES**. It is compatible with the shared class cache that is created by a master JVM with the JVM profile DFHJVMCC. If you want a JVM to use the shared class cache, you can name DFHJVMPC as the JVM profile on the PROGRAM resource definition for the

request, or you can name your own JVM profile based on DFHJVMPC that states CLASSCACHE=YES. JVMs created using JVM profiles that state CLASSCACHE=YES are known as worker JVMs.

The JVM profile for CICS-supplied system programs, DFHJVMCD, states **CLASSCACHE=NO**. This JVM profile is specified by the PROGRAM resource definition for the default request processor program, DFJIIRP. Enterprise bean requests that invoke the CICS-supplied CIRP request processor transaction will therefore use this JVM profile. If you want these enterprise beans to use the shared class cache, change DFHJVMCD to state CLASSCACHE=YES.

If you specify CLASSCACHE=YES in a JVM profile, certain options in the JVM profile and JVM properties file are ignored. If these options are found in the JVM profile or JVM properties file for a worker JVM, CICS does not pass them on to the JVM. If values for these options are required, they are taken from the JVM profile and JVM properties file for the master JVM that initializes the shared class cache. These options are not used in the CICS-supplied sample JVM profile DFHJVMPC. If you have converted another JVM profile to use the shared class cache, you can either remove the options (by commenting out or deletion) from the JVM profile or JVM properties file, or leave them there. The *CICS System Definition Guide* tells you what options and system properties are treated in this way.

The options that are ignored for a worker JVM include the REUSE option, which specifies the level of reusability for the JVM. “How JVMs are reused” on page 85 explains the levels of reusability for JVMs. Worker JVMs inherit their level of reusability from the master JVM, so they do not need the REUSE option in their JVM profiles. The master JVM and worker JVMs in a CICS region can be resettable or continuous, depending on the REUSE setting in the master JVM profile. If you choose to make your master JVM and worker JVMs into continuous JVMs, you need to note the considerations for programming and for application design which are described in “Continuous JVMs (REUSE=YES)” on page 86.

Also among the options that are ignored for a worker JVM are the options that specify the library path (in the JVM profile), the trusted middleware class path (in the JVM profile), and the shareable application class path (in the JVM properties file). These class paths are taken from the values for the master JVM. If you have converted an existing JVM profile to use the shared class cache, you need to ensure that directory paths specified by these options in the JVM profile or its JVM properties file are transferred to the library path, the middleware class path or the shareable application class path for the master JVM. “Adding application classes to the class paths for a JVM” on page 128 tells you how to do this.

Managing the shared class cache

“The shared class cache” on page 89 explains how the shared class cache works, and how resettable JVMs benefit from using it. “Setting up the shared class cache” on page 106 tells you how to set up a shared class cache.

Once you have set up a shared class cache in your CICS region, you might need to perform the following management tasks:

Starting the shared class cache

You can start the shared class cache in three ways:

1. You can **set the shared class cache to start at CICS initialization**, by setting the JVMCCSTART system initialization parameter to YES. (This setting also enables autostart.)
2. You can **enable autostart for the shared class cache**. If you enable autostart, the shared class cache is started 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. When you first use CICS, autostart is enabled by default. If you have disabled autostart, there are three ways to enable it again:
   a. To enable autostart for the next CICS execution, set the JVMCCSTART system initialization parameter to either YES, or AUTO. YES makes the shared class cache start at CICS initialization, and also enables autostart. AUTO just enables autostart, and does not make the shared class cache start at CICS initialization.
   b. When CICS is running, use the CEMT SET CLASSCACHE AUTOSTARTST command (or the equivalent EXEC CICS command) to enable autostart.
   c. If you are terminating (phasing out, purging or forcepurging) the shared class cache, use the AUTOSTARTST option on the CEMT PERFORM CLASSCACHE command (or the equivalent EXEC CICS command) to enable autostart.

   Use the CEMT INQUIRE CLASSCACHE command (or the equivalent EXEC CICS command) to find out the current status of autostart for the shared class cache. 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 CEMT SET CLASSCACHE command or the CEMT PERFORM CLASSCACHE command (or the equivalent EXEC CICS commands), 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.
3. When CICS is running, if you have disabled autostart, or if autostart is enabled but no JVM has yet required the shared class cache, you can **start the shared class cache manually** by entering a CEMT PERFORM CLASSCACHE START command (or the equivalent EXEC CICS command). When you use this command, you can specify the size of the shared class cache (CACHESIZE option), and the profile that is used for the master JVM that initializes the shared class cache (PROFILE option).

If your CICS system is WARM or EMERGENCY started, and the shared class cache was started when the system shut down, then it is started at CICS initialization. This happens whatever the autostart status is, unless the JVMCCSTART system initialization parameter is specified as an override at startup, in which case the behaviour specified by the system initialization parameter is used.

When you start the shared class cache by an explicit command or by the autostart feature, there is a short delay while the master JVM initializes the shared class cache. If CICS receives requests during this period for worker JVMs that require the use of the shared class cache, the requests wait until the startup process is complete and the shared class cache is ready.

When you have decided how and when you normally prefer the shared class cache to start in your CICS system, use the JVMCCSTART system initialization parameter to set the normal behaviour for the shared class cache. The CICS default setting for the JVMCCSTART system initialization parameter is AUTO. This setting means that

when you start CICS, the shared class cache does not start up immediately. However, autostart is enabled, so the shared class cache starts automatically when the first worker JVM requires it. This is the situation when you first use CICS.

If you prefer the shared class cache to start up during CICS initialization, so that it is ready and waiting when the first worker JVM requires it, set the JVMCCSTART system initialization parameter to YES. This setting enables autostart, so if you stop the shared class cache, it starts again automatically when the first worker JVM requires it.

If you prefer the shared class cache not to start at all until you enter a specific command, set the JVMCCSTART system initialization parameter to NO. This setting disables autostart, so when you want to start the shared class cache, you need either to enter a CEMT PERFORM CLASSCACHE START command (or the equivalent EXEC CICS command), or to enable autostart by one of the methods described above. If a Java program needs to run in a JVM that uses the shared class cache, and the shared class cache has not been started, and autostart is disabled, then the program cannot run.

Adjusting the size of the shared class cache

When the master JVM initializes the shared class cache, the amount of storage in the cache is fixed. When the storage in the shared class cache becomes full, worker JVMs cannot add any more classes to it, and the JIT compiler might not be able to carry out any further processing of the classes that are already loaded. You can use the CEMT INQUIRE CLASSCACHE command (or the equivalent EXEC CICS command) to report on the size of the shared class cache (CACHESIZE), and the amount of free space within it (CACHEFREE, which is part of the extended display for the shared class cache in CEMT).

Use the system initialization parameter JVMCCSIZE to specify the initial setting for the size of the shared class cache. The default is 24MB.

The size that you specify for the shared class cache needs to be sufficient to contain:

- The classes for your applications, except for any classes on the standard class path (the CLASSPATH option in the JVM profile), because those classes are not placed in the shared class cache.
- The JIT-compiled code for the classes.

Bear in mind that the JIT-compiling process happens at variable times during the execution of your applications. To identify the total size of the classes and JIT-compiled code for an application, run the application repeatedly in a development environment, using the shared class cache. While you are running the application, monitor the amount of free space in the shared class cache, which is reported by the CACHEFREE parameter on the CEMT INQUIRE CLASSCACHE command (or the equivalent EXEC CICS command). After a sufficient number of uses of the application (perhaps around one thousand), you should find that the amount of free space has stabilised, meaning that the JIT-compiling process is largely complete. Repeat this process for each application that will be using the shared class cache. Add together the amount of storage used by each application, and add on a suitable safety margin (to allow for any late JIT-compiling or future application changes), to arrive at an approximate size for the shared class cache.

If the storage in your shared class cache becomes full, worker JVMs can continue to use the classes and compiled code that are already present in it. However, if a

worker JVM subsequently tries to add a new class or the results of JIT-compilation to the shared class cache, the worker JVM throws a java.lang.OutOfMemoryError. If you find that this is happening, you need to increase the size of the shared class as soon as possible.

If you have made the initial size of the shared class cache too small (or too large), there are two ways to change it without changing the system initialization parameter:

1. When the status of the shared class cache is STARTED, you can use the CEMT PERFORM CLASSCACHE RELOAD command (or the equivalent EXEC CICS command) to create a new shared class cache. The RELOAD option does not work if the shared class cache has not been started. Specify the size for the new shared class cache by using the CACHESIZE option on the command. This causes the least disruption to worker JVMs that are using the shared class cache.
2. When the status of the shared class cache is STOPPED (either because it has not yet been started on this CICS execution, or because you have stopped it manually), you can use the CEMT PERFORM CLASSCACHE START command (or the equivalent EXEC CICS command) to start the shared class cache. Specify the size for the new shared class cache by using the CACHESIZE option on the command. If you do not want the shared class cache to remain active, you can then shut it down again (see "Terminating the shared class cache" on page 116).

When you specify a new size for the shared class cache while CICS is running, subsequent CICS restarts use the new value, unless the system is INITIAL or COLD started, or the JVMCCSIZE system initialization parameter is specified as an override at startup. In these cases, the value from the JVMCCSIZE system initialization parameter is used.

Updating classes or JAR files in the shared class cache

The shared class cache contains:

- The IBM-supplied middleware that you need to run enterprise beans and Java applications, and any other middleware classes that you have specified (on the trusted middleware class path).
- Any application classes that are loaded by shared application class loaders, including classes on the shareable application class path, and classes that are loaded from a DJAR.

"The structure of a JVM" on page 64 has more details about these classes.

If any of these classes or JAR files change, you need to update them in the shared class cache, because they are not automatically reloaded. This does **not** apply to classes loaded by the nonshareable class loader, that is, classes on the standard class path specified by the CLASSPATH option in the JVM profile, because they are loaded into the individual worker JVMs.

If you need to update any of the classes or JAR files in the shared class cache, first update the classes or files on your system. Next, phase out the old shared class cache and create a new shared class cache. The new shared class cache will contain the new classes or JAR files that you have placed on your system.

You can phase out the old shared class cache using one of three commands, depending on how you want the new shared class cache to be introduced. Once you have entered one of these commands, either CICS will create a new shared

class automatically, or you will need to create a new shared class cache manually. The three commands that you can use are listed here, with a description of what happens when you use each command, and what you need to do next. Read the list to identify the command that is most appropriate for your situation. Table 6 on page 116, following the list, summarizes when you should use each command.

The commands you can use to update classes or JAR files in the shared class cache are:

- **CEMT PERFORM CLASSCACHE RELOAD (or the equivalent EXEC CICS command)**.

  This command creates a new shared class cache using the new classes or JARs. You can only use this command when the status of the shared class cache is STARTED. When you reload the shared class cache, worker JVMs, both those that are already allocated to tasks and those that are allocated to tasks after you issue the command, continue to use the existing shared class cache until the new shared class cache is ready. When the new shared class cache is ready, subsequent requests for worker JVMs are given a worker JVM that uses the new cache. These new worker JVMs are started as they are requested by applications, and they replace the worker JVMs that are using the old shared class cache. The worker JVMs that are using the old shared class cache are allowed to finish running their current Java programs, and then they are terminated. The old shared class cache is deleted when all the worker JVMs that are dependent on it have been terminated.

  CEMT PERFORM CLASSCACHE RELOAD is the least disruptive of the commands listed here, but it does mean that the old versions of the class or JAR files continue to be used until the process is complete. CEMT PERFORM CLASSCACHE RELOAD has no effect on standalone JVMs that are not sharing the class cache.

  When you have entered CEMT PERFORM CLASSCACHE RELOAD, you do not need to take any further action, because CICS automatically creates the new shared class cache as a result of the command.

- **CEMT PERFORM CLASSCACHE PHASEOUT, PURGE or FORCEPURGE (or the equivalent EXEC CICS command)**.

  This command terminates all the worker JVMs that are dependent on the shared class cache, and then deletes the shared class cache itself. You can choose to purge or forcepurge the worker JVMs, or allow them to finish running their current Java programs before they are deleted. New JVMs that start up after you issue the command **cannot** use the shared class cache that is being terminated. This command has no effect on standalone JVMs that are not sharing the class cache.

  When you have entered CEMT PERFORM CLASSCACHE PHASEOUT, PURGE or FORCEPURGE:

  – If autostart is enabled, as soon as a new JVM requests the use of the shared class cache, a new shared class cache is started, and this new shared class cache contains the new versions of the classes or JAR files. There will be a slight delay while the new shared class cache is initialized, during which requests will wait. All subsequent JVMs that require the shared class cache will use the new shared class cache.

  – If autostart is disabled, you need to take action to ensure that a new shared class cache is started. You can use the AUTOSTARTST option on the CEMT PERFORM CLASSCACHE PHASEOUT, PURGE or FORCEPURGE command (or the equivalent EXEC CICS command) to enable autostart, in which case a new shared class cache is created as soon as a new JVM requests the use of the shared class cache. Alternatively, if you want to keep

autostart disabled, you need to start a new shared class cache using the CEMT PERFORM CLASSCACHE START command (or the equivalent EXEC CICS command). You can enter this command while the old shared class cache is being terminated; you do not need to wait for termination to complete. Enter the command as soon as you can, because while the status of the shared class cache is STOPPED, requests to run a Java application in a JVM whose profile requires the use of the shared class cache (that is, requests for worker JVMs) will fail. After you enter the command, making the status of the shared class cache STARTING, the requests will wait. The new shared class cache that you start (whether manually or by enabling autostart) contains the new versions of the classes or JAR files.

After the new shared class cache starts, the old shared class cache remains in the system until all the worker JVMs that are dependent on it have been terminated, and then it is deleted. You can use the CEMT INQUIRE CLASSCACHE command (or the equivalent EXEC CICS command) to report on any old shared class caches in your system, and the number of JVMs that are dependent on them.

- **CEMT SET JVMPOOL PHASEOUT, PURGE or FORCEPURGE (or the equivalent EXEC CICS command)**.

  This command terminates all the JVMs in the JVM pool, both those sharing the class cache and those running independently as standalone JVMs, and it terminates the shared class cache. You can choose to purge or forcepurge the JVMs, or allow them to finish running their current Java programs before they are deleted.

  When you have entered CEMT SET JVMPOOL PHASEOUT, PURGE or FORCEPURGE (or the equivalent EXEC CICS command):

  – For standalone JVMs that do not use the shared class cache, CICS will start these automatically as they are needed. New standalone JVMs that start up after you have issued the command will use the new versions of the classes or JAR files. They can start up right away, and do not need to wait until all the JVMs in the pool have been terminated.

  – For worker JVMs that use the shared class cache:

    - If autostart is enabled, the result is the same as with the CEMT PERFORM CLASSCACHE PHASEOUT, PURGE or FORCEPURGE command. A new shared class cache is started as soon as a new JVM requests it.

    - If autostart is disabled, you need to start a new shared class cache using the CEMT PERFORM CLASSCACHE START command (or the equivalent EXEC CICS command). You can enter this command while the old shared class cache is being terminated; you do not need to wait for termination to complete. When autostart is disabled, and the status of the shared class cache is STOPPED (that is, after you have entered the CEMT SET JVMPOOL PHASEOUT, PURGE or FORCEPURGE command), requests to run a Java application in a JVM whose profile requires the use of the shared class cache (that is, requests for worker JVMs) will fail. After you enter the CEMT PERFORM CLASSCACHE START command, making the status of the shared class cache STARTING, the requests will wait rather than fail. To avoid the possible failure of requests for JVMs, if you are planning to restart the shared class cache, it is advisable to use the CEMT SET CLASSCACHE AUTOSTARTST command (or the equivalent EXEC CICS command) to enable autostart before using the CEMT SET JVMPOOL PHASEOUT, PURGE or FORCEPURGE command. This ensures that a new shared class cache is started as soon as it is needed.

As with the CEMT PERFORM CLASSCACHE PHASEOUT, PURGE or FORCEPURGE command, the old shared class cache remains in the system until all the worker JVMs that are dependent on it have been terminated, and then it is deleted.

Table 6 summarizes when you should use each command to update classes or JAR files in the shared class cache.

Table 6. Updating classes or JARs in the shared class cache

Situation	Suitable command
• You want to allow new JVMs requiring the shared class cache to use the old classes or JARs until the new shared class cache is ready. • You have no standalone JVMs, or you do not want to update this type of JVM.	CEMT PERFORM CLASSCACHE RELOAD (or the equivalent EXEC CICS command)
• You want to ensure that all new JVMs requiring the shared class cache from now on must wait until the new shared class cache is ready, and not use the old classes or JARs. • You have no standalone JVMs, or you do not want to update this type of JVM.	CEMT PERFORM CLASSCACHE PHASEOUT, PURGE or FORCEPURGE (or the equivalent EXEC CICS command), using the AUTOSTARTST option to enable autostart if it is not already enabled
• You want to update the classes or JARs in standalone JVMs, as well as in the shared class cache.	CEMT SET JVMPOOL PHASEOUT, PURGE or FORCEPURGE (or the equivalent EXEC CICS command), followed as soon as possible by CEMT PERFORM CLASSCACHE START (unless autostart is enabled, in which case you do not need to use CEMT PERFORM CLASSCACHE START)

You can use the CEMT INQUIRE CLASSCACHE command (or the equivalent EXEC CICS command) to report on any old shared class caches in your system (OLDCACHES), and the number of JVMs that are dependent on them (PHASINGOUT). If you want to check the status of the JVMs themselves, including standalone JVMs, you can use the CEMT INQUIRE JVM command (or the equivalent EXEC CICS command) to report on all the JVMs in the JVM pool, including those that are being phased out. (The INQUIRE JVM command does not find the master JVM that initializes the shared class cache. It only finds worker JVMs and standalone JVMs.)

Terminating the shared class cache

If you want to terminate the shared class cache without restarting it, use either of the following two commands:

- CEMT PERFORM CLASSCACHE PHASEOUT, PURGE or FORCEPURGE (or the equivalent EXEC CICS command). This command terminates the shared class cache and any worker JVMs that are dependent on it. If the shared class cache is not restarted (either by a command or by the autostart feature), JVMs that need to use the shared class cache cannot run. JVMs that are not using the shared class cache are not affected by this command.
- CEMT SET JVMPOOL PHASEOUT, PURGE or FORCEPURGE (or the equivalent EXEC CICS command). This command terminates all the JVMs in the JVM pool, both those sharing the class cache and those running independently

as standalone JVMs, and terminates the shared class cache. Standalone JVMs can restart as needed, but if the shared class cache is not restarted (either by a command or by the autostart feature), JVMs that need to use the shared class cache cannot run.

With both of these commands, you can choose to purge or forcepurge the JVMs, or allow them to finish running their current Java programs before they are deleted.

When you enter these commands, if autostart is enabled for the shared class cache, a new shared class cache is created as soon as a JVM requests its use. If you want to prevent this—that is, you want to terminate the shared class cache without restarting it—then you need to disable autostart. You can disable autostart in three ways:

1. When you are entering a CEMT PERFORM CLASSCACHE PHASEOUT, PURGE or FORCEPURGE command (or the equivalent EXEC CICS command) to terminate the shared class cache, use the AUTOSTARTST option to disable autostart. (This option is not available if you are using the CEMT SET JVMPOOL PHASEOUT, PURGE or FORCEPURGE command.)
2. Before you enter a command to terminate the shared class cache, use the CEMT SET CLASSCACHE AUTOSTARTST command (or the equivalent EXEC CICS command) to disable autostart.
3. To disable autostart for the next CICS execution, set the JVMCCSTART system initialization parameter to NO.

Remember that if a Java program needs to run in a JVM that uses the shared class cache, and the shared class cache has been terminated and not restarted, and autostart is disabled, then the program cannot run.

You can use the CEMT INQUIRE CLASSCACHE command (or the equivalent EXEC CICS command) to find out the current status of autostart for the shared class cache. 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 CEMT SET CLASSCACHE command or the CEMT PERFORM CLASSCACHE command (or the equivalent EXEC CICS commands), 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.

If you do not want to restart the shared class cache, and the worker JVMs that are dependent on it remain active for too long, you can repeat the CEMT PERFORM CLASSCACHE PURGE or FORCEPURGE command, or the CEMT SET JVMPOOL PURGE or FORCEPURGE command (or the equivalent EXEC CICS commands), to attempt to purge the tasks that are using the JVMs. You should only repeat these commands if autostart for the shared class cache is **disabled**. The commands operate on both the most recent shared class cache, and any old shared class caches in the system that still have JVMs dependent on 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

Messages from the master JVM that initializes the shared class cache are written to the HFS file specified by the CLASSCACHE_MSGLOG option in the JVM profile for the master JVM. The default name for this file is `dfhjvmccmsg.log`.

To report on the status of the shared class cache, use the CEMT INQUIRE CLASSCACHE command (or the equivalent EXEC CICS command). The shared class cache can be in one of four states:

STARTING
The shared class cache is being initialized. If autostart is enabled, the shared class cache is starting either because CICS received a request to run a Java application in a JVM whose profile requires the use of the shared class cache, or because an explicit CEMT PERFORM CLASSCACHE START command (or the equivalent EXEC CICS command) was issued. If autostart is disabled, the shared class cache is starting because an explicit CEMT PERFORM CLASSCACHE START command (or the equivalent EXEC CICS command) was issued. If CICS receives requests during this period for worker JVMs that require the use of the shared class cache, the requests wait until the startup process is complete and the shared class cache is ready. If initialization of the shared class cache is unsuccessful, any waiting requests for worker JVMs fail.

STARTED
The shared class cache is ready, and it can be used by worker JVMs.

RELOADING
A CEMT PERFORM CLASSCACHE RELOAD command (or the equivalent EXEC CICS command) has been issued, and a new shared class cache is being loaded to replace the existing shared class cache. Worker JVMs, both those that were already allocated to tasks and those that were allocated to tasks after the command was issued, continue to use the existing shared class cache until the new shared class cache is ready.

STOPPED
The shared class cache has either not been initialized on this CICS execution, or it has been stopped by a CEMT PERFORM CLASSCACHE PHASEOUT, PURGE or FORCEPURGE or CEMT SET JVMPOOL PHASEOUT, PURGE or FORCEPURGE command (or the equivalent EXEC CICS commands). If autostart is disabled, requests to run a Java application in a JVM whose profile requires the use of the shared class cache (that is, requests for worker JVMs) will fail. If autostart is enabled, a new shared class cache will be initialized 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.

The CEMT INQUIRE CLASSCACHE command (or the equivalent EXEC CICS command) also tells you:

- The status of autostart for the shared class cache (AUTOSTARTST).
- The size of the shared class cache (CACHESIZE) and the amount of free space in it (CACHEFREE).
- The date and time that the current shared class cache was started (STARTTIME for the EXEC CICS command, or DATESTARTED and TIMESTARTED for the CEMT command).
- The JVM profile for the master JVM that initializes the shared class cache (PROFILE).
- The level of reusability for the master JVM, which is inherited by all the worker JVMs in the CICS region (REUSEST).
- Whether there are any old shared class caches in the region that are waiting for worker JVMs that are dependent on them to be phased out (OLDCACHES). If the status of the current shared class cache is STOPPED, then it is included in the number of old shared class caches.

- The number of worker JVMs that are dependent on an old shared class cache, and are being phased out (PHASINGOUT).
- The total number of worker JVMs in the region that are dependent on a shared class cache, old or current (TOTALJVMS).

To report on the status of the JVMs in the JVM pool, use the CEMT INQUIRE JVM command (or the equivalent EXEC CICS command). This 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. The INQUIRE JVM command does not find the master JVM that initializes the shared class cache. It only finds worker JVMs and standalone JVMs.

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.

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. “Choosing a JVM profile and JVM properties file” on page 96 summarizes the considerations you need to take into account, and the changes that you might want to make to the JVM profile.
2. Check the programming considerations that apply to each of the possible levels of reusability for a JVM: resettable JVMs, continuous JVMs, and single-use JVMs. Ensure that your application design takes these into account, and that you have carried out appropriate testing. “Programming for different types of JVM” on page 120 explains the programming considerations.
3. 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 126 tells you how to do this.
4. 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 128 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 111 tells you how to start the shared class cache or enable autostart.

CORBA stateless objects and enterprise beans do not have a PROGRAM resource definition as such. The PROGRAM resource definition that is relevant to CORBA stateless objects and enterprise beans is that for the request processor program. To enable these applications 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 14, "Configuring CICS for IIOP," on page 165 explains how to configure CICS as a CORBA participant, and Chapter 17, "Setting up an EJB server," on page 227 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 shareable application class path, by using the `ibm.jvm.shareable.application.class.path` system property that will be 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 shareable application class path. "Adding application classes to the class paths for a JVM" on page 128 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 shareable application 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 128.

Programming for different types of JVM

The level of reusability for a JVM is controlled by the REUSE option in the JVM profile for the JVM. When developing Java applications, you need to bear in mind the type of JVM in which the application is intended to run.

The levels of reusability for a JVM are:

- Continuous (option REUSE=YES)
- Resettable (option REUSE=RESET)
- Single-use (option REUSE=NO)

"How JVMs are reused" on page 85 explains the three levels of reusability, the situations for which each level of reusability is appropriate, and the relative performance of each level of reusability.

Persistent Reusable Java Virtual Machine User's Guide, SC34-6201, has more detailed information about developing Java applications to run in a JVM.

Programming considerations for continuous JVMs

When you are developing Java applications to run in a continuous JVM, you can create persistent items that might be of use to future executions of the same application in the same JVM, but you also need to ensure that programs do not change the state of the JVM in undesirable ways, or leave any unwanted state in the JVM.

Protecting the state of a continuous JVM

The continuous JVM does not isolate invocations of Java programs from changes made to the JVM by previous invocations of programs in the same JVM. User application classes, as well as middleware 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 (if not prohibited by a security manager), 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. In a resettable JVM, such actions would be considered unresettable actions, and cause the JVM to be destroyed.

Unresettable actions are not recorded in a continuous JVM. To help eliminate unresettable actions where they are not desired, during the development process, you can test the program in a resettable JVM (with the option REUSE=RESET). Resettable JVMs record unresettable actions, and you can use this information to help you re-design the application if necessary. “Programming considerations for resettable JVMs” on page 122 tells you how to make a JVM record unresettable actions. When you know that the program does not change the state of the JVM in undesirable ways, you can move to using it in a continuous JVM.

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 can use this to your advantage in designing your Java applications if you want information to persist from one program invocation to the next. Because static state and object instances referenced through static state are not reset between JVM reuses in a continuous JVM, 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.) This is also possible with a resettable JVM, but would require the development of middleware code, which is a step up in complexity from developing application code.

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.

However, you must 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. Static variables in a continuous JVM can exist through a number of CICS tasks, and their value might not be predictable. 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). 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.

Further programming tips for a continuous JVM:

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 2 security policy mechanism” on page 329 has more information about applying a security policy.

Accessing DB2

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.

Middleware

If you are writing middleware to run in a continuous JVM, note that a continuous JVM does not invoke the `ibmJvmTidyUp` method to request the middleware classes to perform cleanup. The middleware classes must perform any cleanup that is required without being prompted by this request. (The CICS-supplied middleware does perform cleanup without a request from the JVM.) *Persistent Reusable Java Virtual Machine User's Guide*, SC34-6201, explains how middleware should be written.

Programming considerations for resettable JVMs

When you are developing Java applications to run in a resettable JVM, you need to ensure that the programs do not perform unresettable actions or leave cross-heap references in the JVM. In a resettable JVM, if an unresettable event (caused by an unresettable action or by a cross-heap reference still in scope) is recorded, the JVM is destroyed and cannot be reused, so CICS incurs the CPU cost of initializing a new JVM.

Unresettable actions are listed in the document *Persistent Reusable Java Virtual Machine User's Guide*, SC34-6201. You should log the unresettable actions and cross-heap references during the development process in order to identify and eliminate them.

To log unresettable actions and cross-heap references during testing of your Java programs, include the system properties listed below in the JVM properties file for the JVM that you are using. There is also one option to be included in the JVM profile. The *CICS System Definition Guide* has more detailed information about each of these system properties. In particular, note that when you include some of these system properties in the JVM properties file with any value, the function is enabled. To disable the function, you need to comment out or remove the system property; there is no value you can specify for the system property that disables the function.

ibm.jvm.events.output={event.log | *path* | stderr | stdout}
: This system property enables event logging in the JVM, and it must be specified in order to enable the other system properties related to event logging. The CICS-supplied sample JVM properties file `dfjjvmpr.props` specifies the file `event.log`, which is created in the directory defined by the WORK_DIR option in the JVM profile. You can also store the text records describing the events in a HFS file of your choice, or in the stderr or stdout file for the JVM. Bear in mind that the output from multiple JVMs will be interleaved in the file `event.log`, or in a HFS file that you have chosen, or in the stderr or stdout files if only one file is used. If you are only obtaining output from a single JVM, you can specify a single file for this system property. If you are obtaining output from multiple JVMs, you should specify `stderr` or `stdout`, and also ensure that the `-generate` option is used for the STDERR or STDOUT option in the JVM profile, to generate a separate output file for each JVM. Alternatively, if you specify `stderr` or `stdout`, you can use the USEROUTPUTCLASS option in the JVM profile to redirect the output to another destination of your choice and add headers to it (see “Redirecting JVM output” on page 135).

ibm.jvm.unresettable.events.level={max | min}
: This system property specifically enables the logging of unresettable events (caused by an unresettable action or by a cross-heap reference still in scope), and sets the level of logging required. Specifying `min` produces a list of reason codes that define the unresettable events found, and specifying `max` produces the reason codes and also a stack trace where appropriate.

ibm.jvm.crossheap.events=on
: This system property specifically enables the logging of cross-heap references. Cross-heap references are references between the middleware heap and the transient heap in the JVM. They are logged to the event output destination at the time that each reference is created. The log entry includes a full stack trace to identify the line of code that created the cross-heap reference.

  Most of the cross-heap references that are logged will be removed before the JVM is reset, through the normal actions of the CICS and JVM code, and through any actions that your application takes for this purpose. However, if any cross-heap references are not removed before the JVM is reset, they cause the JVM to perform a trace-for-unresettability check. Any references that are found to be in live objects trigger unresettable events, which cause the JVM to be marked as unresettable and destroyed. Any references that are found to be in unreferenced middleware heap objects (garbage) are reported as reset trace events, which do not cause the JVM to be destroyed, but have still wasted processor time by causing the trace-for-unresettability check. You should

therefore ensure that all cross-heap references created by your applications are removed from the JVM before it is reset.

ibm.jvm.resettrace.events=on
This system property specifically enables the logging of reset trace events. Reset trace events are caused by cross-heap references that are still present in out-of-scope JVM objects (garbage) in the JVM at reset time. (If the cross-heap reference is still in scope, it causes an unresettable event.) Reset trace events do not cause the JVM to be marked as unresettable and destroyed, but you should still eliminate the cross-heap references that caused them, because the trace-for-unresettability check that is required for these cross-heap references reduces the performance of the JVM.

java.compiler=NONE
The activities of the Java just-in-time (JIT) compiler can interfere with the logging of unresettable events, reset trace events and cross-heap events. During the development process, specify the system property `java.compiler=NONE` (the word NONE must be in upper case) in the JVM properties file to turn off the JIT compiler for the JVM. Remember to turn the JIT compiler back on when you have finished investigating unresettable events, reset trace events and cross-heap events in your application.

Use the information from the output in your event log to eliminate the causes of unresettable events and reset trace events from your Java programs. The following example shows an unresettable event in an event log:

```
[EVENT 0xa]
TIME=30/08/2003 at 15:14:33.107
THREAD=EXAMPLE.TASK100.SAMP (0:22c8abf0)
CLASS=UnresettableEvent
DESCRIPTION=Attempt to load a native library in untrusted code
STACK=
     JAVA STACK TRACE
[END EVENT]
```

In this example, the reason code for the unresettable event is `0xa`. The meaning of the reason codes for unresettable events can be found in Appendix A of the document *Persistent Reusable Java Virtual Machine User's Guide*, SC34-6201.

Where an unresettable event is caused by an unresettable action, rewrite your application code to remove the unresettable action. Your Java programs should not perform unresettable actions when they are used in a production environment. If a program absolutely has to perform unresettable actions, you should run it in a single-use JVM instead (see “Single-use JVMs (REUSE=NO)” on page 88).

Where an unresettable event or a reset trace event is caused by a cross-heap reference, you can use the memory location listed for the event to identify the cross-heap reference recorded in the event log which is responsible for triggering the event. You can then use the stack trace associated with the cross-heap reference to help you to fix the problem. The document *Persistent Reusable Java Virtual Machine User's Guide*, SC34-6201, has more information about debugging reset trace events. You might have to perform compensatory actions in application code to cause a cross-heap reference to be removed, which could include closing files or streams, emptying collections, or other kinds of clean-up activity. If you cannot remove the cross-heap reference in application code, consider contacting your IBM support representative for further advice.

Invocations of Java programs that run in a resettable JVM are completely isolated from subsequent invocations of programs in the same JVM. This means that the programs cannot pass on state to subsequent invocations.

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 Java programs that must perform an unresettable action, and cannot at present be redesigned to eliminate this action so that they could run in a continuous JVM or a resettable JVM.

To improve performance, you should redesign these Java programs as soon as you can. Run the programs in a development environment, using a resettable JVM (with the option REUSE=RESET in the JVM profile) with unresettable event logging enabled, to identify the unresettable actions, and use this information to help you redesign the program. The programs can then be run in a continuous JVM or a resettable 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.

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 142 has more information about this.

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.

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, depending on the type of JVM.

- If an application running in a resettable JVM starts threads, the CICS task does not complete until all the threads that were started by user code, as well as the IPT, have finished executing. When all active threads have been stopped, control is returned to CICS. Starting a thread is an unresettable action, so the JVM is destroyed when the Java program has finished using it. Although this maintains isolation, it has a significant impact on performance.
- 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.

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 resettable JVM, the CICS task cannot complete until all activity has ended for sockets created using the `java.net` classes, and the sockets are automatically closed when the task completes. Creating and using sockets are not unresettable actions, but starting a thread to manage a socket is an unresettable action.
- 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. A security manager can be used for both resettable and continuous JVMs. 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.

Setting up a PROGRAM resource definition for a Java program to run in a JVM

When an application makes a request to run a Java program, it can make the request in various ways: for example, it can make an enterprise bean request, start a transaction, or link to or call the program by name. “How CICS creates JVMs” on page 71 explains how CICS locates the PROGRAM resource definition in each case. That topic also gives fuller information about some of the attributes mentioned in this topic. 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 section and move on to “Adding application classes to the class paths for a JVM” on page 128.

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 (EXECKEY attribute)” on page 72 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. “Setting up JVM profiles and JVM properties files” on page 94 tells you how to select or create JVM profiles and their associated JVM properties files.

As JVM profiles are HFS 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 HFS 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. (“Setting up JVM profiles and JVM properties files” on page 94 tells you how to select or create JVM profiles and their associated JVM properties files.) For each Java program, when you have specified the name of the JVM profile that CICS is to use for the JVM (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. You can edit JVM profiles and JVM properties files in a standard text editor.

If you are setting up CORBA stateless objects and enterprise beans, and you just need to know how to specify the JAR file or any additional classes on the shareable application class path, you can skip straight to “Including CORBA stateless objects and enterprise beans on the shareable application class path” on page 131. If you are setting up a standard Java application, or if you want to know more about how the different class paths should be used, carry on reading.

“Classes in a JVM” on page 64 explains the classes that are present in a JVM: system classes and standard extension classes, middleware classes, and application classes. That topic also explains the four class paths to which you can add the classes that your application needs. The class path you choose determines how the JVM treats the class.

When you add any class to a class path, remember that:

- Your application classes should not perform unresettable actions, that is, actions that modify the state of a JVM in such a way that it cannot be properly reset. The classes can perform these actions, but they cause resettable JVMs to be marked as unresettable and to be destroyed rather than re-used, and in continuous JVMs they might have an undesirable effect on subsequent program invocations in the JVM. “How JVMs are reused” on page 85 explains how to avoid unresettable actions.
- If you want to add classes to different class paths, they need to be in separate directories or JAR files. For example, if an application includes some classes that you want to place in the shared class cache (which need to go on the shareable application class path), and other classes that you prefer to be placed in the individual JVMs (which need to go on the standard class path), you should place each type of class in a separate directory.
- If the JVM for this application is to use the shared class cache (see “The shared class cache” on page 89), then classes that you add to most of the class paths

must be placed in the class paths in the JVM profile and JVM properties file for the master JVM that initializes the shared class cache, rather than in the JVM profile and JVM properties file for the JVM where the application will run. This applies to middleware classes, shareable application classes, and any items that you add to the library path. This is because the library path (defined by the LIBPATH option in the JVM profile), the trusted middleware class path (built from the CICS_DIRECTORY, TMPREFIX, and TMSUFFIX options in the JVM profile), and the shareable application class path (defined by the `ibm.jvm.shareable.application.class.path` system property in the JVM properties file), are all ignored for a worker JVM, and are taken instead from the JVM profile and JVM properties file for the master JVM. Only the standard class path (specified by the CLASSPATH option in the JVM profile) is taken from the JVM profile for the JVM itself, rather than from the JVM profile for the master JVM that initialises the shared class cache.

- The name of the class itself (including the name of the package, if the program has been built as a package) is not actually 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 HFS 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 126). So if the program has been built as a package, 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. 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. “Options in JVM profiles” in the *CICS System Definition Guide* shows some examples of paths.
- Use a colon as the separator between paths that you specify on a class path. (This is defined by the `path.separator` system property for the JVM, which you can change.) To include line breaks, use a backslash and a blank (\ ). “Rules for coding JVM profiles and JVM properties files” in the *CICS System Definition Guide* has a full explanation of how to code class paths and other items in a JVM profile or JVM properties file.

As explained in “Classes in a JVM” on page 64, you should add classes to class paths as follows:

1. Choose the **library path** for any native C dynamic link library (DLL) files that are to be loaded into the JVM by trusted code. This might include the DLL files needed to use the DB2 JDBC drivers, or any native code associated with a class that you are using to redirect JVM output (named on the USEROUTPUTCLASS option in the JVM profile).

   To include items on the library path, use the LIBPATH option in the JVM profile that you have selected for the application (or for worker JVMs, in the profile for the master JVM that initialises the shared class cache). “Options in JVM profiles” in the *CICS System Definition Guide* has more information about this option.
2. Choose the **trusted middleware class path** for middleware classes that can be trusted by the JVM to manage their own state across a JVM-reset. This class path is normally used for classes for middleware supplied by IBM or by another vendor, which are not included in the standard JVM setup for CICS. Trusted

middleware classes are permitted to change the JVM environment even if the JVM is resettable, so for this reason you should not normally place your own application classes on the trusted middleware class path.

To include classes on the trusted middleware class path, use the TMPREFIX or TMSUFFIX option in the JVM profile that you have selected for the application (or for worker JVMs, in the profile for the master JVM that initialises the shared class cache). Using TMPREFIX places the class at the beginning of the trusted middleware class path, and using TMSUFFIX places it at the end of the path. The trusted middleware class path also includes the path specified by the CICS_DIRECTORY option in the JVM profile, which points to the CICS-supplied JAR files. “Options in JVM profiles” in the *CICS System Definition Guide* has more details about all these options. For CICS, these options are used instead of the `ibm.jvm.trusted.middleware.class.path` JVM system property.

3. Choose the **shareable application class path** for application classes that you want to be cached. For standalone JVMs, the classes will be cached in the JVM, kept across JVM reuses, and reinitialized if the JVM is reset. For worker JVMs, they will be obtained from the shared class cache. Adding application classes to this class path, rather than to the standard class path, produces the best performance in a resettable JVM, and it should be your normal choice for loading application classes in a production environment.

   To include classes on the shareable application class path, use the system property `ibm.jvm.shareable.application.class.path`, in the JVM properties file that is referenced by the JVM profile that you have selected for the application (or for worker JVMs, in the JVM properties file for the master JVM that initialises the shared class cache). “System properties for JVMs” in the *CICS System Definition Guide* has more details about this system property.

4. Choose the **standard class path** for nonshareable application classes, that is, application classes that you do not want to be shared by other JVMs or across JVM resets. Classes on this class path are not placed in the shared class cache. They are loaded into the JVM. If the JVM is defined as resettable, classes on this class path are discarded when the JVM is reset, and reloaded from HFS files each time the JVM is reused. If the JVM is defined as a continuous JVM, however, nonshareable application classes are kept intact from one JVM reuse to the next.

   You should not normally place application classes on the standard class path without a good reason for doing so, because it causes a degradation in performance in a resettable JVM, and for worker JVMs (both resettable and continuous) it uses more storage than having a single copy of the classes in the master JVM. You might find it convenient to use this class path during application development in a non-production environment if your JVMs are resettable, because it means you do not have to phase out the JVM pool in order to update class definitions. (If your JVMs are continuous, you still need to phase out the JVM pool.) Occasionally, you might decide to use this class path for classes that are used infrequently, if you prefer to incur the performance cost in a resettable JVM of reloading the class each time it is required, rather than the storage cost of keeping the class in the JVM or in the shared class cache.

   To include classes on the standard class path, use the CLASSPATH option in the JVM profile that you have selected for the application. This class path is always taken from the profile for the JVM itself, not from the profile for the master JVM that initialises the shared class cache. “Options in JVM profiles” in the *CICS System Definition Guide* has more details about this option. For CICS, this option is used instead of the `java.class.path` JVM system property.

Including CORBA stateless objects and enterprise beans on the shareable application class path

As explained in “Enabling applications to use a JVM” on page 119, for CORBA stateless objects and enterprise beans, you need to include the following items on the shareable application 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.

You do not need to include the deployed JAR files (DJARs) for enterprise beans on the class path.

Specify these items on the shareable application class path by using the `ibm.jvm.shareable.application.class.path` system property that will be used by the JVM for the request processor program. This means that:

- If the JVM for the request processor program is a standalone JVM that does not use the shared class cache, then use the `ibm.jvm.shareable.application.class.path` system property in the JVM properties file that is referenced by the JVM profile named in the JVMPROFILE option of the PROGRAM definition for the request processor program. For example, if your CORBA stateless objects or enterprise beans use the default request processor program DFJIIRP (which is named by the default CIRP transaction on REQUESTMODEL definitions), and DFJIIRP is set to use the JVM profile DFHJVMCD (which is the default JVM profile for CICS-supplied system programs), then you need to specify the paths to the classes in the JVM properties file `dfjjvmcd.props` (which is the JVM properties file referenced by DFHJVMCD).
- If the JVM used by the request processor program is a worker JVM that uses the shared class cache, then use the `ibm.jvm.shareable.application.class.path` system property in the JVM properties file that is referenced by the JVM profile for the master JVM that initializes the shared class cache. For example, take the case where you have created a request processor program that uses the JVM profile DFHJVMPC (a JVM profile for JVMs that use the shared class cache), so that your CORBA stateless objects or enterprise beans use the shared class cache, and your master JVM uses the JVM profile DFHJVMCC (which is the default JVM profile for a master JVM). In this situation, you need to specify the paths to the classes in the JVM properties file `dfjjvmcc.props`, which is the JVM properties file referenced by DFHJVMCC, instead of in the JVM properties file `dfjjvmpc.props`, which is referenced by DFHJVMPC.

When you are adding these items to the shareable application class path, remember:

- The name of the class itself is not actually 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 128.
- Use a colon as the separator between paths that you specify on a class path. (This is defined by the `path.separator` system property for the JVM, which you can change.) To include line breaks, use a backslash and a blank (\ ). “Rules for

coding JVM profiles and JVM properties files" in the *CICS System Definition Guide* has a full explanation of how to code class paths and other items in a JVM profile or JVM properties file.

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 clearing up unrequired JVMs."How CICS creates JVMs" on page 71, "How CICS manages JVMs in the JVM pool" on page 75 and "How CICS allocates JVMs to applications" on page 79 explain how CICS performs these 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 75 explains the issues associated with MAXJVMTCBS, and what happens when an MVS storage constraint occurs. The *CICS Performance Guide* tells you how to work out an appropriate setting for the MAXJVMTCBS system initialization parameter.
- 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."
- Terminate all the JVMs in the JVM pool, or disable the JVM pool so that it cannot service new requests. See "Terminating or disabling the JVM pool" on page 134.
- Redirect messages from JVM internals and output from Java applications running in a JVM, and add time stamps and headers to the records. You can create a merged log file containing the output from multiple JVMs, or a file containing the output for a single program instance or task. See "Redirecting JVM output" on page 135.
- Control JVM tracing. See "Controlling tracing for JVMs" on page 140.
- Tune the JVM pool as a whole, and your individual JVMs, to achieve optimum performance. The *CICS Performance Guide* tells you how to do this.

Monitoring JVM activity

You can use CICS commands and statistics to monitor:

- The JVM pool.
- The JVMs that CICS has in the JVM pool, and how CICS assigns them to requests.
- The JVM profiles that CICS is using to create JVMs, and the activity for each JVM profile.
- The Java programs that run in JVMs.

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. 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).

The INQUIRE JVM command does not find the master JVM that initializes the shared class cache. It only finds worker JVMs and standalone JVMs.

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. 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 HFS 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 times JVMs of this profile were marked unresettable and destroyed (because an application performed an unresettable action in a JVM defined as resettable).
- 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.

Terminating or disabling the JVM pool

CICS reduces the number of active JVMs automatically if the workload does not require them. If a JVM is inactive for 30 minutes, it is discarded.

You can terminate all the JVMs in the JVM pool by using a CEMT SET JVMPOOL PHASEOUT, PURGE or FORCEPURGE command (or the equivalent EXEC CICS command). When you use this command, all the JVMs in the pool, both worker JVMs using the shared class cache and standalone JVMs running independently of the shared class cache, are terminated. The shared class cache is also terminated once all the worker JVMs that were dependent on it have been terminated. On the CEMT SET JVMPOOL command:

- If you use the PHASEOUT option, the JVMs are marked for deletion, but they are only terminated when they finish running their current Java programs.

- If you use the PURGE option, the tasks using JVMs are purged, and the JVMs are terminated.
- If you use the FORCEPURGE option, the tasks using JVMs are forcepurged, and the JVMs are terminated.

You can also disable the JVM pool so that it cannot service new requests, by using a CEMT SET JVMPOOL DISABLED command (or the equivalent EXEC CICS command). 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 CEMT SET JVMPOOL ENABLED command (or the equivalent EXEC CICS command).

Redirecting JVM output

By default, output from Java applications running in a JVM is written to the HFS files that are named by the STDOUT and STDERR options in the JVM profile for the JVM. The file named by the STDOUT option is used for System.out requests, and the file named by the STDERR option is used for System.err requests. The output files are located in the working directory named by the WORK_DIR option in the JVM profile.

You can specify a fixed file name for each of the output files, in which case the output from multiple JVMs is appended to the named file, and the output is interleaved. However, the records are not given headers. Alternatively, you can use the **-generate** option on the STDOUT and STDERR options to specify that file names should be generated when the JVM is started up. If you do this, each JVM can have its own output files, identified by a time stamp and the applid of the CICS region. Otherwise, the output from the JVMs in all your CICS regions will be written to the same output files. However, note that the use of the **-generate** option is not recommended in a production environment, because it can be detrimental to the performance of your JVMs. “Options in JVM profiles” in the *CICS System Definition Guide* tells you more about the STDOUT and STDERR options, and “Customizing or creating JVM profiles and JVM properties files” on page 102 tells you how to customize the options in a JVM profile.

To gain more control over the output from your JVMs, as well as specifying the STDOUT and STDERR options, you can use the USEROUTPUTCLASS option in a JVM profile to name 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. The HFS 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.

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. However, it can be useful to specify the USEROUTPUTCLASS option during application development. This is because even if you use the **-generate** option on the STDOUT and STDERR options, the generated output files can only be differentiated by the applid of the CICS region and the time of generation. The USEROUTPUTCLASS option enables developers using the same CICS region to separate out their own JVM output, and direct it to an identifiable destination of their choice. The USEROUTPUTCLASS option can be

used for most types of JVM, with the exception of the master JVM that initializes the shared class cache, because the output redirection class will never be invoked by the activities of the master JVM.

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, DFHJVMPC 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 behaviour of both these classes is described later in this topic.

The classes are shipped as a middleware class file `dfjoutput.jar`, which is in the directory `/usr/lpp/cicsts/cicsts31/lib`, where `cicsts31` is a user-defined value that you chose for the CICS_DIRECTORY variable used by the DFHIJVMJ job during CICS installation. 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 do this.

Table 7 shows what output from JVMs can and cannot be intercepted by the class named by the USEROUTPUTCLASS option. The class that you use must be able to deal with all the types of output that it might intercept.

Table 7. JVM output intercepted by Java class named by USEROUTPUTCLASS

Output type	**Intercepted by class**
System.out output	Yes
System.err output	Yes
JVM internal messages	Yes
Unresettable events log (controlled by ibm.jvm.events.output system property)	Yes (unless another destination file is explicitly named in the system property). See note 1.
Language Environment stdout and stderr output	Not intercepted
JAVADUMP	Not intercepted
HPITRACE	Not intercepted

Note:

1. The destination for information about unresettable events in a resettable JVM is controlled by the ibm.jvm.events.output system property in the JVM properties file for the JVM. If you have used that system property to name a specific HFS file as the destination for the unresettable events information, then the information goes to that file, and is not intercepted. (The default ibm.jvm.events.output system property in the CICS-supplied sample JVM properties files names the file event.log as the destination.) If the system property names the destination as stdout or stderr, then the information is intercepted by the class named on the USEROUTPUTCLASS option in the JVM profile, and directed to the destination specified by that class. If the ibm.jvm.events.output system property is null, then this output is not produced at all.

The class that you are using must be present in a directory on the trusted middleware class path used by the JVM (to which you can add paths by using the TMPREFIX or TMSUFFIX option in the JVM profile). If you are using your own

class in place of the supplied sample class, any associated native code for your class must be present on the library path used by the JVM (specified by the LIBPATH option in the JVM profile), and must be explicitly loaded using the System.loadLibrary() call, either at class load time via a static initializer, or in the class constructor. (This avoids the need to include doPrivileged() blocks around the loadLibrary call when you are running with Java security active.) Note that if the JVM is to use the shared class cache (if CLASSCACHE=YES is specified in the JVM profile), you will need to include the class and any associated native code in the trusted middleware class path and library path that are specified in the JVM profile for the master JVM that initializes the shared class cache, rather than those specified in the JVM profile for the JVM itself. The CICS-supplied sample JVM profile for the master JVM is DFHJVMCC, and the JVM properties file that it references is `dfjjvmcc.props`.

Also bear in mind that the Java programs that will run in JVMs that use the USEROUTPUTCLASS option should include appropriate exception handling to deal with the exceptions that might be thrown by a class named on the USEROUTPUTCLASS option. The CICS-supplied sample classes handle all exceptions internally, so they do not return any errors to the calling program.

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 HFS 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 HFS 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 HFS files `/work_dir/applid/stdout/CSJO` and `/work_dir/applid/stderr/CSJE` as described above, or if these files are unavailable, to the HFS 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 HFS 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 HFS files, one for stdout output and one for stderr output, that are uniquely named using a task number (in the format `Task.`*`tasknumber`*). The HFS 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 middleware class file `dfjoutput.jar`, which is in the directory `/usr/lpp/cicsts/cicsts31/lib`, where `cicsts31` is a user-defined value that you chose for the CICS_DIRECTORY variable used by the DFHIJVMJ job during CICS installation. 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.

Problem determination for JVMs

Many of the usual sources of CICS diagnostic information contain information that applies to JVMs, including:

- Abend codes and error messages. *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 132 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. “JVM domain trace points” in *CICS Trace Entries* has details of these trace points.

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 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.

The CICS documentation provides more information about some of these diagnostic tools and interfaces, as follows:

- “Controlling tracing for JVMs” on page 140 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. CICS enables you to control JVM tracing by using the CETR transaction and by using CICS system initialization parameters. The JVM trace information that is produced is output as CICS trace (specifically, as instances of CICS trace point SJ 4D01).
- “Debugging an application that is running in a CICS JVM” on page 142 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 145.

Many more diagnostic tools and interfaces are available for the JVM. The *IBM Developer Kit and Runtime Environment, Java 2 Technology Edition, Version 1.4.2 Diagnostics Guide*, SC34-6358, 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. CICS uses the JVM's **ibm.dg.trc.external** system property, which can be set either in the JVM properties file or through CETR, to control tracing for the JVM. However, the JVM has several other system properties that can be set to output trace. The *IBM Developer Kit and Runtime Environment, Java 2 Technology Edition, Version 1.4.2 Diagnostics Guide* has information about all 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 nonsystem heap.
- The HPROF profiler, which is shipped with the IBM Software Developer Kit for z/OS, Java 2 Technology Edition, Version 1.4.2, 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).

If you are using enterprise beans, Chapter 24, “Dealing with CICS enterprise bean problems,” on page 321 has more information about issues that apply specifically to them.

Controlling tracing for JVMs

You can control tracing for JVMs using the CICS-supplied transaction CETR. With CETR, you can activate tracing for each transaction that uses the JVM. You can set several levels of tracing (or a user-defined option string) using the JVM trace options, turn tracing on or off at each level for transactions, or reset to the supplied settings. JVM tracing can produce a large amount of output, so you should normally activate it for special transactions, rather than turning it on globally for all transactions. “Using JVM trace options” in *CICS Supplied Transactions* tells you how to use CETR to control tracing for JVMs.

The default JVM trace options that are provided in CICS use the JVM trace point level specifications. The default settings for JVM Level 0 trace, JVM Level 1 trace, and JVM Level 2 trace specify LEVEL0, LEVEL1, and LEVEL2 respectively, so they map to the Level 0, Level 1 and Level 2 trace point levels for JVMs. A Level 0 trace point is very important, and this classification is reserved for extraordinary events and errors. Note that unlike CICS exception trace, which cannot be switched off, the JVM Level 0 trace is normally switched off unless JVM tracing is required. The Level 1 trace points and Level 2 trace points provide deeper levels of tracing. The JVM trace point levels go up to Level 9, which provide in-depth component detail.

It is suggested that you keep the CICS-supplied level specifications, but if you find that another JVM trace point level is more useful for your purposes than one of the default levels, you could change the level specification to map to your preferred JVM trace point level (for example, you could specify LEVEL5 instead of LEVEL2 for the JVMLEVEL2TRACE option). Note that enabling a JVM trace point level enables that level and all levels above it, so for example, if you activate JVM Level 1 trace for a particular transaction, you receive Level 0 trace points for that transaction as well. This means that you only need to activate the deepest level of tracing that you require, and the other levels are activated as well.

You can add further parameters to the basic level specifications for JVM Level 0 trace, JVM Level 1 trace, and JVM Level 2 trace, if you want to include or exclude particular components or trace point types at the selected trace levels. If you want to create more complex specifications for JVM tracing which use multiple trace point levels, or if you do not want to use trace point levels at all in your specification, use the JVMUSERTRACE option to create a trace option string that includes the parameters of your choice. “Defining tracing for JVMs” in the *CICS Problem Determination Guide* has information about the JVM trace options that you can set using the JVM Level 0 trace, JVM Level 1 trace, JVM Level 2 trace, and JVM User trace levels. There is further information about JVM trace and about problem determination for JVMs in the *IBM Developer Kit and Runtime Environment, Java 2 Technology Edition, Version 1.4.2 Diagnostics Guide*, SC34-6358, which is available to download from www.ibm.com/developerworks/java/jdk/diagnosis/.

There are alternatives to using CETR to control tracing for JVMs. You can **set the JVM trace options** by:

- Using the CICS system initialization parameters JVMLEVEL0TRACE, JVMLEVEL1TRACE, JVMLEVEL2TRACE, and JVMUSERTRACE (see the *CICS*

System Definition Guide) to set the default trace options for JVMs in the JVM pool in your CICS region. You can only supply these parameters at CICS startup time; you cannot define them in the DFHSIT macro. You can then use CETR to view and change these options, if you want. These system initialization parameters do not activate JVM tracing, they only set the default JVM trace options.

- Using the EXEC CICS INQUIRE JVMPOOL command to inquire on the JVM trace options you have set for the JVM pool, and the EXEC CICS SET JVMPOOL command to change them. (Note that the JVM trace options are not available on the CEMT equivalents for these commands.)

You can **activate JVM trace** by:

- Using the CICS system initialization parameters SPCTRSJ, which applies to special tracing , or STNTRSJ (see the definition for STNTRxx in the *CICS System Definition Guide*), which applies to standard tracing, to activate JVM trace at startup. Specify level numbers 29–32 to activate the levels of JVM trace that you require. You can only supply these parameters at CICS startup time; you cannot define them in the DFHSIT macro.
- Using the EXEC CICS SET TRACETYPE command to set trace levels 29–32 for the SJ component.

Remember that JVM trace should normally only be activated for special transactions. If you are activating JVM trace by one of these alternative methods, you should normally use the SPCTRSJ system initialization parameter rather than the STNTRSJ system initialization parameter, and use the SPECIAL option on the EXEC CICS SET TRACETYPE command, rather than the STANDARD option.

If you need to trace a JVM during its whole lifetime, including start-up and reset as well as the periods when it is being used by a transaction, you can set and activate trace options using the **ibm.dg.trc.external** system property in the JVM properties file that is referenced by the JVM profile. (“Customizing or creating JVM profiles and JVM properties files” on page 102 tells you how to set system properties for a JVM.) This system property has to be used with care, as JVM tracing can produce large amounts of output in a very short time. “Defining tracing for JVMs” in the *CICS Problem Determination Guide* has information about the JVM trace options that you can set using this system property.

When CICS starts to use or re-use a JVM, it ensures that any trace options that you have set and activated using CETR are applied. Activating or deactivating a trace option using CETR overrides any setting for that trace option in the **ibm.dg.trc.external** system property. For example, a trace option that is activated in the system property, but deactivated using CETR, will be deactivated when CICS starts to use or re-use the JVM. If you use CETR to activate any trace options that are not referred to in the **ibm.dg.trc.external** system property, the trace options that you have specified in CETR are *added to* any trace options that you have set using the **ibm.dg.trc.external** system property. The trace output will then reflect all the trace options that you requested in both CETR and the system property.

JVM trace appears as CICS trace points in the JVM domain. When you activate JVM trace by setting trace levels 29–32 for the SJ component, each JVM trace point that is generated appears as an instance of CICS trace point SJ 4D01. If the JVM trace facility fails, CICS issues the trace point SJ 4D00.

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. These trace

points can be activated using the CETR Component Trace screens, as described in "Selecting tracing by component" in the *CICS Problem Determination Guide*. 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 *CICS Trace Entries* 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. 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. The debugging process is summarized in Figure 9.

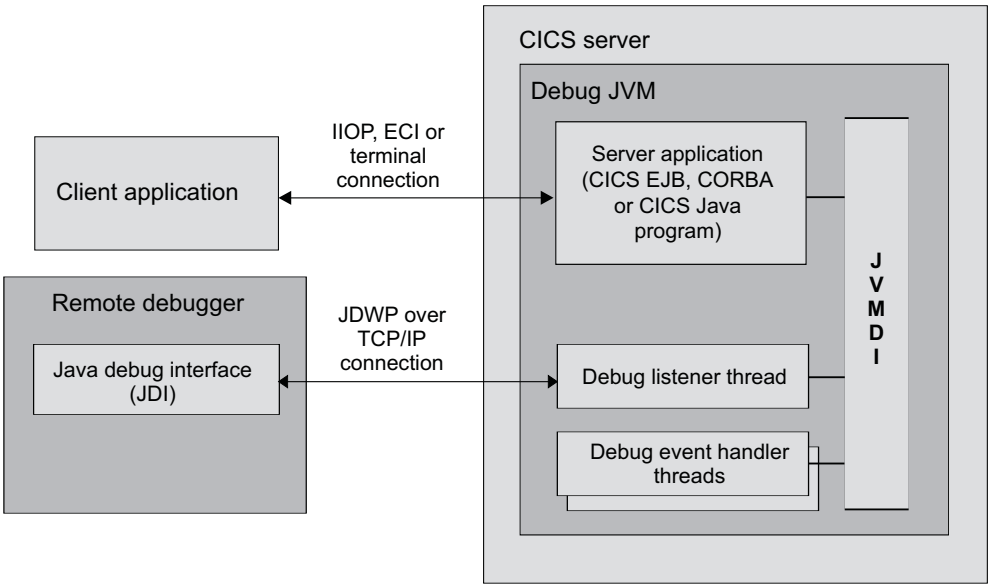

Figure 9. Debugging in the CICS JVM using JPDA.

JPDA consists of the following layered APIs:

Java Debug Interface (JDI)
This is a Java programming language interface providing support for remote debugging. This is the highest level interface in the architecture, and can be used to implement a remote debugger user interface without having to write any code that runs in the application JVM or understand the protocol between the debugger and the JVM. Most third party debuggers that support JPDA currently use this API.

Java Debug Wire Protocol (JDWP)
This API defines the format of the flows that run between the application JVM and the debugger user interface. This protocol is for use by debuggers that need to exploit the communication at a lower level than the JDI, and for JVM suppliers or more advanced debugger developers who need to support the standard connection architecture from the application JVM side.

Java Virtual Machine Debug Interface (JVMDI)
This is a low-level native interface within the JVM. It defines the services a Java virtual machine must provide for debugging, and can be used by advanced debugger developers who wish to implement debugger code that runs inside the application JVM (to implement an alternative transport mechanism for debugger connection, for example).

When you start the JVM in debug mode, the JVMDI interface is activated and additional threads are started in the JVM. One of these threads handles communication with the remote debugger, the others monitor the application that is running in the JVM. You can issue commands in the remote debugger, for example to set break points or to examine the values of variables in the application. These commands are activated by the listener and event handler threads in the JVM.

There is more information about JPDA on the web site `java.sun.com/products/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 must set some options in the JVM profile for the JVM. “Customizing or creating JVM profiles and JVM properties files” on page 102 explains the procedure for customizing options in a JVM profile.

The specific options required for debugging are as follows:

Xdebug This option is required to start the JVM in debug mode; that is, with the JPDA interfaces active.

Xrunjdwp=(suboption=...,suboption=...)
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’re 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:6789)
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.

The *CICS System Definition Guide* 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, the JVM profiles for worker JVMs that specify CLASSCACHE=YES, and the JVM profile for the master JVM that initializes the shared class cache. JVM debugging is not supported for shared classes, and if you configure these JVM profiles for debug, CICS ignores your setting. DFHJVMPC and DFHJVMCC are the CICS-supplied sample profiles for worker and master JVMs respectively.
- **Avoid configuring for debug** the CICS-supplied sample JVM profiles DFHJVMPR and DFHJVMPS, 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. “Redirecting JVM output” on page 135 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

When you use plugin programs to debug Java applications, you need to:

- Code the programs to the standards required of trusted middleware code. Middleware is responsible for resetting itself correctly at the end of a transaction and, if necessary, for reinitializing at the beginning of a new transaction, in order

to isolate different applications from each other. *Persistent Reusable Java Virtual Machine User's Guide*, SC34-6201 explains how middleware should be written.

- Use the TMSUFFIX option in the appropriate JVM profile to include the programs on the trusted middleware class path for the JVM which will be used by the application that is to be debugged. "Adding application classes to the class paths for a JVM" on page 128 tells you how to do this; classes for plugin programs can be added in the same way as classes for ordinary applications.

If the classes for plugin programs are placed on other class paths, they might not be accessible to the correct classloader, and could also cause resettable JVMs to be marked as unresettable. Debug plugins can be used with resettable, continuous and single-use JVMs (with REUSE=RESET, REUSE=YES or REUSE=NO in the JVM profile), provided that the classes are placed on the trusted middleware class path.

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 10 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 10. Definitions of the DebugControl and Plugin interfaces

The code fragment in Figure 11 on page 147 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 11. 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:

- `com.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.
- `com.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.
- `com.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.

The *CICS System Definition Guide* tells you about the system properties available for JVMs. “Setting up JVM profiles and JVM properties files” on page 94 tells you how to customize them.

Part 4. CICS and IIOP

This Part tells you what you need to know to configure CICS to support distributed IIOP applications.

Chapter 12. 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 27, “Stateless CORBA objects,” on page 357.

`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 16, “What are enterprise beans?,” on page 199.

Some common CORBA terminology

The following 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 13. The IIOP request flow

The following diagram shows the execution flow of an incoming request:

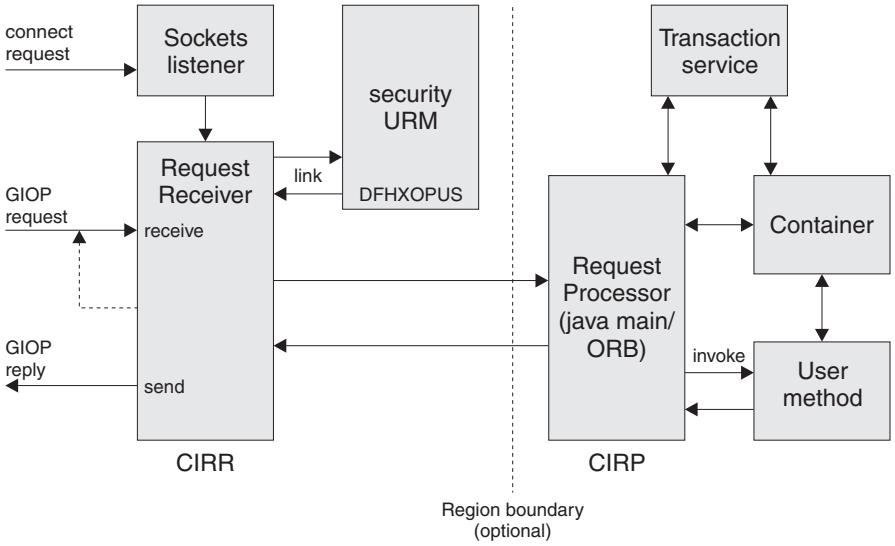

Figure 12. IIOP request execution flow

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 *OS/390 V2R8.0 SecureWay Communication Server: IP Configuration, SC31-8513-03*.

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 the *CICS Customization Guide* for guidance on writing a customized distributed routing program. See the *CICSPlex System Manager Managing Workloads* 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.

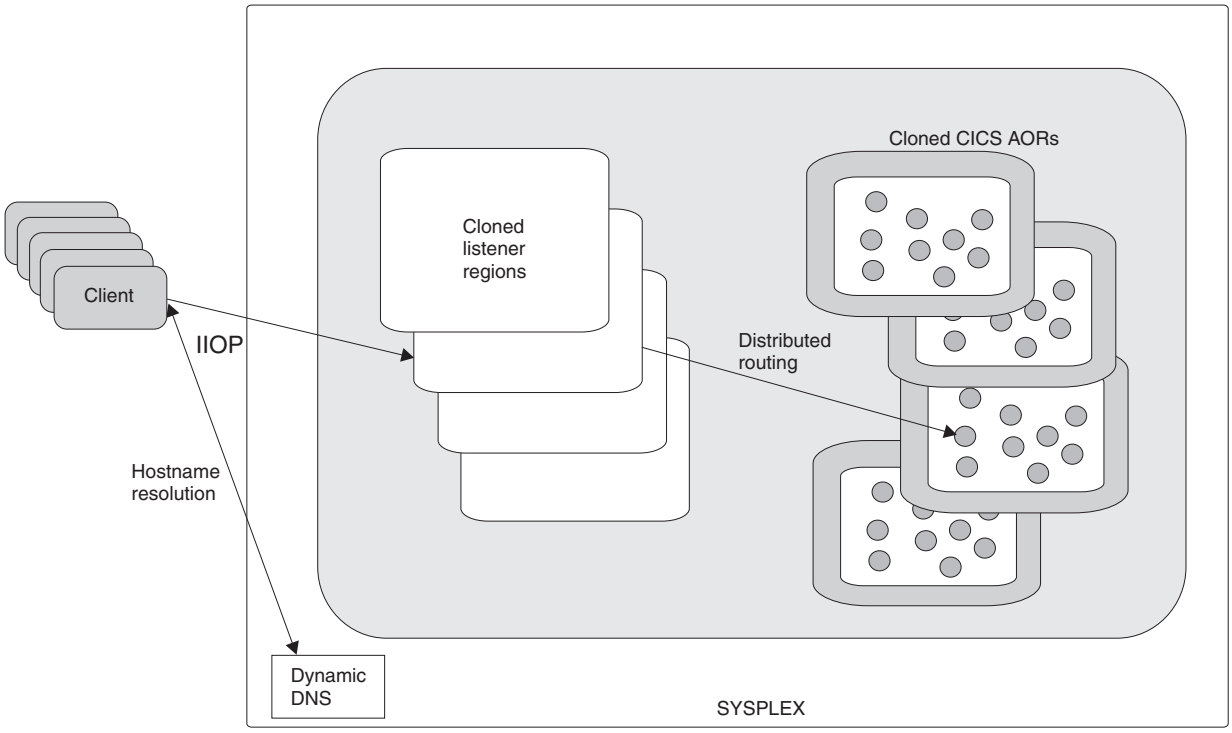

Figure 13. A CICS logical server. In this example, the logical server consists of a set of cloned "listener" regions and a set of cloned AORs. Connection optimization by means of dynamic DNS registration is used to balance client connections across the listener regions. Distributed routing is used to 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 name space. 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

The following diagram shows an example 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

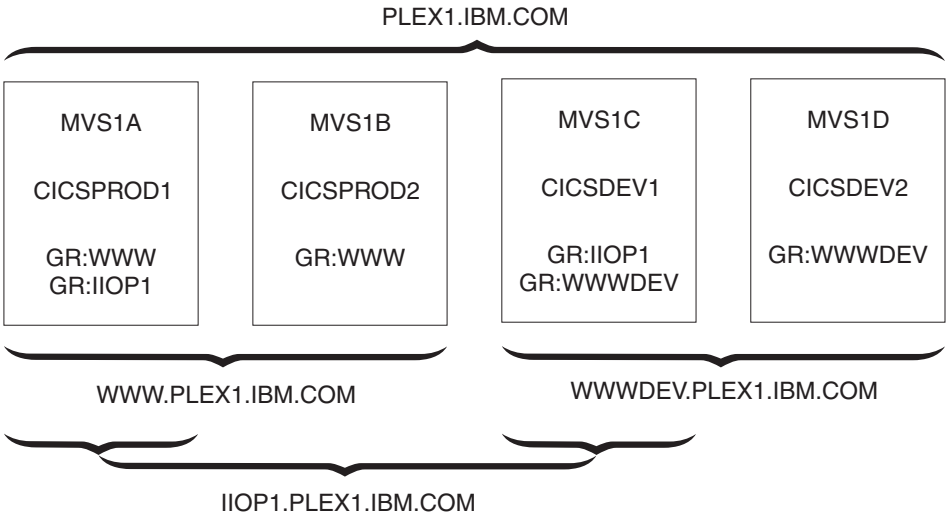

Figure 14. CICSplex using DNS connection optimization

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

The following 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 simply 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.)

Authentication of IIOP requests

Authentication
is the process by which a service accurately establishes the authenticity of a user making a request.

Identification
is the process by which the identity of a user is established. Typically, the term *user ID* is used to denote the user's identity; in Java parlance, the term *principal* is used.

The two processes are related because, in many cases, the information used to authenticate a user is also used for identification. For example, in a scheme that uses a user ID and password, the user ID alone identifies the user, while the combination of user ID and password authenticates the user.

Authentication is provided by one of the following mechanisms:
- Basic authentication
- SSL client certificate authentication
- Asserted identity authentication

See the *CICS RACF Security Guide* for more information.

For IIOP requests, you can identify the user in the following ways:
- Using SSL client authentication—see the *CICS RACF Security Guide* for more information.
- If SSL client authentication does not provide a user ID, you can write a user-replaceable IIOP security program to provide one. Specify the name of your security program on the URM attribute of the TCPIPSERVICE definition for the port. See “Using the IIOP user-replaceable security program” on page 189 for more information.
- The client can supply a user ID directly. Typically this is done as part of the authentication process.

  You can identify users in this way when you use basic authentication with the HTTP and ECI application protocols
- If none of these mechanisms provides a user ID, the CICS default user ID is used.

The authentication and identification schemes are specified in the CORBASERVER and TCPIPSERVICE resource definitions. Each CORBASERVER is associated with one or more TCPIPSERVICE definitions; each TCPIPSERVICE supports a different mechanism for authentication and identification:
- The ASSERTED attribute of the CORBASERVER names a TCPIPSERVICE that supports inbound IIOP with asserted identity authentication.
- The BASIC attribute of the CORBASERVER names a TCPIPSERVICE that supports inbound IIOP with basic authentication.
- The CLIENTCERT attribute of the CORBASERVER names a TCPIPSERVICE that supports inbound IIOP with SSL client certificate authentication.
- The SSLUNAUTH attribute names a TCPIPSERVICE that supports inbound IIOP with SSL encryption and no client authentication.
- The UNAUTH attribute names a TCPIPSERVICE that supports inbound IIOP with no authentication.

Note:
1. To change the association between an installed CORBASERVER definition and its TCPIPSERVICE definitions, you must discard and reinstall the CORBASERVER definition.
2. If you use SSL encryption, or SSL client certificate authentication, you must configure your CICS system to support SSL. See the *CICS RACF Security Guide*.

An enterprise bean can use the `getCallerPrincipal()` method to obtain information about the client which is contained in the certificate. See “Deriving distinguished names” on page 336 for more details.

The derived USERID is passed with the IIOP request to the request processor, for authentication of the request execution. If the request processor is executing in a different CICS region, the transmission of the USERID follows CICS rules for CONNECTION authentication.

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 189.

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 the *CICS RACF Security Guide* for more information.

IIOP users are recommended to specify SEC=YES and ATTACHSEC(IDENTIFY).

Chapter 14. Configuring CICS for IIOP

Important

If you are setting up a CICS EJB server (to support enterprise beans) we recommend that you start at Chapter 17, “Setting up an EJB server,” on page 227, which contains the specific requirements for enterprise bean support, rather than here.

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. However, the specific requirements for enterprise beans are not addressed here. See Chapter 17, “Setting up an EJB server,” on page 227 for these further requirements.

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. Thus, to configure CICS as an IIOP server or client, set up the following host software environment:

- A z/OS system at Version 1.4 or later, with UNIX Systems Services and HFS
- Language Environment configured and active
- CICS
- The IBM Software Developer Kit for z/OS, Java 2 Technology Edition, Version 1.4.2. This uses persistent reusable JVM technology. It is available from :

  `http://www.s390.ibm.com/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 178
- “Setting up CICS for IIOP” on page 179

You might also need to perform one of these steps:

- “Setting up an LDAP server” on page 168
- “Setting up a COS Naming Directory Server” on page 178

If you choose “Setting up an LDAP server” on page 168, you should read “The LDAP namespace structure” on page 174.

Setting up the host system for IIOP

To support IIOP you need to perform the following system tasks:

1. “Giving CICS regions access to z/OS UNIX System Services and HFS directories and files” on page 53. As part of this task, you will:
   a. Give CICS access to the HFS directories and files that are needed to create JVMs
   b. Create and give CICS access to the HFS working directory that you have specified for input, output, and messages from the JVMs

2. “Setting up JVM profiles and JVM properties files” on page 94. During this task, you will:
   a. Enable CICS to locate JVM profiles and their associated JVM properties files.
   b. Choose appropriate JVM profiles for your CORBA stateless objects and enterprise beans.
   c. If necessary, customize the JVM profiles and JVM properties files 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 to the JVM properties files.)

   Bear in mind when reading “Setting up JVM profiles and JVM properties files” on page 94 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, the JVM properties file used is that specified on the JVM profile.
   - The default JVM profile, specified on the PROGRAM definition of the default request processor program, is DFHJVMCD.
   - The default JVM properties file, specified on the default JVM profile, DFHJVMCD, is `dfjjvmcd.props`.
   - If you plan to use the default JVM profile and JVM properties file with your CORBA stateless object and enterprise bean requests, then you need only to locate DFHJVMCD and `dfjjvmcd.props` and customize them for your CICS region, as described in “Setting up JVM profiles and JVM properties files” on page 94.

     If you plan to use customized JVM profiles or properties files, you should still make the changes to DFHJVMCD and `dfjjvmcd.props` 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 HFS. 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 an HFS 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 and HFS directories and files” on page 53 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 *CICS Transaction Server for z/OS Installation Guide*.
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, such as the IBM SecureWay Directory, which is shipped with the IBM SecureWay Security Server, an optional feature of OS/390 and z/OS.
  - IBM SecureWay Directory is available for Windows 32 or System/390®.
  - If you use an LDAP name server on your System/390, enterprise beans from CICS and WebSphere can interoperate more readily in a shared namespace. See “Setting up an LDAP server” on page 168.
- A Corba Object Services (COS) Naming Directory Service, such as that provided with IBM WebSphere Application Server Version 6.
  - COS Naming Servers run on an external machine.
  - Any industry-standard COS Naming Serice that supports JNDI Version 1.2 can be used. Among others, you might choose the COS Naming Service supplied with IBM WebSphere Application Server Advanced Edition for AIX®, Version 3.5 and later.

  See “Setting up a COS Naming Directory Server” on page 178

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.

Specify the Web address (URL) and TCP/IP port number of your name server using the `com.ibm.cics.ejs.nameserver` property in your JVM properties files. The supplied sample JVM profiles contain examples of how to do this, and the *CICS System Definition Guide* has more detailed information.

Important:

1. You must specify the location of your name server on the `com.ibm.cics.ejs.nameserver` property in all the JVM 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 `dfjjvmcd.props` properties file referenced by 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 the JVM properties files referenced by 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 the *CICS System Definition Guide*.

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/390, 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 Name Space.

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 1.4.2 base, rather than the WebSphere context factory supplied with CICS. See “SUN LDAP Context Factory” on page 282 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 174) 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 the *CICS System Definition Guide*. 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 Name Space 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/390, for any of the following reasons:

1. The security configuration needs changing to cope with the CICS regions being introduced. See “The LDAP namespace structure” on page 174 and “Security considerations” on page 175 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/390 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 name space 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/390 systems working with the differing containerdn 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/390, these are the steps necessary to configure a new LDAP server:

1. Install the WebSphere naming schema
2. Add a new suffix
3. Build the system name space root node (containerdn)
4. Build the legacyRoot node below the name space 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:

```
com.ibm.cics.ejs.nameserver=ldap://wibble.ibm.com:389
com.ibm.ws.naming.ldap.containerdn=ibm-wsnTree=t1,o=WASNaming,c=US
com.ibm.ws.naming.ldap.noderootrdn=ibm-wsnName=legacyRoot,ibm-wsnName=PLEX2,
      ibm-wsnName=domainRoots
java.naming.security.authentication=simple
java.naming.security.principal=cn=CICSSystems,c=US
java.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 authorised 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/cicsts31/utils/namespace/WebSphereNamingSchema.ldif on HFS.

   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 OS/390 or 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 authorised 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.ibm.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 an OS/390 or 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 name space root node (containerdn)

   An ldif file to build the root of the system name space (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 userids are defined. To understand how they are used, see “Security considerations” on page 175.

   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 name space 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 authorised principal is the distinguished name of a user with sufficient authority on the nameserver to write entries.

   A specific example is:

```
ldapmodify-h wibble.ibm.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 authorised to write to the namespace>
              -credentials <password for that principal>
             [-force]
```

For example:

```
DFHBuildSNS    -ldapserver ldap://wibble.ibm.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 176.

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 system properties and adding them to your JVM properties files

The system properties that you can use in JVM properties files include six, described below, that relate to the use of an LDAP namespace for JNDI. The *CICS System Definition Guide* has full descriptions of each of these system properties.

- If you have just set up this LDAP namespace you will know the values that you used to do so. 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.

  It is unlikely that the security principal or credentials can be discovered by browsing the namespace.

com.ibm.cics.ejs.nameserver
: is the URL for the LDAP server being configured. In the preceding example it is *ldap://wibble.ibm.com:389*

com.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.

com.ibm.ws.naming.ldap.noderootrdn
: can be determined from the domain 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.

java.naming.security.authentication
: is set to *simple* if CICS must authenticate itself to LDAP in order 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 in order to write to it, do not set a value for this system property.

Important: If you do specify this system property, you also need to specify java.naming.security.principal and java.naming.security.credentials. Since these hold the UserID and password that CICS requires to access the secure LDAP service, you need to 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. You should ensure that the JVM properties files are secure, with update authority restricted to system administrators.

java.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, it would be advisable to use at least the *cn=CICSUser,c=US* distinguished name specified in dfhsns.ldif, since that ID is only able to write to a particular area of the LDAP namespace (the containerdn and below).

If you want even tighter security, the principal could be *cn=CICSSystems,c=US*. There is extra LDAP configuration to be performed if you use this ID, see “Security considerations” on page 175' for a full discussion of CICS LDAP security configuration.

java.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 need to specify them in all the JVM properties files that are used by CORBA applications or enterprise beans.

In particular, be sure to specify them in the dfjjvmcd.props properties file referenced by 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.

You also need to specify these system properties in the JVM properties files referenced by 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.

The only JVM properties file that never needs to include this information is a JVM properties file that you are only using for the master JVM that initializes the shared class cache, because this JVM is not used to run applications. The CICS-supplied sample JVM properties file for the master JVM is dfjjvmcc.props.

The *CICS System Definition Guide* tells you the rules for coding system properties in a JVM properties file.

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 customisation 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 name space 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 name space 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 176.

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 name space.

Security considerations

If you specified that CICS must authenticate itself to LDAP in order to write to it, by coding the system property **java.naming.security.authentication=***simple* in your JVM properties files, you now have a choice between

- “Security at the containerdn level” on page 176, or
- “Security at the CICS region level” on page 176.

To help you decide, a very simplified view of part of the LDAP namespace is shown in Figure 15 on page 176.

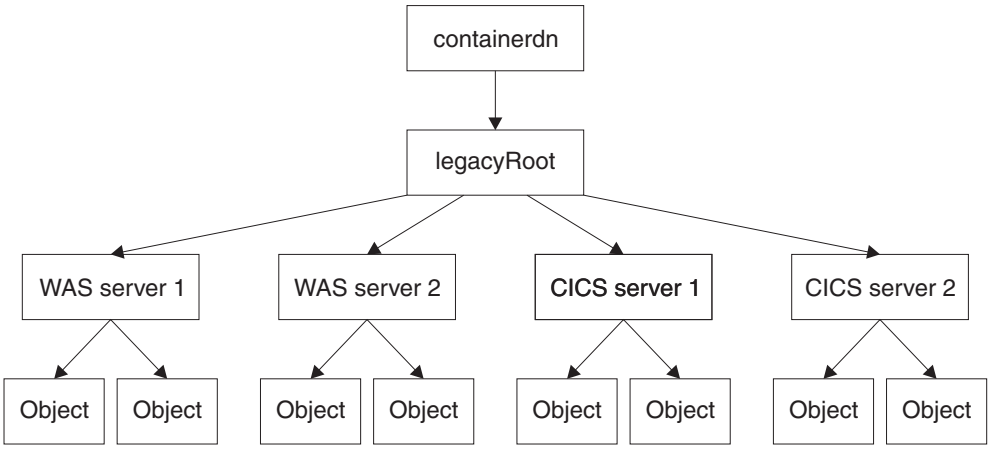

Figure 15. Simplified view of part of an LDAP namespace

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 name space 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 *java.naming.security.principal* and *java.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 *java.naming.security.principal* and *java.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 name space root node”. This involves some additional steps. Ensure that this userid and its password appear in the system properties *java.naming.security.principal* and *java.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 15), 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” on page 177).
- Ensure the CICS runtime principal can write to it.

- Specify the CICS runtime principal and credentials using the system properties *java.naming.security.principal* and *java.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.ibm.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 Name Space 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 Name Space.
- 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 name space, 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. 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 the following definitions in CICS:

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 160
   - 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 189.

   For example:

```
DEFINE TCPIPSERVICE(IIOPNSSL) GROUP(DFH$IIOP)
     DESCRIPTION(IIOP TCPIPSERVICE with no SSL support)
     URM(DFHXOPUS)          BACKLOG(5)             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 *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 need to perform the following CICS tasks:

- "Defining CICS start-up jobstream"
- "Defining CICS resources" on page 181

Defining CICS start-up jobstream

The following parameters must be defined 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 HFS directory containing the JVM profiles that you are using for your applications. "Enabling CICS to locate the JVM profiles and JVM properties files" on page 94 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. 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 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

The following CICS resources must be defined and installed. You can define CICS resources online using CEDA (see the *CICS Resource Definition Guide*); from a CICS application using EXEC CICS CREATE (see the *CICS System Programming Reference*); using the DFHCSDUP offline utility (see the *CICS Operations and Utilities Guide*); or by using CICSPlex SM (see the *CICSPlex System Manager Concepts and Planning*).

`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 180.

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 287.

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/cicsts31/JVMProfiles` HFS directory (where `cicsts31` is the value of the CICS_DIRECTORY variable used by the DFHIJVMJ job during CICS installation). "Setting up JVM profiles and JVM properties files" on page 94 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 178.

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 229.

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 229.

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 190.

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 16 on page 185. 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 the *CICS Resource Definition Guide*).

Figure 16 on page 185 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.

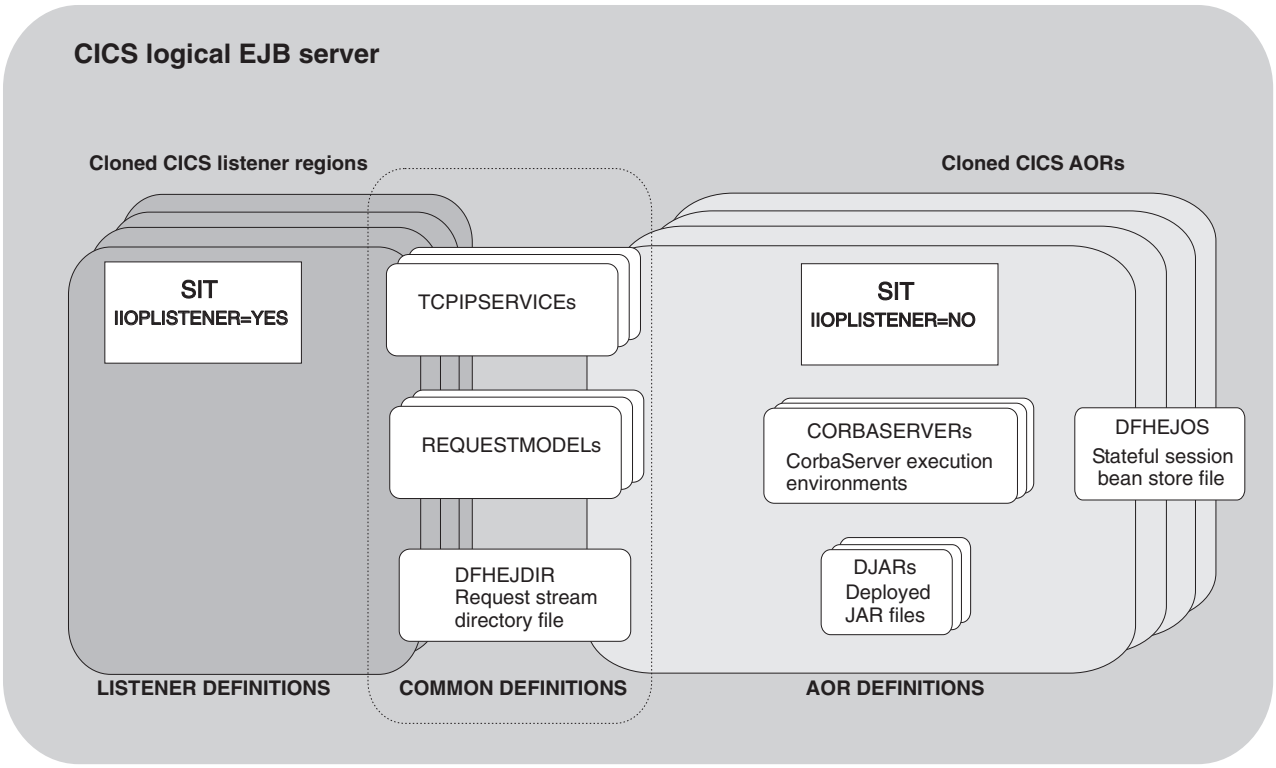

Figure 16. Resource definitions in a CICS logical server. The picture shows which definitions are required in the listener regions, which in the AORs, and which in both.

Chapter 15. 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 the user in the following ways:

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 189 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
(1C)	FULLWORD	4	lBeanName
(20)	FULLWORD	4	BeanInterfaceType

Offset Hex	Type	Len	Name
(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 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 DFLTUSER 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 192.

- 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 192 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 193.

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 neccesary 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

Table 8. CICS specific CORBA minor codes (continued)

Code	CICS component detecting problem
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 Part tells you what you need to know to develop and use enterprise beans in CICS.

Chapter 16. 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[1] to access CICS resources.

1. 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 23, “The CCI Connector for CICS TS,” on page 305.)

Figure 17 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 17 are explained in the sections that follow.

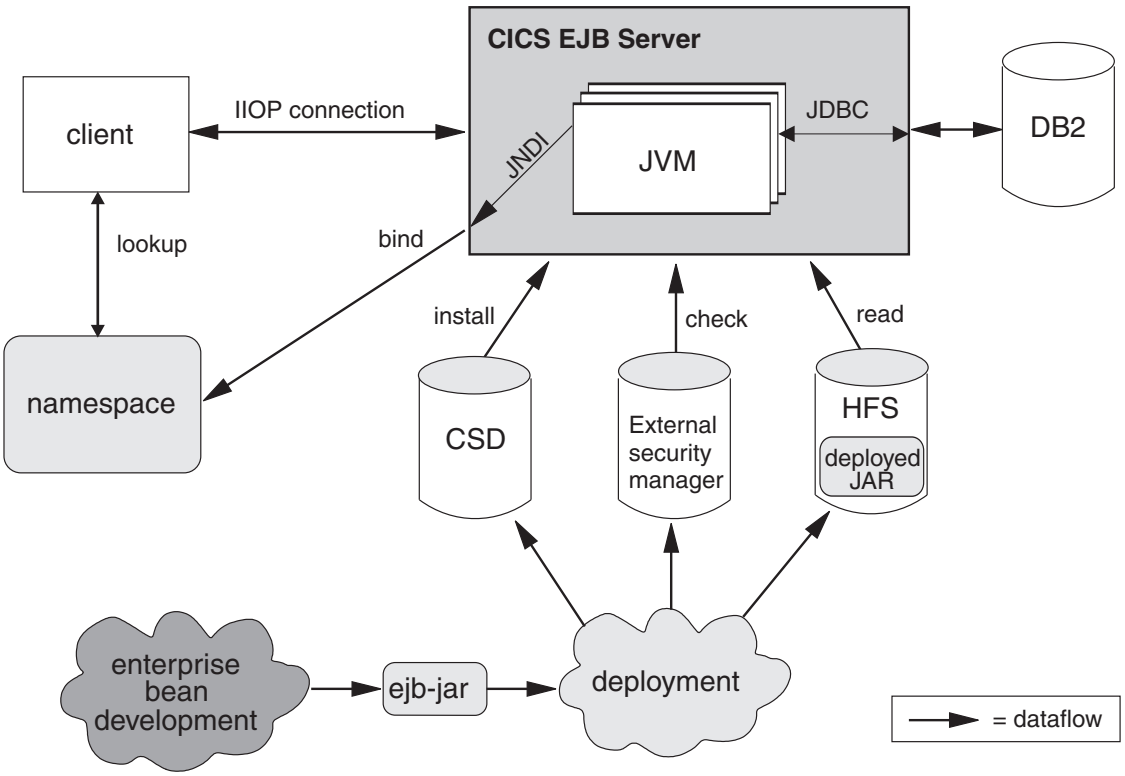

Figure 17. A CICS EJB application server. Enterprise beans developed on a workstation are installed into the EJB server by a process known as deployment. They are executed in a JVM at the request of a client program. The details of this picture are explained in the sections that follow.

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 215.

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 151.)

  CICS Transaction Server for z/OS, Version 3 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).[2]
- 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 lifecycle, state management, security, and transaction management:

`Lifecycle`
: 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 lifecycle 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.

2. The javax.transaction.UserTransaction interface is used by session beans that manage their own transactions, as described later in this chapter.

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 204.) 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 lifecycle, transaction management, security, and persistence are defined in an associated XML document called a **deployment descriptor**. See “Deploying enterprise beans—overview” on page 212.

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.

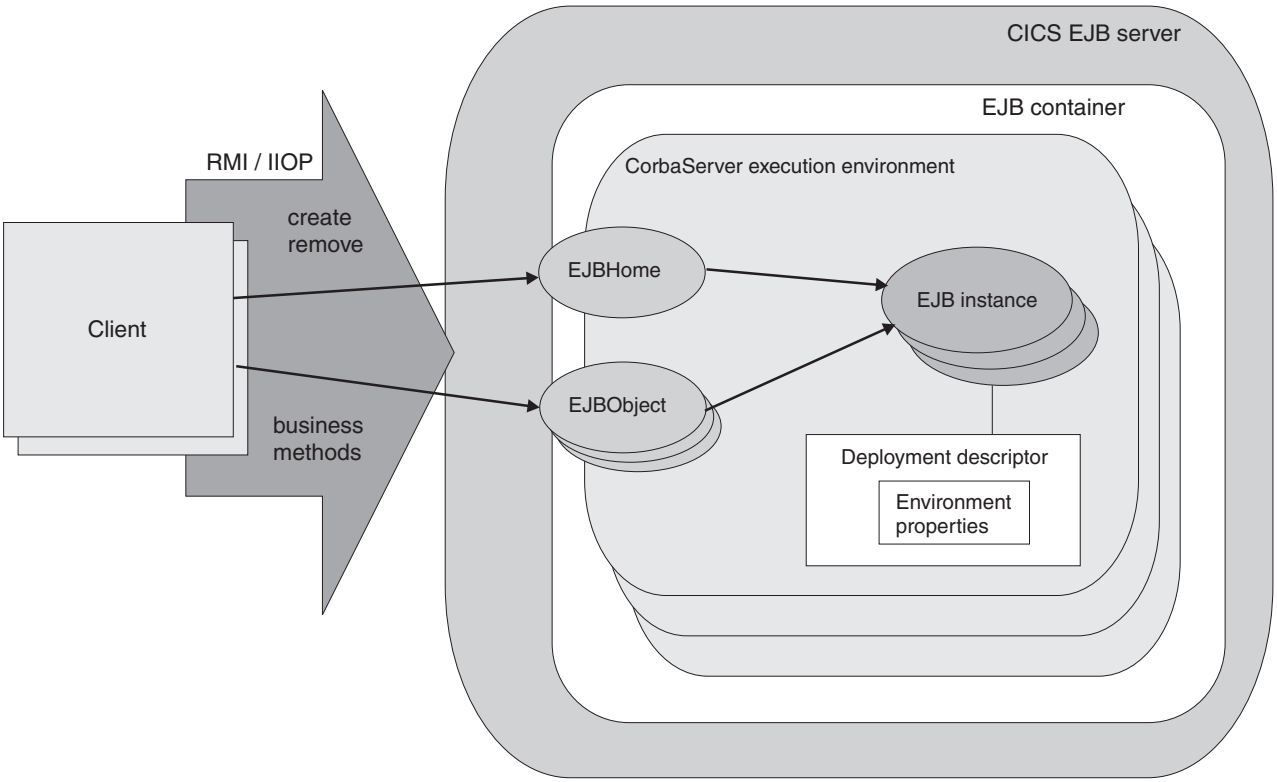

Figure 18. Enterprise bean objects in a CICS EJB server. The EJB container manages and provides services to the enterprise beans contained within it. When a bean is deployed, the deployment tool generates the EJB home and component interface classes.

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 208.
- 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 simply 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

Important

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.

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 208.

- Is recoverable—it survives a server crash.

Session beans and entity beans compared

Table 9 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[3] using an implementation of the Java Transaction Service (JTS) or the CORBA Object Transaction Service (OTS). These transactions 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.

3. Transactions possessing atomicity, consistency, isolation, and durability. Jim Gray and Andreas Reuter, *Transaction Processing: Concepts and Techniques*, 1993.

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 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 the *CICS RACF Security Guide* for information about configuring CICS to use SSL.

Access control

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 337.

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 2 security manager

The security of the enterprise beans container environment is protected by the Java 2 security policy mechanism and is independent of CICS security. The security policy mechanism is one of the components that make up the Java 2 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 behaviour.

To use JDBC or SQLJ from enterprise beans 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 unless you disable the mechanism.

Enterprise beans—user tasks

Typically, several people are involved in the development and deployment of applications that use enterprise beans:

- The bean provider
- The application assembler
- The deployer
- The system administrator

Note: In smaller organizations, one person may be responsible for more than one of these tasks.

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 *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:[4]

1. A **deployment tool** (such as the Assembly Toolkit (ATK), described 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 hierarchical file system (HFS) used by z/OS.

   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 *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.)

4. This simplified description of the deployment process assumes that you're using RDO rather than BAS.

- 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.)

  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 HFS 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 217 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 HFS. 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 19 on page 214 shows the deployment process.

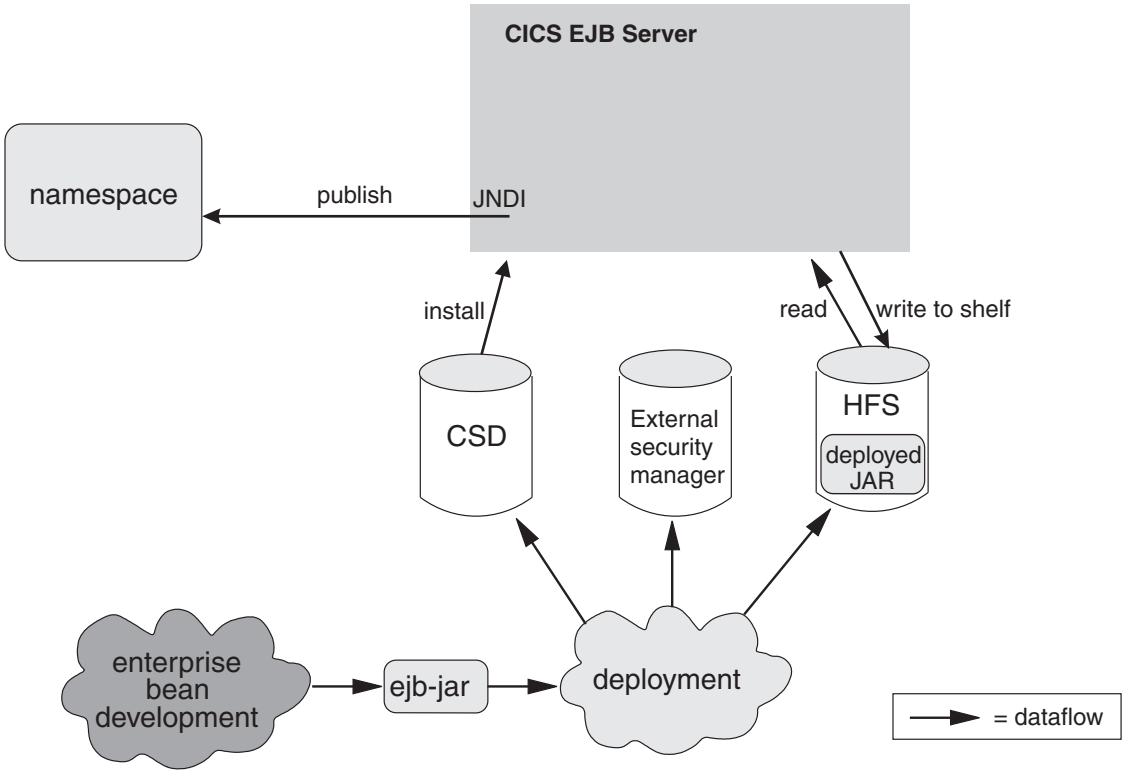

Figure 19. Deploying enterprise beans into a CICS EJB server. A deployment tool is used to perform code generation on the ejb-jar file containing the bean classes. The transformed file is stored as a deployed JAR file on HFS. An RDO definition of the deployed JAR file is created and installed in CICS, 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 the following 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 13, “The IIOP request flow,” on page 155.

Figure 20 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.

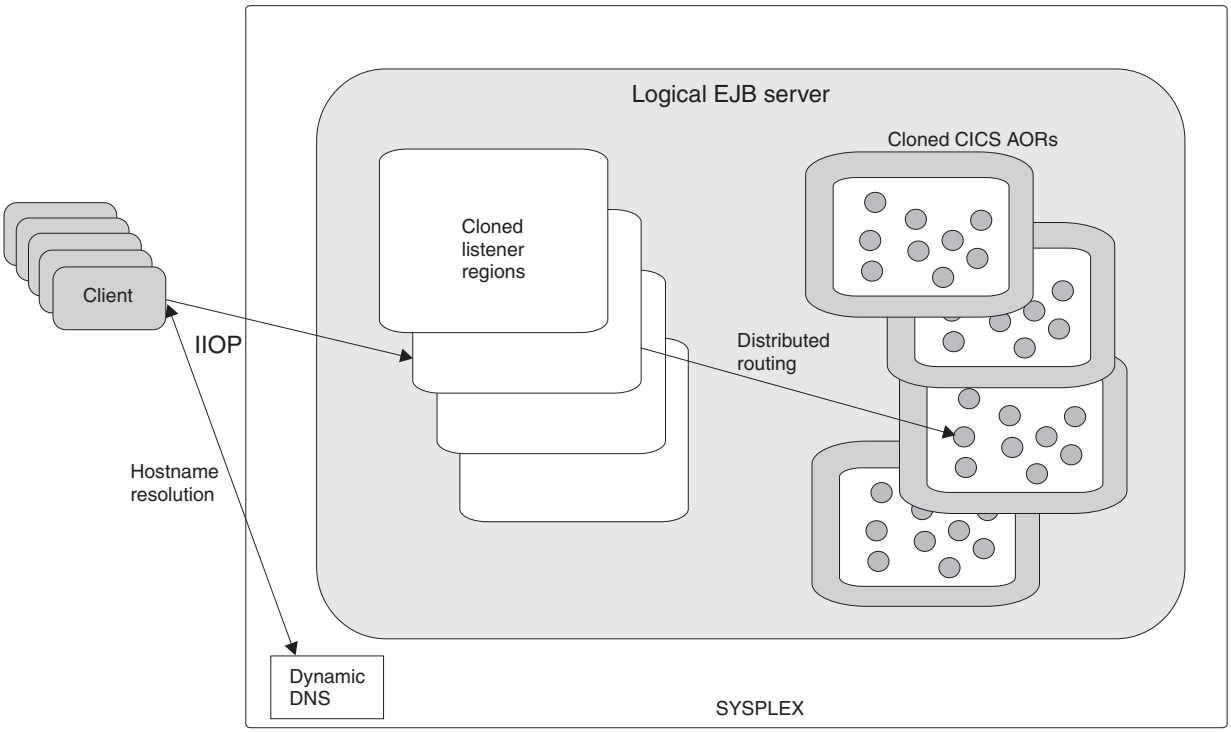

Figure 20. A CICS logical EJB server. The logical server consists of a set of cloned listener regions and a set of cloned AORs. In this example, connection optimization by means of dynamic DNS registration is used to balance client connections across the listener regions. Distributed routing is used 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*, prior to 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*, prior to the current method call; *not* simply 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

Important

It is strongly recommended that all the regions in a logical EJB server—both listeners and AORs—should be at the same level of CICS.

In simplified form, the steps involved in setting up a CICS logical EJB server to support enterprise beans are:

1. Create a set of cloned CICS Transaction Server for z/OS, Version 3 Release 1 listener regions.Each of the listener regions must have the IIOPLISTENER system initialization parameter set to YES.
2. Create a set of cloned CICS Transaction Server for z/OS, Version 3 Release 1 AORs. Each of the AORs must:
   - 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 HFS. For example, you might create an HFS directory called `/var/cicsts/`. To do this, you need an HFS userid with write authority to the directory path to be used by CICS. Having created the shelf directory, you must give the AORs' userids full access to it—read, write, and execute.
4. Create a deployed JAR file (pickup) directory on HFS. For example, you might create an HFS directory called `/var/cicsts/pickup`. The AORs must have at least read access to it.

   Note: If your AORs are to contain more than one CorbaServer execution environment:
   - You must create a separate pickup directory for each CorbaServer.
   - It is recommended that you 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. (To assign transaction IDs to bean methods, you 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 RM for enterprise beans, and the CREA CICS-supplied transaction 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. This means that the service on this port will accept unauthenticated inbound IIOP requests.

- Some REQUESTMODEL definitions. In a single-region EJB server, these are only required if the default TRANSID, CIRP, is unsuitable. In a multi-region logical server, however, they are required if you want to route method requests across several AORs. (The TRANSACTION definition for CIRP specifies DYNAMIC(NO).) They are required too if, for example, you want to segregate your IIOP workload by transaction ID.

  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 HFS. 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.

  Note:

  a. 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 whose JVMClass is com.ibm.cics.iiop.RequestProcessor.

  b. 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 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.

  Note: 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 documentation of the CORBASERVER and TCPIPSERVICE resource definitions.

  On the SHELF option, specify the fully-qualified name of the HFS shelf directory that you created in step 3. (Because the CORBASERVER definition will be installed on all the AORs in the logical server, 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.)

  On the DJARDIR option, specify the fully-qualified name of the HFS 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) will be shared by all the AORs in the logical server. On each AOR, when a CORBASERVER definition is installed, CICS scans

the CorbaServer's pickup directory and installs any deployed JAR files it finds there. It copies them to its shelf sub-directory and dynamically creates and installs DJAR definitions for them.

Specify AUTOPUBLISH(YES). This causes 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 non-recoverable.

  To share DFHEJDIR and DFHEJOS across multiple regions, you could, 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.

  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: For clarity's sake, we're assuming that there's only one CorbaServer in the logical server. To create another CorbaServer, you'll need 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 do this, 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 HFS. Store the deployed JAR files in the CorbaServer's pickup directory.
8. Start all the CICS regions. On each of the listener regions, the definitions to be installed from the CSD are:
   - 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:
   - The TCPIPSERVICE definition.
   - The REQUESTMODEL definitions.

   Note: 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.

- The CORBASERVER definition.
- The file definitions for DFHEJDIR and DFHEJOS.

Note: 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). It is only necessary to create static (CSD-installed) DJAR definitions for deployed JAR files that you place in other HFS 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.

   Note: You can put deployed JAR files in the pickup directory after CICS has performed its initial scan at the time the CORBASERVER definition was installed. If you do so, you can force CICS to perform another scan by issuing a CORBASERVER PERFORM SCAN command. This command can be issued 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 will be automatically bound into the JNDI namespace.

    If you had specified AUTOPUBLISH(NO), you would need to issue a PERFORM CORBASERVER(CorbaServer_name) PUBLISH command on at least one of the AORs. This command can be issued using EXEC CICS, the CEMT master terminal transaction, the RM for enterprise beans, or via a CICSPlex SM EUI or 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'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*.

Figure 21 on page 221 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.

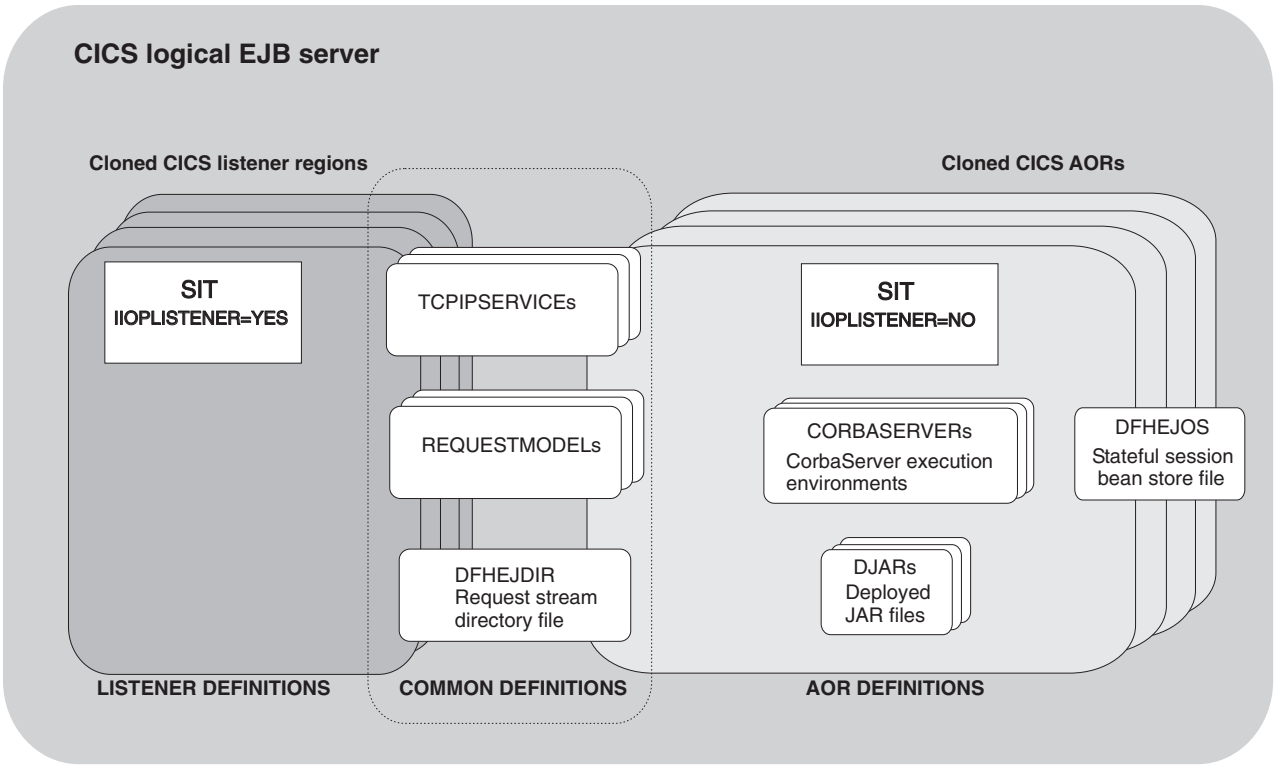

Figure 21. Resource definitions in a CICS logical EJB server. The picture shows which definitions are required in the listener regions, which in the AORs, and which in both.

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
- Delete an instance of the bean

For example:

```
// 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
- Delete the bean

For example:

```
// 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 23, “The CCI Connector for CICS TS,” on page 305.

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

Some of the benefits of using enterprise beans are:

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 behaviour and runtime settings are defined through attributes that can be changed when the enterprise bean is deployed.

Multitier 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 23, “The CCI Connector for CICS TS,” on page 305.

Requirements for EJB support

Hardware

There are no specific hardware requirements for enterprise beans, over and above those for CICS Transaction Server for z/OS, Version 3 Release 1 itself.

Software

The software requirements for enterprise beans are:

- IBM Software Developer Kit for z/OS, Java 2 Technology Edition, Version 1.4.2.

  Note: There are two versions of the IBM Software Developer Kit for z/OS, Java 2 Technology Edition Version 1.4, a 31-bit and a 64-bit version. CICS TS 3.1 supports only the 31-bit version, which must be at the 1.4.2 level.
- 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 OS/390 and 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).

Note: The ATK is included in WebSphere Studio Enterprise Developer Version 5.1, which is shipped with CICS TS 3.1 as a marketing promotion.

Chapter 17. Setting up an EJB server

This chapter contains the following topics:

- "Setting up a single-region EJB server" tells you how to create a minimal CICS EJB server consisting of a single listener/AOR.
- "Testing your EJB server" on page 234 tells you how to check that your single-region EJB server is correctly configured.
- "Setting up a multi-region EJB server" on page 235 tells you how to develop your single-region CICS EJB server into one consisting of multiple listener regions and multiple AORs, that is capable of supporting workload balancing.
- "Migrating an EJB server to CICS Transaction Server for z/OS, Version 3 Release 1" on page 238 tells you how to update a back-level EJB server to CICS TS for z/OS, Version 3.1.

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 235.

Important

- For clarity's sake, we're assuming that:
  1. You start from a basic, non-customized, CICS Transaction Server for z/OS, Version 3 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, and the default JVM properties file, `dfjjvmcd.props`. After you've got your first EJB server up and running, you may want to customize your JVM profile and properties file. How to do this is described in "After running the EJB IVP—optional steps" on page 233.
- 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 21, "Deploying enterprise beans," on page 289.
- The rest of this section is split into two parts:
  - "Before running the EJB IVP" on page 228 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 233 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. Actions are required on:

1. z/OS or Windows NT, depending on the type of name server that you use
2. HFS
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 167.

Actions required on HFS

To perform the tasks in this section, you need an HFS userid with write authority to the directory path to be used by CICS.

Create the following directories on HFS, 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 HFS 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 and HFS directories and files” on page 53 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 an HFS 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 and HFS directories and files” on page 53.
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 an HFS 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 HFS 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 may already have performed some of these steps.

1. Install the IBM Software Developer Kit for z/OS, Java 2 Technology Edition, Version 1.4.2, which provides a Java Virtual Machine (JVM) featuring persistent reusable JVM technology. This is available from `www.s390.ibm.com/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 179.

   Bear in mind when reading “Setting up CICS for IIOP” on page 179 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 HFS directory that is specified by the JVMPROFILEDIR system initialization parameter. When you install CICS, the CICS-supplied default and sample JVM profiles are placed in the directory `/usr/lpp/cicsts/cicsts31/JVMProfiles`, where `cicsts31` is the value that you chose for the CICS_DIRECTORY variable used by the DFHIJVMJ job during CICS installation. The default value of CICS_DIRECTORY is “cicsts31”. The default value of JVMPROFILEDIR is `/usr/lpp/cicsts/cicsts31/JVMProfiles`. That is, the supplied setting for JVMPROFILEDIR points to the default directory for the sample JVM profiles. If you chose a different name during CICS installation for the directory containing the sample JVM profiles (that is, if you chose a non-default value for the CICS_DIRECTORY variable used by the DFHIJVMJ job), or if you have created your own JVM profiles in a directory other than the samples directory, you need to do one of the following:
     – Change the value of the JVMPROFILEDIR system initialization parameter.
     – Link to your profiles from the directory specified by JVMPROFILEDIR by means of UNIX soft links.
   - 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 `com.ibm.cics.ejs.nameserver` property in all the JVM properties files that are used by CORBA applications or enterprise beans—including the `dfjjvmcd.props` properties file that CICS uses to to publish deployed JAR files.

  For detailed information about defining the location of your name server, see the *CICS System Definition Guide*.
- 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 21, “Deploying enterprise beans,” on page 289.
  - 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 235, 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 HFS shelf directory that you created in step 2 of “Actions required on HFS” on page 228.

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 HFS deployed JAR file directory (pickup directory) that you created in step 3 of “Actions required on HFS” on page 228.

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 documentation of the CORBASERVER and TCPIPSERVICE definitions.

– 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) Because we're creating a single-region server, the value of the HOST option of the CORBASERVER definition must match that of the IPADDRESS option of the TCPIPSERVICE definition. (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 `com.ibm.cics.ejs.nameserver` property in your JVM properties file.

   For example, to use `tnameserv`, the lightweight COS Naming Directory Server supplied with Java 1.3 and later, specify:

   ```
   com.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:

```
com.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 168.

If you are using a standard COS Naming Directory Server you might specify:

```
com.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:

```
com.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 `com.ibm.cics.ejs.nameserver` property in the *CICS System Definition Guide*.

The location of the JVM properties file is specified on the JVMPROPS statement in your JVM profile. (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 HFS 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 22, “Updating enterprise beans in a production region,” on page 293.

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 18, “Running the EJB IVP,” on page 245. 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 or more sample JVM profiles and JVM properties files, or create your own JVM profiles and JVM properties files for use with enterprise beans, rather than using the default JVM profile DFHJVMCD. 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 and JVM properties files” on page 94 tells you how to select a suitable JVM profile and JVM properties file and customize them, or if you prefer, how to create your own JVM profile and JVM properties file based on one of the supplied sample profiles. Follow the procedures in that section to customize or create your JVM profile and JVM properties file.

When you have customized or created your JVM profile and JVM properties file, in order for them 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 181, 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 HFS directory specified by the JVMPROFILEDIR system initialization parameter.

Important: You must specify the location of your name server on the `com.ibm.cics.ejs.nameserver` property in all the JVM properties files that are used by CORBA applications or enterprise beans—including the `dfjjvmcd.props` properties file that CICS uses to to publish deployed JAR files. For detailed information about defining the location of your name server, see the *CICS System Definition Guide*.

Testing your EJB server

This section tells you how to check that your single-region CICS EJB server is configured correctly. It contains:

- "Running the EJB IVP"
- "Using the EJB "Hello World" sample"
- "Using the EJB Bank Account sample" on page 235
- "Using your own enterprise beans" on page 235

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 228. You may or may not have completed the steps in "After running the EJB IVP—optional steps" on page 233. 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 18, "Running the EJB IVP," on page 245.

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 228. You may or may not have completed the steps in "After running the EJB IVP—optional steps" on page 233.

For details of the EJB "Hello World" application, and instructions on how to install it, see "The EJB "Hello World" sample application" on page 251.

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 228. You may or may not have completed the steps in "After running the EJB IVP—optional steps" on page 233.

For details of the EJB Bank Account application, and instructions on how to install it, see "The EJB Bank Account sample application" on page 259.

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 21, "Deploying enterprise beans," on page 289.

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. It assumes that you have already created a single-region EJB server, as described in "Setting up a single-region EJB server" on page 227.

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.

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 22 on page 238 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 158.
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 26, “CICSPlex SM with enterprise beans,” on page 347.
   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 22 on page 238, 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.

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 22, “Updating enterprise beans in a production region,” on page 293.

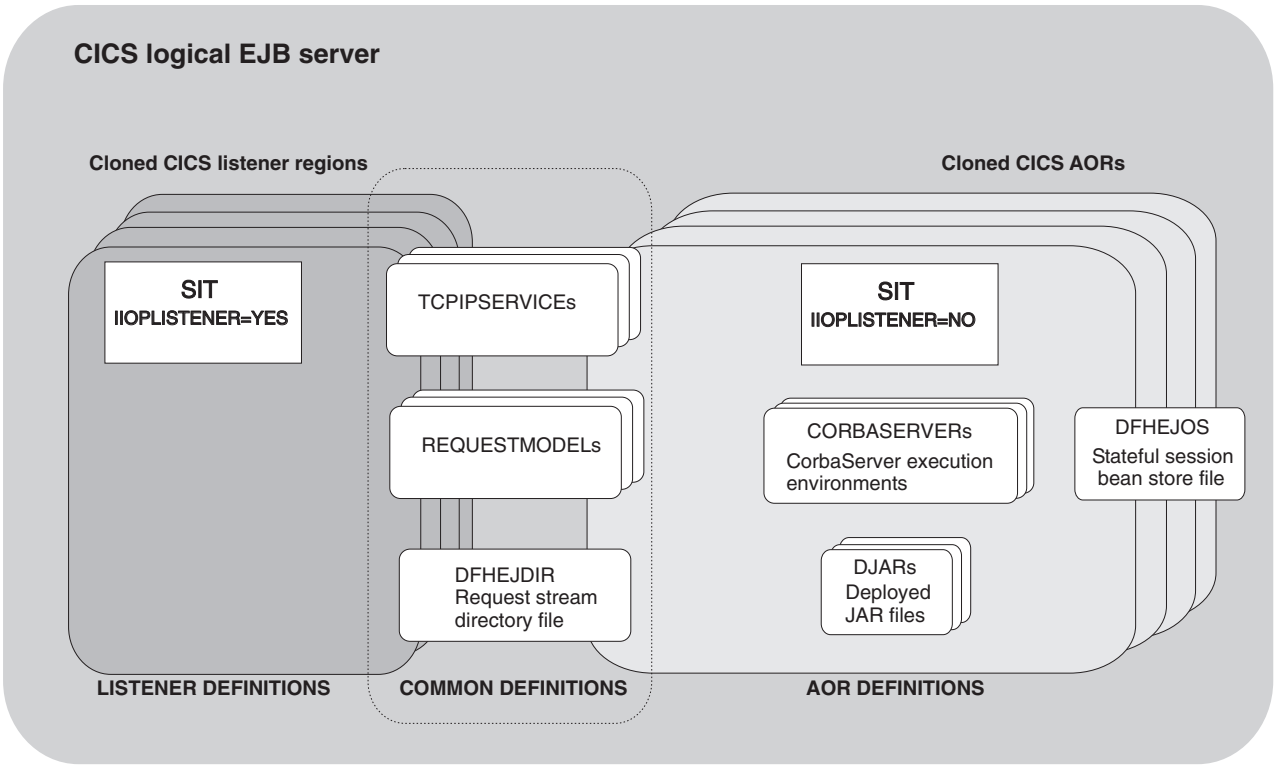

Figure 22. Resource definitions in a multi-region CICS EJB server. The picture shows which definitions are required in the listener regions, which in the AORs, and which in both.

Migrating an EJB server to CICS Transaction Server for z/OS, Version 3 Release 1

Upgrading a single-region CICS EJB/CORBA server

To migrate a single-region CICS EJB/CORBA server to CICS Transaction Server for z/OS, Version 3 Release 1:

1. Quiesce the workload.
2. Shut down the region.
3. Upgrade the region to CICS Transaction Server for z/OS, Version 3 Release 1, following the standard migration procedures described in *CICS Transaction Server for z/OS Migration from CICS TS Version version_number*, where `version_number` is the version number of your back-level CICS release.
4. Review “Migration tips” on page 243, which describes some of the changes in EJB/CORBA support between CICS TS for z/OS, Version 2.2 and CICS Transaction Server for z/OS, Version 3 Release 1. You should also refer to “Setting up a single-region EJB server” on page 227, which describes in detail how to set up a single-region EJB server in CICS TS for z/OS, Version 3.1.
5. Restart the region.
6. Republish the Interoperable Object References (IORs) for all the enterprise beans and stateless CORBA objects processed by the server. To do this, issue a PERFORM CORBASERVER(*CorbaServer_name*) PUBLISH command. This command can be issued using EXEC CICS, CEMT, the Resource Manager for

enterprise beans, or from a CICSPlex SM EUI or WUI view. Remember to issue a separate command for each CorbaServer in the region.

Upgrading a multi-region CICS EJB/CORBA server

To migrate a multi-region CICS EJB/CORBA server to CICS Transaction Server for z/OS, Version 3 Release 1, you can use any of the following methods:

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 238, except that:

   a. You must upgrade all the regions to CICS Transaction Server for z/OS, Version 3 Release 1 before restarting the server. Again, follow the standard migration procedures described in *CICS Transaction Server for z/OS Migration from CICS TS Version version_number*, where `version_number` is the version number of your back-level CICS release.

   b. You should refer to “Setting up a multi-region EJB server” on page 235, which describes in detail how to set up a multi-region EJB server in CICS TS for z/OS, Version 3.1.

   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 3.1, logical server and gradually migrate 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 3.1 server, perhaps consisting of just two regions—one listener and one AOR. As more applications are migrated, you can expand the CICS TS for z/OS, Version 3.1 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 3.1 multi-region EJB server, follow all the steps in “Setting up a single-region EJB server” on page 227 and “Setting up a multi-region EJB server” on page 235.

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”” on page 240.

Performing a “rolling upgrade”
Important

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. 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 3 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. “Migrating the listener regions” on page 241
4. “Migrating the AORs” on page 241
5. “Tidying up” on page 243

Requirement:

Your server must consist of separate listener and application-owning regions. This is because the migration 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:

1. Review “Migration tips” on page 243.
2. If you are migrating 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 3.1 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 3.1 request receiver (listener).
3. Set the AUTOPUBLISH option on all your CORBASERVER definitions to `NO`. Setting a CorbaServer to autopublish IORs into the JNDI name spaces could disrupt the migration 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 a migration aid. 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 3.1 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 3 Release 1 checks the DYRLEVEL field and routes requests accordingly.

Migrating the listener regions:

1. Quiesce a listener region and bring it down.
2. Upgrade this single listener region to CICS Transaction Server for z/OS, Version 3 Release 1, following the standard migration procedures described in *CICS Transaction Server for z/OS Migration from CICS TS Version version_number*, where `version_number` is the version number of your back-level CICS release.

   Important:

   a. If you upgrade a CSD from CICS TS **2.2** to CICS TS for z/OS, Version 3.1 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 3.1) 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 3.1 CSD.

      This step is necessary because, in CICS TS for z/OS, Version 3.1, 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 3.1-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 3.1, refer to Chapter 14, “Configuring CICS for IIOP,” on page 165.
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.

Migrating the AORs:

1. Quiesce an AOR and bring it down.
2. Update this single AOR to CICS Transaction Server for z/OS, Version 3 Release 1, following the standard migration procedures described in *CICS Transaction Server for z/OS Migration from CICS TS Version version_number*.

   If you are migrating 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 “Migration tips” on page 243.

   Important:

   a. If you upgrade a CSD from CICS TS **2.2** to CICS TS for z/OS, Version 3.1 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 3.1) in the startup group list of these regions.

b. At this stage, don't enable any new, CICS TS for z/OS, Version 3.1-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 3.1 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 work load can be processed by a single region.
   - Wherever possible, all the beans used by an application should be migrated at the same time. For example, if bean A is known to call bean B the two beans should be migrated together. If this is not possible, bean A should be migrated first.

     This is particularly important if you are migrating 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 3.1 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 3.1 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 migrated separately.

   Re-publish the selected DJARs to the JNDI name space, 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 3.1 AORs.
   - Any clients that look up the IOR of a re-published bean in the name space get the new IOR in CICS TS for z/OS, Version 3.1 format. Your customized routing program or CICSPlex SM directs such requests to the CICS TS for z/OS, Version 3.1 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 3.1 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 3.1 regions.

- It becomes less important, when selecting deployed JAR files to re-publish, to choose those whose entire work load can be processed by a single region—because there are more AORs in the CICS TS for z/OS, Version 3.1 pool.

Eventually, all the AORs will be running CICS TS for z/OS, Version 3.1 and processing 100% of the workload.

Tidying up:

1. If required, reset the AUTOPUBLISH option on your CORBASERVER definitions to `YES`.
2. Enable any CICS TS for z/OS, Version 3.1-specific resource definition options that you want to use.

Migration tips

This section briefly lists some of the ways in which EJB and Java support has changed between CICS TS for z/OS, **Version 2.2** and CICS Transaction Server for z/OS, Version 3 Release 1. All these changes are described in detail in Chapter 9, "Setting up Java support," on page 53. They are listed here, together with some general tips, as a reminder of things to be aware of when migrating an EJB server to CICS TS for z/OS, Version 3.1.

1. In CICS TS 2.2, JVM profiles were stored in a PDS member. In all later releases, including CICS TS for z/OS, Version 3.1, they are stored in the HFS directory pointed to by the JVMPROFILEDIR system initialization parameter.
2. The default JVM profile used by CorbaServers in CICS TS 2.2 was DFHJVMPR. In all later releases, including CICS TS for z/OS, Version 3.1, it is DFHJVMCD.
3. The default JVM properties file used by CorbaServers in CICS TS 2.2 was `dfjjvmpr.props`. In all later releases, including CICS TS for z/OS, Version 3.1, it is `dfjjvmcd.props`.
4. Don't enable any new, CICS TS for z/OS, Version 3.1-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.
5. From a CICS TS for z/OS, Version 3.1 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 3.1 format. **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 3.1 AOR you must first retract it; furthermore, because earlier CICS releases do not understand the format of CICS TS for z/OS, Version 3.1 IORs, *you must retract it from a CICS TS for z/OS, Version 3.1 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 3.1 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 3.1 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 migrated to CICS TS for z/OS, Version 3.1, 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 for z/OS, Version 3.1 and CICS TS 2.3 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 3.1 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 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 3.1 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 240, the problem (of a GIOP 1.2 message being received in a CICS TS 2.2 region) will not occur.
3. CICS TS for z/OS, Version 3.1 and CICS TS 2.3 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 3.1 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 240, this problem will not occur.

Chapter 18. Running the EJB IVP

The EJB Installation Verification Program (IVP) is a simple application that CICS installers can use to verify the CICS EJB environment. It uses a simple 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 simple enterprise bean.
- HFS 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

The rest of the chapter contains the following topics:

Prerequisites for the EJB IVP

To run the EJB IVP, you need:

- 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 227.
- 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 167. Alternatively, you can use the lightweight `tnameserv` COS Naming Server supplied with Java.

Note:

1. We're assuming that you're testing a single-region CICS EJB server.
2. For the purposes of running the IVP, you need only to have completed the steps in "Before running the EJB IVP" on page 228. You may or may not have completed the steps in "After running the EJB IVP—optional steps" on page 233.
3. Before starting, make sure 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

Installing the EJB IVP requires actions on:
1. HFS
2. CICS
3. The client, on z/OS UNIX System Services

HFS setup

The IVP uses the same CICS enterprise bean as the EJB "Hello World" sample application described in "The EJB "Hello World" sample application" on page 251. Thus, on HFS, 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 228.

Note: Both the source and executable code of the enterprise bean is in the `HelloWorldEJB.jar` file.

The samples directory is: `/usr/lpp/cicsts/cicsts31/samples/ejb/helloworld`, where `cicsts31` is the value of the CICS_DIRECTORY variable used by the DFHIJVMJ job during CICS installation.

Remember that HFS names are case-sensitive.

CICS setup

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 229.
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.

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/cicsts31/samples/ejb/helloworld
   ```

   where `cicsts31` is the value of the CICS_DIRECTORY variable used by the DFHIJVMJ job during CICS installation.
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 and JVM properties file.)
   a. Modify the JAVA_HOME variable to your IBM SDK 1.4.2 installation directory, as indicated by the comments in the script. The line to be changed is:

      ```
      JAVA_HOME=/usr/lpp/<Java SDK 1.4.2 installation directory>/J1.4
      ```

   b. Modify the CICS_DIRECTORY variable to your CICS installation directory, as indicated by the comments in the script. The line to be changed is:

      ```
      CICS_DIRECTORY=/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

First, check that the name server is running.

Note: 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.

Next, 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 "1.4.2"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.2)
Classic VM (build 1.4.2, J2RE 1.4.2 IBM z/OS Persistent Reusable VM build
 cm142-yyyymmdd (JIT enabled: jitc))
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:

1. In the above messages, **yyyymmdd** is the date on which the SDK was built.
2. In this example, a COS Naming Server has been used. If you use an LDAP name server, similar messages are produced.

3. 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.

When you have finished running the IVP, you should:

1. Discard the resource definitions that you created in `mygroup`.
2. 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 19. Running the sample EJB applications

Important

The sample EJB applications require a CICS EJB server. You must configure CICS, as described in Chapter 17, “Setting up an EJB server,” on page 227, 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 18, “Running the EJB IVP,” on page 245.

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 259.

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 23 on page 252 shows the components of the sample application.

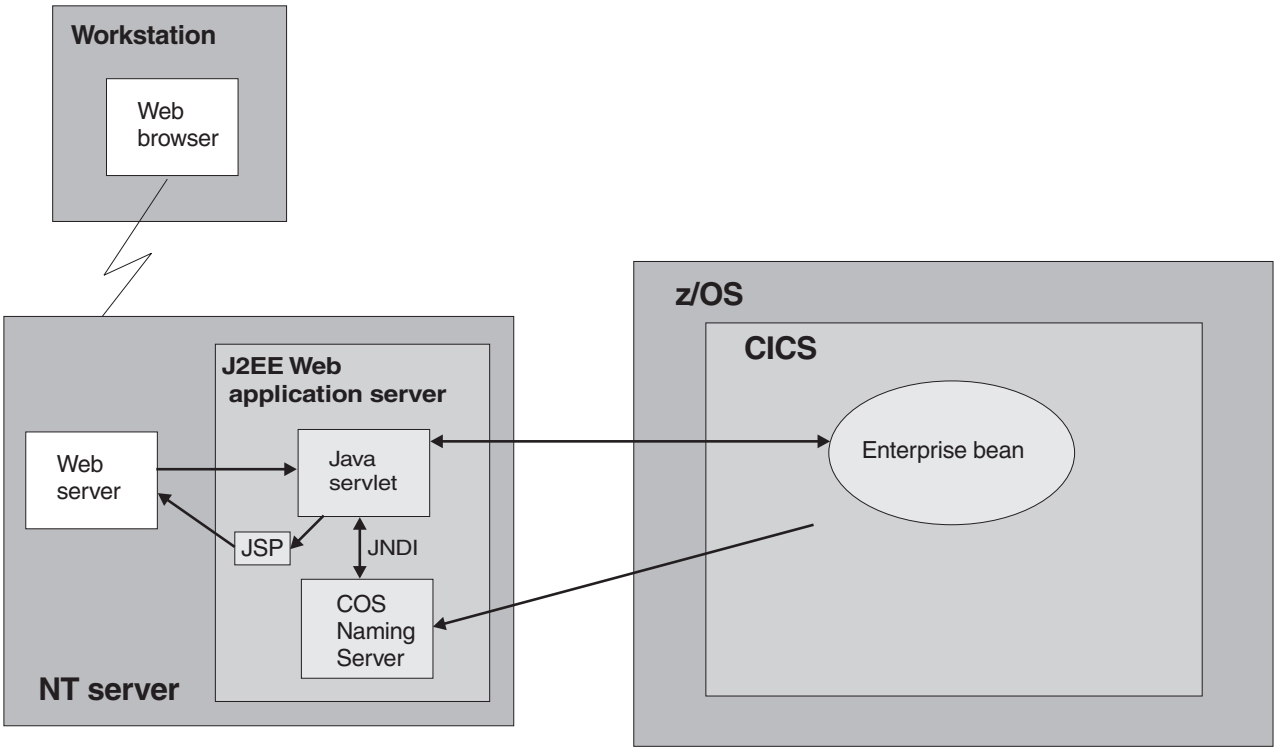

Figure 23. Overview of the EJB "Hello World" sample application. The main elements of the sample are a Java servlet and an enterprise bean. In this example, the servlet is running in a Web application server on a Windows NT server; a COS Naming Server is used. Other configurations are possible. For example, 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

To run the EJB "Hello World" sample, you need:

- A CICS EJB server. The way to set one up is described in Chapter 17, "Setting up an EJB server," on page 227.
- 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 228.

Supplied components of the EJB "Hello World" sample

Table 10 lists the files 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	HFS samples 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.
`HelloWorldCLI.jar`	JAR file	HFS samples directory: see Note.	Client EJB stubs required by the servlet.

Table 10. Supplied components of the EJB "Hello World" sample (continued)

Filename	Type	Default location	Comments
`HelloWorldEJB.jar`	Deployed JAR file	HFS 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	HFS 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 HFS samples directory is `/usr/lpp/cicsts/cicsts31/samples/ejb/helloworld` where `cicsts31` is the value of the CICS_DIRECTORY variable used by the DFHIJVMJ job during CICS installation.			

Installing the EJB "Hello World" sample

Installing the EJB "Hello World" sample requires actions on:

1. HFS. 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.

HFS setup

If necessary, on HFS 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 228 and specified on the DJARDIR option of the CORBASERVER definition.
3. The samples directory is: `/usr/lpp/cicsts/cicsts31/samples/ejb/helloworld`, where `cicsts31` is the value of the CICS_DIRECTORY variable used by the DFHIJVMJ job during CICS installation.
4. Remember that HFS names are case-sensitive.
5. The `HelloWorldEJB.jar` file contains both the source and executable code for the enterprise bean.

CICS setup

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 229.

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. From the HFS 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/cicsts31/samples/ejb/helloworld
```

where cicsts31 is the value of the CICS_DIRECTORY variable used by the DFHIJVMJ job during CICS installation.

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

To test the application:

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 24 on page 256 appears.

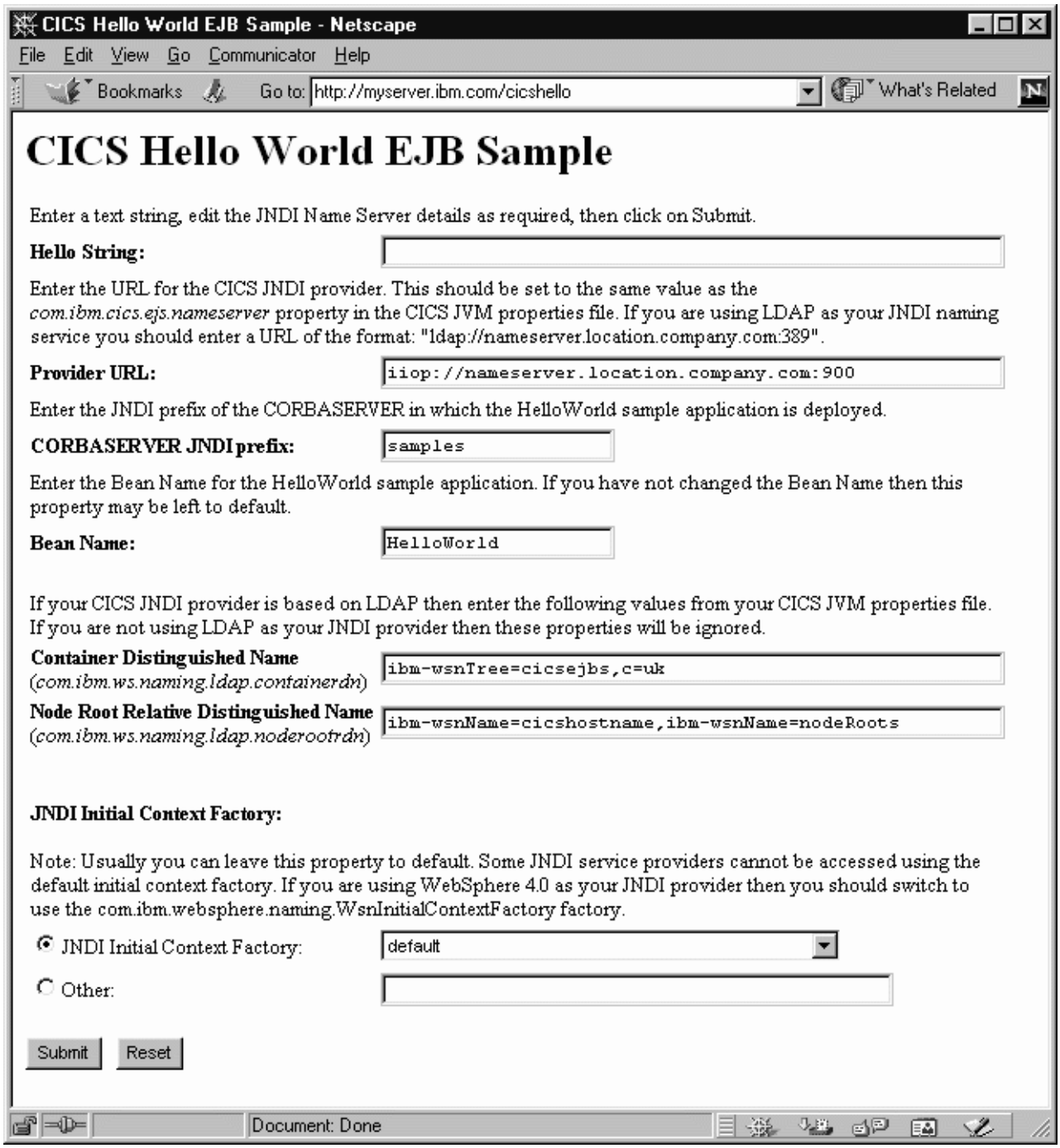

Figure 24. Opening screen of the EJB "Hello World" sample application

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 com.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:

  com.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 com.ibm.cics.ejs.nameserver property in the *CICS System Definition Guide*.

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 com.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 com.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 258 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 23, “The CCI Connector for CICS TS,” on page 305.
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 25 on page 260 shows the components of the sample application.

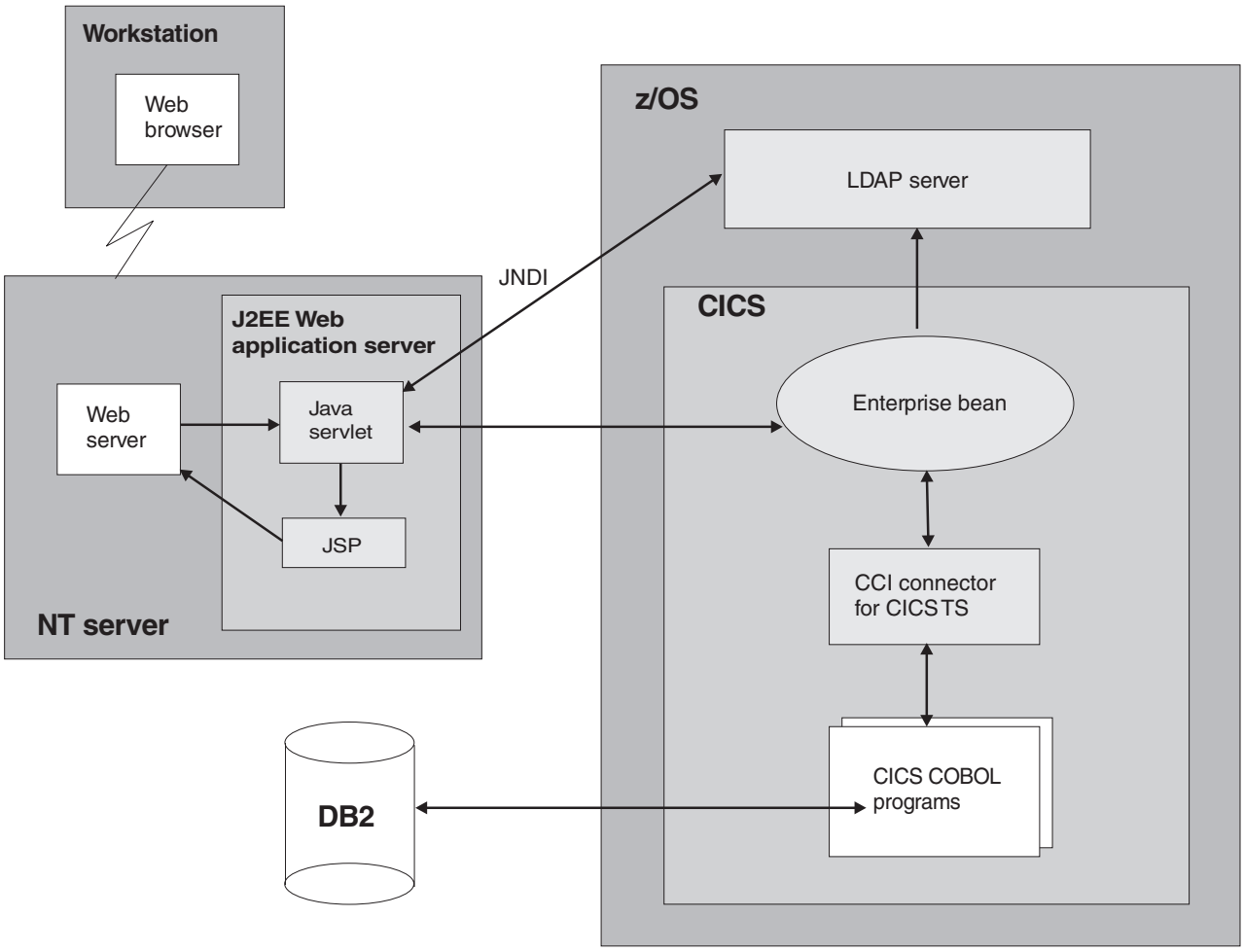

Figure 25. Overview of the EJB Bank Account sample application. The main elements of the sample are a Java servlet, an enterprise bean, two CICS server programs, and two DB2 data tables. The sample extracts customer details from the data tables and returns them to the user. In this example, the servlet 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

To run the EJB Bank Account sample, you need:

- A CICS EJB server. The way to set one up is described in Chapter 17, “Setting up an EJB server,” on page 227.
- DB2 Version 5 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 228.

Supplied components of the EJB Bank Account sample

Table 12 lists the files 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	HFS samples directory: see Note.	The Web components of the sample application—Java servlet classes and source files; HTML and JSPs.
`readme.txt`	Text file	HFS 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	HFS samples directory: see Note.	Client EJB stubs required by the servlet.
`SampleEJB.jar`	Deployed JAR file	HFS 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 HFS samples directory is `/usr/lpp/cicsts/cicsts31/samples/ejb/bankaccount` where `cicsts31` is the value of the CICS_DIRECTORY variable used by the DFHIJVMJ job during CICS installation.			

Security of the EJB Bank Account sample

We recommend 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 the IBM Asserted Identity protocol to allow Web-tier client applications running in WebSphere Application Server for z/OS to propagate their existing userids 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 the *CICS RACF Security Guide*.

Whichever authentication method you choose, you need (among other things) to:

1. Provide authorisation information in the deployment descriptor of the enterprise bean in CICS. This authorisation 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 authorised 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, please 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). 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:

1. Copy the `SampleEJB.jar` file from the HFS samples directory to your workstation. You can do this using FTP in binary mode, or any other method of your choice. The HFS samples directory is `/usr/lpp/cicsts/<cics_name>/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/<cics_name>/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.

   At this point, in order to familiarise yourself with ATK, you can browse through the contents of the JAR file. For more information about the EJB deployment descriptor, see “Enterprise beans—the deployment descriptor” on page 204.
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 HFS. 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:

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.

   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. 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 343.)

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 189.

   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 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 *CICS Resource Definition Guide*.

- If you are using the IBM Asserted Identity protocol for encryption, authentication, and identity propagation, you must:
  - Configure WebSphere Application Server for z/OS to authenticate users.
  - 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 security.

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:

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 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$EJB2 also contains a CORBASERVER and a TCPIPSERVICE definition. However, these are for reference only. It is strongly recommended that you set up your EJB server, as described in Chapter 17, "Setting up an EJB server," on page 227, *before* attempting to install the sample programs. If you do this, you don't need the CORBASERVER and TCPIPSERVICE definitions in DFH$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 229.

      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 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 313.
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. From the HFS 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/cicsts31/samples/ejb/bankaccount
```

where `cicsts31` is the value of the CICS_DIRECTORY variable used by the DFHIJVMJ job during CICS installation.

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 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.

Testing the EJB Bank Account sample

To test the application:

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 26 on page 270 appears.

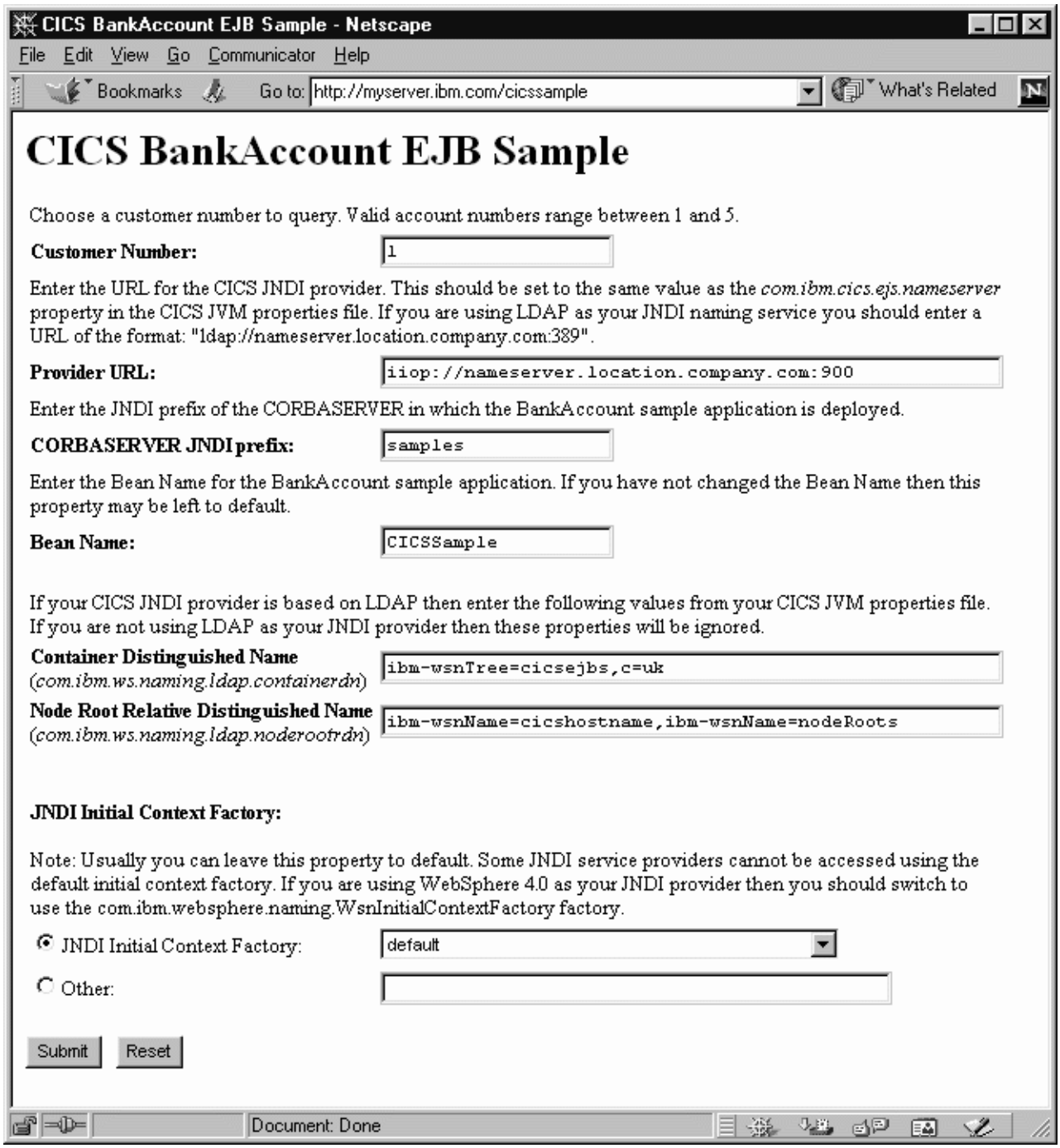

Figure 26. Opening screen of the EJB Bank Account sample application

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 com.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:

  com.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 com.ibm.cics.ejs.nameserver property in the *CICS System Definition Guide*.

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 com.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 com.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 13 on page 272 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`
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 27 on page 273 shows the result of a successful enquiry.

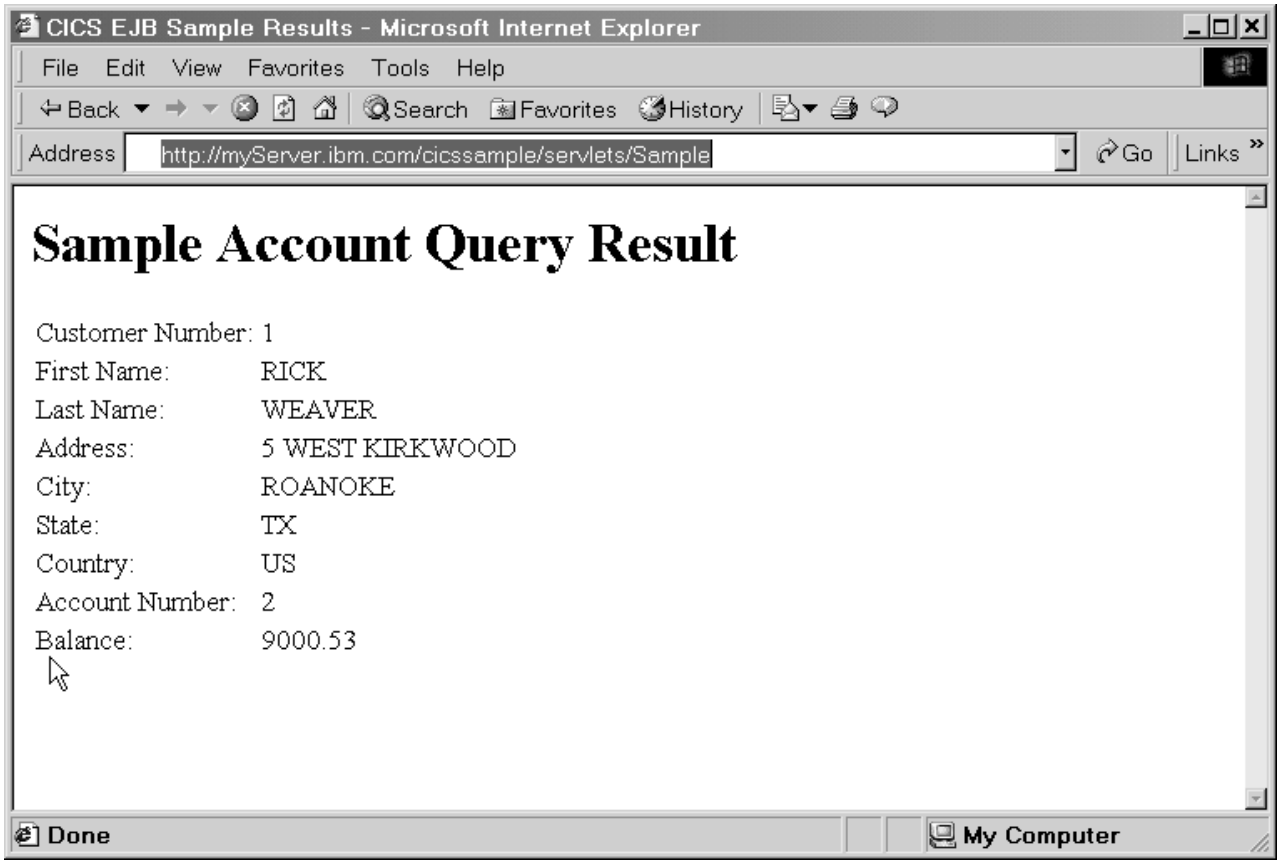

Figure 27. Results screen of the EJB Bank Account sample application

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.

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 313.

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 20. Writing enterprise beans

You can write session beans that use the interfaces defined by Sun Microsystem's *Enterprise JavaBeans Specification, Version 1.1*, which is described at `http://www.javasoft.com/products/ejb`. The interfaces used by these beans are mapped to CICS services and resources and the beans are portable to any other EJB-compliant server.

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 1.4 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. (Currently, CICS supports SDK 1.4.2.)
- 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 289.

“Coding a session bean” on page 276 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 23, “The CCI Connector for CICS TS,” on page 305 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 21, “Deploying enterprise beans,” on page 289 for a description of the deployment process, and “Using CICS deployment tools for enterprise beans” on page 290 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 21, “Deploying enterprise beans,” on page 289 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 277.

Then you need to compile and package your program:

1. “Compiling the code” on page 279
2. “Packaging the code” on page 279

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 276 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. 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 166 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 simply 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 28 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.

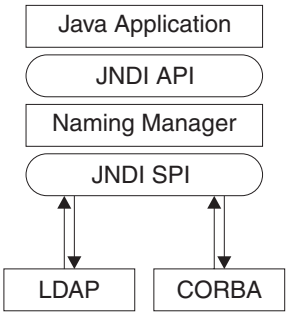

Figure 28. JNDI structure

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.dital . An overview is available at http://www.javasoft.com/products/jndi/tutorial/getStarted/overview/index.dital .

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 168 and “Setting up a COS Naming Directory Server” on page 178, and for details of the JVM properties that are needed, see the *CICS System Definition Guide*

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 name space (that is the structured name space 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 name space 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

The next listing shows 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.hursley.ibm.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 whilst looking up and calling the HelloWorld bean:");
      e.printStackTrace();
     }
  }
}
```

As noted in the the comments, the first four properties are fixed, the remaining three match settings for your CICS region (Albeit the com.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 HFS 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 **com.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 HFS 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.hursley.ibm.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.hursley.ibm.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 whilst 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 276. 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

```
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.wobble.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.wobble.com:900". This value should be the name of your nameserver, and must match the java.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 166 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

```
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 whilst 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, simply 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 273.)

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 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 wishes 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, and
- 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 HFS area where CICS is installed. This jar 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 implentation of the client's object request broker (ORB).

Using EDF with enterprise beans

To use EDF to test enterprise beans, you must:

- 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 21. Deploying enterprise beans

The concept of deployment is introduced in “Deploying enterprise beans—overview” on page 212. This section explains the process of deploying enterprise beans into a CICS EJB server in more detail.

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”

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 5.0 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 *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 *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 the *CICS Supplied Transactions* manual

CREA and CREC can be used without needing to access a 3270 terminal. For details of such access, see 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:

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 *CICS Operations and Utilities Guide*.

Store in HFS pickup directory

Store a copy of the deployable JAR file in the HFS pickup directory of the CorbaServer in which you want to run the bean. You can do this using FTP, NFS, or SMB. If the HFS 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 *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 *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 shareable application class path that will be used by the JVM for the request processor program. “Enabling applications to use a JVM” on page 119 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 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 HFS directory that you intend to use in the production region.
   b. Install the production CORBASERVER definition.
   c. Create an 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 shareable application class path that will be used by the JVM for the request processor program in your production system.

Note: If you want to update enterprise beans in a production region, see Chapter 22, “Updating enterprise beans in a production region,” on page 293.

Chapter 22. 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 HFS 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 29 on page 295. 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.

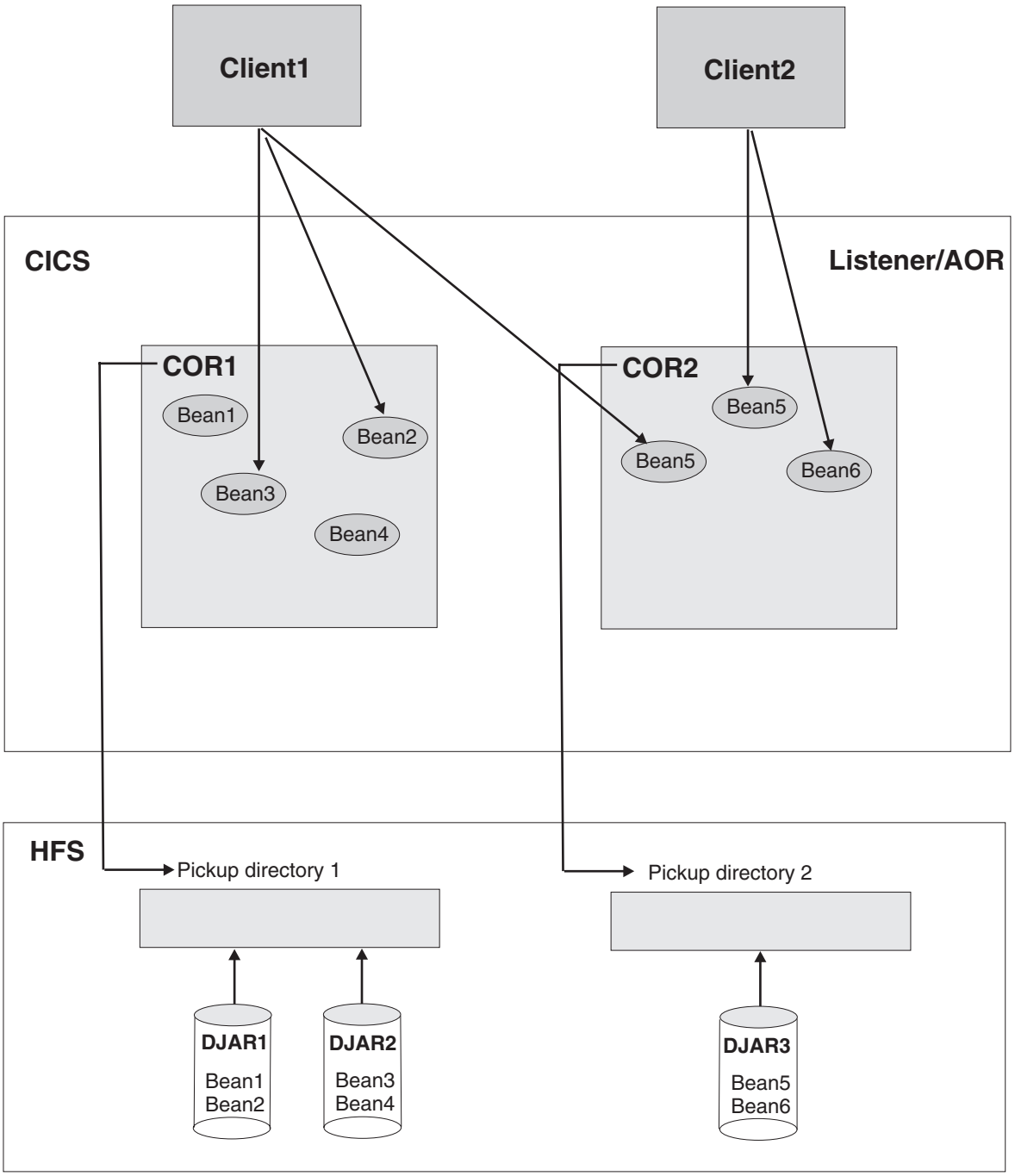

Figure 29. A CICS EJB production region. The clients are invoking bean methods in CorbaServers COR1 and COR2.

You are recomended 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 29 on page 295, 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

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) 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.

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 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 29 on page 295 (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. 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

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 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 296 and "Solutions for a multi-region EJB server" on page 300 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 23. The CCI Connector for CICS TS

This chapter describes the CCI Connector for CICS TS. It covers the following topics:

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-proprietory, 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 "Migrating from the CICS Connector for CICS TS to the CCI Connector for CICS TS" on page 320.

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.dital`.

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 30 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 30. 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 31 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 31. 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 32. 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.

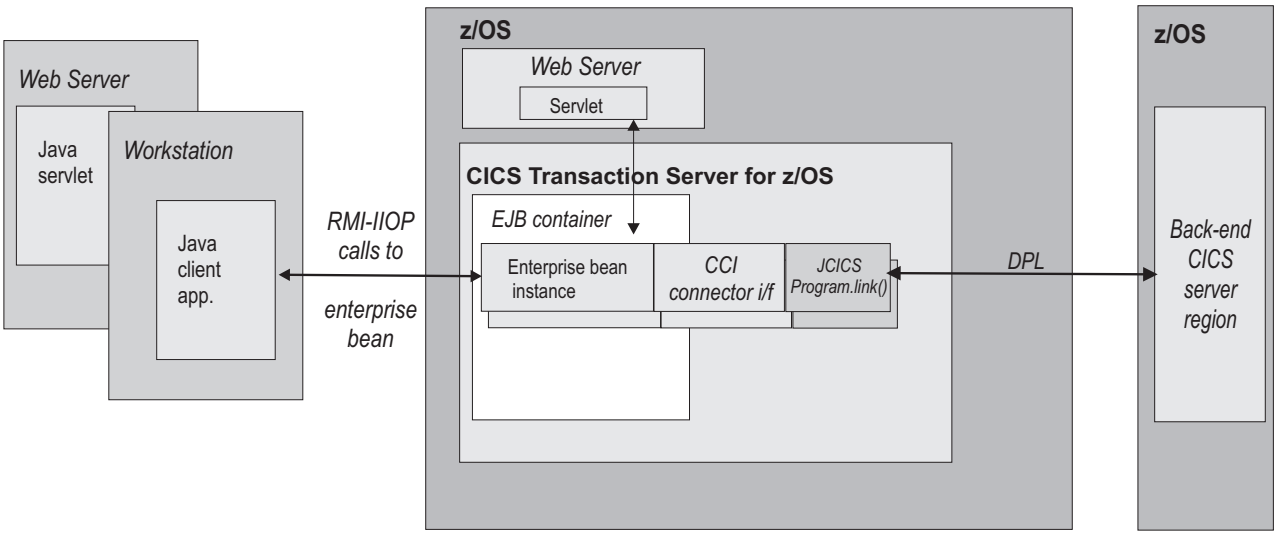

Figure 32. A CICS enterprise bean uses the CCI Connector for CICS TS to connect to a CICS server program.

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

1. The CCI Connector for CICS TS helps you to build powerful server components that make use of existing CICS programs.
2. 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.

3. 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.
4. 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.
5. Using the connector from a CICS session bean results in a simple, two-tier deployment model: Client → CICS TS.
6. 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.
7. 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 317.
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 259.

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 315.

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 316.

Using the CCI Connector for CICS TS

CICS Java components that use the CCI Connector for CICS TS can be programmed in two ways. You can:

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 31 on page 307. 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 313.)
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 the following 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` HFS 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/cicsts31/lib` directory (where `cicsts31` 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

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 315.

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 316.
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 317.

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 167 and “Specifying the location of the JNDI name server” on page 167.

Installing the publish and retract sample programs

This section describes how to install the `CICSConnectionFactoryPublish` and `CICSConnectionFactoryRetract` programs. How to install the CCI Connector application is described in “Installing the CCI Connector sample” on page 318.

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/cicsts31/samples/cci` directory (where `cicsts31` is the value of the CICS_DIRECTORY variable used by the DFHIJVMJ job during CICS installation). Also installed into the `/usr/lpp/cicsts/cicsts31/samples/cci` directory are the source (.java) files of the programs.

To install the `CICSConnectionFactoryPublish` and `CICSConnectionFactoryRetract` programs:

1. Add the JAR file containing the programs, `/usr/lpp/cicsts/cicsts31/samples/cci/CICSCCISamples.jar`, to the CLASSPATH 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/cicsts31/JVMProfiles` directory (where `cicsts31` is the value of the CICS_DIRECTORY variable used by the DFHIJVMJ job during CICS installation).
2. Place your edited version of DFHJVMPR in the HFS directory specified on the JVMPROFILEDIR system initialization parameter. (In a default CICS installation, JVMPROFILEDIR specifies `/usr/lpp/cicsts/cicsts31/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.

Publishing a connection factory using CICSConnectionFactoryPublish

The CICSConnectionFactoryPublish program:

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 name space.

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 33. 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

To look up a previously-published connection factory in the JNDI namespace used by CICS, use code such as the following:

```
// 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 317.

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 name space.

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 34. 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 35 on page 318 shows the components of the sample application.

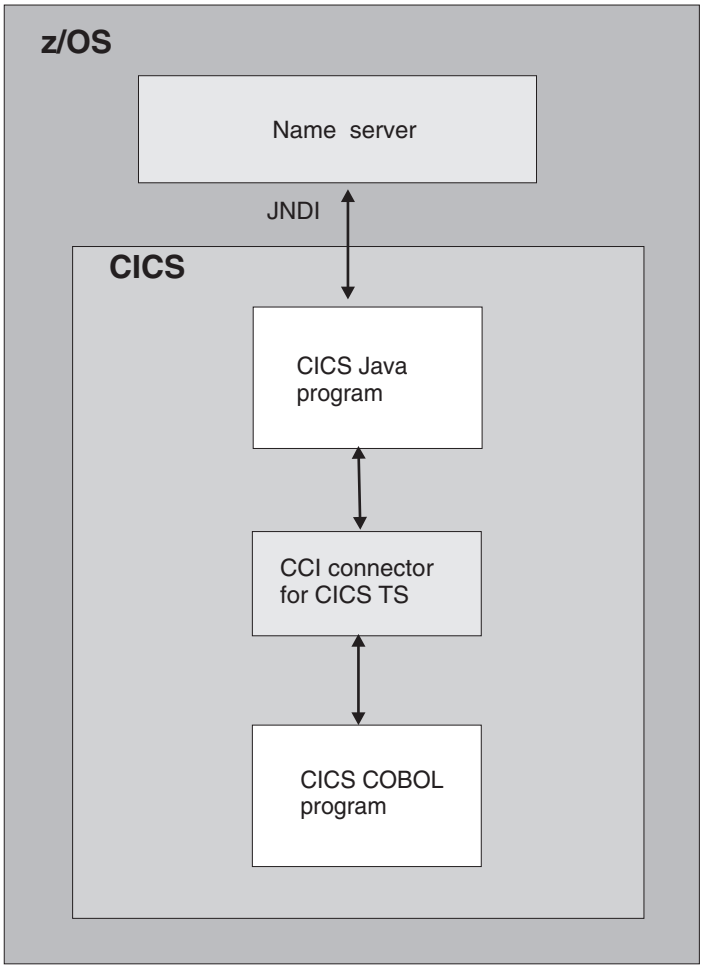

Figure 35. Overview of the CCI Connector sample application. The main elements of the sample are a CICS Java program and a CICS COBOL server program. The Java program uses the CCI Connector for CICS TS to link to the COBOL server program. The CICS connection factory can be published to either a COS Naming 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 228. 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

1. If you have not already done so when running the `CICSConnectionFactoryPublish` and `CICSConnectionFactoryRetract` samples, add the JAR file containing the sample programs, `/usr/lpp/cicsts/cicsts31/samples/cci/CICSCCISamples.jar`, to the CLASSPATH 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/cicsts31/JVMProfiles` directory (where `cicsts31` is the value of the CICS_DIRECTORY variable used by the DFHIJVMJ job during CICS installation).

Place your edited version of DFHJVMPR in the HFS directory specified on the JVMPROFILEDIR system initialization parameter. (In a default CICS installation, JVMPROFILEDIR specifies `/usr/lpp/cicsts/cicsts31/JVMProfiles`.)

2. Ensure that the `connector.jar` and `dfjcci.jar` files are in the "trusted middleware" classpath used by the JVM.

   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/cicsts31/lib` directory, as described in "Compiling CCI applications" on page 313. *These directories are in the default trusted middleware classpath*. Thus, assuming that your trusted middleware classpath is the same as, or based on, the default path, you shouldn't need to do anything.
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 315.
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

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

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.

Migrating 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 3.1**	**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.
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 24. 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.

This section includes the following topics:

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. (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

This section includes the following topics:

CICS enterprise bean errors and messages

There are a variety of places to look for error messages from CICS, the main ones are as follows:

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 *CICS Messages and Codes* 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 *CICS Messages and Codes* 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 *CICS Messages and Codes* 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 *CICS Messages and Codes* for a listing.

JVM trace

JVM trace can aid in the diagnosis of problems in the Java Virtual Machine (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.

"Controlling tracing for JVMs" on page 140 tells you about the different ways to activate JVM tracing and change the JVM trace options. To summarize, you can control JVM trace using:

- The CICS-supplied transaction CETR, which you can use to change the JVM trace options, and to activate JVM tracing. You can use the CETR transaction to define JVM tracing dynamically on the running CICS system.
- The CICS system initialization parameters JVMLEVEL0TRACE, JVMLEVEL1TRACE, JVMLEVEL2TRACE, and JVMUSERTRACE, which you can use to set the default JVM trace options for the CICS region, and the SPCTRSJ and STNTRSJ system initialization parameters, which you can use to activate JVM tracing. You can only supply these parameters at CICS startup time; you cannot define them in the DFHSIT macro. You can use this method to define tracing for JVMs at CICS initialization.
- The EXEC CICS INQUIRE JVMPOOL and EXEC CICS SET JVMPOOL commands (but not their CEMT equivalents), which you can use to change the current JVM trace options for the CICS region.
- The EXEC CICS SET TRACETYPE command, which you can use to activate JVM tracing.
- The **ibm.dg.trc.external** system property in the JVM properties file that is referenced by the JVM profile for a JVM, which you can use to set and activate initial trace options for a particular JVM. You should only use this system property with care.

The first two methods, using CETR and using the CICS system initialization parameters, are most similar to the methods that you would use to define tracing for other components.

When you set trace levels 29–32 for the SJ component and activate JVM trace, each JVM trace point that is generated appears as an instance of CICS trace point SJ 4D01. If the JVM trace facility fails, CICS issues the trace point SJ 4D00.

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 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 *CICS Trace Entries* has details of all the standard trace points in the SJ domain.

Debugging Java applications in CICS

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 142 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/products/jpda/`

Using EJB client runtime diagnostics

Most of the error messages issued by the client are of limited use if the problem is actually 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 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 273.)

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 actually 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* manual
- the *CICS System Programming Reference* manual
- the *CICS Supplied Transactions* manual
- the *CICS Supplied Transactions* manual

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 25. Managing security for enterprise beans

The following security mechanisms can be used with enterprise beans. You can implement any combination of these.

Java2 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 2 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 the *CICS RACF Security Guide*. For information about using SSL with enterprise beans see "Authentication of IIOP requests" on page 161.

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 the *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 337.

Protecting Java applications in CICS by using the Java 2 security policy mechanism

The security of the enterprise beans container environment is protected by the Java 2 security policy mechanism and is independent of CICS security. The security policy mechanism is one of the components that make up the Java 2 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 will do whatever it is asked. If you want to use Java 2 security to protect a Java application or enterprise bean from performing potentially unsafe actions, you need to enable a security manager for the Java virtual machine (JVM) in which the application or enterprise bean executes. If no security manager is enabled, then by default, the JVM runs without Java 2 security. A default security manager is supplied with the Java 2 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) which are assigned to code sources. Every time the JVM executes code within 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 2 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 2 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. The CICS-supplied enterprise beans policy file, `dfjejbpl.policy`, contains the permissions that you need for this purpose. You need to 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 JVM properties file for the JVM. "Enabling a Java security manager and specifying policy files for a JVM" tells you how to do this.

If you need more information about Java 2 security than is provided here, refer to the Java 2 documentation.

Note: Java 2 security with JDBC or SQLJ

To use JDBC or SQLJ from enterprise beans that execute in a JVM with a Java 2 security policy mechanism active, you must use the JDBC 2.0 driver provided by DB2 Version 7 or later. The JDBC 1.2 driver provided by DB2 does not support Java 2 security, and will fail with a security exception unless you disable the mechanism (by deactivating the security manager for the JVM). You will also need to modify your additional policy file to grant permissions to the JDBC driver. "Enabling a Java security manager and specifying policy files for a JVM" tells you more about this.

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 JVM properties file for the JVM. The JVM properties file specifies the system properties for a JVM, including the security manager and policy files. It is associated with the JVM profile for a JVM. "How CICS creates JVMs" on page 71 explains what JVM profiles and JVM properties files are, and how CICS uses them when it starts up a JVM. "Setting up JVM profiles and JVM properties files" on page 94 contains full information on how to choose and customize JVM profiles and JVM properties files for a JVM.

To summarize the essential information from those topics, a JVM profile is a text file stored on HFS, which contains options that determine the characteristics of a JVM. When an application wants to run a Java program in a JVM, it requests a JVM with a particular profile by specifying that JVM profile in the JVMPROFILE attribute of the PROGRAM definition that relates to the Java program. For enterprise beans

and IIOP applications, this is the PROGRAM definition for the initial program used by the request processor transaction definition (which is by default CIRP). This program definition is usually DFJIIRP, and the JVMPROFILE that it specifies is usually the CICS-supplied sample JVM profile DFHJVMCD. When you install CICS, the sample JVM profiles are placed in the HFS directory `/usr/lpp/cicsts/cicsts31/JVMProfiles`, where `cicsts31` is your chosen value for the CICS_DIRECTORY symbol.

The JVM profile references a JVM properties file, which is another text file stored on HFS, containing the system properties for the JVM. The **JVMPROPS** option on the JVM profile names the JVM properties file that CICS uses when setting up a JVM with that profile. The CICS-supplied sample JVM properties file associated with DFHJVMCD is called `dfjjvmcd.props`. When you install CICS, the sample JVM properties files are placed in the HFS directory `/usr/lpp/cicsts/cicsts31/props`.

For each JVM profile that your Java applications and enterprise beans request, if you want JVMs with that profile to run with Java 2 security, you need to modify the JVM properties file that is associated with the JVM profile, to enable the default security manager and specify a suitable policy file. When you have located the relevant JVM properties file for each JVM profile that you want to use Java 2 security, customize the following system properties in the JVM properties file:

java.security.manager
This system property indicates the Java security manager to be enabled for the JVM. To enable the default Java 2 security manager, include this system property in one of the following formats:

```
java.security.manager=default
```

or

```
java.security.manager=""
```

or

```
java.security.manager=
```

All these statements have the effect of enabling the default security manager. If you do not include the **java.security.manager** system property in your JVM properties file, then the JVM runs without Java 2 security enabled. If you need to disable Java 2 security for a JVM, comment out this system property.

java.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/java142/J1.4/lib/security/java.policy`, where the `java142/J1.4` subdirectory names are the default values when you install the IBM Software Developer Kit for z/OS, Java 2 Technology Edition, Version 1.4.2. The default security manager always uses this default policy file to determine the security policy for the JVM, and you can use the **java.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 2 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:

```
java.security.policy=/usr/lpp/cicsts/cicsts31/lib/security/dfjejbpl.policy
```

where *cicsts31* is your chosen value for the USSDIR installation parameter that you defined when you installed CICS TS. "The CICS-supplied enterprise beans policy file, dfjejbpl.policy" on page 333 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. 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 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 **java.security.policy** system property in the JVM properties file, 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 2 security for your JVMs, you still need to add the **java.security.manager** system property to your JVM properties files to enable a Java security manager.

Specifying policy files to apply to all JVMs

As an alternative to using the **java.security.policy** system property in a JVM properties file 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 2 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`. It is provided by CICS in:

```
/usr/lpp/java142/J1.4/lib/security/java.security
```

where the `java142/J1.4` subdirectory names are the default values when you install the IBM Software Developer Kit for z/OS, Java 2 Technology Edition, Version 1.4.2.

The default security properties file already includes the name of the default policy file, `/usr/lpp/java142/J1.4/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 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/cicsts31/lib/security/dfejbpl.policy
```

where *cicsts31* 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 **java.security.policy** system property in the JVM properties file for the JVM, and inserting a double equals sign (= =). For example, if you include the system property:

```
java.security.policy==/usr/lpp/cicsts/cicsts31/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, dfjejbpl.policy

The CICS-supplied enterprise beans policy file, `dfjejbpl.policy`, is based on the security policy recommended 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 36 on page 334.

In Java 2, 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 36 on page 334.

2. Grants any code source only the permissions outlined in the *Enterprise JavaBeans* specification, Version 1. See the default 'grant' block in Figure 36:
   - 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. 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/cicsts31//-" {
     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 36. 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 336.

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 migrated 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 289 for information about deployment tools.
- See Chapter 20, “Writing enterprise beans,” on page 275 for information about writing enterprise beans.

- See “The deployment tools for enterprise beans in a CICS system” on page 289 for information about deployment tools.
- See Chapter 20, “Writing enterprise beans,” on page 275 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 “Giving CICS regions access to z/OS UNIX System Services and HFS directories and files” on page 53.

Access to HFS files used by 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/`*`cicsts31`* directory structure and classes	Read and execute	Contains the CICS-supplied Java code (see note 2).
`/usr/lpp/java142/J1.4/bin` and `/usr/lpp/java142/J1.4/bin/classic` directories	Read and execute	Contain the IBM Java 2 persistent reusable 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 `java.security.policy` property is specified in the JVM system properties file.
System properties file	Read	Required by CICS 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. *cicsts31* is your chosen value for the USSDIR installation parameter that you defined when you installed CICS TS.
3. The `java142/J1.4` subdirectory names are the default values when you install the IBM Software Developer Kit for z/OS, Java 2 Technology Edition, Version 1.4.2.
4. The CICS working directory is defined by the WORK_DIR parameter in the JVM profile.
5. The 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 37 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  'CICSTS31.CICS.CICS.DFHEJDIR' NOTIFY(cics_sys_admin_id) UACC(NONE)
PERMIT 'CICSTS31.CICS.CICS.DFHEJDIR' ID(cics_id1,...,cics_group1,..,cics_groupn)
        ACCESS(UPDATE)
ADDSD  'CICSTS31.CICS.CICS.DFHEJOS'  NOTIFY(cics_sys_admin_id) UACC(NONE)
PERMIT 'CICSTS31.CICS.CICS.DFHEJOS'  ID(cics_id1,...,cics_group1,..,cics_groupn)
        ACCESS(UPDATE)
```

Figure 37. An example of RACF commands used to authorize access to CICS data sets

See 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 3 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, *CICSTS31.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 341

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.”

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 343.

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 340.

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 341

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 341

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 Codepage 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
\	back slash	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 parethesis	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 Codepage 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

Figure 38 on page 342 shows a fragment of a deployment descriptor that includes security role information. It contains:

- 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 38. 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 *CICS Operations and Utilities Guide*.

Implementing security roles

Access to enterprise bean methods is based on the concept of **security roles**. These are described in “Security roles” on page 337.

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 340 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 345 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 345 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 EJBROLEPRFX 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_DIRECTORY/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_DIRECTORY/lib directory. CICS_DIRECTORY is the HFS 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 wish 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

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 EJBROLEPRFX 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 *CICS Messages and Codes*.

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 26. CICSPlex SM with enterprise beans

This chapter describes the following:

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, enquiry 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 enquiry 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.
- The end user interface (EUI) - to allow the definition, enquiry and management of enterprise bean objects through a traditional 3270 interface via MVS/TSO. See the *CICSPlex System Manager Operations Views Reference* for information.

CICSPlex SM definition support for enterprise beans

Business Application Services (BAS) is the CPSM component concerned with the definition and installation of CICS resources—see *CICSPlex System Manager Managing Business Applications*. 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 *CICSPlex System Manager Managing Business Applications*

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 *CICSPlex System Manager Managing Business Applications*.

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 178 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 190 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 *CICSPlex System Manager Managing Business Applications*.

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 know 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 the *CICSPlex System Manager Managing Business Applications*.

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 347. All of the interactive operator services provided through the CICS CEMT and CEOT transactions are functionally replicated in CICSPlex SM via the EUI, or through a web browser window. 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 enquiry 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

As stated previously, there are three paths of inquiry regarding the state of your EJB objects with CICSPlex SM:

- For inquiries through the CICSPlex SM Application Programming Interface, you should refer to the *CICSPlex System Manager Application Programming Reference* (for details of the available CICSPlex SM API commands), in conjunction with the *CICSPlex System Manager Resource Tables Reference* (for details of the attributes and actions allowed against each CICSPlex SM object (resource table)).
- For inquiries through the CICSPlex SM Web User Interface, you should refer to the *CICSPlex System Manager Web User Interface Guide*. Note that the rationale of the Web User Interface is for users to tailor and configure their inquiry structure according to the requirements (and authority) of their operators. However, to assist new users to get online as easily as possible with the Web User Interface, a starter set is provided that comprises an inquiry suite similar in structure to that of the traditional CICSPlex SM EUI. Within this starter set are a set of menus and panels under the link labelled "Enterprise Java component views".
- For inquiries through the traditional 3270 end user interface (EUI) via TSO/MVS, you should refer to the *CICSPlex System Manager Operations Views Reference* for details of the available CICSPlex SM views.

  Note: The EJB menu command is ENTJAVA, and is available as a direct command, or as an item under the main OPERATE menu.

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 *CICSPlex System Manager Managing Workloads*).

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 enquiry. 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 the *CICSPlex System Manager Managing Workloads*.

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.” 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 the *CICSPlex System Manager Managing Workloads*.

CICSPlex SM resource monitoring considerations 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 351. 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 the *CICSPlex System Manager Concepts and Planning*. Full details of the monitoring function is given in *CICSPlex System Manager Managing Resource Usage*.

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 *CICSPlex System Manager Concepts and Planning*. Full detail of the RTA function is given in *CICSPlex System Manager Managing Resource Usage*.

Part 6. Using stateless CORBA objects

This Part tells you what you need to know to develop stateless IIOP applications.

Chapter 27. 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 JVM is reset, causing any state data to be lost.

This implies that every remote method must 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 JVM is reset.

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 365.

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 shareable application class path in the JVM properties file 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 shareable application class path in the JVM properties file 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 HFS 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.

You can use the CICS CEMT command (see the *CICS Supplied Transactions* manual) to issue the PERFORM command, or you can issue EXEC CICS PERFORM (see the *CICS System Programming Reference* manual) 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 simply a `bankaccount` might imply that the instance always represented Mr X's account.

Creating the Interface Definition Language (IDL)

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 357, rather than the RMI-IIOP approach). The RMI-IIOP approach is described in “Developing an RMI-IIOP stateless CORBA application” on page 365.

If you're using the CORBA development style to create a stateless CORBA object application, your first step will be to create an OMG IDL file that contains the definitions of interfaces the server implementation will support. 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 may or may not work.

The proxies and skeletons provide the object-specific information needed for an ORB to distribute a method invocation.

Figure 39 on page 361 shows how the same IDL file is used to generate different classes used by the client and the server.

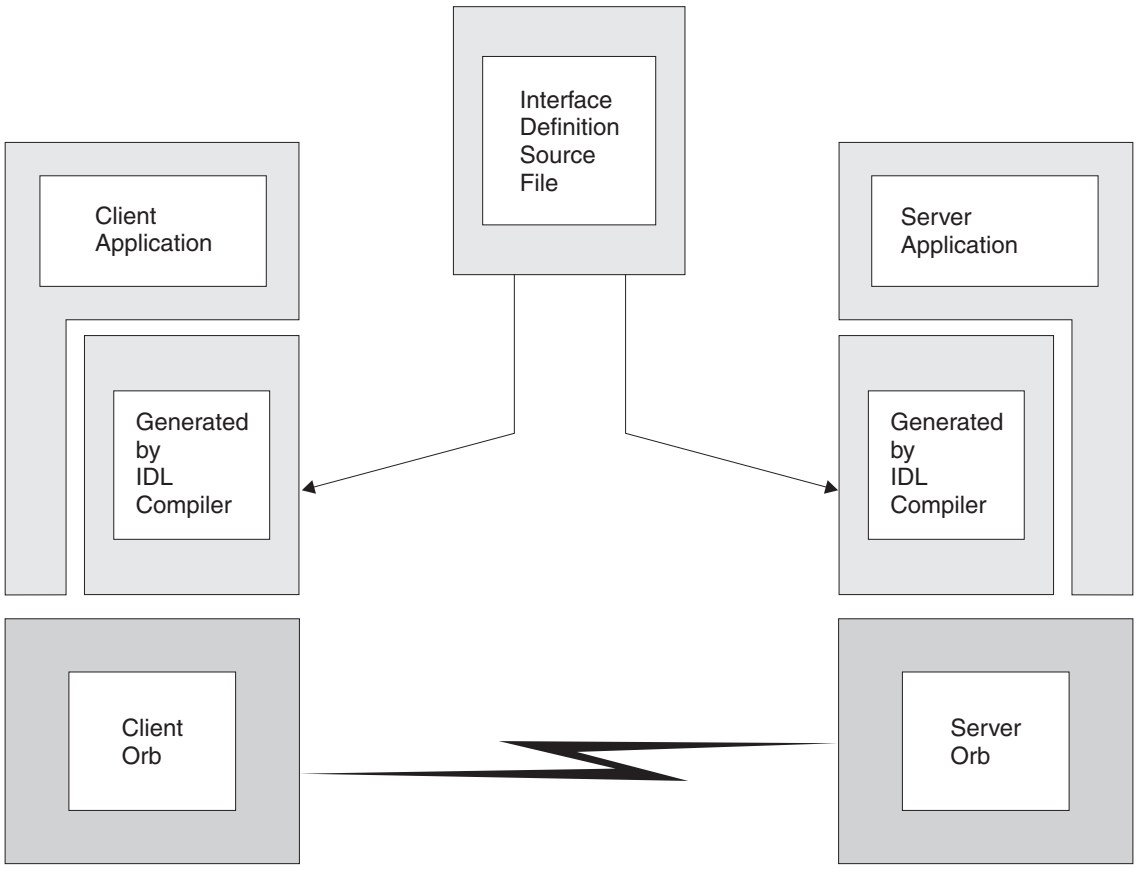

Figure 39. IDL and generated code

Developing an IIOP server program

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 357, rather than the RMI-IIOP approach). The RMI-IIOP approach is described in “Developing an RMI-IIOP stateless CORBA application” on page 365.

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. The following steps are required:

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
```

If you are using a version of the IDL compiler from Java 1.4, 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
Required if you are using the version of the IDLJ compiler supplied with Java 1.4 SDK. 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 shared application classpath, `ibm.jvm.shareable.application.class.path`, in the JVM properties file.

IDL example

The following example describes a bank account whose contents can be queried and updated. Note that this 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);
};
};
```

In this example, the module name is `bank`, the interface name is `BankAccount` and the Operations are `querybalance`, and `setbalance`.

Server implementation

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 `cicsts31`.

Resource definition for example

You must have:

- 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

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 357, rather than the RMI-IIOP approach). The RMI-IIOP approach is described in “Developing an RMI-IIOP stateless CORBA application” on page 365.

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 166. 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.

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/<`*`cicsts31`*`>/lib/omgcos.jar`

Where *cicsts31* 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 (approach 2 in “Developing stateless CORBA objects” on page 357, 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 21, “Deploying enterprise beans,” on page 289.

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 chose to use your own inheritance hierarchy if you wish. 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 364). 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 HFS 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 CICS shareable application 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

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

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.

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.

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. 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.)

  Serializable objects used in the bean interfaces will be expressed in IDL as CORBA valuetypes. 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 the following codepages:

- UCS2—the standard Java codeset (Unicode)
- UTF-8

Chapter 28. Migrating IIOP applications from CICS TS 1.3

CICS implemented an enhanced CORBA ORB in CICS TS for z/OS, Version 2. This means that, if you have existing CICS TS OS/390, Version 1.3 IIOP applications, you can exploit some new function but you will also need to make some changes to the applications, or to the execution environment.

You need to make the following changes:

Environment

CICS replaced **dfjcorb.jar** with **dfjorb.jar** in CICS TS for z/OS, Version 2. See Chapter 14, “Configuring CICS for IIOP,” on page 165 for more information about setting up your environment.

Resource definition

CORBASERVER

You now need to provide and install a CORBASERVER resource definition to define and initialize the execution environment for the IIOP application. Note that the installation of a CORBASERVER is a phased process that may complete at some time after the install is initiated. You can use INQUIRE CORBASERVER commands to verify that the CORBASERVER has installed correctly. See the *CICS Resource Definition Guide* for more information about the CORBASERVER resource definition.

REQUESTMODEL

You need to make some changes to the REQUESTMODEL resource definition. You should use the MODULE, INTERFACE, and OPERATION attributes instead of the OMGMODULE, OMGINTERFACE, and OMGOPERATION attributes, which continue to be supported for migration purposes only. New fields are added to identify the related CORBASERVER and to support Enterprise beans.

Generic pattern matching has been changed to allow only zero or more characters followed by a '*'. In cases where several different generic patterns match a given string, there is now a simple rule for choosing the most specific match. The longest generic pattern results in the most specific match. See the *CICS Resource Definition Guide* for more information about the REQUESTMODEL resource definition.

TCPIPSERVICE

The new PROTOCOL parameter of the TCPIPSERVICE resource definition for the IIOP port must be set to IIOP.

If you are using the Domain Name System (DNS) connection optimization, you now need to define a groupname in the DNSGROUP parameter. In CICS TS 1.3, DNS was active for all TCPIPSERVICEs with names beginning with 'D'. This is now replaced by use of the DNSGROUP and GRPCRITICAL TCPIPSERVICE parameters. See “Domain Name System (DNS) connection optimization” on page 158 for more information about using DNS.

There are new SSL options. See “Authentication of IIOP requests” on page 161 for more information about the use of SSL. See the *CICS Resource Definition Guide* for more information about the TCPIPSERVICE resource definition.

PROGRAM

All IIOP programs must now be defined as JVM programs. You will need to modify existing PROGRAM definitions to add the JVM , JVMCLASS, and JVMPROFILE options. See the *CICS Resource Definition Guide* for more information about the PROGRAM resource definition.

Files

You will need to provide and define a DFHEJDIR and a DFHEJOS file. These must be defined and available before any CORBASERVERs are installed. See Chapter 14, “Configuring CICS for IIOP,” on page 165 for more information about setting up your IIOP environment.

Security URM

You need to change any IIOP security user-replaceable programs to support the new and changed fields in the updated COMMAREA structure. The URM is now called only if it is specified in the TCPIPSERVICE definition for the IIOP port. It is no longer possible to update the transaction identifier from the URM. The sample DFHXOPUS is still supplied. See “Obtaining a CICS user ID” on page 187 for more information about supplying a URM

IDL

CICS does not provide the **dfjcidl.jar** file in CICS TS for z/OS, Version 3.1. Instead, you can use the pre-compiled IDL in the **omgcos.jar** file:

- CosNaming
- CosTransactions
- CosLifeCycle

Alternatively, you can use the idlj compiler from the SDK to generate Java statements from IDL.

GenFacIOR

The offline GenFacIOR utility is no longer needed. You should use the PERFORM CORBASERVER PUBLISH command to publish the CORBASERVER resource definition defining the execution environment for this IIOP request. PUBLISH causes a stringified IOR (called `genfac.ior`) of the GenericFactory class to be created and stored on the shelf (an HFS directory associated with the CorbaServer), and published to the nameserver. You can download the IOR to your client workstation from the shelf using ftp, or your client can use the JNDI interface to obtain the IOR from the nameserver. All existing stringified IOR files need to be recreated. For more information, see “Defining name servers” on page 166.

IIOP messages > 32K

In CICS TS 1.3, CICS used temporary storage to pass IIOP messages larger than 32k to the request processor, and you needed to define TSMODELS for temporary storage queue prefixes DFIO and DFJO. The request streams logic manages these messages in a different way in CICS TS for z/OS, Version 2 and later, and these TS queues are no longer needed.

JVM

IIOP applications execute in the JVM. CICS Transaction Server for z/OS, Version 3 Release 1 does not provide runtime support for applications that have been processed by the VisualAge for Java, Enterprise Edition for OS/390 bytecode binder (hpj) to run as Java program objects in CICS. You will need to set up the JVM environment as described in Chapter 9, “Setting up Java support,” on page 53, and define your programs as JVM programs.

Chapter 29. Using the IIOP samples

The following 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.

This chapter describes the samples and tells you how to run them. The following topics are covered:

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:

- A z/OS system, Version 2.8 or later, with UNIX Systems Services and HFS
- Language Environment configured and active
- CICS
- The IBM Software Developer Kit for z/OS, Java 2 Technology Edition, Version 1.4.2. This is available from :

  `http://www.s390.ibm.com/java`

Then follow these steps to set up the IIOP environment:

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. 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 the *CICS RACF Security Guide*.

4. Create a shelf directory on HFS and give the CICS region userid full access to it. See "Giving CICS regions access to z/OS UNIX System Services and HFS directories and files" on page 53 for guidance.
5. Choose a suitable JVM profile and JVM properties file and ensure that CICS is able to locate them, as described in "Setting up JVM profiles and JVM properties files" on page 94.
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/cicsts31/
   ```

   where *cicsts31* is your chosen value for the USSDIR installation parameter that you defined when you installed CICS TS.

   JAVA_HOME
   The installation directory prefix of the SDK. This is:

   ```
   /usr/lpp/java142/J1.4/
   ```

   where *java142/J1.4* is defined when you install the IBM Software Developer Kit for z/OS, Java 2 Technology Edition, Version 1.4.2.

7. Ensure that the following files are added to a suitable class path in the JVM profile or JVM properties file:

   •

   The sample Java source and makefiles that are stored in the z/OS UNIX System Services HFS 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 128 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 whilst it is running. See the *CICS Supplied Transactions* 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 the *CICS Resource Definition Guide* for information about TCIPSERVICE and CORBASERVER resource definitions..

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 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 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.

    You can use the CICS CEMT master terminal command (see the *CICS Supplied Transactions*) to issue the PERFORM command, or you can issue the EXEC CICS PERFORM command (see the *CICS System Programming Reference*) from a CICS application.

    You can download the IOR to your client workstation (in ascii mode) from the shelf using ftp.

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" on page 377
- "Building the client side HelloWorld application" on page 377
- "Running the HelloWorld sample application" on page 377

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 end of the class path, `ibm.jvm.shareable.application.class.path`, in the default JVM properties file, dfjjvmcd.props.

To build the programs, enter the following command from $CICS_HOME/samples/dfjcorb/HelloWorld/server:

```
make
```

This 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.

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 `$CICS_HOME/lib` directory on HFS. Alternatively you may choose to build these classes from the original OMG-supplied IDL. In this case a copy of the relevant IDL files is available in `$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

Run the client application using:

```
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 379

Creating the VSAM file

Define the VSAM file to hold the bank account data, using the following 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 end of the class path, `ibm.jvm.shareable.application.class.path`, in the default JVM properties file, dfjjvmcd.props.

To build the programs, enter the following command from $CICS_HOME/samples/dfjcorb/BankAccount/server:

```
make
```

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.

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 377.

Running the BankAccount sample application

The following steps are required to run the sample application:

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

Bibliography

The CICS Transaction Server for z/OS library

The published information for CICS Transaction Server for z/OS is delivered in the following forms:

The CICS Transaction Server for z/OS Information Center
The CICS Transaction Server for z/OS Information Center is the primary source of user information for CICS Transaction Server. The Information Center contains:

- Information for CICS Transaction Server in HTML format.
- Licensed and unlicensed CICS Transaction Server books provided as Adobe Portable Document Format (PDF) files. You can use these files to print hardcopy of the books. For more information, see "PDF-only books."
- Information for related products in HTML format and PDF files.

One copy of the CICS Information Center, on a CD-ROM, is provided automatically with the product. Further copies can be ordered, at no additional charge, by specifying the Information Center feature number, 7014.

Licensed documentation is available only to licensees of the product. A version of the Information Center that contains only unlicensed information is available through the publications ordering system, order number SK3T-6945.

Entitlement hardcopy books
The following essential publications, in hardcopy form, are provided automatically with the product. For more information, see "The entitlement set."

The entitlement set

The entitlement set comprises the following hardcopy books, which are provided automatically when you order CICS Transaction Server for z/OS, Version 3 Release 1:

Memo to Licensees, GI10-2559
CICS Transaction Server for z/OS Program Directory, GI10-2586
CICS Transaction Server for z/OS Release Guide, GC34-6421
CICS Transaction Server for z/OS Installation Guide, GC34-6426
CICS Transaction Server for z/OS Licensed Program Specification, GC34-6608

You can order further copies of the following books in the entitlement set, using the order number quoted above:

CICS Transaction Server for z/OS Release Guide
CICS Transaction Server for z/OS Installation Guide
CICS Transaction Server for z/OS Licensed Program Specification

PDF-only books

The following books are available in the CICS Information Center as Adobe Portable Document Format (PDF) files:

CICS books for CICS Transaction Server for z/OS

General

CICS Transaction Server for z/OS Program Directory, GI10-2586
CICS Transaction Server for z/OS Release Guide, GC34-6421

CICS Transaction Server for z/OS Migration from CICS TS Version 2.3, GC34-6425
CICS Transaction Server for z/OS Migration from CICS TS Version 1.3, GC34-6423
CICS Transaction Server for z/OS Migration from CICS TS Version 2.2, GC34-6424
CICS Transaction Server for z/OS Installation Guide, GC34-6426

Administration

CICS System Definition Guide, SC34-6428
CICS Customization Guide, SC34-6429
CICS Resource Definition Guide, SC34-6430
CICS Operations and Utilities Guide, SC34-6431
CICS Supplied Transactions, SC34-6432

Programming

CICS Application Programming Guide, SC34-6433
CICS Application Programming Reference, SC34-6434
CICS System Programming Reference, SC34-6435
CICS Front End Programming Interface User's Guide, SC34-6436
CICS C++ OO Class Libraries, SC34-6437
CICS Distributed Transaction Programming Guide, SC34-6438
CICS Business Transaction Services, SC34-6439
Java Applications in CICS, SC34-6440
JCICS Class Reference, SC34-6001

Diagnosis

CICS Problem Determination Guide, SC34-6441
CICS Messages and Codes, GC34-6442
CICS Diagnosis Reference, GC34-6899
CICS Data Areas, GC34-6902
CICS Trace Entries, SC34-6443
CICS Supplementary Data Areas, GC34-6905

Communication

CICS Intercommunication Guide, SC34-6448
CICS External Interfaces Guide, SC34-6449
CICS Internet Guide, SC34-6450

Special topics

CICS Recovery and Restart Guide, SC34-6451
CICS Performance Guide, SC34-6452
CICS IMS Database Control Guide, SC34-6453
CICS RACF Security Guide, SC34-6454
CICS Shared Data Tables Guide, SC34-6455
CICS DB2 Guide, SC34-6457
CICS Debugging Tools Interfaces Reference, GC34-6908

CICSPlex SM books for CICS Transaction Server for z/OS

General

CICSPlex SM Concepts and Planning, SC34-6459
CICSPlex SM User Interface Guide, SC34-6460
CICSPlex SM Web User Interface Guide, SC34-6461

Administration and Management

CICSPlex SM Administration, SC34-6462
CICSPlex SM Operations Views Reference, SC34-6463
CICSPlex SM Monitor Views Reference, SC34-6464
CICSPlex SM Managing Workloads, SC34-6465
CICSPlex SM Managing Resource Usage, SC34-6466
CICSPlex SM Managing Business Applications, SC34-6467

Programming

CICSPlex SM Application Programming Guide, SC34-6468
CICSPlex SM Application Programming Reference, SC34-6469

Diagnosis

CICSPlex SM Resource Tables Reference, SC34-6470
CICSPlex SM Messages and Codes, GC34-6471
CICSPlex SM Problem Determination, GC34-6472

CICS family books

Communication

CICS Family: Interproduct Communication, SC34-6473
CICS Family: Communicating from CICS on System/390, SC34-6474

Licensed publications

The following licensed publications are not included in the unlicensed version of the Information Center:

CICS Diagnosis Reference, GC34-6899
CICS Data Areas, GC34-6902
CICS Supplementary Data Areas, GC34-6905
CICS Debugging Tools Interfaces Reference, GC34-6908

Other CICS books

The following publications contain further information about CICS, but are not provided as part of CICS Transaction Server for z/OS, Version 3 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 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

Books from related libraries

This section lists the non-CICS books that are referred to in this manual.

IBM Developer Kit and Runtime Environment, Java 2 Technology Edition, Version 1.4.2 Diagnostics Guide, SC34-6358

Persistent Reusable Java Virtual Machine User's Guide, SC34-6201

Determining if a publication is current

IBM regularly updates its publications with new and changed information. When first published, both hardcopy and BookManager® softcopy versions of a publication are usually in step. However, due to the time required to print and distribute hardcopy books, the BookManager version is more likely to have had last-minute changes made to it before publication.

Subsequent updates will probably be available in softcopy before they are available in hardcopy. This means that at any time from the availability of a release, softcopy versions should be regarded as the most up-to-date.

For CICS Transaction Server books, these softcopy updates appear regularly on the *Transaction Processing and Data Collection Kit* CD-ROM, SK2T-0730-xx. Each

reissue of the collection kit is indicated by an updated order number suffix (the -xx part). For example, collection kit SK2T-0730-06 is more up-to-date than SK2T-0730-05. The collection kit is also clearly dated on the cover.

Updates to the softcopy are clearly marked by revision codes (usually a # character) to the left of the changes.

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.

Some accessibility features may not be available when using the Application Assembly Tool (AAT), which is a component of WebSphere Application Server. You should consult the documentation that comes with WebSphere Application Server to determine which accessibility features are available when using AAT.

Index

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

W

X

Z

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 wish 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.

Adobe and the Adobe logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Microsoft, Windows, and Windows NT 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.

Sending your comments to IBM

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM.

Feel free to comment on what you regard as specific errors or omissions, and on the accuracy, organization, subject matter, or completeness of this book.

Please limit your comments to the information in this book and the way in which the information is presented.

To ask questions, make comments about the functions of IBM products or systems, or to request additional publications, contact your IBM representative or your IBM 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.

You can send your comments to IBM in any of the following ways:

- By mail, to this address:

  IBM United Kingdom Limited
  User Technologies Department (MP095)
  Hursley Park
  Winchester
  Hampshire
  SO21 2JN
  United Kingdom
- By fax:
  - From outside the U.K., after your international access code use 44–1962–816151
  - From within the U.K., use 01962–816151
- Electronically, use the appropriate network ID:
  - IBMLink: HURSLEY(IDRCF)
  - Internet: idrcf@hursley.ibm.com

Whichever you use, ensure that you include:

- The publication title and order number
- The topic to which your comment applies
- Your name and address/telephone number/fax number/network ID.

Product Number: 5655-M15

SC34-6440-07

Spine information:

CICS Transaction Server for z/OS Java Applications in CICS

Version 3
Release 1