Java applications in CICS

Version 2 Release 1
Java applications in CICS

Version 2 Release 1
First edition (March 2001)

This edition applies to Version 2 Release 1 of CICS Transaction Server for z/OS, program number 5697-E93, 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.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

At the back of this publication is a page entitled “Sending your comments to IBM”. If you want to make comments, but the methods described are not available to you, please address them to:

IBM United Kingdom Laboratories, Information Development,
Mail Point 095, Hursley Park, Winchester, Hampshire, England, SO21 2JN.

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

© Copyright International Business Machines Corporation 1999, 2001. All rights reserved.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
# Contents

**Preface** ............................................. xi
What this information is about .......................... xi
Who should read this information ........................ xi

**Part 1. Java™ Development Roadmaps** ................ 1

Chapter 1. Java development roadmaps .................. 3
Java application roadmap for CICS ........................ 3
CICS IIOP application roadmap ............................ 3
CICS enterprise beans roadmap ............................ 3

**Part 2. Developing Java applications for CICS** ........... 5

Chapter 2. Java applications in CICS .................. 7
Types of Java application in CICS ....................... 7

Chapter 3. What you need to know about CICS .......... 9
CICS transactions ........................................ 9
CICS tasks ............................................... 10
CICS application programs ............................... 10
CICS services ............................................ 10

Chapter 4. Java programming in CICS .................. 13
The JCICS class library ................................... 13
Changes from CICS Transaction Server for OS/390 Version 1 Release 3 13
Translation ............................................... 14
JavaBeans ............................................... 14
Library structure ....................................... 14
CICS resources ......................................... 15
Command arguments ..................................... 15
Using the Java Record Framework ....................... 15
System.out and System.err ............................. 16
Threads ................................................. 16
JCICS command reference ................................ 16
CICS exception handling in Java programs ............. 17
Error handling and abnormal termination ............... 18
APPC mapped conversations .............................. 19
Basic Mapping Support (BMS) ............................ 19
Diagnostic services .................................... 19
Environment services .................................... 20
File services ........................................... 22
Program services ....................................... 24
Scheduling services .................................... 25
Serialization services ................................... 25
Storage services ........................................ 25
Temporary storage queue services ....................... 25
Terminal services ...................................... 26
Transient data queue services ........................... 26
Unit of Work (UOW) services ........................... 27
Unsupported CICS services .............................. 27
JCICS exception mapping ................................ 27
Using JCICS ............................................ 28
<table>
<thead>
<tr>
<th>Page</th>
<th>Chapter</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>108</td>
<td>13</td>
<td>Setting up an EJB server</td>
</tr>
<tr>
<td>109</td>
<td>13</td>
<td>Before running the IVP</td>
</tr>
<tr>
<td>110</td>
<td>13</td>
<td>After running the IVP</td>
</tr>
<tr>
<td>113</td>
<td>13</td>
<td>Testing your EJB server</td>
</tr>
<tr>
<td>115</td>
<td>13</td>
<td>Running the EJB IVP</td>
</tr>
<tr>
<td>119</td>
<td>13</td>
<td>Using the “Hello World” sample</td>
</tr>
<tr>
<td>119</td>
<td>13</td>
<td>Using the EJB CICS sample</td>
</tr>
<tr>
<td>119</td>
<td>13</td>
<td>Using your own enterprise beans</td>
</tr>
<tr>
<td>120</td>
<td>13</td>
<td>Setting up a multi-region EJB server</td>
</tr>
<tr>
<td>123</td>
<td>14</td>
<td>Prerequisites for the EJB IVP</td>
</tr>
<tr>
<td>123</td>
<td>14</td>
<td>Installing the EJB IVP</td>
</tr>
<tr>
<td>123</td>
<td>14</td>
<td>CICS setup</td>
</tr>
<tr>
<td>124</td>
<td>14</td>
<td>Unix System Services setup</td>
</tr>
<tr>
<td>125</td>
<td>14</td>
<td>Running the EJB IVP</td>
</tr>
<tr>
<td>127</td>
<td>15</td>
<td>The “Hello World” sample application</td>
</tr>
<tr>
<td>127</td>
<td>15</td>
<td>What the “Hello World” sample does</td>
</tr>
<tr>
<td>128</td>
<td>15</td>
<td>Prerequisites for the “Hello World” sample</td>
</tr>
<tr>
<td>128</td>
<td>15</td>
<td>Supplied components of the “Hello World” sample</td>
</tr>
<tr>
<td>129</td>
<td>15</td>
<td>Installing the “Hello World” sample</td>
</tr>
<tr>
<td>132</td>
<td>15</td>
<td>Testing the “Hello World” sample</td>
</tr>
<tr>
<td>133</td>
<td>15</td>
<td>The EJB CICS sample application</td>
</tr>
<tr>
<td>133</td>
<td>15</td>
<td>What the EJB CICS sample does</td>
</tr>
<tr>
<td>134</td>
<td>15</td>
<td>Prerequisites for the EJB CICS sample</td>
</tr>
<tr>
<td>135</td>
<td>15</td>
<td>Supplied components of the EJB CICS sample</td>
</tr>
<tr>
<td>136</td>
<td>15</td>
<td>Installing the EJB CICS sample</td>
</tr>
<tr>
<td>137</td>
<td>15</td>
<td>Testing the EJB CICS sample</td>
</tr>
<tr>
<td>141</td>
<td>15</td>
<td>A note about data conversion</td>
</tr>
<tr>
<td>145</td>
<td>16</td>
<td>Coding a session bean</td>
</tr>
<tr>
<td>146</td>
<td>16</td>
<td>Coding the home interface</td>
</tr>
<tr>
<td>146</td>
<td>16</td>
<td>Coding the remote interface</td>
</tr>
<tr>
<td>147</td>
<td>16</td>
<td>Coding the bean implementation</td>
</tr>
<tr>
<td>147</td>
<td>16</td>
<td>Compiling the code</td>
</tr>
<tr>
<td>149</td>
<td>16</td>
<td>Packaging the code</td>
</tr>
<tr>
<td>149</td>
<td>16</td>
<td>Writing the client program</td>
</tr>
<tr>
<td>149</td>
<td>16</td>
<td>Creating object references in the namespace</td>
</tr>
<tr>
<td>150</td>
<td>16</td>
<td>Using JNDI to obtain bean references</td>
</tr>
<tr>
<td>152</td>
<td>16</td>
<td>Using the client program</td>
</tr>
<tr>
<td>153</td>
<td>16</td>
<td>Using EDF with enterprise beans</td>
</tr>
<tr>
<td>155</td>
<td>17</td>
<td>Installing and configuring CICS deployment tools for EJB technology</td>
</tr>
</tbody>
</table>
Hardware and software requirements for CICS deployment tools for EJB technology

**Hardware**

**Software**

Installing and configuring the CICS JAR development tool for EJB technology

Configuration considerations

Installing and configuring the CICS development deployment tool for EJB technology

Setting up the WebSphere components of the CICS development deployment tool for EJB technology

Configuring the deployment configuration file

Setting up the CICS components of the CICS development deployment tool

Setting up more complex development deployment tool configurations

Installing and configuring the CICS production deployment tool for EJB technology

Chapter 18. Deploying enterprise beans

Using CICS deployment tools for EJB technology

The CICS JAR development tool for EJB technology

The deployment descriptor

Code generation

Using the CICS JAR development tool for EJB technology

The GUI

the Deployment descriptor

Using the CICS code generation utility for EJB technology from the CICS JAR development tool GUI

Using the CICS code generation utility for EJB technology as a stand-alone tool

The CICS development deployment tool for EJB technology

DJAR generation and the CICS development deployment tool

Request model generation and the CICS development deployment tool

How the CICS development deployment tool resolves bindings

Using the CICS development deployment tool for EJB technology

The CICS production deployment tool for EJB technology

Using the CICS production deployment tool for EJB technology

The CICS production deployment tool as a GUI

The CICS production deployment tool as an offline utility

The final stages of deployment

Applying generated resource definitions

Publishing names to the JNDI

Ensuring that the JAR file is stored in HFS

Chapter 19. The CICS Connector for CICS TS

Overview of the CICS Connector for CICS TS

What are CICS connectors?

The CICS Connector for CICS TS

The background—accessing CICS programs from Java

Benefits of the CICS Connector for CICS TS

A sample application

Using the CICS Connector for CICS TS’s CCF interface

Setting the CCF interface attributes

Introducing VisualAge for Java Enterprise Access Builder

Data conversion and the CICS Connector for CICS TS

The CICS Connector for CICS TS—using the CTG API

ECIRequest

JavaGateway
### Developing the IIOP client program
- Client example

<table>
<thead>
<tr>
<th>Chapter 24. Migrating IIOP applications from CICS TS 1.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setting up the IIOP sample environment</td>
</tr>
<tr>
<td>Running the IIOP HelloWorld sample</td>
</tr>
<tr>
<td>Building the server side HelloWorld application</td>
</tr>
<tr>
<td>Building the client side HelloWorld application</td>
</tr>
<tr>
<td>Running the HelloWorld sample application</td>
</tr>
<tr>
<td>Running the IIOP BankAccount sample</td>
</tr>
<tr>
<td>Creating the VSAM file</td>
</tr>
<tr>
<td>Building the server side BankAccount application</td>
</tr>
<tr>
<td>Building the client side BankAccount application</td>
</tr>
<tr>
<td>Running the BankAccount sample application</td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>Chapter 26. VisualAge for Java, ET/390</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running a Java program object in CICS</td>
</tr>
<tr>
<td>Run-time requirements</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 27. Java hot-pooling concepts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open transaction environment TCBs</td>
</tr>
<tr>
<td>H8 mode TCB allocation for hot-pooling</td>
</tr>
<tr>
<td>Using hot-pooling with ET/390</td>
</tr>
<tr>
<td>Defining Language Environment run-time options</td>
</tr>
<tr>
<td>Allocating storage</td>
</tr>
<tr>
<td>Defining hot-pooled Java program objects</td>
</tr>
<tr>
<td>Programming for Java hot-pooling</td>
</tr>
<tr>
<td>Static storage and static initializers</td>
</tr>
<tr>
<td>Programming interfaces</td>
</tr>
<tr>
<td>Protection keys</td>
</tr>
<tr>
<td>Exit programming interface (XPI)</td>
</tr>
<tr>
<td>Hot-pooling exit program</td>
</tr>
<tr>
<td>Security</td>
</tr>
<tr>
<td>Problem determination</td>
</tr>
</tbody>
</table>

### Part 8. Appendixes

<table>
<thead>
<tr>
<th>Bibliography</th>
</tr>
</thead>
<tbody>
<tr>
<td>CICS Transaction Server for z/OS</td>
</tr>
<tr>
<td>CICS books for CICS Transaction Server for z/OS</td>
</tr>
<tr>
<td>CICSPlex SM books for CICS Transaction Server for z/OS</td>
</tr>
<tr>
<td>Other CICS books</td>
</tr>
<tr>
<td>Determining if a publication is current</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Notices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trademarks</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sending your comments to IBM</th>
</tr>
</thead>
</table>
Preface

What this information is about

This information tells you how to develop and use Java applications and Enterprise Java beans in CICS. It is designed to be accessed online through the CICS Information Center, but is also available as a pdf, in a book format, for printing.

Reviewers may see this information presented in a sequential book format, but should be aware that it is designed to be accessed randomly, using hypertext links to move logically through the topics.

Who should read this information

This information is intended for experienced Java application programmers, who have little experience of CICS, and no great need to know more about CICS than is necessary to develop and run Java programs. It should also be useful to experienced CICS users and system programmers, who need to know about CICS requirements for the Java support.
Part 1. Java™ Development Roadmaps

This Part tells you all the steps needed to implement Java applications in CICS®. It covers the following tasks:

- "Java application roadmap for CICS" on page 3
- "CICS IIOP application roadmap" on page 3
- "CICS enterprise beans roadmap" on page 3
Chapter 1. Java development roadmaps

The following roadmaps show you the steps to follow to develop the different Types of Java application in CICS that you can use:

- Java application roadmap for CICS
- CICS IIOP application roadmap
- CICS enterprise beans roadmap

Java application roadmap for CICS

1. Write a Java application, using the JCICS classes to access CICS services and resources. See Chapter 4. Java programming in CICS on page 13.
2. Use the Java Virtual Machine in CICS to execute your application. See Chapter 6. About the JVM on page 37 and Chapter 7. Configuring the JVM on page 47.

CICS IIOP application roadmap

2. Write your IIOP server application, also known as a 'stateless CORBA object'. See Developing stateless CORBA objects on page 247, Creating the Interface Definition Language (IDL) on page 249, and Developing an IIOP server program on page 250.
3. Write your client program. See Developing the IIOP client program on page 253.

CICS enterprise beans roadmap

1. Set up CICS as an IIOP server, see Chapter 10. Configuring CICS for IIOP on page 71.
2. Configure the CICS TCP/IP listener region, see Setting up TCP/IP for IIOP on page 72.
3. Set up the host system environment, see Setting up the host system for IIOP on page 71.
4. Make sure that all the steps in Setting up a single-region EJB server on page 113 have been performed, or Setting up a multi-region EJB server on page 120 if you are using load balancing across multiple CICS regions.
5. Implement any security controls required by your system, see Chapter 21. Managing security for enterprise beans on page 227.
6. Code your session bean. See Coding a session bean on page 146 if you are not using an Integrated Development Environment (IDE).
7. Install and configure the CICS deployment tools. See Chapter 17. Installing and configuring CICS deployment tools for EJB technology on page 155.
8. Follow the deployment process described in Chapter 18. Deploying enterprise beans on page 173, using the tools as described in Using CICS deployment tools for EJB technology on page 173.
9. Write the client program. See Writing the client program on page 149.
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. It covers the following topics:

- "Chapter 2. Java applications in CICS" on page 7
- "Chapter 3. What you need to know about CICS" on page 9
- "Chapter 4. Java programming in CICS" on page 13
- "Chapter 5. Using the JCICS sample programs" on page 31
Chapter 2. 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 in two different ways:
1. Using a Java Virtual Machine (JVM) that is executing under CICS control.
2. Using VisualAge for Java, Enterprise Edition for OS/390 (ET/390) to bind the Java bytecode into OS/390 executable files, known as Java program objects, that are stored in MVS PDSE libraries and executed by CICS in a Language Environment (LE) run-unit, similarly to C++.

Types of Java application in CICS

You can use the following types of Java program in CICS:

CICS applications for Java
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 dfjcics.jar and can be downloaded to your workstation. It is also available with some releases of VisualAge for Java.

JCICS applications can be bound by the ET/390 bytecode binder to form Java Program objects that can be loaded and executed by CICS, or they can be run in the CICS JVM.

You can read more about JCICS in "The JCICS class library" on page 13.

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 in 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 CosTransactions::TransactionalObject, then the object will be 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 can execute only in the CICS JVM.
You can read more about CICS stateless CORBA objects in Chapter 23, *Stateless CORBA objects* on page 247.

**Enterprise beans**

Enterprise beans are portable Java components that comply with Sun Microsystems' *Enterprise JavaBeans Specification, Version 1.1 (EJB)* specification. 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 platform.

The Enterprise JavaBeans 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 can execute only in the CICS JVM.

You can read more about Enterprise beans in Chapter 12, *What are enterprise beans?* on page 87.

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

<table>
<thead>
<tr>
<th>FEATURE</th>
<th>non-IIOP CICS appl.</th>
<th>CICS stateless CORBA object</th>
<th>CICS session bean</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>outbound</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td>inbound</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>IIOP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>APPC/MRO outbound</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>APPC/MRO inbound</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>EXEC CICS SYNCPPOINT</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>Units of Work</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXEC CICS SYNCPPOINT</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td></td>
<td>outbound</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td>inbound</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>OTS transactions</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Container managed</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>Bean managed</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>factory publication to JNDI</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Application metadata</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>hpj bytecode binder</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>Secure sockets Layer (SSL)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Secure sockets Layer (SSL)</td>
<td>outbound</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td>inbound</td>
<td>NO</td>
<td>YES</td>
</tr>
</tbody>
</table>
Chapter 3. What you need to know about CICS

CICS is a transaction processing subsystem. This means that it provides services for you 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, with fast response.

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. This is usually from an end-user at a terminal, but may also be made from a Web page, from a remote workstation program, from an application in another CICS system or 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 what 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 operation that is recoverable; it can be committed or backed out as an entirety as a result of programmed command or 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, 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, OO 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 you use CICS commands, or the Java and C++ class libraries to request CICS services.

The use of the CICS command level programming interface, 'EXEC CICS', which is used in COBOL, OO COBOL, C, C++, PL/I or assembler programs is described in the [CICS Application Programming Reference](#) and the [CICS System Programming Reference](#).

In Java programs, you can use the JCICS classes to access CICS services and link to CICS application programs written in other languages. The types of Java program that you can write are listed in [Types of Java application in CICS](#) on page 7 and JCICS is described in [The JCICS class library](#) on page 13.

You can write enterprise beans that use the interfaces defined in the Sun Microsystem’s *Enterprise JavaBeans Specification, Version 1.1*, which is available at [http://www.javasoft.com/products/ejb](http://www.javasoft.com/products/ejb). CICS implements this specification by mapping program requests transparently to underlying CICS services. You can also use the JCICS classes to call CICS services directly, but then your Enterprise beans will not be portable to non-CICS servers.

CICS services

CICS provides the following services, which you can access through the JCICS and EXEC CICS programming interfaces. CICS services managers are traditionally called 'control', such as terminal control or 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, and to support 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 is a means of making data readily available to multiple transactions. Data is kept in queues created as required by a program and 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 is also available to multiple transactions, and is kept in queues, but they 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 items to it can act as a trigger to start a specific transaction (for example, to process the queue).

- Access to data in other databases (including DB2), through interfaces with database products.

**Communications services**
CICS provides commands giving access to a wide range of terminals: displays, printers and workstations, using SNA and TCP/IP protocols. You can write programs that use APPC (Advanced Program to Program Communication) commands to start and communicate with other programs in remote systems, using SNA protocols. CICS APPC implements the peer-to-peer distributed application model. Management of CICS Management of SNA and TCP/IP networks is provided by CICS terminal control.

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), supporting 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 systems are automatically routed by CICS to the owning system.

**Distributed Program Link (DPL)**
Program LINK requests for a program defined as existing on a remote CICS system are automatically routed to the owning system. 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 remote, CICS system 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 systems are automatically routed to the owning system. Responses to the end-user are routed back to the system that received the request.

**Unit-of Work services**
When CICS creates a new task to run a transaction, a new unit-of-work is started automatically. CICS does not provide a BEGIN command. CICS transactions are always executed in-transaction. CICS provides a SYMCPNT 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 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 CICS provided interfaces.

If you write enterprise beans using transaction services provided by commands defined by the Java Transaction Service (JTS), these commands (including BEGIN) are implemented in CICS by mapping them to underlying unit-of-work services.

**Program services**
- CICS provides commands enabling you to link or transfer control to another program, and return.

**Diagnostic services**
- CICS provides commands enabling you to trace and dump program activity.

**Other services**
- CICS provides other services, such as journaling, timer, storage management, that are not available through the JCICS interface, but are described in the CICS Application Programming Guide.
Chapter 4. Java programming in CICS

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

You write Java programs on a workstation, or in the OS/390 UNIX System Services shell, using an editor of your choice, or in a visual composition environment such as VisualAge.

CICS provides a Java class library, known as JCICS, supplied in dfjcics.jar. 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 CICS EXEC API are supported. See [The JCICS class library](#) for a description of the JCICS API.

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 in two different ways:

1. Using a Java Virtual Machine (JVM) that is executing under CICS control. This is the recommended method.

2. Using VisualAge for Java, Enterprise Edition for OS/390 to bind the Java bytecode into OS/390 executable files, known as Java program objects, that are stored in OS/390 PDSE libraries and executed by CICS in a Language Environment (LE) run-unit, similarly to C++.

   **Note:** CICS TS for z/OS Version 2 supports hpj-bound Java program objects that were developed using CICS TS OS/390 Version 1 Release 3 and its associated tooling. Such program objects will run in CICS TS for z/OS Version 2 unchanged, but CICS TS for z/OS Version 2 CICS TS V2 provides no support for developing new Java program objects nor for modifying existing Java program objects. The version of JCICS supplied in CICS TS for z/OS Version 2 should not be used to compile Java program objects.

You can read about VisualAge for Java, Enterprise Edition for OS/390 in [Chapter 26. VisualAge for Java, ET/390](#) and the CICS JVM in [Chapter 6. About the JVM](#).

### The JCICS class library

The Java class library for CICS, JCICS, supports most of the functions of the EXEC CICS API commands, with the limitations described in [JCICS command reference](#) on page 16.

The JCICS classes are fully documented in JAVADOC html that is generated from the class definitions. This is available through the CICS Information Center, and can be found at [JCICS Javadoc class Reference](#).

### Changes from CICS Transaction Server for OS/390 Version 1 Release 3

If you use the `xct1()` method in the `Program` class, a `TransferOfControlException` is thrown to the issuing program, even if it completes successfully.
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 VisualAge for Java, serialized, and manipulated using the JavaBeans API. The beans 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 `new` 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 setter methods at run-time.

Library structure

JCICS library components fall 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 for 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 and it provides the superclass for all other CICS error classes. These are used for severe errors.

JCICS defines CicsException as a subclass of Exception. CicsException provides the superclass for all CICS exception classes. This includes the CicsConditionException classes such as InvalidQueueIdException, which represents the CICS QIDERR condition.

See [Error handling and abnormal termination](#) on page 18 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 by using CEDA or CICSPlex SM BAS. See the [CICS Resource Definition Guide](#) or [CICSPlex System Manager Concepts and Planning](#) 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.

**Note:** Before using CEDA, you may need to use CEOT to enable your terminal to handle mixed case input— for details, see [CICS Supplied Transactions](#).

### Command arguments

Many CICS programming commands, such as [Distributed Program Link](#) on page 11 pass data in a structure known as a COMMAREA. This, and similar 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 or 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 also 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.

### Using the Java Record Framework

The main purpose of the Java Record Framework is to provide run-time support for accessing application record data (such as VSAM files, or COMMAREAs). See [CICS Application Programming Guide](#) for a description of VSAM files.
You can use the framework as the base for record-oriented file input/output, as well as for record-based message passing schemes. You can also use the framework for new applications, and for applications accessing existing files.

The Java Record Framework is part of VisualAge for Java. You can find out more about it in the VisualAge documentation, which is supplied in HTML format with the product.

**System.out and System.err**

If a CICS task is being driven from a terminal (the terminal is called a principal facility in this case) then CICS automatically creates two Java PrintWriters that can be used as standard out and standard error streams and are mapped to the task’s terminal. If the task does not have a terminal as its principal facility, the streams are sent to System.out and System.err. The two streams are public fields in the Task called out and err. System.out and System.err are mapped to the CICS transient data queues CESO and CESE respectively. Your CICS systems 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, or use the DFHSTDWT sample program described in the CICS Customization Guide. DFHSTDWT is supplied with the CICS pregenerated system in CICSTS21.CICS.SDFHLOAD.

**Threads**

Only one thread (the initial thread) can access the CICS API. You can create other threads but you must route all requests to the CICS 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()
- xctI()
- setNextTransaction(), setNextCOMMAREA()
- commit(), rollback()
- returning an AbendException

**Note:** Multiple threads are not supported by ET/390.

**JCICS command reference**

Some of the options and services available through the EXEC CICS API are also available from 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:

- "Error handling and abnormal termination" on page 18
- "CICS exception handling in Java programs" on page 17
- "APPC mapped conversations" on page 19
- "Basic Mapping Support (BMS)" on page 19
- "Diagnostic services" on page 19
- "Environment services" on page 20
- "File services" on page 20
- "Program services" on page 20
- "Scheduling services" on page 20
- "Serialization services" on page 20
- "Storage services" on page 20

16  CICS Transaction Server: Java applications in CICS
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.

In addition, 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 and 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 again 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 that the Language Environment product is installed and active on your OS/390 system in order to run Java applications. You should not define the Language Environment run-time option TRAP=OFF, as 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 each command is supported is described below:

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

HANDLE CONDITION
To handle a specific condition, such as PGMIDERR, you can use a catch clause that names the appropriate exception, in this case InvalidProgramException. Alternatively, you can 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, and 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; the RESP field in the EIB must be checked after each CICS command to detect a condition.

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 27 for a description of the relationship between CICS conditions and Java exceptions.

Error handling and abnormal termination

<table>
<thead>
<tr>
<th>Methods</th>
<th>JCICS class</th>
<th>EXEC CICS commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>abend(), forceAbend()</td>
<td>Task</td>
<td>ABEND</td>
</tr>
</tbody>
</table>
ABEND
To initiate an ABEND from a Java program, you invoke the Task.abend(String) method, supplying an ABEND code. This will cause 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), then CICS will terminate and roll-back the transaction.

ABEND CANCEL
To initiate an ABEND that cannot be handled, you invoke the Task.forceAbend(String) method, supplying an ABEND code. As described above, this will cause 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, it will terminate and roll back the transaction. You should only catch AbendCancelException for notification purposes and then you should re-throw it.

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

APPC mapped conversations:

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

Basic Mapping Support (BMS)

<table>
<thead>
<tr>
<th>Methods</th>
<th>JCICS class</th>
<th>EXEC CICS Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>sendControl()</td>
<td>TerminalPrincipalFacility</td>
<td>SEND CONTROL</td>
</tr>
<tr>
<td>sendText()</td>
<td>TerminalPrincipalFacility</td>
<td>SEND Text</td>
</tr>
<tr>
<td></td>
<td>Not supported</td>
<td>SEND MAP, RECEIVE MAP</td>
</tr>
</tbody>
</table>

Diagnostic services

<table>
<thead>
<tr>
<th>Methods</th>
<th>JCICS class</th>
<th>EXEC CICS Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Not supported</td>
<td>DUMP</td>
</tr>
<tr>
<td>Methods</td>
<td>JCICS class</td>
<td>EXEC CICS Commands</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>enterTrace()</td>
<td>EnterRequest</td>
<td>ENTER</td>
</tr>
<tr>
<td>enableTrace(), disableTrace()</td>
<td>Region, Task</td>
<td>TRACE</td>
</tr>
</tbody>
</table>

### Environment services

CICS environment services provide access to CICS data areas, parameters and resource attributes that are relevant to an application program. The only EXEC CICS commands and options that have equivalent JCICS support are:

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

### ADDRESS

The following support is provided for the ADDRESS options. See [CICS Application Programming Reference](#) for information about the EXEC CICS ADDRESS command.

### COMMAREA

A COMMAREA contains user data that is passed with a command, such as `link()`. The COMMAREA pointer is passed automatically to the linked program by the `CommAreaHolder` argument. See "Command arguments" on page 15 for more information.

### EIB

The EXEC Interface Block 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 that is 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.

### CWA

The Common Work Area (CWA) contains global user data, sharable between tasks. This option is not supported in JCICS.

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

The following support is provided for the ASSIGN options:

<table>
<thead>
<tr>
<th>Methods</th>
<th>JCICS class</th>
<th>EXEC CICS Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>getABCODE()</td>
<td>AbendException</td>
<td>ASSIGN ABCODE</td>
</tr>
<tr>
<td>getAPPLID()</td>
<td>Region</td>
<td>ASSIGN APPLID</td>
</tr>
<tr>
<td>getCWA()</td>
<td>Region</td>
<td>ASSIGN CWAILENG</td>
</tr>
<tr>
<td>getName()</td>
<td>TerminalPrincipalFacility or</td>
<td>ASSIGN FACILITY</td>
</tr>
<tr>
<td></td>
<td>ConversationPrincipalFacility</td>
<td></td>
</tr>
<tr>
<td>getFCI()</td>
<td>Task</td>
<td>ASSIGN FCI</td>
</tr>
<tr>
<td>getNetName()</td>
<td>TerminalPrincipalFacility or</td>
<td>ASSIGN NETNAME</td>
</tr>
<tr>
<td></td>
<td>ConversationPrincipalFacility</td>
<td></td>
</tr>
<tr>
<td>getPrinSysid()</td>
<td>TerminalPrincipalFacility</td>
<td>ASSIGN PRINSYSID</td>
</tr>
<tr>
<td></td>
<td>or ConversationPrincipalFacility</td>
<td></td>
</tr>
<tr>
<td>getProgramName()</td>
<td>Task</td>
<td>ASSIGN PROGRAM</td>
</tr>
<tr>
<td>getQNAME()</td>
<td>Task</td>
<td>ASSIGN QNAME</td>
</tr>
<tr>
<td>getSTARTCODE()</td>
<td>Task</td>
<td>ASSIGN STARTCODE</td>
</tr>
<tr>
<td>getSysid()</td>
<td>Region</td>
<td>ASSIGN SYSID</td>
</tr>
<tr>
<td>getTCTUA()</td>
<td>TerminalPrincipalFacility</td>
<td>ASSIGN TCTUAILENG</td>
</tr>
<tr>
<td>getTERMCODE()</td>
<td>TerminalPrincipalFacility</td>
<td>ASSIGN TERMCODE</td>
</tr>
<tr>
<td>getTWA()</td>
<td>Task</td>
<td>ASSIGN TWAILENG</td>
</tr>
<tr>
<td>getUserid(), Task.getUserid()</td>
<td>Task, TerminalPrincipalFacility or</td>
<td>ASSIGN USERID</td>
</tr>
<tr>
<td></td>
<td>ConversationPrincipalFacility</td>
<td></td>
</tr>
</tbody>
</table>

No other ASSIGN options are supported.

INQUIRE SYSTEM
The following support is provided for the INQUIRE SYSTEM options:

<table>
<thead>
<tr>
<th>Methods</th>
<th>JCICS class</th>
<th>EXEC CICS Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>getAPPLID(), getSYSid()</td>
<td>Region</td>
<td>INQUIRE SYSTEM</td>
</tr>
</tbody>
</table>

No other INQUIRE SYSTEM options are supported.

INQUIRE TASK
The following support is provided for the INQUIRE TASK options:

<table>
<thead>
<tr>
<th>Methods</th>
<th>JCICS class</th>
<th>EXEC CICS Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>getAPPLID(), getSYSid()</td>
<td>Task</td>
<td>INQUIRE TASK FACILITY</td>
</tr>
<tr>
<td>getSTARTCODE()</td>
<td>Task</td>
<td>INQUIRE TASK STARTCODE</td>
</tr>
<tr>
<td>get TransactionName()</td>
<td>Task</td>
<td>INQUIRE TASK TRANSACTION</td>
</tr>
<tr>
<td>getUserid()</td>
<td>Task</td>
<td>INQUIRE TASK USERID</td>
</tr>
</tbody>
</table>

Note:
FACILITY
You can find the name of the task’s principal facility by calling the getName() method on the task’s principal facility, which in turn can 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 or NETNAME
The following support is provided for INQUIRE TERMINAL or NETNAME options:

<table>
<thead>
<tr>
<th>Methods</th>
<th>JCICS class</th>
<th>EXEC CICS Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>Terminal.getUser(), getUserid()</td>
<td>Terminal, ConversationalPrincipalFacility</td>
<td>INQUIRE TERMINAL USERID&lt;br/&gt;INQUIRE NETNAME USERID</td>
</tr>
</tbody>
</table>

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 that are treated by CICS as though they were separate KSDS files in their own right, which means they have separate FD entries. However, there are a few differences between KSDS, ESDS (primary index) and ESDS (secondary index) accessing, which means that they cannot always use a common interface. CICS does not support access to an RRDS through a secondary index. 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 CICS Application Programming Guide for more information about data sets.

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

The Java interfaces relating to File Control therefore separate into five categories:

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

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

KSDS  Containing the interface specific to KSDS files.
ESDS  Containing the interface specific to ESDS files accessed via Relative Byte Address (RBA - its primary index).

RRDS  Containing the interface specific to RRDS files accessed via Relative Record Number (RRN - its primary index).

For each file, there are two objects that can be operated on; the file object and the file browse 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 file browse object represents a browse operation on a file (there can be more than one active browse against a given file at any time, distinguished by a REQID). Methods can be invoked against these objects to perform the following operations:

- ENDBR
- READNEXT
- READPREV
- RESETBR

A file browse 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:

<table>
<thead>
<tr>
<th>KSDS primary or secondary index</th>
<th>ESDS secondary index</th>
<th>CICS File command</th>
</tr>
</thead>
<tbody>
<tr>
<td>KeyedFile.read()</td>
<td>KeyedFile.read()</td>
<td>READ</td>
</tr>
<tr>
<td>KeyedFile.readForUpdate()</td>
<td>KeyedFile.readForUpdate()</td>
<td>READ UPDATE</td>
</tr>
<tr>
<td>KeyedFile.readGeneric()</td>
<td>KeyedFile.readGeneric()</td>
<td>READ GENERIC</td>
</tr>
<tr>
<td>KeyedFile.rewrite()</td>
<td>KeyedFile.rewrite()</td>
<td>REWRITE</td>
</tr>
<tr>
<td>KSDS.write()</td>
<td>KSDS.write()</td>
<td>WRITE</td>
</tr>
<tr>
<td>KSDS.delete()</td>
<td></td>
<td>DELETE</td>
</tr>
<tr>
<td>KSDS.deleteGeneric()</td>
<td></td>
<td>DELETE GENERIC</td>
</tr>
<tr>
<td>File.unlock()</td>
<td>File.unlock()</td>
<td>UNLOCK</td>
</tr>
<tr>
<td>KeyedFile.startBrowse()</td>
<td>KeyedFile.startBrowse()</td>
<td>START BROWSE</td>
</tr>
<tr>
<td>KeyedFile.startGenericBrowse()</td>
<td>KeyedFile.startGenericBrowse()</td>
<td>START BROWSE GENERIC</td>
</tr>
<tr>
<td>KeyedFileBrowse.next()</td>
<td>KeyedFileBrowse.next()</td>
<td>READNEXT</td>
</tr>
<tr>
<td>KeyedFileBrowse.previous()</td>
<td>KeyedFileBrowse.previous()</td>
<td>READPREV</td>
</tr>
</tbody>
</table>
This table shows the classes and methods for non-keyed files. ESDS and RRDS are accessed by their primary indexes:

<table>
<thead>
<tr>
<th>ESDS primary index</th>
<th>RRDS primary index</th>
<th>CICS File command</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESDS.read()</td>
<td>RRDS.read()</td>
<td>READ</td>
</tr>
<tr>
<td>ESDS.readForUpdate()</td>
<td>RRDS.readForUpdate()</td>
<td>READ UPDATE</td>
</tr>
<tr>
<td>ESDS.rewrite()</td>
<td>RRDS.rewrite()</td>
<td>REWRITE</td>
</tr>
<tr>
<td>ESDS.write()</td>
<td>RRDS.write()</td>
<td>WRITE</td>
</tr>
<tr>
<td>File.unlock()</td>
<td>File.unlock()</td>
<td>UNLOCK</td>
</tr>
<tr>
<td>ESDS.startBrowse()</td>
<td>RRDS.startBrowse()</td>
<td>START BROWSE</td>
</tr>
<tr>
<td>ESDS_Browse.next()</td>
<td>RRDS_Browse.next()</td>
<td>READNEXT</td>
</tr>
<tr>
<td>ESDS_Browse.previous()</td>
<td>RRDS_Browse.previous()</td>
<td>READPREV</td>
</tr>
<tr>
<td>ESDS_Browse.reset()</td>
<td>RRDS_Browse.reset()</td>
<td>RESET BROWSE</td>
</tr>
<tr>
<td>FileBrowse.end()</td>
<td>FileBrowse.end()</td>
<td>END BROWSE</td>
</tr>
</tbody>
</table>

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 file browse 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:

<table>
<thead>
<tr>
<th>Methods</th>
<th>JCICS class</th>
<th>EXEC CICS Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>link()</td>
<td>Program</td>
<td>LINK</td>
</tr>
<tr>
<td>SetNextTransaction()</td>
<td>TerminalPrincipalFacility</td>
<td>RETURN</td>
</tr>
<tr>
<td>setNextCOMMAREA()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>xctl()</td>
<td>Program</td>
<td>XCTL</td>
</tr>
<tr>
<td></td>
<td>Not supported</td>
<td>SUSPEND</td>
</tr>
</tbody>
</table>

**LINK and XCTL**

You can transfer control to another program that is defined to CICS using the
link() and xctl() methods. These programs can be in any language supported by CICS, except another JVM program as a CICS task can only have one JVM program in its stack.

**Note:** 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 just terminate as normal. The pseudoconversational functions are supported by methods in the TerminalPrincipalFacility class; setNextTransaction() is equivalent to using the TRANSID option of RETURN, and setNextCOMMAREA() replaces the COMMAREA option. These methods can be invoked at any time during the running of the program, and will 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

<table>
<thead>
<tr>
<th>Methods</th>
<th>JCICS class</th>
<th>EXEC CICS Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>cancel()</td>
<td>StartRequest</td>
<td>CANCEL</td>
</tr>
<tr>
<td>get*() methods</td>
<td>StartRequest</td>
<td>RETRIEVE</td>
</tr>
<tr>
<td>issue()</td>
<td>StartRequest</td>
<td>START</td>
</tr>
</tbody>
</table>

### Serialization services

<table>
<thead>
<tr>
<th>Methods</th>
<th>JCICS class</th>
<th>EXEC CICS Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>dequeue()</td>
<td>SynchronisationResource</td>
<td>DEQ</td>
</tr>
<tr>
<td>enqueue(), tryEnqueue()</td>
<td>SynchronisationResource</td>
<td>ENQ</td>
</tr>
</tbody>
</table>

### 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, and you must ensure that they are of the necessary length.

### Temporary storage queue services

<table>
<thead>
<tr>
<th>Methods</th>
<th>JCICS class</th>
<th>EXEC CICS Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>delete()</td>
<td>TSQ</td>
<td>DELETEQ TS</td>
</tr>
<tr>
<td>readItem(), readNextItem()</td>
<td>TSQ</td>
<td>READQ TS</td>
</tr>
</tbody>
</table>
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, 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.

**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**

<table>
<thead>
<tr>
<th>Methods</th>
<th>JCICS class</th>
<th>EXEC CICS Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>converse()</td>
<td>TerminalPrincipalFacility</td>
<td>CONVERSE</td>
</tr>
<tr>
<td>receive()</td>
<td>TerminalPrincipalFacility</td>
<td>RECEIVE</td>
</tr>
<tr>
<td>send()</td>
<td>TerminalPrincipalFacility</td>
<td>SEND</td>
</tr>
</tbody>
</table>

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

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

---

**Transient data queue services**

<table>
<thead>
<tr>
<th>Methods</th>
<th>JCICS class</th>
<th>EXEC CICS Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>delete()</td>
<td>TDQ</td>
<td>DELETEQ TD</td>
</tr>
<tr>
<td>readData(), readDataConditional()</td>
<td>TDQ</td>
<td>READQ TD</td>
</tr>
<tr>
<td>writeData()</td>
<td>TDQ</td>
<td>WRITEQ TD</td>
</tr>
</tbody>
</table>
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 the CICS NOSUSPEND logic. If a QBUSY condition is detected, it is returned to the application immediately as `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

<table>
<thead>
<tr>
<th>Methods</th>
<th>JCICS class</th>
<th>EXEC CICS Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>commit(), rollback()</td>
<td>Task</td>
<td>SYNCPOINT</td>
</tr>
</tbody>
</table>

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

### JCICS exception mapping

<table>
<thead>
<tr>
<th>CICS condition</th>
<th>Java Exception</th>
<th>CICS condition</th>
<th>Java Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALLOCERR</td>
<td>AllocationErrorException</td>
<td>CBIDERR</td>
<td>InvalidControlBlockIdException</td>
</tr>
<tr>
<td>CCERROR</td>
<td>CCERRORException</td>
<td>DISABLED</td>
<td>FileDisabledException</td>
</tr>
<tr>
<td>DSIDERR</td>
<td>FileNotFoundException</td>
<td>DSSTAT</td>
<td>DestinationStatusChangeException</td>
</tr>
<tr>
<td>DUPKEY</td>
<td>DuplicateKeyException</td>
<td>DUPREC</td>
<td>DuplicateRecordException</td>
</tr>
<tr>
<td>END</td>
<td>EndException</td>
<td>ENDDATA</td>
<td>EndOfDataException</td>
</tr>
<tr>
<td>ENDFILE</td>
<td>EndOfFileException</td>
<td>ENDINPT</td>
<td>EndOfFileIndicatorException</td>
</tr>
<tr>
<td>ENQBUSY</td>
<td>ResourceUnavailableException</td>
<td>ENVDEFERR</td>
<td>InvalidRetrieveOptionException</td>
</tr>
<tr>
<td>EOC</td>
<td>EndOfChainIndicatorException</td>
<td>EODS</td>
<td>EndOfDataSetIndicatorException</td>
</tr>
<tr>
<td>EOF</td>
<td>EndOfFileIndicatorException</td>
<td>ERROR</td>
<td>ErrorException</td>
</tr>
<tr>
<td>EXPIRED</td>
<td>TimeExpiredException</td>
<td>FILENOTFOUND</td>
<td>FileNotFoundException</td>
</tr>
<tr>
<td>FUNCERR</td>
<td>FunctionErrorException</td>
<td>IGREQID</td>
<td>InvalidREQIDPrefixException</td>
</tr>
</tbody>
</table>

Table 2. Java exception mapping
Table 2. Java exception mapping (continued)

<table>
<thead>
<tr>
<th>CICS condition</th>
<th>Java Exception</th>
<th>CICS condition</th>
<th>Java Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>IGREQCD</td>
<td>InvalidDirectionException</td>
<td>ILLOGIC</td>
<td>LogicException</td>
</tr>
<tr>
<td>INBFMH</td>
<td>InboundFMHException</td>
<td>INVERRTERM</td>
<td>InvalidErrorTerminalException</td>
</tr>
<tr>
<td>INVEXITREQ</td>
<td>InvalidExitRequestException</td>
<td>INVLCDC</td>
<td>InvalidLDCException</td>
</tr>
<tr>
<td>INVMPSZ</td>
<td>InvalidMapSizeException</td>
<td>INVPARTNSET</td>
<td>InvalidPartitionSetException</td>
</tr>
<tr>
<td>INVPARTN</td>
<td>InvalidPartitionException</td>
<td>INVREQ</td>
<td>InvalidRequestException</td>
</tr>
<tr>
<td>INVTSREQ</td>
<td>InvalidTSRequestException</td>
<td>IOERR</td>
<td>IOException</td>
</tr>
<tr>
<td>ISCINVREQ</td>
<td>ISCInvalidRequestException</td>
<td>ITEMERR</td>
<td>ItemErrorException</td>
</tr>
<tr>
<td>JIDERR</td>
<td>InvalidJournalIdException</td>
<td>LENGER</td>
<td>LengthErrorException</td>
</tr>
<tr>
<td>MAPERROR</td>
<td>MapErrorException</td>
<td>MAPFAIL</td>
<td>MapFailureException</td>
</tr>
<tr>
<td>NAMEERROR</td>
<td>NameErrorException</td>
<td>NODEIDERR</td>
<td>InvalidNodeIdException</td>
</tr>
<tr>
<td>NOJBUFSP</td>
<td>NoJournalBufferSpaceException</td>
<td>NONVAL</td>
<td>NotValidException</td>
</tr>
<tr>
<td>NOPASSBKRD</td>
<td>NoPassbookReadException</td>
<td>NOPASSBKWR</td>
<td>NoPassbookWriteException</td>
</tr>
<tr>
<td>NOSPACE</td>
<td>NoSpaceException</td>
<td>NOSPOOL</td>
<td>NoSpoolException</td>
</tr>
<tr>
<td>NOSTART</td>
<td>StartFailedException</td>
<td>NOSTG</td>
<td>NoStorageException</td>
</tr>
<tr>
<td>NOTALLOC</td>
<td>NotAllocatedException</td>
<td>NOTAUTH</td>
<td>NotAuthorisedException</td>
</tr>
<tr>
<td>NOTFND</td>
<td>RecordNotFoundException</td>
<td>NOTOPEN</td>
<td>NotOpenException</td>
</tr>
<tr>
<td>OPENERR</td>
<td>DumpOpenErrorException</td>
<td>OVERFLOW</td>
<td>MapPageOverflowException</td>
</tr>
<tr>
<td>PARTNFAIL</td>
<td>PartitionFailureException</td>
<td>PGMIDERR</td>
<td>InvalidProgramIdException</td>
</tr>
<tr>
<td>QBUSY</td>
<td>QueueBusyException</td>
<td>QIDERR</td>
<td>InvalidQueueIdException</td>
</tr>
<tr>
<td>QZERO</td>
<td>QueueZeroException</td>
<td>RDATT</td>
<td>ReadAttentionException</td>
</tr>
<tr>
<td>RETPAGE</td>
<td>ReturnedPageException</td>
<td>ROLLEDBACK</td>
<td>RolledBackException</td>
</tr>
<tr>
<td>RTEFAIL</td>
<td>RouteFailedException</td>
<td>RTESOME</td>
<td>RoutePartiallyFailedException</td>
</tr>
<tr>
<td>SELNERR</td>
<td>DestinationSelectionErrorException</td>
<td>SESSBUSY</td>
<td>SessionBusyException</td>
</tr>
<tr>
<td>SESSIONERR</td>
<td>SessionErrorException</td>
<td>SIGNAL</td>
<td>InboundSignalException</td>
</tr>
<tr>
<td>SPOLBUSY</td>
<td>SpoolBusyException</td>
<td>SPOLErr</td>
<td>SpoolErrorException</td>
</tr>
<tr>
<td>STRELERR</td>
<td>STRELERGErreption</td>
<td>SUPPRESSED</td>
<td>SuppressedException</td>
</tr>
<tr>
<td>SYSBUSY</td>
<td>SystemBusyException</td>
<td>SYsidERR</td>
<td>InvalidSystemIdException</td>
</tr>
<tr>
<td>TASKIDERR</td>
<td>InvalidTaskIdException</td>
<td>TCIDERR</td>
<td>TCIDERRException</td>
</tr>
<tr>
<td>TERMERR</td>
<td>TerminalException</td>
<td>TERMIDERR</td>
<td>InvalidTerminalIdException</td>
</tr>
<tr>
<td>TRANSIDERR</td>
<td>InvalidTransactionIdException</td>
<td>TSIoERR</td>
<td>TSIoErrorException</td>
</tr>
<tr>
<td>UNEXPIN</td>
<td>UnexpectedInformationException</td>
<td>USERIDERR</td>
<td>InvalidUserIdException</td>
</tr>
<tr>
<td>WRBRK</td>
<td>WriteBreakException</td>
<td>WRONGSTAT</td>
<td>WrongStatusException</td>
</tr>
</tbody>
</table>

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 then 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
Writing the main method

CICS will attempt to pass control to method `main(CommAreaHolder)` in the class specified by the JVMCLASS option of the PROGRAM resource definition for Java programs executed under control of the JVM, or by the hpj `-main` option for Java programs processed by the ET/390 binder. If this method is not found, it will try to invoke `main(String[])` from that class.

Creating objects

To create an object you need to:

- Declare a reference, for example:
  
  ```java
  TSQ tsq;
  ```

- Use the `new` operator to create an object
  
  ```java
  tsq = new TSQ();
  ```

- Use the `setName` method to give it a name
  
  ```java
  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:

```java
// 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());
    }
  }

  // A method to check the queue operations
  public static void checkQueue() {
    try {
      // Create a queue
      TSQ tsq = new TSQ();
      tsq.setName("JCICSTSQ");

      // Write items to the queue
      String[] transactions = {"Transaction 1", "Transaction 2", "Transaction 3"];`
// Return from the application
return;
}
Sample programs are provided to demonstrate the use of JCICS classes and the combination of Java programs with CICS programs in other languages.

The Java source files are installed in OS/390® UNIX® System Services HFS with makefiles to build the sample programs to execute in the CICS JVM.

The sample programs are run by entering a transaction on a 3270 CICS screen. The following sample programs are provided:

**HelloWorld samples**
Two simple 'Hello World' programs are supplied. The JHE1 transaction runs a sample that uses only Java services and the JHE2 transaction runs a sample that uses JCICS. The JCICS sample demonstrates the use of the JCICS TerminalPrincipalFacility class.

**ProgramControl sample**
This sample demonstrates the use of the JCICS Program class. A transaction, JPC1, invokes a Java class that constructs a COMMAREA and LINKs to a C program (DFH$LCCA) that processes the COMMAREA, updates it and returns. The Java program then checks the data in the COMMAREA and schedules a pseudoconversational transaction to be started with the changed data in its COMMAREA.

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.

**TDQ transient data sample**
This sample shows you how to use the TDQ class. It consists of a single transaction, JTD1, which 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 TSQ class. It consists of a single transaction, JTS1, which invokes a single Java class, TSQ.ClassOne, and uses an AUXILIARY temporary storage queue. This sample also shows you how to build a 'Common' class as a dll, which can be shared with other Java programs.

---

**Building the sample programs**

The Java source and makefiles are stored in the OS/390 UNIX System Services HFS during CICS installation. To build the samples in the OS/390 UNIX System Services environment, you will need to define the following environment variables. You can define these in the profile for the OS/390 UNIX System Services (for example) using the `export` command, or you can enter the export command manually when OS/390 UNIX System Services is running.

**CICS_HOME**
The installation directory prefix of CICS TS. This is

```
/usr/lpp/cicsts/cicsts21
```

where `cicsts21` is defined by the USSDIR installation parameter when you installed CICS TS (defaulting to cicsts21). For example:
export CICS_HOME=/usr/lpp/cicsts/cicsts21

JAVA_HOME
This specifies the path for IBM Developer Kit for OS/390, Java 2 Technology Edition subdirectories. This is
/usr/lpp/java130s/J1.3/

where /java130s/J1.3/ is defined when you install the IBM Developer Kit for OS/390.

- $CICS_HOME/samples/dfjcics contains the makefiles.
- $CICS_HOME/samples/dfjcics/examples contains the Java source

The following CICS C language programs are stored in SDFHSAMP during CICS installation. They are LINKed by the Java sample programs. You need to compile and translate these supplied C programs and link them into a load library in the CICS DFHRPL concatenation, and define all CICS resources as described in "Defining CICS resources":

- DFH$LCCA
- DFH$JSAM

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.

Building the Java samples
To run the samples using the JVM, 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 HFS directory.

Defining CICS resources
CICS Resource definitions for all the sample PROGRAMS and TRANSACTIONS are supplied in the group DFH$JVM. You must install this group in order to run the samples.

Running the Java samples
Follow the procedure described to run each sample:
- "Running the Hello World sample" on page 33
- "Running the Program Control sample" on page 33
- "Running the TDQ sample" on page 33
- "Running the TSQ sample" on page 34
Running the Hello World sample

This sample uses the following Java classes:

- HelloWorld (PROGRAM name DFJ$JHE1)
- HelloCICSWorld (PROGRAM name DFJ$JHE2)

and the following C language CICS program:

- DFH$JSAM

Run the JHE1 CICS transaction to execute the Java standard application, or the JHE2 transaction to execute the JCICS application. You should receive the following message from JHE1 on System.out:

Hello from a regular Java application

and the following message from JHE2 on Task.out:

Hello from a Java CICS application

Running the Program Control sample

This sample uses the following Java classes:

- ProgramControlClassOne (PROGRAM name DFJ$JPC1)
- ProgramControlClassTwo (PROGRAM name DFJ$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()

Running the TDQ sample

This sample uses the following Java class:

- TDQ.ClassOne (PROGRAM name DFJ$JTD1)

and the following C language CICS program:

- DFH$JSAM

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 class:

- 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()
Part 3. Using the JVM

This Part tells you what you need to know to use the Java Virtual Machine (JVM) in CICS. It covers the following topics:

- "Chapter 6. About the JVM" on page 37
- "Chapter 7. Configuring the JVM" on page 47
Chapter 6. About the JVM

CICS provides the support you need to run a Java application program in an OS/390 Java Virtual Machine (JVM) executing under the control of a CICS region. CICS support for the JVM 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 OS/390 platform at http://www.s390.ibm.com/java/

The JVM created by IBM Developer Kit for OS/390 Java 2 Technology Edition Version 1.3.S (the persistent reusable JVM, or just reusable JVM for short) includes two optimizations designed for the execution of CICS transactions. These are:

- The serial reuse of a JVM for multiple transactions, avoiding most of the initialization costs. Serial reuse entails resetting the state of the JVM between uses.
- An optimized garbage collection scheme, enabled by the clean separation of short lived application objects from the long-lived classes, objects and native state (that is, non-Java or C language), which are reset.

Enabling serial reuse of the JVM

Serial reuse of a JVM is enabled by dividing the classes contained in the JVM into three parts:

- The OS/390 JVM code, which provides the base services in the JVM.
- The middleware, which provides services that access resources. These include the JCICS interfaces classes, JDBC, JNDI, and so on.
- The user application.

Middleware classes have privileges that are not available to the application, and which enable optimizations through the caching of state (loading of classes and native libraries, for example) to be used by multiple applications. However, middleware is also responsible to reset itself correctly at the end of a transaction and, if necessary, to reinitialize at the beginning of a new transaction in order to isolate different applications from each other. Classes are classified as middleware by virtue of their inclusion on the ibm.jvm.trusted.middleware.class.path. The trusted middleware class path property is built automatically by CICS from the paths specified on the CICS_DIRECTORY, JAVA_HOME, TMPREFIX, and TMSUFFIX parameters defined in the JVM profile (see the CICS System Definition Guide for details of these parameters and the trusted middleware class path respectively).

Not all applications are able to exploit serial reuse. If the application uses Java interfaces that modify the state of the JVM in a way that can't be safely reset (such as changing system properties, closing the standard output stream, or loading a native library), the JVM is not reused. The storage used by the JVM is recovered and a new JVM is initialized to provide a safe environment for subsequent applications. The JVM monitors the use of interfaces that prevent safe resetting, and the events that prevent reuse are logged.

Enterprise beans and CICS programs that execute on a single Java thread using interfaces defined by the Enterprise JavaBeans specification, or by the JCICS classes, are normally able to exploit serial reuse.
You can run the JVM in a mode that does not attempt serial reuse by specifying Xresettable=NO in the JVM profile, but this should be necessary only if the application needs to use Java facilities that modify the state of the JVM in an uncontrolled way.

Application classes are defined as follows:

- For classes that are enterprise beans, CICS manages the loading of the JAR files by means of the DJAR definitions.
- For classes that are executed directly through a CICS program definition, or CORBA applications:
  - By inclusion on the CLASSPATH setting in the JVM profile.
  - By inclusion in the ibm.jvm.shareable.application.class.path system property, defined in the system properties file referenced by the JVMPROPS parameter in the JVM profile.

Defining classes in the ibm.jvm.shareable.application.class.path system property provides additional optimization by caching the classes in the JVM and reinitializing them. This is the recommended configuration for the best performance. If you use the CLASSPATH, classes are reloaded from HFS files each time the JVM is reused.

- For utility classes:
  - If the utility classes are used by enterprise beans, they must be defined in the ibm.jvm.shareable.application.class.path system property.
  - If the utility classes are used by classes that are executed directly through a CICS program definition, the utility classes can be defined in CLASSPATH or in the ibm.jvm.shareable.application.class.path system property.

OS/390 JVM classes have a special status that allows the objects they create to be associated with middleware or the application, depending on the kind of class that invokes their construction. The arrangement of the objects at run-time is explained in more detail below. The OS/390 JVM classes do not need to be included on a classpath.

The run-time structure of the JVM

The reusable JVM manages run-time storage in several segregated heaps whose characteristics can, to some extent, be individually tuned using parameters in the JVM profile. These are:

**The transient heap**

This heap contains objects constructed by application classes, and any objects constructed by OS/390 JVM classes as a result of calls from application classes. It also contains any application classes, including their static data, that are loaded from the CLASSPATH. Segregation of this heap from the middleware heap improves performance of garbage collection.

**The middleware heap**

This heap contains objects constructed by middleware classes and any objects constructed by OS/390 JVM classes as a result of calls from middleware classes. It also contains static data for the middleware classes and the OS/390 JVM classes and other string constant data.

**The system heaps**

The main system heap contains the class definitions for all the classes except
those application classes loaded from the CLASSPATH that are reloaded every time they are used. This includes the middleware class definitions and the pooled string constant data.

The other system heap contains cached application class definitions. This is called the application class system heap (ACSH).

---

**How CICS manages JVMs in a pool**

CICS maintains a pool of JVMs, in which JVMs may be in use or available for reuse, as shown in Figure 1.

---

Each JVM begins execution on an MVS TCB allocated from a pool of open TCBs managed by CICS. Open TCBs can also be used for Java hotpooling, and the total number of TCBs is limited by the MAXOPENTCBS system initialization parameter. CICS controls the numbers of TCBs of each type and adjusts the number in response to the work load. You should adjust the MAXOPENTCBS setting according to the amount of storage below 16M that is available in your system. You should also restrict the number of active transactions in the system (through the MXT system initialization parameter, for example) to maintain a JVM pool that always has JVMs free to satisfy new requests. CICS reduces the number of active JVMs automatically if the work load does not require them.

---

**How CICS selects the right type of JVM**

JVMs are allocated to a CICS transaction that requests execution of a Java program. This is illustrated in Figure 2 on page 40. The JVM characteristics (for example, heap size and class path) required by the Java program are defined by naming a JVM profile on the CICS program resource definition, which also indicates the static main method that is the entry point of the application program. In the case of enterprise beans, the entry point is the CICS CORBA request processor, supplied with CICS.
The selection mechanism matches requests with a JVM that has the correct configuration to run them. The configuration is identified by the profile name that was used to create the JVM.

If CICS can’t find a JVM of the right type, and cannot create a new JVM because of the MAXOPENTCBS limit, CICS re-initializes an existing JVM to ensure the required characteristics. The best performance is achieved when there are many more JVMs than the number of JVM types. The fewer the number of JVM types you have, the more chance there is of an existing JVM matching the program requirement, thus avoiding the overhead of re-initialisation.

**Use of resource definitions for JVM selection**

The specification of the JVM environment suitable for a request is found by reference to the CICS program definition. The program resource definition is determined from the request, which could originate as one of the following:

- An EJB request that matches a REQUESTMODEL, which specifies a transaction identifier
- A 3270 or START request that specifies a transaction identifier
- An EXEC CICS LINK request, or an ECI or EXCI call that names the program directly
- An entry in a program list table post-initialization (PLTP1)
The first two of these types of request are shown in Figure 3.

Some IIOP and EJB requests are processed using an existing request processor transaction. Such requests run in existing JVM environments, even if there is a REQUESTMODEL that matches the requests. The TRANSID in any matching REQUESTMODEL is used only when a new request processor transaction is required.

In each case, a program definition is used to specify the name of the Java class whose public static main method is to be invoked, and the characteristics of the JVM needed to run it.

![Figure 3. Resolution of Java class name and JVM profile to determine JVM type](image)

**Selecting an open TCB for the JVM**

CICS manages a pool of open TCBs up to the limit set by the MAXOPENTCBS system initialization parameter. At any one time, the pool can consist of some TCBs that are currently allocated to tasks, and others that have been allocated but are now are free. For example, if the maximum number of open TCBs is set at 100, at a particular time the pool could consist of 50 open TCBs, not all of which are allocated. CICS attaches a new TCB when it can't find a suitable match with a free TCB.

To run a Java application program in a JVM, CICS needs a J8 mode TCB under which it can run the JVM. CICS associates the JVM profile name with a J8 mode TCB when a JVM is created. When selecting a TCB to allocate from the pool of open TCBs, CICS searches for an exact match with the JVM profile. If a matching TCB is not already available in the pool, and the maximum number of open TCBs has not been reached, CICS creates a new TCB and JVM to associate with the JVM profile. If the number of open TCBs is at the limit, and there are no free J8
mode TCBs, the least recently used of all the currently free open TCBs is deleted and a new TCB allocated. If all TCBs are in use, the requesting task waits until an existing task terminates and frees the TCB it was using. The process of matching J8 mode TCBs to JVM requests, based on the JVM profile, is summarized in the following steps:

1. If the transaction already has a J8 mode TCB allocated, it is used.
2. If there is a free J8 mode TCB that matches the JVM profile, it is allocated and used.
3. If the number of open TCBs is below the MAXOPENTCBS limit, a new J8 mode TCB is created, and associated with the JVM profile.
4. If the number of open TCBs is at the MAXOPENTCBS limit, but there is a free J8 mode TCB that does not match the JVM profile it is used. In this case, a new JVM has to be created and its profile associated with the TCB.
5. If the number of open TCBs is at the MAXOPENTCBS limit, and there is not a free J8 mode TCB, but there is a free open TCB of another mode, it is destroyed and a J8 mode TCB is created, together with a new JVM. This process is called stealing and, although this involves a high overhead, it enables the request to be satisfied.
6. If the number of open TCBs is at the MAXOPENTCBS limit and there is no free open TCB of any mode, the task is suspended until one becomes free.

CICS uses a similar allocation process to allocate H8 TCBs for hpj-compiled Java programs running in hot-pooling mode (see [H8 mode TCB allocation for hot-pooling" on page 273 for details]). If hot-pooling and JVM Java programs run in the same CICS region, contention for open TCBs can cause allocation requests to steal free TCBs that do not match in order to satisfy a request, resulting in high overhead. Because JVMs incur more overhead to initialize than a hot-pooling enclave, you should try to preserve the pool of JVMs by protecting them from TCB stealing to satisfy HPJ program requests. You can use transaction classes (TRANCLASS resource definitions) to achieve the necessary separation.

CICS restrictions on use of persistent reusable JVM

The JVM introduced in CICS TS 1.3 is not supported, and any Java programs that executed under CICS TS 1.3 must be migrated to Java 2 to run under the reusable JVM. Application migration issues are discussed at: http://java.sun.com/products/jdk/1.3/compatibility.html#incompatibilities1.3 and in http://java.sun.com/products/jdk/1.2/compatibility.html. Support for the reusable JVM completely replaces the JVM support provided in CICS TS Release 3, but configuration options allow the reusable JVM to be run in the same mode with small modifications to your customized initialization options. This might be necessary to execute programs that use Java interfaces that make the JVM non-resettable, such as multi-threading. It might also be necessary for compatibility reasons: for example, the old mode calls DFHJVMAT, which is not available in the new mode.

Native code migration issues are described by the read.me file that is in the /usr/lpp/java130s/J1.3/doc directory when you have installed IBM Developer Kit for OS/390, Java 2 Technology Edition (the reusable JVM).

A stack of programs formed by a succession of EXEC CICS LINK commands, or JCICS program invocations within the same CICS task, cannot contain more than one JVM. Distributed program link (DPL) requests are not restricted in this way.
Note: This is a restriction for CORBA client Java applications, which execute through the VisualAge for Java, Enterprise ToolKit for OS/390 bytecode binder in CICS TS Release 3 and can make local EXEC CICS LINK calls (directly or through an intermediate program) to a JVM. Now, the CORBA client applications execute in a JVM and cannot, therefore, make the same call.

Resource definitions for the JVM

In common with non-Java applications, CICS requires that you define the resources needed to run a Java application in a JVM. The JVM is started by the CICS Java launcher, which uses a set of standard options that are supported in the reusable JVM runtime environment. There are also some non-standard options that are subject to change in future releases of the Java language specification.

The first step is to specify the appropriate Java attributes on the CICS program resource definition. These are as follows:

**JVM**
Specify YES if the program is a Java bytecode program that requires a JVM in which to execute.

**JVMCLASS**
Specify the name of the main class in the Java program. Note that class name in this context includes the package name as well as the Java class, with a period (.) used as a separator. Also, names are case sensitive and must be entered with exactly the correct spelling. For example, `com.ibm.cics.iiop.RequestProcessor`, which is the class specified on the CICS IIOP request processor program, DFJIIRP.

Note: It is possible that the CICS terminal that you use for specifying CICS resource definitions in the CSD (using CEDA, for example) is normally defined with UCTRAN(YES) or UCTRAN(TRANID). In this case, you can use the CEOT transaction to switch UCTRAN mode for the session only, enabling you to enter mixed-case input.

**JVMPROFILE**
Specify the name of the profile that CICS is to use to provide the JVM characteristics. Among other things, the JVM profile specifies the name of the JVM properties file.

Thus, the program resource definition refers to the name of a JVM profile, which in turn specifies the name of a JVM properties file. CICS provides two sample profiles, DFHJVMPR and DFHJVMP. These refer to their corresponding properties file, dfjjvmp.props and dfjjvmp.props as follows:

- Profile DFHJVMPR specifies JVMPROPS=dfjjvmp.props
- Profile DFHJVMP specifies JVMPROPS=dfjjvmp.props

CICS loads the JVM profile from the PDS named on the DFHJVM DD statement in the CICS startup JCL. You can maintain JVM profile members in a PDS using MVS text editing facilities in TSO.
Authorizing CICS region userids to UNIX system services

In addition to the traditional DFSMS-managed data sets, such as partitioned data sets (PDS), CICS requires access to UNIX system services and the hierarchical file store (HFS) in order to create a JVM. This means that the CICS region userid must be defined to UNIX system services with a user identifier (UID). This is the first step in enabling CICS to access the required HFS directories and files, such as those specified on the JVMPROPS, LIBPATH, CLASSPATH, and WORK_DIR parameters.

When a CICS region requests an OS/390 UNIX function for the first time, RACF:
- Verifies that the user (the CICS region user ID) is defined as an OS/390 UNIX user.
- Verifies that the user’s current connect group is defined as an OS/390 UNIX group.
- Initializes the control blocks needed for subsequent security checks.

To ensure the CICS region meets these security requirements:
- Add a GID to the RACF group profile for the RACF group that is to be defined as the default group of the CICS region userid
- Add a UID to the RACF user profile for the CICS region user ID, and connect the user ID to the RACF group that has the required GID

For the HFS directories specified by the CLASSPATH, LIBPATH, and JVMPROPS parameter, the CICS region requires read and execute access only. However, for the WORK_DIR directory path that CICS uses for the stdout and stderr files, CICS requires write access. Table 3 gives you the basic minimum list of directories and files to which CICS requires access in order to create a JVM to execute a Java program. Add to this any of your own directories and files that you need for your applications. If the JVM is to run enterprise beans, add to this list the extra permissions needed for EJB support.

Table 3. JVM directories and files

<table>
<thead>
<tr>
<th>Directories and files</th>
<th>Minimum group permissions</th>
</tr>
</thead>
<tbody>
<tr>
<td>The /usr/lpp/cicsts/cicsts21/props directories containing the JVM system properties file (see note 1).</td>
<td>Read and execute</td>
</tr>
<tr>
<td>The /usr/lpp/cicsts/cicsts21/lib directories (see note 2). The /lib directory contains the CICS-supplied JAR files, such as dfjcics.jar and dfjcsi.jar.</td>
<td>Read and execute</td>
</tr>
<tr>
<td>The /usr/lpp/java130s/J1.3/bin and /usr/lpp/java130s/J1.3/bin/classic directories that contain the IBM Java 2 persistent reusable JVM code (see note 3).</td>
<td>Read and execute</td>
</tr>
<tr>
<td>The working directory required for CICS regions, as specified on the WORK_DIR parameter in the JVM profile.</td>
<td>Read, write and execute</td>
</tr>
</tbody>
</table>

Notes:
1. This is the default directory used for the CICS-supplied properties files, dfjjvmpr.props and dfjjvmps.props, when you install CICS TS.
2. The cicsts21 subdirectory name is a user-defined value which you specify (optionally) in the DFHISTAR installation job.
3. The java130s/J1.3 subdirectory names are the default values when you install the IBM Developer Kit for OS/390, Java 2 Technology Edition.
For information about RACF facilities for controlling access to UNIX system services, see the *OS/390 Security Server RACF Security Administrator’s Guide*, SC28-1915.

For information about the UNIX facilities that you can use to control access to HFS files and directories, see *OS/390 UNIX System Services Planning*, SC28-1890

**Group naming considerations**

You grant other users permissions to HFS resources by connecting them, through their user profile, to a group defined with the GID associated with the directories or files, or by setting the permissions for everyone to have access. The latter is not recommended for use in your production environment, and you should plan your directory and file permissions accordingly. To do this requires planning, and giving some thought to naming conventions for the groups to which you want to connect your CICS regions.

**The /usr/lpp/cicsts/cicsts21/lib and classes directories**

These directories and file contents are created during the installation of CICS TS. As a result, they are associated with the UID and GID of the system programmer who installs the product. This may not be the GID of the group you want to use, in which case you can change the GID to one owned by a more suitable group name. For example, you might want to associate these directories with a system-wide group name such as CICSTS21, with GID of 610..

**The /usr/lpp/java130s/J1.3/bin and classic directories**

These directories and file contents are created during the installation of IBM Developer Kit, Java 2 Technology Edition, and are associated with the UID and GID of the system programmer installing the product. As in the case of CICS TS, consider changing to a more suitable group name, such as IBMJAVA2.

To change the group (GID) of a file, the superuser or the file owner can enter a `chgrp` command, specifying either a RACF group name or a GID. The file owner must have the new group as his group or one of his supplementary groups.
Chapter 7. Configuring the JVM

This section contains the following topics:

- "Defining resources for the JVM"
- "Debugging in the CICS JVM"

Defining resources for the JVM

To enable CICS to create a JVM for the execution of a Java program:

1. Create the JVM profile that specifies the options that are suitable for your Java program and store this profile as a member of a partitioned data set (PDS).
2. Add a DFHJVM DD statement, referencing the PDS containing the JVM profiles, to the CICS startup JCL.
3. Specify the name of the JVM profile on the Java program’s resource definition.

The Java launcher options that you want CICS to use when creating a JVM are specified as a set of JVM options that are read by CICS from the DFHJVM data set. To help you get started, CICS supplies two versions of JVM initialization options, referred to as JVM profiles, as members of XDFHENV. These CICS-supplied JVM profiles are called DFHJVMPR and DFHJVMPS. DFHJVMPR specifies the resettable option for persistent, reusable JVM; DFHJVMPS specifies a JVM that is used once only and destroyed.

To enable CICS to run a JVM, specify in your CICS startup job stream a DD statement for DDNAME DFHJVM that references the PDS containing the JVM profiles. Programs that require a JVM are specified with JVM(YES) in their PROGRAM resource definition. The following is an example of the DD statement for DFHJVM:

//DFHJVM DD DSN=CICSTS21.CICS.XDFHENV,DISP=SHR

Note: Do not specify a member name on the data set defined by DFHJVM. CICS obtains the member name from the program resource definition for the Java program that requires the JVM.

Specify the JVM profile required for your Java program on the JVMPROFILE attribute of the CICS program resource definition.

You can modify the supplied values in the sample JVM profiles, using the TSO editor.

For JVMs that are not resettable, you can also specify that you want the user-replaceable module DFHJVMAT to be called at JVM initialization, to examine and modify the options.

Debugging in the 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 CICS 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 4.

**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 understands 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, some JVM initialization options must be set. The required options are specified for a CICS JVM via a JVM profile in the DFHJVM data set. See CICS System Definition Guide for more information.

The specific options required for debugging are as follows:

Xdebug=YES
This is needed to start the JVM in debug mode (that is, with the JPDA interfaces active)

Xnoagent=YES
This option disables an emulation mode that the JVM provides for compatibility with older (Java 1.1.x) debuggers. This emulation mode is not supported.

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=yes) 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.

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.

Xresettable=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.

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.
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. The default profiles DFHJVMPR and DFHJVMPS are used by internal CICS Java programs (for example when enterprise beans are installed and published) so you should not normally configure these profiles for debug. Instead you should create a separate JVM profile specifically for debug use, and set the appropriate CICS program definition (for example, program definition DFJIIRP for enterprise beans) to use this debug JVM profile. Note that to modify CICS 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.

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 CICS 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 (for example VisualAge for Java provides an option that can be set when a compiled application is exported from the IDE).

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 the 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 class name 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.

There are three Java exit points:

- A CICS EJB container debug plugin providing methods that are called immediately before and after an EJB method is invoked.
- A CICS CORBA debug 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 CICS Java program is invoked.

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 13 for more information).

The code fragment in Figure 5 shows an example implementation of the DebugControl interface.

```java
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 {
    // initialiser, called when plugin is registered
    public void init();
}
```

**Figure 5. Definitions of the DebugControl and Plugin interfaces**

The code fragment in Figure 6 on page 52 shows an example 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:

- `com.ibm.cics.server.debug.EJBPlugin = <classname>`
  
  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 = <classname>`
  
  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 = <classname>`
  
  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 CICS 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.

See [CICS System Definition Guide](#) for more information.

```java
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 6. Sample implementation of the DebugControl and Plugin interfaces*
This Part tells you what you need to know to configure CICS to support distributed IIOP applications. It covers the following topics:

- "Chapter 8. IIOP support in CICS" on page 55
- "Chapter 9. The IIOP request flow" on page 59
- "Chapter 10. Configuring CICS for IIOP" on page 71
- "Chapter 11. Processing IIOP requests" on page 79
Chapter 8. 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 2.1.

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 the CORBA 2.1 API, except for Dynamic Invocation Interface (DII), Dynamic Skeleton Interface (DSI), and GIOP fragments.
- Support for IIOP 1.1
- Support for both inbound and outbound IIOP requests. IIOP applications can act as both client and server.
- Support for transactional objects. CICS Transaction Server for z/OS, Version 2 Release 1 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 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, then 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 9]

ORB function is implemented in CICS by:
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 are supported in JVM mode only. The VisualAge for Java, Enterprise Edition for OS/390 bytecode binder cannot be used.

You can read more about CICS stateless CORBA objects in Chapter 23.

**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 applications using the CICS 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 will not be portable to a non-CICS platform.

Enterprise beans are supported in JVM mode only. The VisualAge for Java, Enterprise Edition for OS/390 bytecode binder cannot be used.

You can read more about enterprise beans in Chapter 12. What are enterprise beans?

Some common CORBA terminology

The following terms are used throughout this information segment:

**OMG**  The Object Management Group. The consortium of software organizations that has defined the CORBA architecture.

**CORBA**  The Common Object Request Broker Architecture. An architecture and a specification for distributed object-oriented computing.

**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 support common CORBA protocols and IDL.
IIOP  The Internet Inter-Orb Protocol. An industry standard that defines formats and protocols to provide client/server semantics for distributed object-oriented applications in a TCP/IP network. It is part of the CORBA architecture.

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.

Module  An IDL packaging construct containing interfaces. This maps to a Java package.

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 class. In CORBA terminology, the client request specifies, in IDL, an interface that defines the server object.

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.

IOR  Interoperable Object Reference. In a distributed environment this provides enough information to locate the server and the object.

Stub or proxy  This is generated by the client IDL compiler. It is used by the ORB to convert a local object reference to an IOR, and invoke translation of object datatypes from/to the IIOP message syntax.

Skeleton  This is generated by the server IDL compiler. It is used by the ORB to parse the message into a method call on a local (to the server) object.
Chapter 9. The IIOP request flow

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

![Diagram of the IIOP request execution flow]

**Figure 7. 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 DFHIRRS, 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 Module (URM) 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 received, 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 (via its associated request stream). The directory, DFHEJDIR, relates request streams (and request processor instances) to OTS transactions and 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**
LocateRequests have no operation or parameters. They are passed to a new instance of the request processor.

**CancelRequest**
A CancelRequest message notifies a server that the client is no longer expecting a reply for a specified pending Request or LocateRequest message. This is an advisory message only, no reply is expected. A CancelRequest received during fragment processing causes the request in progress to be terminated. All other CancelRequest messages are ignored.

**MessageError**
A MessageError 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.

**Request processor**
The request processor manages the execution of the IIOP request. It:
- locates the object identified by the request
- calls the container to process the bean method for an enterprise bean request
- processes the request itself for a request for a stateless CORBA object, (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 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.

**Note:** The listener regions and AORs may be separate or combined into listener/AORs.

---

**Workload balancing of IIOP requests**

To balance client connections across the listener regions, you can use IP routing or connection optimization by means of Domain Name System (DNS) registration, and you can use CICSPlex SM or the CICS distributed routing program to balance OTS transactions across a set of cloned AORs.

**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 call invocations across CICS application owning regions (AORs). The dynamic selection of the target is made using CICS services, or CICSPlex SM Workload Management, to select the least loaded or most efficient application region. CICS invokes a customized version of the distributed routing program, DFHDSRP, 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 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.
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. WLM must be configured in goal mode on all hosts within the sysplex. TCP/IP stacks can also register with 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 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 WLM for registration.

**Name resolution example**

The following diagram shows an example cicsplex consisting of 4 CICS regions, each executing on separate OS/390 machines within a sysplex. The systems are named MVS1A, MVS1B, MVS1C and MVS1D, with the CICS 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 2 TCPIPSERVICE 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 1 TCPIPSERVICE definition specifying a group_name of WWW.
- The region CICSDEV1 running on machine MVS1C has 2 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 1 TCPIPSERVICE definition specifying a group_name of WWWDEV.

The names that a client can access are:

- PLEX1.IBM.COM will return the IP address of any of the machines in the plex
- WWW.PLEX1.IBM.COM will return either the address of MVS1A or MVS1B
- IIOP1.PLEX1.IBM.COM will return either the address of MVS1A or MVS1C

Figure 9. CICSPLEX using DNS connection optimization
• WWWDEV.PLEX1.IBM.COM will return 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

DNSGROUP specifies the location parameter passed on the IWMSRSRG register call to Workload Manager. The value may be up to 18 characters with trailing blanks ignored. This parameter is referred to as group_name by the OS/390 TCP/IP DNS documentation and is the 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 a 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 remains 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 then the group is deregistered from WLM.

To implement DNS connection optimization for IIOP requests, (including enterprise beans) the following CORBASERVER options must be defined:

• The HOSTNAME option of the CORBASERVER definition must have a HOSTNAME defining the corresponding 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 (CosNaming) nameserver.
The COS Naming Server must be configured to allow it to look up and resolve the generic host name.

**Authentication of IIOP requests**

Identification is the means by which an end-user client is given an identity, which in Java terms is known as a **Principal**, and in CICS terms means a USERID in the external security manager (RACF or equivalent). Authentication is a technique by which the server confirms that the client really is who he or she claims to be. This involves the use of a password that is only known by the client. Although IIOP can contain a SECIOP layer that sits below the GIOP layer, this is not implemented in CICS. Identification is provided in CICS by either of two mechanisms:

- A USERID provided by a user replaceable module (URM) whose name is defined in the TCPIPSERVICE resource definition for the port.
- A USERID provided by the Secure Sockets Layer (SSL)

When the SSL(CLIENTAUTH) option is used on the TCPIPSERVICE definition for the IIOP port, then identification and authentication are both provided by the Secure Sockets Layer (SSL). See the [CICS Resource Definition Guide](#) for information about the authentication options that can be specified in the TCPIPSERVICE.

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.

**Secure Sockets Layer (SSL)**

In client authentication the password that authenticates the client is the client’s private key, which is retained inside the client’s computer system. The authentication is derived from the fact that this private key is used for encryption during the SSL handshake. The X.509 certificate itself does not authenticate the client: it is only the fact that the certificate has successfully been used in the SSL handshake that authenticates the client, because it confirms that the client holds the private key that is associated with the certificate. The mapping of the certificate to the USERID provides the identification. A mapping from a certificate to a USERID exists when the certificate is registered to a USERID in an External Security Manager (ESM) such as RACF®. See the [CICS RACF Security Guide](#) for more information about SSL.

The following system initialization parameters and resource definitions have attributes that are relevant to the implementation of SSL security:

**System initialization parameters for SSL**

The **KEYRING** system initialization parameter is required for secure sockets layer (SSL) support to identify the RACF key ring containing certificates used in the SSL handshake. **KEYRING** specifies the name of a key ring within the ESM database that contains the keys and certificates used by CICS support for the SSL.

The maximum length of the KEYRING parameter is 47 characters.

Each certificate in a key ring is identified by its certificate label (up to 32 characters in length). The certificate label can be specified in the CERTIFICATE attribute in either the TCPIPSERVICE definition or the CORBASERVER definition.
The key ring should include a default certificate. If no default is present, a CERTIFICATE attribute must be specified in all CORBASERVERs and TCPIPSERVICEs.

The TCPIPSERVICE certificate is used in SSL to represent the server certificate for inbound SSL. The CORBASERVER certificate is used in SSL to represent the client certificate for outbound SSL connections. It is also used to obtain a distinguished name (DN) for input to DFHEJDNX. For more information, see "Deriving distinguished names" on page 234.

The SEC parameter is independent of KEYRING, and whether specified YES or NO, does not affect execution of the Java security API getCallerPrincipal(). However, the distinguished name contained in the java.security.Principal object returned by getCallerPrincipal() is influenced by the SEC system initialization parameter. See "Deriving distinguished names" on page 234.

**Resource definition for SSL**
The following resources have attributes relevant to SSL security:
- CORBASERVER
- TCPIPSERVICE

**CORBASERVER:** A CORBAServer defines the attributes of an execution environment for enterprise beans and stateless CORBA objects. CORBAServer definitions are installed in AORs.

**CERTIFICATE**
The CERTIFICATE attribute specifies the label (1–32 characters) of the certificate in the key ring that is to be used as a client certificate in the SSL handshake for outbound IIOP connections.

If this option is not specified, the default certificate for the key ring is used.

The distinguished name within the specified (or default) certificate provides inputs to the distinguished name user-replaceable program, DFHEJDNX.

If the KEYRING system initialization parameter has been specified, the selected certificate (either the specified one or the default one) must be present in the key ring, otherwise the install will fail.

If KEYRING is omitted, SSL must be NO and CERTIFICATE must be omitted on all CORBASERVER definitions.

**SSL(\{NO\|YES\|CLIENTCERT\})**
The SSL attribute specifies whether Secure Sockets Layer is used and whether authentication must be performed using a client certificate.

- NO specifies that SSL is not used.
- YES specifies that SSL is used. You must specify a value for the SSLPORT.
- CLIENTCERT specifies that SSL is used and that the client must present a certificate registered to a userid. You must specify a value for SSLPORT.

**SSLPORT(number)**
The SSLPORT attribute specifies the TCP/IP port number to be used for SSL communication to this logical EJB/Corba server. The port number should be 684 (the ‘well-known’ port for CORBA-IIOP with SSL) or within the range 1024–65535.

If SSL is NO, the value of this option is ignored.

You must not specify the same port number for PORT and SSLPORT.
PORT

The PORT attribute is used for the non-SSL communication and should be number 683 or within the range 1024–65535.

TCP/IPSERVICE:

One or two TCPIPSERVICE definitions may be required for each CORBASERVER. See the CICS Resource Definition Guide for full details of TCPIPSERVICE options. These TCPIPSERVICE definitions must specify PROTOCOL=IIOP. One of these corresponds to the PORT on the CORBASERVER and the other corresponds to SSLPORT. The one corresponding to PORT requires SSL(NO). The one corresponding to SSLPORT requires the following:

- SSL(YES) if SSL(YES) is specified on CORBASERVER
- SSL(CLIENTAUTH) and AUTHENTICATE(CERTIFICATE) if SSL(CLIENTCERT) is specified on CORBASERVER

CERTIFICATE

You can select a particular certificate within the key database by specifying the certificate label in the CERTIFICATE field of the TCPIPSERVICE definition.

When KEYRING is specified as a system initialization parameter, specifying CERTIFICATE must be the label of a certificate in the keyring. If CERTIFICATE is omitted, one of the certificates in the keyring must be specified as the default.

Note that CERTIFICATE cannot be specified if KEYRING is omitted, and CERTIFICATE cannot be specified with SSL(NO).

AUTHENTICATE(NO|CERTIFICATE)

Specifies whether the client is to be authenticated by a certificate or not. You can only specify CERTIFICATE if SSL (CLIENTAUTH) is also specified.

SSL(NO|YES|CLIENTAUTH)

Specifies whether authentication of IIOP clients uses the SSL protocol.

PORTNUMBER

Specifies the TCP/IP port upon which this service listens. It must correspond to PORT or SSLPORT in the CORBASERVER definition.

The well known ports are those 0 through 1023. The well known ports for CORBA IIOP are:

- 683 (non SSL)
- 684 (SSL)

You should avoid using other IIOP servers on the same MVS image that may also use the well known ports.

URM

Specifies the name of a user-replaceable module to be called as the identifier mechanism, the request receiver IIOP security URM (sample DFHXOPUS) must be specified in a TCPIPSERVICE definition. (For more information, see "Using the IIOP security URM" on page 79). It is not called when the TCPIPSERVICE requires a client certificate to be provided, and the client has provided a certificate mapped to a userid.

PROTOCOL(IIOP|HTTP)

Specifies the protocol of the messages on this service. This must be IIOP for a TCPIPSERVICE that will process requests for stateless CORBA objects or enterprise beans.

SSL values are specified in the TCPIPSERVICE and CORBASERVER resource definitions. These values must be consistent:
If the CORBASERVER definition specifies SSL(CLIENTCERT), the TCPIPSERVICE must specify SSLTYPE(CLIENTAUTH) and AUTHENTICATE(CERTIFICATE). This means that the client is required to send an SSL certificate which is registered to an ESM USERID. If you are using RACF, you must also specify the name of the RACF key ring holding the certificate in the CICS system initialization parameter KEYRING. See [CICS RACF Security Guide] for information about registering a certificate to a USERID in RACF.

If the CORBASERVER definition specifies SSL(YES), the TCPIPSERVICE can specify either of the following combinations of options:
– SSL(CLIENTAUTH) and AUTHENTICATE(NO). This means that the client is asked for an SSL certificate and, if it sends one, CICS will use any USERID configured for it
– SSL(YES) and AUTHENTICATE(NO). This means that the client is not asked for an SSL certificate.

The following table shows how the USERID is established from CICS defaults and SSL, before the URM is called. A blank means that this value is not applicable:

<table>
<thead>
<tr>
<th>SSL in TCPIPSERVICE</th>
<th>SSL in CORBASERVER</th>
<th>Certificate registered?</th>
<th>USERID</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO</td>
<td>NO</td>
<td></td>
<td>DFLTUSER</td>
</tr>
<tr>
<td>YES</td>
<td></td>
<td></td>
<td>DFLTUSER</td>
</tr>
<tr>
<td>CLIENTAUTH, AUTHENTICATE (NO)</td>
<td>YES</td>
<td>NO</td>
<td>DFLTUSER</td>
</tr>
<tr>
<td>CLIENTAUTH, AUTHENTICATE (CERTIFICATE)</td>
<td>YES</td>
<td>YES</td>
<td>reg.USERID</td>
</tr>
<tr>
<td>CLIENTCERT</td>
<td>YES</td>
<td></td>
<td>NO_PERMISSION error</td>
</tr>
</tbody>
</table>

Where reg.USERID is the registered certificate USERID, and DFLTUSER is the USERID specified by the CICS system initialization parameter DFLTUSER (which defaults to CICSUSER).

If AUTHENTICATE (CERTIFICATE) is specified in the TCPIPSERVICE resource definition, then a valid X.509 certificate is required from the client, and it MUST map to a valid USERID in the external security manager (RACF). If such a certificate is not received, the IIOO request is rejected and a CORBA NO_PERMISSION exception is sent in response.

If a valid (not defaulted) RACF USERID is obtained, then this is passed with the request to the request processor and the IIOO security URM is not called. Also, if you specify SSL(CLIENTAUTH) and AUTHENTICATE(NO), and the client provides a certificate that RACF can map to a USERID, then the RACF USERID associated with the certificate is used and the URM is not called.

In all other cases the URM is called, and can override the default USERID.

The IIOO security URM
This is an optional identification mechanism. It is not an authentication mechanism, but it is a way to supply a CICS USERID. If used, the URM name is specified in the TCPIPSERVICE resource definition for the IIOO port, and is called by the IIOO
request processor. The URM is primed with the value defined by the system initialization parameter DFLTUSER (which defaults to CICSUSER), and can override it. You can write your own module to supply a USERID, or use the sample, DFHXOPUS, that is provided. CICS checks the generated USERID with RACF to confirm that the request receiver transaction is allowed to initiate work on behalf of this USERID, before the IIOP request is routed to a request processor. See “Using the IIOP security URM” on page 79.

**CONNECTION authentication**

The client USERID is only transmitted from the Listener region to the AOR when 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 10. Configuring CICS for IIOP

This chapter describes what you need to do to configure CICS as a CORBA participant. You will need to do this to run all IIOP based applications, including enterprise beans, but the additional specific requirements for enterprise beans are not addressed here. See [Chapter 13. Setting up an EJB server on page 113] for these further requirements.

To configure CICS as an IIOP server or client, you need to set up the following host software environment:

- An OS/390 Version 2.8 or later system, with UNIX Systems Services and HFS
- Language Environment® configured and active
- CICS TS Release 2.1
- The IBM Developer Kit for OS/390, Java 2 Technology Edition, with the special enhancement that provides a persistent, reusable Java Virtual Machine (JVM). This 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

Configuration of CICS to support IIOP inbound and outbound requests requires setup of the CICS system, and also setup of the host OS/390 system environment. You need to perform the following steps:

- "Setting up the host system for IIOP"
- "Setting up TCP/IP for IIOP" on page 72
- "Setting up CICS for IIOP" on page 73

Setting up the host system for IIOP

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

- [Defining hfs files](#)
- [Defining nameservers](#)

Defining hfs files

You need to define the following hfs files and directories:

**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 [Authorizing CICS region userids to UNIX system services](#) on page 44 for guidance.

**System properties**

This is a text file that defines the execution environment for the JVM. The system properties file can have any name. It is specified by the JVMPROPS parameter of the JVM profile in the DFHJVM data set. See [Chapter 7. Configuring the JVM on page 47] for more information about the content of JVM
profiles and setup requirements for the JVM. Sample properties files are supplied in the SDFHENV partitioned dataset. The default properties file is dfjjvmpr.props

**shelf directory**

Every CORBASERVER must specify the name of a shelf directory on HFS. When DJARs are installed, CICS copies the deployed JAR file into a sub-directory of the shelf root directory. Also when you issue a PERFORM CORBASERVER PUBLISH command, the IOR of the CorbaServer is written to the sub-directory. You need to create the shelf directory on HFS and give the CICS region userid full access to it. See "Authorizing CICS region userids to UNIX system services" on page 44 for guidance.

**Defining nameservers**

You may need to define two nameservers:

1. If you are using Domain Name System connection optimization, the listener regions need to be configured to communicate with the same name server on the host 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 nameserver. 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 nameserver from CICS by issuing the commands PERFORM CORBASERVER PUBLISH, or PERFORM DJAR PUBLISH.

To enable this, you must set up a CORBA Object Services (COS) Naming Directory Server that supports the Java Naming and Directory Interface (JNDI) Version 1.2. You can use, for example, the WebSphere™ Application Server Advanced Edition running on an external Windows NT® machine. WebSphere Application Server Advanced Edition is shipped with CICS for this purpose.

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 COS Naming Directory Server. Specify its Web address (URL) in the `java.naming.provider.url` property in your JVM properties file.

**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. Set the system initialization parameter TCPIP to YES in the CICS startup jobstream for every CICS region where the Listener is required.

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. See
  "Resource definition for SSL" on page 66
• The DNSGROUP name if DNS connection optimization is to be used. See
  "Resource definition for DNS connection optimization" on page 64
• the name of the user replaceable module (URM) to be called to associate this
  request with a CICS USERID for security or workload management purposes.
  If omitted, no URM is called. A sample URM, DFHXOPUS, is supplied, see
  "Using the IIOP security URM" on page 73.

For example:

```
DEFINE TCPIPSERVICE(IIOPNSSL) GROUP(DFH$IIOP)
   DESCRIPTION(IIOP TCPIPSERVICE with no SSL support)
   URM(DFHXOPUS) BACKLOG(5)
   TRANSACTION(CIRR) SSL(NO)
   STATUS(CLOSED) PROTOCOL(IIOP)
```

See 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 communicate with the same DNS name
server on the host that the MVS orkload manager is configured to use. You can
define the nameserver 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

Notes:
1. Both the client and the CICS server must use the same TCP/IP nameserver
2. The nameserver 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 74

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

**MAXOPENTCBS**
- 10 maximum is recommended

**TCPIP**
- YES

**KEYRING**
- Required if you are using Secure Sockets Layer (SSL) authentication with certificates registered to RACF. See "System initialization parameters for SSL" on page 65

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 [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.

**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.

**DFHJVM**
- A partitioned dataset containing the JVM profiles that set environment variables to control the initialization of the Java Virtual Machine (JVM). See "Chapter 7. Configuring the JVM" on page 47 for more information about the content of JVM profiles and setup requirements for the JVM.

Initializing the JVM

Setup for the JVM is described in "Chapter 7. Configuring the JVM" on page 47 and you will find information about performance considerations in [CICS Performance Guide](#).

Defining CICS resources

The following CICS resources must be defined and installed. Group DFHIIOP containing default definitions is supplied. You should include this group in your DFHLIST. You can define CICS resources online using CEDA, see [CICS Resource Definition Guide](#); from a CICS application using EXEC CICS CREATE, see [CICS System Programming Reference](#); using the DFHCSDUP offline utility, see [CICS Operations and Utilities Guide](#); or by using CICSPlex SM, see [CICSPlex System Manager Concepts and Planning](#).

**Note:** Before using CEDA, you may need to use CEOT to enable your terminal to handle mixed case input— for details, see [CICS Supplied Transactions](#).

**TCPIPSERVICE**
- Provide and install TCPIPSERVICE resource definitions to configure the CICS Listener to receive IIOP requests and call the IIOP request receiver. The
TCP/IP SERVICE resource definition also specifies load-balancing and security options. See "Setting up TCP/IP for IIOP" on page 72.

REQUESTMODEL

Provide and install REQUESTMODEL resource definitions to enable the request receiver to match the incoming request to a CICS TRANSID, to define execution parameters that are used if a new request processor instance is created to handle the request. The default is CIRP, which specifies the default DFJIIRP PROGRAM. If you choose to use your own TRANSACTION definition, you must define and install it, but it must specify a PROGRAM definition with the JVMCLASS parameter must be set to com.ibm.cics.iioop.RequestProcessor. See "Obtaining a CICS TRANSID" on page 82. Note that REQUESTMODEL resource definitions for enterprise beans are built and installed for you if you use the Development Deployment Tool, see "Request model generation and the CICS development deployment tool" on page 183.

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.

PROGRAM and TRANSACTION

You must install PROGRAM and TRANSACTION definitions for the CICS supplied request receiver and request processor programs. Sample resource definitions are supplied in group DFHIIOPI.

The TRANSACTION definition for the request receiver should be CIRR. The following default definition is supplied in DFHIIOPI:

```
DEFINE TRANSACTION(CIRR) GROUP(DFHIIOPI)
  PROGRAM(DFHIIRRS) TWASIZE(0)
  PROFILE(DFHCICST) STATUS(ENABLED)
  TASKDATALOC(ANY) TASKDATAKEY(USER)
  RUNAWAY(SYSTEM) SHUTDOWN(ENABLED)
  PRIORITY(1) TRANCLASS(DFHTCLO0)
  DTIMEOUT(NO) TPURGE(NO)
  SPURGE(YES) ISOLATE(NO)
  RESSEC(NO) CMDSEC(NO)
  RESTART(NO)
  DESCRIPTION(Default CICS IOP Request Receiver transaction)
```

If you do not use the default PROGRAM definition for the request processor (DFJIIRP) you can provide one with any name, but the JVMCLASS parameter must be set to com.ibm.cics.iioop.RequestProcessor. You must specify the name of your JVM profile on the JVMPROFILE option of the PROGRAM definition for the request processor program, if you do not want to use the default DFHJVMPR. If necessary, customize the JVM profile. CICS supplies sample JVM profiles in the SDFHENV partitioned data set. (See Chapter 7. Configuring the JVM on page 47 for more information about the JVM profile.)

The following DFHJIIRP sample definition is supplied in group DFHIIOPI:

```
DEFINE PROGRAM(DFJIIRP) GROUP(DFHIIOPI)
  DESCRIPTION(CICS IOP Request Processor)
  JVM(YES)
  JVMCLASS(com.ibm.cics.iioop.RequestProcessor)
  LANGUAGE(LE370)
  RELOAD(NO)
  EXECKEY(USER)
  RESIDENT(NO)
  USAGE(NORMAL)
```
USELPACOPY(NO)
STATUS(ENABLED)
CEDF(NO)
DATALLOCATION(ANY)
DYNAMIC(NO)

**Note:** Note that the CEDF attribute can be set to YES for debugging purposes. See [Using EDF with enterprise beans](#) on page 153.

Also provide and install PROGRAM resource definitions for all stateless CORBA object server programs, with JVM(YES) and JVMPROFILE specified.

**DJAR**
Provide and install DJAR resource definitions for any enterprise beans. Note that DJAR resource definitions are built and installed for you if you use the Development Deployment Tool, see [DJAR generation and the CICS development deployment tool](#) on page 182.

**FILE**
Provide and install FILE resource definitions for the DFHEJDIR and DFHEJOS files. There are sample FILE definitions in the CICS-supplied RDO group DFHEJVS, sample coupling facility FILE definitions in the CICS-supplied RDO group, DFHEJCF, and sample VSAM RLS FILE definitions in the CICS-supplied RDO group, DFHEJVR. (DFHEJVS, DFHEJCF, and DFHEJVR are not included in the default CICS startup group list, DFHLIST.)

The following diagram 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 10. 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 11. 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 USERID

The request receiver initially assumes the CICS default USERID, as specified by the system initialization parameter DFLTUSER. This can then overridden by

1. A USERID provided by the Secure Sockets Layer (SSL) authentication mechanism
2. A USERID provided by a user replaceable module (URM) whose name is defined in the TCPIPSERVICE resource definition for the port

Using Secure Sockets Layer (SSL)

To use SSL to provide a USERID, you must:

- specify SSL(CLIENTAUTH) in the TCPIPSERVICE definition for the IIOP port
- The client must own and present an X.509 certificate in the SSL handshake
- The X.509 client certificate presented in the SSL handshake may be associated with a valid USERID in the external security manager (RACF).

If AUTHENTICATE (CERTIFICATE) is also specified in the TCPIPSERVICE resource definition, then a valid X.509 certificate is required from the client, and it MUST map to a valid USERID in the external security manager (RACF). If such a certificate is not received, the IIOP request is rejected and a CORBA NO_PERMISSION exception is sent in response.

See "Authentication of IIOP requests" on page 65 for more information.

Using the IIOP security URM

You may optionally provide a User Replaceable Module (URM), to examine elements of the incoming IIOP request and generate a USERID. You must specify the name of the URM you want to use in the TCPIPSERVICE resource definition, and also supply a PROGRAM resource definition for it. If you do not specify a URM name in the TCPIPSERVICE, no URM is called. If you specify SSL(CLIENTAUTH) and AUTHENTICATE (CERTIFICATE) in the TCPIPSERVICE, then the RACF USERID associated with the certificate is used and the URM is not called. Additionally, if you specify SSL(CLIENTAUTH) and AUTHENTICATE(NO), and the client provides a certificate that RACF can map to a USERID, then the RACF USERID associated with the certificate is used and the URM is not called.

A sample URM, DFHXOPUS, is supplied

The URM may use CICS services, such as Task Related User Exits to access DB2, and application parameters encoded within the body of the request.

If you specify a URM, but omit the PROGRAM resource definition for it, CICS will attempt to build a resource definition for it (autoinstall), but if this fails, or your URM does not supply a USERID, a default USERID is used. This is the RACF USERID.
associated with the SSL client certificate, if there is one. Otherwise, it is the
USERID specified, or allowed to default, in the CICS system initialization
DFTUSER parameter.

The following COMMAREA is passed to the URM. 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

<table>
<thead>
<tr>
<th>Offset</th>
<th>Type</th>
<th>Len</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0)</td>
<td>STRUCTURE</td>
<td>80</td>
<td>sXOPUS</td>
</tr>
<tr>
<td>(0)</td>
<td>CHARACTER</td>
<td>4</td>
<td>standard_header</td>
</tr>
<tr>
<td>(4)</td>
<td>FULLWORD</td>
<td>4</td>
<td>pIIOPData</td>
</tr>
<tr>
<td>(8)</td>
<td>FULLWORD</td>
<td>4</td>
<td>lIIOPData</td>
</tr>
<tr>
<td>(C)</td>
<td>FULLWORD</td>
<td>4</td>
<td>pRequestBody</td>
</tr>
<tr>
<td>(10)</td>
<td>FULLWORD</td>
<td>4</td>
<td>lRequestBody</td>
</tr>
<tr>
<td>(14)</td>
<td>CHARACTER</td>
<td>4</td>
<td>corbaserver</td>
</tr>
<tr>
<td>(18)</td>
<td>FULLWORD</td>
<td>4</td>
<td>pBeanName</td>
</tr>
<tr>
<td>(1C)</td>
<td>FULLWORD</td>
<td>4</td>
<td>lBeanName</td>
</tr>
<tr>
<td>(20)</td>
<td>FULLWORD</td>
<td>4</td>
<td>BeanInterfaceType</td>
</tr>
<tr>
<td>(24)</td>
<td>FULLWORD</td>
<td>4</td>
<td>pModule</td>
</tr>
<tr>
<td>(28)</td>
<td>FULLWORD</td>
<td>4</td>
<td>lModule</td>
</tr>
<tr>
<td>(2C)</td>
<td>FULLWORD</td>
<td>4</td>
<td>pInterface</td>
</tr>
<tr>
<td>(30)</td>
<td>FULLWORD</td>
<td>4</td>
<td>lInterface</td>
</tr>
<tr>
<td>(34)</td>
<td>FULLWORD</td>
<td>4</td>
<td>pOperation</td>
</tr>
<tr>
<td>(38)</td>
<td>FULLWORD</td>
<td>4</td>
<td>lOperation</td>
</tr>
<tr>
<td>(3C)</td>
<td>FULLWORD</td>
<td>4</td>
<td>userid</td>
</tr>
<tr>
<td>(40)</td>
<td>FULLWORD</td>
<td>4</td>
<td>transid</td>
</tr>
<tr>
<td>(44)</td>
<td>FULLWORD</td>
<td>4</td>
<td>flag_bytes</td>
</tr>
<tr>
<td>(48)</td>
<td>FULLWORD</td>
<td>4</td>
<td>return_code</td>
</tr>
<tr>
<td>(4C)</td>
<td>FULLWORD</td>
<td>4</td>
<td>reason_code</td>
</tr>
</tbody>
</table>

**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 USERID

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.

RETCODE is set to RCUSRID (X'01') if a USERID is being returned. The URM should return all other fields unchanged, or unpredictable results will occur.

See the CICS Customization Guide for information about installing user replaceable modules.
Using DFHXOPUS

The CICS supplied sample URM, 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.

If a USERID has not been found using these procedures, DFHXOPUS returns the USERID specified, (or allowed to default to CICSUSER), in the CICS system initialization DFLTUSER.

Obtaining a CICS TRANSID

To associate the incoming GIOP formatted request with a CICS transaction id, you need to provide and install REQUESTMODEL resource definitions for all the possible requests that CICS can process. CICS compares fields in the request against values defined in the REQUESTMODELS, to find the REQUESTMODEL that most exactly matches. The selected REQUESTMODEL provides the TRANSID name that is used if a new request processor instance is created to handle the request. If no match is found, a default TRANSID (CIRP) is used. REQUESTMODELS define requests for enterprise beans, stateless CORBA objects, or both. They specify:
- MODULE and INTERFACE patterns to match against requests for stateless CORBA objects
- Bean names for matching enterprise beans.
- OPERATION patterns to match against a bean method name or an IDL operation for a stateless CORBA object
- 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:

```plaintext
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)
   LANGUAGE(LE370) RELOAD(NO) EXECKEY(USER)
```
See "Dynamic routing" 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.

**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.

**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.
Part 5. Using enterprise beans

This Part tells you what you need to know to develop and use enterprise beans in CICS. It covers the following topics:

- “Chapter 12. What are enterprise beans?” on page 87
- “Chapter 13. Setting up an EJB server” on page 113
- “Chapter 15. Running the sample EJB applications” on page 127
- “Chapter 16. Writing enterprise beans” on page 145
- “Chapter 19. The CICS Connector for CICS TS” on page 205
- “Chapter 21. Managing security for enterprise beans” on page 227
- “Chapter 22. CICSPlex SM with Enterprise JavaBeans” on page 237
Chapter 12. What are enterprise beans?

This chapter describes CICS support for the Enterprise JavaBeans (EJB) architecture.

About Enterprise JavaBeans

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” on page 88
- “JavaBeans™ and Enterprise JavaBeans” on page 88
- “The EJB server—overview” on page 90
- “The EJB container—overview” on page 90
- “Enterprise beans—the home and remote interfaces” on page 91
- “Enterprise beans—the deployment descriptor” on page 92
- “Types of enterprise bean” on page 93
- “Enterprise beans—managing transactions” on page 96
- “Enterprise beans—accessing data” on page 97
- “Enterprise beans—security overview” on page 99
- “Enterprise beans—user tasks” on page 99
- “Deploying enterprise beans—overview” on page 101
- “Configuring CICS as an EJB server—overview” on page 103
- “Enterprise beans—what can a client do with a bean?” on page 108
- “Enterprise beans—what can a bean do?” on page 109
- “Benefits of EJB technology” on page 110
- “Requirements for EJB support” on page 111

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 Microsystems'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.

CICS Transaction Server for z/OS Version 2 Release 1 offers partial support for Version 1.1 of the Enterprise JavaBeans specification. Future releases of CICS will support the specification more fully, particularly in regard to security.

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 classes⁠¹ to access CICS resources.
- Use JCICS or the new CICS Connector for CICS TS¹ to link to existing CICS programs written in procedural languages such as COBOL. (For information about the CICS Connector for CICS TS, see page 205.)

Figure 11 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 11 are explained in the sections that follow.

---

Figure 11. 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.

Note: The picture shows an external security manager. EJB resource security—the checking of access to enterprise beans, based on EJB security “roles”—is not supported in CICS TS for z/OS Version 2.1.

JavaBeans™ and Enterprise JavaBeans

JavaBeans and Enterprise JavaBeans are component architectures for the Java language.

---

1. Enterprise beans that use the JCICS classes are not portable to a non-CICS environment.
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 IBM's VisualAge for Java or Symantec's Visual Café.

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 VisualAge for Java, 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 VisualAge for Java, 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.

**Note:** Release 3.5 of VisualAge for Java supports Version 1.0 of the Enterprise JavaBeans specification, but does not support Version 1.1.

If you develop enterprise beans with VisualAge for Java 3.5, you will need to use the CICS JAR development tool for EJB technology and the CICS code generation utility for EJB technology before using one of the CICS deployment tools for EJB Technology. For more information about this, see "Using CICS deployment tools for EJB technology" on page 173.

---

**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 103.

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 55.)
- CICS Transaction Server for z/OS, Version 2 Release 1 supports RMI over IIOP (RMI-IIOP), but not JRMP. (JRMP is a proprietary interface which does not support a transaction service context.)
- 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.

---

² The javax.transaction.UserTransaction interface is used by session beans that manage their own transactions, as described later in this chapter.
The EJB container supports a number of implicit services, including lifecycle, state management, security, transaction management, and persistence:

**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.

**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.

**Persistence**
Individual enterprise beans do not need to retrieve or store persistent data from a database explicitly. The EJB container can automatically manage persistent data 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 92.)

Each deployment results in the creation of a distinct home object (see "Enterprise beans—the home and remote interfaces").

**Enterprise beans—the home and remote 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 remote
As the client invokes operations using these intermediate objects, the container intercepts each method call and inserts the management services.

The home and remote 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.

**The remote interface**

The remote interface allows a client to access the business methods of the enterprise bean. When a client creates or finds an instance of an enterprise bean, the container returns an EJB remote interface object (one per instance). The remote interface 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.

---

**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 101](#).

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.

*Figure 12 on page 93* shows enterprise bean objects in a CICS EJB server.
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 96.

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 96.

- Is recoverable—it survives a server crash.

Session beans and entity beans compared

Table 5 is a summary of the differences between entity and session beans.

**Table 5. Comparison of session and entity beans**

<table>
<thead>
<tr>
<th>Session bean</th>
<th>Entity bean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Represents a single conversation with a client.</td>
<td>Typically, encapsulates persistent business data—for example, a row in a database.</td>
</tr>
<tr>
<td>Typically, encapsulates an action or actions to be taken on business data.</td>
<td></td>
</tr>
<tr>
<td>Is relatively short-lived.</td>
<td>Is relatively long-lived.</td>
</tr>
<tr>
<td>Is created and used by a single client.</td>
<td>May be shared by multiple clients.</td>
</tr>
<tr>
<td>Has no primary key.</td>
<td>Has a primary key, which enables an instance to be found and shared by more than one client.</td>
</tr>
<tr>
<td>Typically, persists only for the life of the conversation with the client. (However, may choose to save information.)</td>
<td>Persists beyond the life of a client instance. Persistence can be container-managed or bean-managed.</td>
</tr>
<tr>
<td>Is not recoverable—if the EJB server fails, it may be destroyed.</td>
<td>Is recoverable—it survives failures of the EJB server.</td>
</tr>
<tr>
<td>May be stateful (that is, have a client-specific state) or stateless (have no non-transient state).</td>
<td>Is typically stateful.</td>
</tr>
<tr>
<td>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. Is not re-entrant.</td>
<td>May or may not be transactional. Must use the container-managed transaction model. If transactional, its state is automatically rolled back on transaction rollback. May be re-entrant.</td>
</tr>
</tbody>
</table>
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 \textbf{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 \textbf{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 \textbf{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 \textbf{bean–managed OTS transactions}. A session bean that uses bean–managed transactions uses methods of the \texttt{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:

\begin{description}
\item[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 \texttt{javax.transaction.TransactionRequiredException} exception and the request fails.
\item[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 \texttt{java.rmi.RemoteException} exception and the request fails.
\item[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.
\end{description}

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.

Enterprise beans—accessing data
CICS enterprise beans can use a variety of methods to access data. The methods available depend on the type of data to be accessed:

Relational data
To access relational data, an enterprise bean can use any of the following methods:

- Use a JCICS LINK command, or the CICS Connector for CICS TS, to link to a program that uses Structured Query Language (SQL) commands to access the data.

- 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. A suitable JDBC driver is available for DB2.

- Use 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 are described in "Using Data Access beans" on page 98.

- Use 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.

Visual Age for Java provides the Enterprise Access Builder for Data to facilitate building such JavaBeans. Use Data Access Builder if you are developing your application outside the Visual Composition Editor, or if you need specialized access to relational data.

- Use 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 EE for OS/390 to access DB2 on OS/390.

**DL/I data**

To access DLI data, an enterprise bean can use a JCICS LINK command, or the CICS Connector for CICS TS, to link to a program that issues EXEC DLI commands to access the data.

**VSAM data**

To access VSAM data, an enterprise bean can use either of the following methods:

- Use a JCICS LINK command, or the CICS Connector for CICS TS, to link to a program that issues CICS File Control commands to access the data.
- Use the JCICS File Control classes to access VSAM directly.

**Notes:**

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 EE 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, enterprise beans can use JDBC calls. However, the recommended method is to use Data Access beans, which package the native JDBC calls with extra function and make them more convenient to use. 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.

**Important**
There are special considerations for using Data Access beans in CICS. For the latest information, ensure that you read the CICS-supplied “How to” file, doc/HOWTO/Data-Access-Beans-HOWTO. This supplements and takes precedence over the documentation of Data Access beans supplied with VisualAge for Java Enterprise Edition.

## Enterprise beans—security overview

EJB security is concerned with authentication and access control.

### Authentication

Authentication of EJB clients uses the TCP/IP secure sockets layer (SSL) protocol. How to configure CICS to use SSL is described in the [CICS Internet Guide](#).  

### Access control

#### EJB security roles

**Important**

EJB security "roles" are not supported in CICS Transaction Server for z/OS Version 2 Release 1.

Access to enterprise beans is based on the concept of security *roles*. An EJB "role" represents a type of user that should have a particular level of access to an application. It maps to a group of users defined to your external security manager (ESM).

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.

#### 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).

## 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 remote 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’ remote 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. (EJB security roles are not supported in CICS TS for z/OS Version 2.1.)
The deployment process is semi-automated. To perform his role, the deployer uses a deployment tool. Deployment tools are provided by VisualAge for Java and 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. They are supplied to the deployer in an ejb-jar file. This file 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:

1. A deployment tool (such as the CICS JAR development tool, described in “The CICS JAR development tool for EJB technology” on page 175) is used to transform the ejb-jar file into a form 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 OS/390 or z/OS hierarchical file system (HFS).

2. CICS resource definitions are created on the CSD. 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.)
   - Deployed JAR files (DJARs), each of which includes the HFS filename of a deployed JAR file.
   - TCP/IP services (for IIOP).
   - Request models (to enable client IIOP requests to be processed correctly).

4. This simplified description of the deployment process assumes that you’re using RDO rather than BAS.
3. Security definitions are added to the external security manager. These specify which “roles” can execute particular beans and methods, and which CICS USERIDs are associated with each role.

Note: EJB security roles are not supported in this release.

4. The resource definitions on the CSD 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
   - Read the deployed JAR from the shelf, parse its XML deployment descriptor, and store the information it contains

5. Using SPI commands, 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.

Figure 13 shows the deployment process.

![Figure 13. 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 on the CSD 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.](image-url)

Note: The picture shows an external security manager. EJB resource security—the checking of access to enterprise beans and the CICS resources they use, based on EJB security “roles”—is not supported in CICS TS for z/OS Version 2.1.
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 9. The IIOP request flow" on page 59.

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**

   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.

We must be clear about what we mean by “new” and “existing” OTS transactions. For the purposes of this chapter:

a. 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.

b. 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.

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.

---

**Figure 14 on page 105** 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.

Setting up a logical EJB server

In simplified form, the steps involved in setting up a CICS logical EJB server to provide access to a specific enterprise bean are:

1. Create a set of cloned CICS TS for z/OS Version 2.1 listener regions.
2. Create a set of cloned CICS TS for z/OS Version 2.1 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).
3. Take the ejb-jar file and perform code generation on it to produce a deployed JAR file on HFS.
4. 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-supplied deployment tool to create some of these definitions.
   - A TCPIPSERVICE.
   - Some REQUESTMODEL definitions. *These are only required if the default TRANSID, CIRP, cannot be used.*

   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 the deployed JAR file on HFS. The value of the
CORBASERVER attribute must be identical with the name of the CorbaServer on the CORBASERVER definition.

- A CORBASERVER definition.
  The 'server ORB' attributes of the CORBASERVER definition (HOST, SSL, and PORT or SSLPORT) must match the corresponding attributes of the TCPIPSERVICE definition (IPADDRESS or DNSGROUP, SSL, and PORTNUMBER respectively). To clarify:
  a. 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.
  b. If the CORBASERVER definition does not support the secure sockets layer—SSL(NO)—the CorbaServer has only one, non-SSL, TCP/IP port. Its PORT number must match the value of PORTNUMBER on the TCPIPSERVICE definition.
  c. If the CORBASERVER definition supports the secure sockets layer—SSL(YES) or SSL(CLIENTCERT)—the CorbaServer has two TCP/IP ports—one which supports the secure sockets layer and one which does not. Either its SSLPORT number or its PORT number must match the value of PORTNUMBER on the TCPIPSERVICE definition. Alternatively, you can install a TCP/IP service on both ports (using two TCPIPSERVICE definitions).

- A DJAR definition.
  The HFSFILE attribute of the DJAR definition points to the deployed JAR file on HFS. The CORBASERVER attribute matches the name of the CorbaServer on the CORBASERVER definition.

- 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, DFHEJC, There are sample VSAM RLS FILE definitions for DFHEJDIR and DFHEJOS in the CICS-supplied RDO group, DFHEJVR. (DFHEJVS, DFHEJC, 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, and that all the enterprise beans you want to deploy are in a single ejb-jar file. To create another CorbaServer, you’ll need another CORBASERVER definition and a second TCPIPSERVICE definition. To deploy beans in other ejb-jar files, you’ll need further DJAR and REQUESTMODEL definitions.

5. 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.

6. On each of the listener regions, install:
   - The TCPIPSERVICE definition
   - The REQUESTMODEL definitions
   - The file definition for DFHEJDIR

   On each of the AORs, install:
   - The REQUESTMODEL definitions
   - The CORBASERVER definition
   - The DJAR definition
   - The file definitions for DFHEJDIR and DFHEJOS

7. Issue a PERFORM CORBASERVER (CorbaServer_name) PUBLISH command on at least one of the AORs. This binds the homes of the enterprise beans into the JNDI namespace. The command can be issued using EXEC CICS, the CEMT master terminal transaction, or via a CICSPlex SM EUI or WUI View.

Figure 15 on page 108 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.
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:

```java
// 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 remote interface
The client can use the bean’s remote 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 CICS 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 CICS Connector for CICS TS to build beans that make use of the power of existing (non-Java) CICS programs.

The CICS Connector for CICS TS is described in "Chapter 19. The CICS Connector for CICS TS" on page 205.

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 remote interfaces of other bean objects
• Invoke the methods of another bean object.

A bean can act as the client of another bean object, as the server of another bean object, or as both.
Manage transactions
Optionally, a session bean can manage its own OTS transactions, rather than using container-managed transactions.

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 CICS 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 19. The CICS Connector for CICS TS on page 205.

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 2 Release 1 itself.

Software

The software requirements for enterprise beans are:

• IBM Developer Kit for OS/390, Java 2 Technology Edition
• A Corba Object Services (COS) Naming Directory Server that supports the Java Naming and Directory Interface (JNDI) Version 1.2. WebSphere Application Server Advanced Edition is shipped with CICS for this purpose.

Note: The JNDI API provides directory and naming functionality for Java applications. It enables a client to locate an enterprise bean. The JNDI is mapped to an external Naming Directory Server. In CICS TS for z/OS Version 2.1, the latter must be a COS Naming Directory Server, which is most conveniently obtained by use of WebSphere Application Server Advanced Edition running on an external Windows NT machine.

• DB2 with Java Data Base Connectivity (JDBC) Version 1.2 extensions.
Chapter 13. 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 119** tells you how to check that your single-region EJB server is correctly configured.
- **“Setting up a multi-region EJB server” on page 120** 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.

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 120.**

**Important**

- For clarity’s sake, we’re assuming that:
  1. You start from a basic, non-customized, CICS TS for z/OS Version 2.1 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, DFHJVMPR, and the default JVM properties file, dfjjvmpr.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 **“Optional steps” on page 117.**
- 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 18. Deploying enterprise beans” on page 173.**
- The rest of this section is split into two parts:
  - **“Before running the IVP”** takes you as far as being able to run the EJB Installation Verification Program, which tests that you have configured CICS correctly as an EJB server.
  - **“After running the IVP” on page 115** finishes the setup of the complete EJB server, including the Naming Directory Server.

Before running the 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.

On CICS, you must:

1. Install the IBM Developer Kit for OS/390, Java 2 Technology Edition, with the special enhancement that provides a persistent, reusable Java Virtual Machine (JVM). This is available from [www.s390.ibm.com/java](http://www.s390.ibm.com/java).

2. 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 CICS region userid full access to it—read, write,
and execute. How to do this is described in Authorizing CICS region userids to
UNIX system services” on page 44.

Note: When you install a DJAR definition, CICS copies the deployed JAR file
into a sub-directory of the shelf root directory specified on the
CORBASERVER definition. (In a multi-region EJB server, each AOR
creates its own sub-directory beneath the shelf root directory.)

3. Configure CICS as an IIOP server. (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 Chapter 10. Configuring
CICS for IIOP” on page 71. Bear in mind when reading Chapter 10. Configuring
CICS for IIOP” on page 71 that:

- The DFHJVM DD statement in the CICS startup job stream must point to the
data set which contains the JVM profiles. For example, the following
DFHJVM DD statement points to the default XDFHENV data set:

  //DFHJVM DD DSN=CICSTS21.CICS.XDFHENV,DISP=SHR

  If you have created your own JVM profile on a different data set, you must
change the DFHJVM DD statement accordingly.

- PROGRAM definitions are not required for enterprise beans as such. The
  only PROGRAM definition required is that for the request processor program.
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.

  If you are using a JVM profile other than the default DFHJVMPR, you must
specify the name of your profile on the JVMPROFILE option of the
PROGRAM definition for the request processor program.

- REQUESTMODEL and DJAR definitions are typically created by the
deployment tool that you use to deploy enterprise beans into CICS.
Deployment is a separate process that occurs after you have set up your EJB
server. It is described in Chapter 18. Deploying enterprise beans” on
page 173.

  The resource definitions that you need to support CICS as an EJB server are
discussed in step 4 and in Actions required on CICS” on page 116.

4. Create FILE definitions for the following files required by CICS:

The “EJB Directory”, DFHEJDIR
is a file containing a request streams directory. You must define
DFHEJDIR as recoverable.

The “EJB Object Store”, DFHEJOS
is a file of stateful session beans that have been passivated. You must
define it as non-recoverable.
**Important**

In a single-region CICS EJB server, it is acceptable to define DFHEJDIR and DFHEJOS as local files. However, in a multiple-region CICS EJB 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.

If you want to use your single-region server as the basis of a multi-region server, you should ensure that DFHEJDIR and 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.

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.)

For reference information about FILE definitions, see the [CICS Resource Definition Guide](#).

5. Define the underlying VSAM data sets for DFHEJDIR and DFHEJOS. (You may already have done this in step 3.) CICS supplies sample JCL to help you do this, in the DFHDEFDS member of the SDFHINST library. How to use the JCL to define the data sets is described in the [CICS System Definition Guide](#).

6. Install the file definitions for DFHEJDIR and DFHEJOS.

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 14. Running the EJB IVP on page 123](#). Alternatively, you can continue with the next section before running the IVP.

In either case, to finish the setup of the complete EJB server, including the Naming Directory Server, you must follow the steps in [After running the IVP](#).

### After running the IVP

To finish the setup of the complete EJB server you need to:

1. Set up a Naming Directory Server
2. Set up CICS to use JNDI
3. Create some additional CICS resource definitions
4. Optionally, customize the JVM profile and JVM properties file.

**Actions required outside CICS**

You must set up a Corba Object Services (COS) Naming Directory Server that supports the Java Naming and Directory Interface (JNDI) Version 1.2. This is most conveniently obtained by use of WebSphere Application Server Advanced Edition.
running on an external Windows NT or Windows 2000 machine. WebSphere Application Server Advanced Edition is shipped with CICS for this purpose. Three CDs are supplied:

2. IBM WebSphere Application Server Advanced Edition for Windows NT—DB2 Universal Database (SBCS) and IBM SecureWay Directory
3. IBM WebSphere Application Server Advanced Edition for Windows NT—DB2 Universal Database (DBC) and IBM SecureWay Directory

To set up the WebSphere Application Server on Windows NT or Windows 2000, insert the first CD (the Application Server and IBM HTTP server CD) and follow the installation instructions. If the setup program does not start automatically, change to the NT directory on the CD and run the setup.exe program.

The CD contains the following readme files:
- Readme.html
- Readme.HTTPSERVER

There is also a Getting Started manual, supplied in both PDF and HTML formats.

Note: CICS also supports the COS Naming Server supplied with IBM WebSphere Application Server Advanced Edition for AIX, Version 3.5. However, this is not supplied with CICS.

Actions required on CICS

1. 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 COS Naming Directory Server. Specify its URL and port number on the java.naming.provider.url property in your JVM properties file. For example:

   java.naming.provider.url=iiop://demojndi.yourcompany.com:900

   The location of the JVM properties file is specified on the JVMPROPS statement in your JVM profile. The default JVM profile is DFHJVMPR, in the XDFHENV data set.

2. Create the following CICS resource definitions:
   - A TCPIPSERVICE.
     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 120, 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.
     For reference information about TCPIPSERVICE definitions, see the CICS Resource Definition Guide.
   - A CORBASERVER definition.
     The 'server ORB' attributes of the CORBASERVER definition (HOST, SSL, and PORT or SSLPORT) must match the corresponding attributes of the TCPIPSERVICE definition (IPADDRESS or DNSGROUP, SSL, and PORTNUMBER respectively). To clarify:
     a. 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.)

b. If the CORBASERVER definition does not support the secure sockets layer—SSL(NO)—the CorbaServer has only one, non-SSL, TCP/IP port. Its PORT number must match the value of PORTNUMBER on the TCPIPSERVICE definition.

Note: The TCP/IP port number that you use must be authorized by your network administrator.

c. If the CORBASERVER definition supports the secure sockets layer—SSL(YES) or SSL(CLIENTCERT)—the CorbaServer has two TCP/IP ports—one which supports the secure sockets layer and one which does not. Either its SSLPORT number or its PORT number must match the value of PORTNUMBER on the TCPIPSERVICE definition. Alternatively, you can install a TCP/IP service on both ports (using two TCPIPSERVICE definitions).

Note: The TCP/IP port number that you use must be authorized by your network administrator.

On the SHELF option, specify the path to the shelf root directory that you created in step 2 of Before running the IVP on page 113.

For reference information about CORBASERVER definitions, see the CICS Resource Definition Guide.

3. Install the TCPIPSERVICE and CORBASERVER definitions.

Optional steps:

1. If necessary, customize the JVM profile to be used by enterprise beans. The JVM profile sets environment variables that control the initialization of the JVM.

   CICS supplies, in the SDFHENV PDS, two sample JVM profiles, two JVM properties files, and a security policy—see Table 6.

   Table 6. CICS-supplied sample JVM profiles, properties files, and security policy

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Associated properties file</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFHJVMPR</td>
<td>JVM profile</td>
<td>dfjjvmp. props</td>
<td>Profile DFHJVMPR is the default. It specifies Xresettable=YES, which causes CICS to make a JVM available for reuse for another task, after the task using the JVM terminates. This is the recommended setting for enterprise beans.</td>
</tr>
<tr>
<td>DFHJVMPS</td>
<td>JVM profile</td>
<td>dfjjvmps. props</td>
<td>DFHJVMPS specifies Xresettable=NO, which causes CICS to make each JVM available for use by a single task only. This setting is not recommended for enterprise beans.</td>
</tr>
<tr>
<td>dfjejbpl.</td>
<td>JVM security</td>
<td></td>
<td>Specifies the JVM's security settings. The location of the security policy file is specified by the java.security.policy property in the JVM properties file.</td>
</tr>
<tr>
<td>policy</td>
<td>policy</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
As supplied, 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. The DFHIJVMJ job is described in the [CICS Transaction Server for z/OS Installation Guide](#). It:

a. Substitutes your own values for the symbol names.
b. Copies the configured profiles to data set XDFHENV.
c. Copies each configured property file to the HFS directory specified by the JVMPROPS statement in its associated JVM profile.
d. Copies the security policy to the /usr/lpp/cicsts/cicsts21/lib/security directory on HFS (where cicsts21 is your chosen value for the CICS_DIRECTORY symbol).

Having run DFHIJVMJ, you may want to customize the JVM profile. To do this, you can do either of the following:

a. Edit the default JVM profile, DFHJVMPR in XDFHENV, with a TSO editor. This is the recommended method.
b. Create your own JVM profile, based on DFHJVMPR. (DFHJVMPS is not recommended for use with enterprise beans.) If you do this:
   1) You must specify the name of your profile on the JVMPROFILE option of the PROGRAM definition for the request processor program. (The supplied PROGRAM definition for the default request processor program, DFJII RP, does not specify a JVM profile, so the default DFHJVMPR is used.)
   2) It's recommended that you place your profile in the XDFHENV data set.

**Note:** You cannot use the DFHJVMAT user-replaceable program to change the initialization values set by a profile, such as DFHJVMPR, that specifies Xresettable=YES.

The JVM initialization parameters that you can specify, and their possible values, are documented in the [CICS System Definition Guide](#). The parameter, if any, that you are most likely to want to change is:

**Xmx=size**

Xmx specifies the maximum total size of the middleware and transient heaps, as a number of bytes, kilobytes, or megabytes, in multiples of 1024 bytes. The letter K is used to indicate kilobytes, and the letter M to indicate megabytes.

The value specified in the default JVM profile, DFHJVMPR, is 32M, which should be adequate for most purposes. If you have large Java applications, you may want to increase this value.

2. If necessary, customize the JVM properties file. This is a text file that sets system properties for the JVM. Each property is specified on a separate line and the property value is delimited by the end of the line. Property values are passed to the JVM for the construction of system properties as if specified by a -D option on a Java command. For example, the following line sets the java.security.policy property:

```java
java.security.policy=file:/usr/lpp/cicsts/lib/security/cicsejb.policy
```

From the property value, CICS constructs the

```java
-Djava.security.policy=file:/usr/lpp/cicsts/lib/security/cicsejb.policy
```

Java launch option.
The location of the JVM properties file is specified on the JVMPROPS statement in the JVM profile. The properties that you can specify, and their possible values, are documented in the New IBM Technology featuring Persistent, Reuseable Java Virtual Machines manual, SC34–5881–00, which is available from www.s390.ibm.com/java.

Testing your EJB server

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

Running the EJB IVP

The easiest way to test your CICS EJB configuration is to run the EJB Installation Verification Program (IVP) supplied with CICS. (The IVP does not test your COS Naming Directory Server.)

The IVP uses the same CICS enterprise bean as the “Hello World” sample application described below, but uses a simpler client program that does not require the use of a Web Server or a Naming Server. The sample consists of:
- A line-mode client program that runs in Unix System Services (USS) on MVS
- An enterprise bean running on the CICS EJB server

For details of the EJB IVP, see [Chapter 14. Running the EJB IVP](#) on page 123.

Using the “Hello World” sample

The easiest way to test your full EJB server, consisting of CICS and your COS Naming Directory Server, is to run the “Hello World” sample application supplied with CICS. “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. Running it successfully confirms that external programs are able to invoke enterprise beans on your CICS EJB server.

For details of the “Hello World” application, and instructions on how to install it, see [The “Hello World” sample application](#) on page 127.

Using the EJB CICS sample

After you’ve run the Hello World” sample successfully, you might want to try something more ambitious. The EJB CICS 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 enterprise bean uses the CICS Connector for CICS TS to link to CICS server programs, which access DB2 data tables.

For details of the EJB CICS application, and instructions on how to install it, see [The EJB CICS sample application](#) on page 133.

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 18. Deploying enterprise beans](#) on page 173.
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 113.

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

   **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. Optionally, you can discard the TCPIPSERVICE resource definition from the AORs, where it's not required.

   **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 16 on page 122 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.

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 62.

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 22. CICSPlex SM with Enterprise JavaBeans" on page 237.
   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.

Note: 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. (REQUESTMODEL definitions are typically created by the deployment tool that you use to deploy enterprise beans into CICS.)
Figure 16. 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.
Chapter 14. 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 the same CICS enterprise bean as the “Hello World” sample application described in the “Hello World” sample application on page 127, but uses a simpler client program that does not require the use of a Web Server or a Naming Server. The sample consists of:

- A line-mode client program that runs in Unix System Services on MVS
- An enterprise bean running on the CICS EJB server

The rest of the chapter contains the following topics:

- Prerequisites for the EJB IVP
- Installing the EJB IVP
- Running the EJB IVP on page 125

Prerequisites for the EJB IVP

To run the EJB IVP, you need:

- A Unix System Services userid and file editor.
- A CICS EJB server. How to set one up is described in Setting up a single-region EJB server on page 113.

Notes:
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 IVP on page 113. You may or may not have completed the steps in After running the IVP on page 115.
3. Sample TCPIPSERVICE, CORBASERVER, and DJAR resource definitions are supplied with the IVP.
4. 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. CICS
2. Unix System Services

CICS setup

On CICS, you must:

1. Make a note of the current setting of the JVMPROPS statement in the JVM profile, then modify it to use the properties file supplied in the sample, as follows:

   JVMPROPS=/usr/lpp/cicsts/cicsts21/samples.ejb/helloworld/IVP.properties

   where cicsts21 is the value of the CICS_DIRECTORY variable used by the DFHIJVMJ job during CICS installation.

2. The CICS-supplied sample group, DFH$EJB, contains TCPIPSERVICE, CORBASERVER, and DJAR definitions for the EJB “HelloWorld” 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.
Note: Before using CEDA, you may need to use CEOT to enable your terminal to handle mixed case input—for details, see the CICS Supplied Transactions manual.

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:
   • For the TCPIPSERVICE resource definition, modify the PORTNUMBER as necessary to a suitable TCP/IP port on your installation.
   • For the CORBASERVER resource definition:
     1) Modify the SHELF to match your shelf root directory on HFS.
     2) Set the HOST to your TCP/IP hostname.
     3) Modify the PORT to match the port number you set in the TCPIPSERVICE definition.
   • For the DJAR resource definition, on the HFSFILE option specify the fully-qualified file name of the HelloWorldEJB.jar deployed JAR file. The HelloWorldEJB.jar file is in the HFS EJB samples directory. The samples directory is:
     /usr/lpp/cicsts/cicsts21/samples/ejb/helloworld

     where cicsts21 is the value of the CICS_DIRECTORY variable used by the DFHIJVMJ job during CICS installation.

c. Install group mygroup to make these definitions known to CICS.

3. Publish the sample enterprise bean:
   CEMT PERFORM DJAR(HELLO) PUBLISH

4. Set the status of the TCPIPSERVICE to OPEN:
   CEMT SET TCPIPSERVICE(EJBTCP1) OPEN

You should see the following messages on the CICS Console:

DFHEJ0701 CorbaServer EJB1 has been created.
DFHEJ1520 CorbaServer EJB1 is now accessible.
DFHEJ0901 DJar HELLO within CorbaServer EJB1 has been created.
DFHEJ1540 DJar HELLO and the Beans it contains are now accessible.
DFHI11028 Name Server not defined for CORBASERVER xxxx being initialized for PROGRAM DFJIIRP
DFHEJ5010 Publishing bean HelloWorld in the Shelf directory shelf/...../EJB1/homeiors as file HelloWorldHome.ior.
DFHS00107 TCPIPSERVICE EJBTCP1 has been opened on port xxxx

where shelf is the shelf root directory on HFS.

Unix System Services setup

On 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/cicsts21/samples/ejb/helloworld

   where cicsts21 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:
   a. Modify the JAVA_HOME variable to your IBM SDK 1.3 installation directory as indicated by the comments in the script. The line to be changed is:
      JAVA_HOME=/usr/lpp/<Java SDK 1.3 installation directory>/J1.3
b. Modify the CICS_HOME variable to your CICS installation directory as indicated by the comments in the script. The line to be changed is:
CICS_HOME=/usr/lpp/cicsts/<CICS installation directory>

c. Modify the CICS_SHELF variable as indicated by the comments in the script. The value must match the shelf directory used by the CORBASERVER definition. The line to be changed is:
CICS_SHELF=<CICS shelf root directory>/<region name>/EJB1

Running the EJB IVP

Run the IVP client program from your Unix System Services working directory by typing runEJBIVP.

On the your Unix System Services terminal, you should see the following messages:

CICS EJB IVP: Querying the Java SDK level
java version "1.3.0"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.3.0)
Classic VM (build 1.3.0, J2RE 1.3.0 IBM OS/390 Persistent Reusable VM build
hm30s-yyyyyymmdd (JIT enabled: jitc))
CICS EJB IVP: Starting the EJB client program
HelloWorld client program started
Located home interface for HelloWorld bean
You said: Hello from CICS EJB IVP client
HelloWorld client program ended
CICS EJB IVP: Completed successfully

Note: In the above messages, yyyyymmdd is the date on which the SDK was built.

On your CICS console, you should see the following message:

CICS EJB hello world sample called with string: Hello from CICS EJB client.

In your JVM stdout file, you should see the following message:

CICS EJB hello world sample: method hello() called with string:
Hello from CICS EJB IVP client

Note: Your JVM stdout file is in the directory specified by the WORK_DIR parameter in the JVM profile and is called
xxxxxxxxx.APPLID.timestampdata.txt, where:
• xxxxxxxxxx is the name specified on the STDOUT parameter in the JVM profile. (The default JVM profile, DFHJVMPR, specifies dfhjvmout).
• APPLID is the CICS APPLID.
• timestampdata is the date and time.

When you have finished running the IVP, you should:
1. Restore the JVMPROPS statement in the JVM profile so that the IVP-specific properties file is no longer used.
2. Discard the resource definitions that you created in mygroup.
Chapter 15. Running the sample EJB applications

Important
The sample EJB applications require a CICS EJB server. You must configure CICS, as described in "Chapter 13. Setting up an EJB server" on page 113, 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. Neither a Web server nor a COS Naming Directory Server is required. See "Chapter 14. Running the EJB IVP" on page 123.

The “Hello World” sample
A simple application that you can use to test your EJB environment, including CICS, your Web server, and your COS Naming Directory Server. See "The "Hello World" sample application".

The EJB CICS 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 CICS sample application" on page 133.

The “Hello World” sample application
“Hello World” is a simple application that you can use to test your EJB environment, including CICS, your Web server, and your COS Naming Directory Server.

What the “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 Java Server Pages (JSPs), running on a Web 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 and sends the phrase to the enterprise bean.
3. The enterprise bean appends the user’s phrase to the string “You said ” and returns the result to the servlet.
4. The servlet uses a Java Server Page to display the result on the user’s browser.

Figure 17 on page 128 shows the components of the sample application.
Prerequisites for the “Hello World” sample

To run the “Hello World” sample, you need:

- A CICS EJB server. How to set one up is described in Chapter 13, Setting up an EJB server on page 113.
- A fully-functional Web server that supports Java Server Pages Version 1.0 or later.
- A Corba Object Services (COS) Naming Directory Server that supports the Java Naming and Directory Interface (JNDI) Version 1.2 or later. How to set one up is described in Actions required outside CICS on page 115.

Supplied components of the “Hello World” sample

Table 7 lists the files supplied with the “Hello World” sample.

<table>
<thead>
<tr>
<th>Filename</th>
<th>Type</th>
<th>Default location</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFH$EJB</td>
<td>Resource definitions</td>
<td>SDFHSAMP</td>
<td>Contains the CICS resource definitions required by the sample application.</td>
</tr>
<tr>
<td>HelloWorldCLI.jar</td>
<td>JAR file</td>
<td>HFS samples directory: see Note.</td>
<td>Client EJB stubs required by the servlet.</td>
</tr>
</tbody>
</table>

Figure 17. Overview of the “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 under WebSphere on a Windows NT server.
Table 7. Supplied components of the “Hello World” sample (continued)

<table>
<thead>
<tr>
<th>Filename</th>
<th>Type</th>
<th>Default location</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>HelloWorldEJB.jar</td>
<td>Deployed JAR file</td>
<td>HFS samples directory: see Note.</td>
<td>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.</td>
</tr>
<tr>
<td>HelloWorldWeb.jar</td>
<td>JAR file</td>
<td>HFS samples directory: see Note.</td>
<td>The Web components of the sample application—Java servlet classes and source files; HTML and JSPs. Files must be unpacked and stored on the Web server.</td>
</tr>
</tbody>
</table>

Note: The default HFS samples directory is /usr/lpp/cicsts/cicsts21/samples.ejb/helloworld

where cicsts21 is the value of the CICS_DIRECTORY variable used by the DFHIJVMJ job during CICS installation.

Installing the “Hello World” sample

Installing the “Hello World” sample requires actions on:
1. CICS
2. The Web server

CICS setup

1. The CICS-supplied sample group, DFH$EJB, contains TCPIPSERVICE, CORBASERVER, and DJAR definitions for the HelloWorld 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.

Note: Before using CEDA, you may need to use CEOT to enable your terminal to handle mixed case input—for details, see the CICS Supplied Transactions manual.

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:
   - For the TCPIPSERVICE resource definition, modify the PORTNUMBER as necessary to a suitable TCP/IP port on your installation.
   - For the CORBASERVER resource definition:
     1) Modify the SHELF to match your shelf root directory on HFS.
     2) Set the HOST to your TCP/IP hostname.
     3) Modify the PORT to match the port number you set in the TCPIPSERVICE definition.
For the DJAR resource definition, on the HFSFILE option specify the fully-qualified file name of the HelloWorldEJB.jar deployed JAR file. The HelloWorldEJB.jar file is in the HFS EJB samples directory. The default samples directory is
/usr/lpp/cicsts/cicsts21/samples.ejb/helloworld

where cicsts21 is the value of the CICS_DIRECTORY variable used by the DFHIJVMJ job during CICS installation.

Note: There is no supplied REQUESTMODEL definition, because it's not necessary to install one. If you don't install one, the sample uses the default transaction ID of CIRP.

c. Install group mygroup to make these definitions known to CICS.

2. After ensuring that the COS Naming Server has been started, publish the enterprise bean to it. At the CICS region console, issue the following command:

   CEMT PERFORM DJAR(HELLO) PUBLISH

3. Set the status of the TCPIPSERVICE to OPEN:

   CEMT SET TCPIPSERVICE(EJBTCP1) OPEN

You should see the following messages on the CICS Console:

DFHEJ0701 CorbaServer EJB1 has been created.
DFHEJ1520 CorbaServer EJB1 is now accessible.
DFHEJ0901 DJar HELLO within CorbaServer EJB1 has been created.
DFHEJ1540 DJar HELLO and the Beans it contains are now accessible.
DFHEJ5010 Publishing bean HelloWorld in the Shelf directory
shelf/...../EJB1/homeiors as file HelloWorldHome.ior.
DFHSO0107 TCPIPSERVICE EJBTCP1 has been opened on port xxxx
at IP address xxx.xxx.xxx.xxx.

where shelf is the shelf root directory on HFS.

Web server setup
On the Web server, you must install the Web components of the “Hello World” sample application. From the HFS EJB samples directory, you need:

- HelloWorldCLI.jar. Includes client EJB stubs required by the sample servlet.
- readme.txt. Contains:
  1. Step-by-step instructions for installing the Web components of the sample on WebSphere Version 3.5 (the Web server that is supplied with CICS).
  2. Hints, tips, and debugging information.

Note: The default samples directory is

/usr/lpp/cicsts/cicsts21/samples.ejb/helloworld

where cicsts21 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 Web server (which may or may not be WebSphere). It is suitable for experienced users. If your Web server is WebSphere Version 3.5 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. Unpack the HelloWorldWeb.jar file, using WinZip or a similar decompression utility.
   
   Store the following files in a directory called cicshello/servlets on the Web server, or as specified by your Web server's guidelines for adding dynamic Web components:
   
   cics/sample/DataBean.class
   cics/sample/HelloWorldServlet.class
   cics/sample/HelloWorldServlet.servlet

   Note: The full servlet class name is cics.sample.HelloWorldServlet.

   Store the following files in a directory called cicshello/web on the Web server, or as specified by your Web server's guidelines for adding static Web components:
   
   index.html
   HelloWorldResults.jsp
   HelloWorldError.jsp

2. Install copies of the following JAR files on the Web server and add them to the Web Application (Servlet Engine) classpath:
   
   • HelloWorldCLI.jar. This must be in the Web Application (Servlet Engine) classpath because it contains the client EJB stubs required by the servlet.
   
   • j2ee.jar. This is required because the CICS enterprise bean called by the servlet is at level EJB 1.1. You can obtain j2ee.jar in either of the following ways:
     
     a. By installing the CICS JAR Development tool from the CICS installation CDs. (How to do this is described in "Installing and configuring the CICS JAR development tool for EJB technology" on page 156.) After you have installed the CICS JAR Development tool, j2ee.jar will be in directory \Program Files\IBM\CICS TS 2.1 Tools\Common
     
     b. By downloading it from java.sun.com/products.

3. Configure the sample servlet, as specified by your Web server's guidelines.
   
   Set the name by which the servlet is known to the Web Application (Servlet Engine) to Hello.
   
   Set the Universal Resource Indicator (URI) of the servlet to /cicshello/Hello. (In WebSphere, the URI is known as the “Servlet Web Path”.)

4. In the index.html file:
   
   a. Check the line:
   
   <FORM METHOD="post" ACTION="/cicshello/Hello">

   Check that the value of the ACTION attribute matches the Universal Resource Indicator of the servlet. (In WebSphere, the URI is known as the “Servlet Web Path”.) If you installed the servlet as suggested in step 3, the
ACTION attribute will be set correctly. You need to change its value only if you configured the servlet with a different Universal Resource Indicator.

b. Optionally, change the default values of the Name Service:, Provider URL:, and JNDI Name: input fields so that they are valid for your installation. This prevents the user having to overtype the values every time the sample is run. (The layout of the opening screen produced by index.html is shown in Figure 18 on page 133.)

1) In the following line, if necessary, change the value attribute of the nameService text field to that of your EJB naming service. This could be either Sun's or IBM's naming service. (As supplied, the IBM naming service is used.)

```
<INPUT TYPE="text" NAME="nameService" ID="nameService"
       value="com.ibm.ejs.ns.jndi.CNInitialContextFactory">
```

2) In the following line, change the value attribute of the providerURL text field to the URL and port number of the COS Naming Server where the enterprise bean is published.

```
<INPUT TYPE="text" NAME="providerURL" ID="providerURL"
       value="iiop://nameserver.location.company.com:900/"> 
```

For example, if the URL of your COS Naming Server is mycosns.ibm.com and its port number is 900, the value attribute should specify "iiop://mycosns.ibm.com:900/".

3) In the following line, change the value attribute of the jndiName text field to the name by which the enterprise bean is known to JNDI.

```
<INPUT TYPE="text" NAME="jndiName" ID="jndiName" value="prefix/HelloWorld">
```

5. In the HelloWorldServlet.servlet file, check the lines:

```
<uri>/cicshello/HelloWorldResults.jsp</uri>
<uri>/cicshello/HelloWorldError.jsp</uri>
```

Check that the lines match the URIs of the Java Server Pages used by the HelloWorldServlet servlet. If you installed the JSPs as suggested in step 1, and configured the servlet as suggested in step 3, these lines will be set correctly.

Note: The names of the JSPs used for the results and error pages are hard-coded in the HelloWorldServlet class.

**Testing the “Hello World” sample**

To test the application:

1. Ensure that all the following are running:
   - The Web server
   - Web Application (Servlet Engine)
   - COS Naming Server
   - CICS region

2. Start a Web browser and point it at the URL of the sample on the Web server—for example:

   http://myServer.ibm.com/cicshello

   The opening screen shown in Figure 18 on page 133 appears.
3. Enter a phrase in the Hello String: field.
4. Check that the Name Service:, Provider URL:, and JNDI Name: fields contain correct values. If they do not, overtype them with the address of your EJB naming service, the URL and port number of your Naming Server (for example, iiop://mycosns.ibm.com:900/), and the name by which the enterprise bean is known to JNDI, respectively.
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 CICS sample application**

The EJB CICS sample demonstrates how you can use enterprise beans to make existing, CICS-controlled, information available to Web users.
What the EJB CICS 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 Java Server Pages, running on a Web 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 V2ACTDB program retrieves information from the accounts data table. The V2CSTDB 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 and sends the number to the enterprise bean.
3. The enterprise bean uses the CCF interface of the CICS Connector for CICS TS to link to the CICS server programs, passing the customer number.
4. 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.
5. The enterprise bean returns the customer’s details to the servlet, which uses Java Server Pages to display them on the user’s browser. If the customer number is not valid, the browser displays an error page.

Figure 19 on page 135 shows the components of the sample application.
Prerequisites for the EJB CICS sample

To run the EJB CICS sample, you need:

- A CICS EJB server. How to set one up is described in "Chapter 13. Setting up an EJB server" on page 113.
- DB2 Version 5 or later.
- A fully-functional Web server that supports Java Server Pages Version 1.0 or later.
- A Corba Object Services (COS) Naming Directory Server that supports JNDI Version 1.2 or later. How to set one up is described in "Actions required outside CICS" on page 113.

Figure 19. Overview of the EJB CICS 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 under WebSphere on a Windows NT server.
Supplied components of the EJB CICS sample

Table 8 lists the files supplied with the EJB CICS sample.

Table 8. Supplied components of the EJB CICS sample

<table>
<thead>
<tr>
<th>Filename</th>
<th>Type</th>
<th>Default location</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>CICS- and DB2-related files</td>
</tr>
<tr>
<td>DFHEBBND</td>
<td>JCL</td>
<td>SDFHSAMP</td>
<td>Binds the DB2 tables to the COBOL server programs for the first time. Use only when installing the sample.</td>
</tr>
<tr>
<td>DFHEBCB1</td>
<td>JCL</td>
<td>SDFHSAMP</td>
<td>Compiles and link-edits the V2ACTDB server program. (Also contains source code.)</td>
</tr>
<tr>
<td>DFHEBCB2</td>
<td>JCL</td>
<td>SDFHSAMP</td>
<td>Compiles and link-edits the V2CSTDB server program. (Also contains source code.)</td>
</tr>
<tr>
<td>DFHEBCBJ</td>
<td>JCL</td>
<td>SDFHSAMP</td>
<td>Procedure used by the DFHEBCB1 and DFHEBCB2 jobs to preprocess, compile, and link-edit the server programs.</td>
</tr>
<tr>
<td>DFHEBDAT</td>
<td>JCL</td>
<td>SDFHSAMP</td>
<td>Inserts sample data into the DB2 data tables.</td>
</tr>
<tr>
<td>DFHEBDEF</td>
<td>Resource definitions</td>
<td>SDFHSAMP</td>
<td>Contains the CICS resource definitions required by the sample application.</td>
</tr>
<tr>
<td>DFHEBGRT</td>
<td>JCL</td>
<td>SDFHSAMP</td>
<td>Grants authority to the DB2 plan.</td>
</tr>
<tr>
<td>DFHEBREB</td>
<td>JCL</td>
<td>SDFHSAMP</td>
<td>Re-binds the DB2 tables to the COBOL server programs. Use each time a server program is recompiled.</td>
</tr>
<tr>
<td>DFHEBTAB</td>
<td>JCL</td>
<td>SDFHSAMP</td>
<td>Defines the DB2 data tables.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Java files</td>
</tr>
<tr>
<td>ccf.jar</td>
<td>JAR file</td>
<td>HFS samples directory: see Note.</td>
<td>VisualAge for Java Common Connector Framework (CCF) classes</td>
</tr>
<tr>
<td>eablib.jar</td>
<td>JAR file</td>
<td>HFS samples directory: see Note.</td>
<td>VisualAge for Java Enterprise Access Builder (EAB) classes</td>
</tr>
<tr>
<td>recjava.jar</td>
<td>JAR file</td>
<td>HFS samples directory: see Note.</td>
<td>VisualAge for Java Record Framework classes</td>
</tr>
<tr>
<td>SampleCLI.jar</td>
<td>JAR file</td>
<td>HFS samples directory: see Note.</td>
<td>Client EJB stubs required by the servlet.</td>
</tr>
</tbody>
</table>
Table 8. Supplied components of the EJB CICS sample (continued)

<table>
<thead>
<tr>
<th>Filename</th>
<th>Type</th>
<th>Default location</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>SampleEJB.jar</td>
<td>Deployed</td>
<td>HFS samples directory: see Note.</td>
<td>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.</td>
</tr>
<tr>
<td>SampleServlet.jar</td>
<td>JAR file</td>
<td>HFS samples directory: see Note.</td>
<td>The Web components of the sample application—Java servlet classes and source files; HTML and JSPs. Files must be unpacked and stored on the Web server.</td>
</tr>
<tr>
<td>SampleVAJ.dat</td>
<td>VAJ</td>
<td>HFS samples directory: see Note.</td>
<td>The source code of the sample packaged as a VisualAge for Java repository file.</td>
</tr>
</tbody>
</table>

**Note:** The default HFS samples directory is /usr/lpp/cicsts/cicsts21/samples.ejb/bankaccount where cicsts21 is the value of the CICS_DIRECTORY variable used by the DFHIJVMJ job during CICS installation.

### Installing the EJB CICS sample

Installing the EJB CICS sample requires actions on:

1. The OS/390 or z/OS server (DB2 and CICS)
2. The Web server

#### z/OS setup

The JCL to define the EJB CICS sample application to DB2 and CICS is contained in the SDFHSAMP partitioned data set.

On OS/390 or z/OS:

1. Compile the CICS COBOL programs. Jobs DFHECBB1 and DFHECBB2 contain the source code of the COBOL server programs. These jobs use the DFHEBCBJ sample procedure to preprocess, compile, and link-edit the programs.
   a. Modify the destination load library in the DFHECBB1 and DFHECBB2 jobs.
   b. Modify the CICS, DB2, COBOL, and Language Environment library references in the DFHEBCBJ procedure.
   c. Run the DFHECBB1 job.
   d. Run the DFHECBB2 job.

**Authority required:** Write access to the datasets whose members are being created: the load module is written to a member of a dataset in the CICS DD DFHRPL concatenation; the SQL statements in the program are written to a member in DB2's DRMLIB.DATA dataset.

2. Define DB2 tables, table space, storage group, and indexes and populate the tables:
   a. Modify the DFHEBTAB and DFHEBDAT jobs to use your DB2 datasets and subsystem ID. The jobs contain comments which indicate which lines must be changed.
b. Run the DFHEBTAB job. This creates DB2 definitions for the sample application.

c. Run the DFHEBDAT job. This does the data insertions for the sample application.

**Authority required:** Ability to create a DB2 database, storage group, tablespace, tables, and indices.

3. Bind the DB2 tables to the COBOL server programs:

   a. Modify the DFHEBBND and DFHEBGRT jobs to point to your DB2 datasets and subsystem ID. The jobs contain comments which indicate which lines must be changed.

   b. Run the DFHEBBND job.

   c. Run the DFHEBGRT job.

**Authority required:** DB2 authority to perform a BIND for this database, and write access to the CICS region.

**Notes:**

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. The *first time* you do a bind you must create the bind and grant authority to the plan. For the sample program, the DFHEBBND and DFHEBGRT jobs handle these functions.

c. If you recompile one of the server programs subsequently and intend it to access DB2, each time you recompile you must:

   1) Run the DFHEBREB (not the DFHEBBND) job.

   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 V2CSTDB program and recompile it, use

```
CEMT SET PROG(V2CSTDB) NEW
```

(V2CSTDB is created by the DFHEBCB2 job and defined to the CICS region in the EJBSAMP resource definition group—see step 4.)

4. Define the programs, deployed JAR file, and DB2 connections used by the sample to CICS. All the required definitions are in the supplied DFHEBDEF file. However, you must modify some of the definitions to suit your environment before installing them. DFHEBDEF contains comments that tell you what needs to be modified.

a. In the DFHEBDEF file:

   1) Modify the DJAR definition:

```
1) On the CORBASERVER option, specify the name of the CorbaServer execution environment in which the bean is to run.

2) On the HFSFILE option, specify the fully-qualified file name of the deployed JAR file. The deployed JAR file is called SampleEJB.jar and is in the HFS EJB samples directory. The default samples directory is /usr/lpp/cicsts/cicsts21/samples.ejb/bankaccount
```

where cicsts21 is the value of the CICS_DIRECTORY variable used by the DFHIJVMJ job during CICS installation.
2) Modify the DB2CONN definitions as indicated by the comments in the file.

3) The PROGRAM definitions do not need to be modified.

Notes:

1) DFHEBDEF also contains a CORBASERVER and a TCPIPSERVICE definition. However, these are for reference only and have been commented out. It is strongly recommended that you set up your EJB server, as described in Chapter 13, Setting up an EJB server on page 113, before attempting to install the sample programs. If you do this, you do not need these CORBASERVER and TCPIPSERVICE definitions.

2) There is no supplied REQUESTMODEL definition, because it’s not necessary to install one. If you don’t install one, the sample uses the default transaction ID, CIRP.

b. Install the modified resource definitions into CICS. 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. All the definitions in DFHEBDEF are in a resource definition group called EJBSAMP.

Authority required: RACF authority to install resource definitions into the CICS region.

5. Add the following VisualAge for Java support classes to the trusted middleware classpath. These classes provide program-connection and COBOL/Java data conversion facilities used by the sample enterprise bean. They are supplied in the HFS EJB samples directory.

ccf.jar
eablib.jar
recjava.jar

The recommended way to add directories and files to the trusted middleware classpath is to specify them on the TMSUFFIX initialization option in your JVM profile. For information about TMSUFFIX, see the CICS System Definition Guide.

6. After ensuring that the COS Naming Server has been started, publish the enterprise bean to it. At the CICS region console, issue the following command:

CEMT PERFORM DJAR(SAMPLE1) PUBLISH

7. Ensure that the DB2 connection status is CONNECTED by issuing the following command at the CICS system console:

CEMT SET DB2CONN CONNECTED

Web server setup

On the Web server, you must install the Web components of the EJB CICS sample application. From the HFS EJB samples directory, you need:

- SampleServlet.jar. Contains the Web components of the sample.
- SampleCLI.jar. Includes client EJB stubs required by the sample servlet.
- readme.txt. Contains:
  1. Step-by-step instructions for installing the Web components of the sample on WebSphere Version 3.5 (the Web server that is supplied with CICS).
  2. Hints, tips, and debugging information.

Note: The default samples directory is...
Important
The rest of this section contains generic instructions for installing the Web components of the sample on a Web server (which may or may not be WebSphere). It is suitable for experienced users. If your Web server is WebSphere Version 3.5 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. Unpack the SampleServlet.jar file, using WinZip or a similar decompression utility.
   Store the following files in a directory called cicssample/servlets on the Web server, or as specified by your Web server’s guidelines for adding dynamic Web components:
   cics/sample/SampleServlet.class
   cics/sample/SampleServlet.servlet
   cics/sample/CustomerData.class
   cics/sample/ErrorData.class
   Note: The full servlet class name is cics.sample.SampleServlet.
   Store the following files in a directory called cicssample/web on the Web server, or as specified by your Web server’s guidelines for adding static Web components:
   index.html
   SampleResults.jsp
   SampleError.jsp

2. Install copies of the following JAR files on the Web server and add them to the Web Application (Servlet Engine) classpath:
   • SampleCLI.jar. This must be in the Web Application (Servlet Engine) classpath because it contains the client EJB stubs required by the servlet.
   • j2ee.jar. This is required because the CICS enterprise bean called by the servlet is at level EJB 1.1. You can obtain j2ee.jar in either of the following ways:
     a. By installing the CICS JAR Development tool from the CICS installation CDs. (How to do this is described in Installing and configuring the CICS JAR development tool for EJB technology on page 156.) After you have installed the CICS JAR Development tool, j2ee.jar will be in directory \Program Files\IBM\CICS TS 2.1 Tools\Common
     b. By downloading it from java.sun.com/products.
   3. Configure the sample servlet, as specified by your Web server’s guidelines.
      Set the name by which the servlet is known to the Web Application (Servlet Engine) to Sample.
      Set the Universal Resource Indicator of the servlet to /cicssample/Sample. (In WebSphere, the URI is known as the “Servlet Web Path”.)
   4. In the index.html file:
      a. Check the line:
         <FORM METHOD="post" ACTION="/cicssample/Sample"
Check that the value of the ACTION attribute matches the Universal Resource Indicator of the servlet. (In WebSphere, the URI is known as the "Servlet Web Path"). If you installed the servlet as suggested in step 3, the ACTION attribute will be set correctly. You need to change its value only if you configured the servlet with a different Universal Resource Indicator.

b. Optionally, change the default values of the Name Service:, Provider URL:, and JNDI Name: input fields so that they are valid for your installation. This prevents the user having to overtype the values every time the sample is run. (The layout of the opening screen produced by index.html is shown in Figure 20 on page 142)

1) In the following line, if necessary, change the value attribute of the nameService text field to that of your EJB naming service. This could be either Sun's or IBM's naming service. (As supplied, the IBM naming service is used.)

```html
<INPUT TYPE="text" NAME="nameService" ID="nameService" value="com.ibm.ejs.ns.jndi.CNInitialContextFactory">
```

2) In the following line, change the value attribute of the providerURL text field to the URL and port number of the COS Naming Server where the enterprise bean is published.

```html
<INPUT TYPE="text" NAME="providerURL" ID="providerURL" value="iiop://nameserver.location.company.com:900">
```

For example, if the URL of your COS Naming Server is mycosns.ibm.com and its port number is 900, the value attribute should specify "iiop://mycosns.ibm.com:900/".

3) In the following line, change the value attribute of the jndiName text field to the name by which the enterprise bean is known to JNDI.

```html
<INPUT TYPE="text" NAME="jndiName" ID="jndiName" value="prefix/CICSSample">
```

5. In the SampleServlet.servlet file, check the lines:

```html
<uri>/cicssample/SampleResults.jsp</uri>
<uri>/cicssample/SampleError.jsp</uri>
```

Check that the lines match the URIs of the Java Server Pages used by the SampleServlet servlet. If you installed the JSPs as suggested in step 1, and configured the servlet as suggested in step 3, these lines will be set correctly.

**Note:** The names of the JSPs used for the results and error pages are hard-coded in the SampleServlet class.

**Testing the EJB CICS sample**

To test the application:

1. Ensure that all the following are running:
   - The Web server
   - Web Application (Servlet Engine)
   - COS Naming Server
   - CICS region
   - DB2 subsystem

2. Start a Web browser and point it at the URL of the sample on the Web server—for example:

   `http://myServer.ibm.com/cicssample`

   The opening screen shown in [Figure 20 on page 142](#) appears.
3. Enter a customer number. (Using the supplied DB2 data, valid customer numbers are 1 through 5).

4. Check that the Name Service:, Provider URL:, and JNDI Name: fields contain correct values. If they do not, overtype them with the address of your EJB naming service, the URL and port number of your Naming Server (for example, iiop://mycosns.ibm.com:900/), and the name by which the enterprise bean is known to JNDI, respectively.

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 21 on page 143 shows the result of a successful enquiry.
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 data conversion**

To represent text data, Java programs always use the Unicode character set, while CICS programs use EBCDIC. When an enterprise bean calls a CICS 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. [Data conversion and the CICS Connector for CICS TS on page 215](#) describes two methods of handling this data conversion. The sample enterprise bean uses method 2. That is, it converts the text data returned by COBOL program V2CSTDB directly from EBCDIC to Unicode in the connector's output record bean. (No conversion is necessary for server program V2ACTDB, nor on input to V2CSTDB, because there are no text values in the communications areas.)

As supplied, the sample assumes that the CICS server code page is 037. To change to a different server code page, modify the `V2CSTDBCommarea` class, either directly or by using the VisualAge for Java Enterprise Access Builder tool.

If you wish, to convert the text data returned by COBOL program V2CSTDB you can use method 1 in [Data conversion and the CICS Connector for CICS TS on page 215](#). A sample conversion template, using DFHCNV macros, is supplied in file...
DFHEBCNV. It shows how to convert from EBCDIC code page 037 to ASCII code page 8859–1 (which is equivalent to Unicode).
Chapter 16. Writing enterprise beans

To write enterprise beans and enable them for execution in the CICS environment, you need to perform the following steps:

Setting up the system

1. Set up CICS as an IIOP server, see [Chapter 10. Configuring CICS for IIOP] on page 71.
2. Configure the CICS TCP/IP listener region, see [Setting up TCP/IP for IIOP] on page 72.
3. Set up the host system environment, see [Setting up the host system for IIOP] on page 71.
4. Make sure that all the steps in [Setting up a single-region EJB server] on page 113 have been performed, or [Setting up a multi-region EJB server] on page 120 if you are using load balancing across multiple CICS regions.
5. Implement any security controls required by your system, see [Chapter 21. Managing security for enterprise beans] on page 227.

Writing enterprise beans

You can write beans that use only the interfaces defined by the Sun Microsystems' 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 enterprise beans that use the JCICS classes to access CICS services and resources directly. These beans are only portable to other CICS EJB compliant servers.

You can write your beans on a workstation using any Integrated Development Environment that supports the Enterprise JavaBeans Specification, Version 1.1.

You should develop all new Java programs for CICS TS 2.1 using an application development environment that supports Java 2 at the SDK 1.3 level. You can use VisualAge for Java 3.5 Enterprise Edition but note that it supports the SDK 1.2 level. Note also that EJBs developed using VisualAge for Java V3.5, which supports the EJB 1.0 specification, need to be migrated to the EJB 1.1 specification level using the supplied JAR development tool [The CICS JAR development tool for EJB technology] on page 175. If you use VisualAge for Java release 3.5, you will need to make the JCICS classes available yourself. You can do this by using ftp to transfer dfjcics.jar from your install location (usr/lpp/cicsts21/lib) to your workstation, then create a project in VisualAge, and import the jar into it.

[ Coding a session bean] on page 146 gives an example of the steps involved in writing a session bean without using an IDE.

You can use the CICS Connector for CICS TS to build enterprise beans that make use of existing CICS programs. See [Chapter 19. The CICS Connector for CICS TS] on page 205 for a description of the CICS connector, 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 OS/390 system or run them from the supplied CD on the target workstation. See "Chapter 17. Installing and configuring CICS deployment tools for EJB technology on page 155" for more information.

See "Chapter 18. Deploying enterprise beans on page 173" for a description of the deployment process, and "Using CICS deployment tools for EJB technology on page 173" 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. It must be loaded into the CICS Jar Development Tool available with CICS Transaction Server 2.1, to create a suitable deployment descriptor and generate code for the bean that tailors it for execution inside CICS. See "The CICS JAR development tool for EJB technology on page 175".

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" on page 147.
3. "Coding the bean implementation" on page 147.

Then you need to compile and package your program:

1. "Compiling the code" on page 149
2. "Packaging the code" on page 149

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:

```java
package casino;

public interface RouletteWheelHome extends javax.ejb.EJBHome {
    public RouletteWheel create() throws javax.ejb.CreateException, java.rmi.RemoteException;
    public RouletteWheel create(int dollars) throws javax.ejb.CreateException, java.rmi.RemoteException;
}
```
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:

```java
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 java.rmi.RemoteException;

    // Check the current status of the wheel.
    public String getCurrentStatus() throws java.rmi.RemoteException;

    // Collect winnings from the wheel (if any!)
    public int collectWinnings() throws java.rmi.RemoteException;
}
```

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 146 is generated by the tooling, so if we need to implement an overloaded create method, we have to do it here:

```java
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 java.rmi.RemoteException {}
    public void ejbPassivate() throws java.rmi.RemoteException {}
    public void ejbRemove() throws java.rmi.RemoteException {}
    public SessionContext getSessionContext() { return ctx; }
    public void setSessionContext(SessionContext ctx) throws java.rmi.RemoteException {
        this.ctx = ctx;
    }

    // The bean state information
    private int wheelValue;
    private int currentCredit;

    // Our create methods

    public void ejbCreate() throws java.rmi.RemoteException, CreateException {
        currentCredit = 100;
    }
}
```
wheelValue = ((int)System.currentTimeMillis())%37;
}

public void ejbCreate(int credit) throws java.rmi.RemoteException, 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 java.rmi.RemoteException {
    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)+"
    // Did the caller win?
    if (color(wheelValue).equalsIgnoreCase(color)) {
        currentCredit+=(amount*2);
        result.append("Well Done! You won $"+amount*2);
    } else {
        currentCredit -= amount;
        result.append("Bad Luck! You lost ");
        result.append(amount);
    }
    result.append(", you now have $"+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 java.rmi.RemoteException {
    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 java.rmi.RemoteException {
    int winnings = currentCredit;
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
```
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) is an API specified in the Java programming language that provides the naming and directory function to Java programs. Figure 22 shows the JNDI structure.

![JNDI Diagram](image)

After an enterprise bean has been registered in a nameserver 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 CORBA Object Services (COS) Naming Directory Server that supports the Java Naming and Directory Interface (JNDI) Version 1.2. You can use, for example, the WebSphere Application Server Advanced Edition running on an external Windows NT machine. See "Defining nameservers" on page 72 for information about using nameservers.

To use JNDI you'll need to set the following system properties in your CICS JVM system properties file. See "Chapter 7. Configuring the JVM" on page 47 for information about defining JVM system properties files. For example:

```java
java.naming.provider.url=iiop://hostname.hursley.ibm.com:port_number
```

The following example shows a client program, Gambler.java, that works with the RouletteWheel bean developed in "Coding a session bean" on page 146. When a bean reference is obtained from the 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 a utility class EJBUtills: This utility class is used by the client program Gambler.

**EJBUtills.java**

```java
import javax.naming.*;
import java.util.Hashtable;

class EJBUtills {
```
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;
}

Notes:

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 nameservers on page 72](#) for information about using nameservers.

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.ejs.ns.jndi.CNInitialContextFactory.

Gambler.java

```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.println("Collecting winnings:");
System.out.println(wheel.collectWinnings());

System.out.println("" );
System.out.println("Removing the roulette wheel");
wheel.remove();

} catch (Exception e) {
    System.err.println("Error whilst gambling:");
e.printStackTrace();
}
}

Notes:
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

Using EDF with enterprise beans

You can use EDF to test enterprise beans by setting the CEDF parameter to YES in the PROGRAM resource definition for DFJIIRP that is supplied in group DFHIIO. You must also take one of the following actions:

- Use a unique transaction id for the create and the business method and put each into a unique TRANCLASS with MAXACTIVE set to one.
- Have all create methods use CIRP and define each a unique transaction id for each set of business methods, or use CIRP for all business methods and define a unique transaction id for each create method.
- Set MAXACTIVE to one in your transaction's TRANCLASS.
Chapter 17. Installing and configuring CICS deployment tools for EJB technology

The steps involved in setting up the application development infrastructure that will allow you to develop and deploy enterprise beans to the CICS EJB server are as follows:

1. Make sure the prerequisite hardware and software are in place. See "Hardware and software requirements for CICS deployment tools for EJB technology" on page 156 for guidance on this.

2. Install the tools on their target workstations. You do not have to install the tools in any particular order.

3. Configure the tools according to the detailed setup instructions. See "Installing and configuring the CICS JAR development tool for EJB technology" on page 156, "Installing and configuring the CICS development deployment tool for EJB technology" on page 157, and "Installing and configuring the CICS production deployment tool for EJB technology" on page 170 for detailed guidance.

The CICS JAR development tool for EJB technology incorporating the CICS code generation utility for EJB technology, and the CICS production deployment tool for EJB technology need to be set up on your application development workstation. The CICS development deployment tool for EJB technology needs to be installed on a workstation running the WebSphere Application Server, which may or may not be on the same machine as the other tools.

The workstation and WebSphere components of the deployment tools are supplied on a CD-ROM as a set of InstallShield packages. The CD image is also accessible via a Web site: http://www.software.ibm.com/software/ts/cics/support/

To install one or more of the deployment tools, run the Setup program from the CD-ROM on the target workstation, or run it from your downloaded copy of the CD image. This starts the InstallShield Wizard, which leads you through the rest of the installation process.

Select Complete to install all three deployment tools, or Custom to control which tools and features to install. Follow the on-screen prompts to install the selected tools. The Wizard also creates a CICS IBM group in your workstation’s Start menu from which you can start the CICS JAR development tool or the CICS production deployment tool. (The CICS development deployment tool is run from a web browser.)

If, later, you decide to re-install the tools, run the Setup program again. Because the InstallShield Wizard determines that the tools are already installed, it offers you 3 options Modify, Repair and Remove (full explanations of each are provided by the InstallShield). For a re-installation take the remove option, then, when the InstallShield process completes, run the Setup program again. This time it will offer the Complete or Custom choice and you can continue as you did before.

You can download any service updates to the CICS deployment tools for EJB technology from the following IBM web site:

http://www.software.ibm.com/software/ts/cics/support/
Hardware and software requirements for CICS deployment tools for EJB technology

The additional hardware and software required to support CICS deployment tools for EJB technology is as follows:

**Hardware**

- One or more client PCs capable of running Microsoft® Windows NT or Microsoft Windows® 2000 together with your Java development environment, such as VisualAge® for Java. The recommended minimum is a 266MHz processor with 64Mb of RAM.
- A server PC platform capable of running WebSphere Application Server. This is required for the CICS development deployment tool and for the run time operation of the CICS EJB server. It is not required for the operation of the other CICS deployment tools. The recommended minimum is a 700MHz processor with, 300Mb of RAM.

**Software**

Install one of the following operating systems on each application development workstation and on the server:

- Microsoft Windows NT, version 4.0, or later

Install the following on each application development workstation:

- One of the following web browsers:
  - Netscape Communicator, version 4.5, or later
  - Microsoft Internet Explorer, version 5.00, or later

(The CICS development deployment tool is the only one of the deployment tools that requires a browser.)

On your server platform only, you need to install WebSphere Application Server version 3.5. WebSphere is an essential prerequisite for the CICS development deployment tool and also provides the necessary JNDI support at runtime.

**Installing and configuring the CICS JAR development tool for EJB technology**

Setting up the CICS JAR development tool involves installation and configuration tasks on your application development workstation. When you install this tool the InstallShield Wizard also installs the CICS code generation utility (CICSCGU). You cannot install one tool without the other.

Use the InstallShield Wizard application to install the required files. Select **Complete** to install all the CICS deployment tools or **Custom** to control which of the tools and features to install on your workstation. Just follow the prompts to complete the installation.
Configuration considerations

You need to check that certain Sun standard EJB interface classes are present on your workstation’s classpath in order to use the CICS JAR development tool and the CICS code generation utility. Specifically:

- Before running either of the tools ensure that the EJB 1.1 standard interface classes are present on your classpath. These classes are available either in file ejb11.jar, which is shipped in the IBM SDK 1.3 on OS/390, or in file j2ee.jar, which is shipped in the Sun J2EE SDK.

- Before you can use the CICS JAR development tool’s facility to migrate EJB 1.0 enterprise beans to EJB 1.1 enterprise beans, ensure that the EJB 1.0 standard interface classes are present on your classpath. These classes are in the file javax.ejb.zip. If this file is not on your workstation, you can download it from the web site http://java.sun.com.

Installing and configuring the CICS development deployment tool for EJB technology

Setting up the CICS development deployment tool involves installation and configuration tasks on the WebSphere Application Server and on your target CICS unit test region. Some prior experience of using WebSphere and CICS would be of benefit in carrying out the relevant tasks.

The basic setup procedure is as follows:

Step 1
If you haven’t already done so, install the WebSphere Application server on your target workstation. It is recommended that you use the same user ID for your WebSphere installation as your workstation’s logon password. This should contain 8 or fewer characters in order to avoid any possible DB2 compatibility issues. See the WebSphere Application Server documentation for detailed guidance on installing and configuring WebSphere.

Step 2
Install and configure the WebSphere components of the development deployment tool on your workstation. See Setting up the WebSphere components of the CICS development deployment tool for EJB technology on page 158 for guidance.

Step 3
Configure the deployment configuration file. See Configuring the deployment configuration file on page 162 for guidance.

Step 4
If you haven’t already done so, configure a CICS EJB server. See Chapter 13. Setting up an EJB server on page 113 for guidance.

Step 5
Set up the CICS components of the development deployment tool. See Setting up the CICS components of the CICS development deployment tool on page 167 for guidance.

Step 6
Set your Web browser so that it does not cache Web pages. See details in Using the CICS development deployment tool for EJB technology on page 184. If you alter this setting for other uses of your browser, you will need to do this each time you use the CICS development deployment tool.
Most of the guidance in this section relates to setting up a basic configuration on a single CICS region. For information about setting up more complex multiple configurations, see "Setting up more complex development deployment tool configurations" on page 169.

**Note:** When specifying file and path names use only the characters in the POSIX portable filename character set. These are:
- Uppercase or lowercase letters A to Z
- Numbers 0 to 9
- Period
- Underscore
- Hyphen

Do not include any null characters in a path name.

### Setting up the WebSphere components of the CICS development deployment tool for EJB technology

This section explains how to set up the web application component of the CICS development deployment tool on your server; and how to configure the deployment configuration file. Some WebSphere experience would be of great benefit in carrying out these tasks.

The web application component of the development deployment tool is configured in much the same way as any standard web application in WebSphere Application Server. The procedure is as follows:

**Step 1**  
Install the files from the InstallShield package.

**Step 2**  
Create a web application in WebSphere Application Server.

**Step 3**  
Add one or more servlets to the web application

### Installing the web application

Use the InstallShield Wizard application to install the required files. The Wizard prompts you for the location of the WebSphere Application Server directory. The displayed default is `C:\WebSphere\AppServer`. InstallShield assumes that the `default_host` directory is located under the `WebSphere\AppServer` directory as follows:

```
<drive>\WebSphere\AppServer\hosts\default_host
```

Installation is more straightforward if you accept this default location. If your `default_host` directory is located elsewhere, see "Installing the web application in a non-default location" on page 153 for installation instructions.

Once you have specified the location of the WebSphere Application Server directory, the Wizard copies the web application files to the appropriate locations.

The final directory structure installed into the `default_host` directory should be as follows:

- `Cics_ejb`
- `Cics_ejb\dcf`
- `Cics_ejb\servlets`
- `Cics_ejb\web`
The contents of these installed directories are as follows:

- The Cics_ejb is the web application base directory.
- The Cics_ejb\dcf directory contains skeleton and sample deployment configuration files.
- The Cics_ejb\servlets directory contains the dfjadwas.jar file and the servlet page list configuration file (CicsEjbAd.servlet).
- The Cics_ejb\web directory contains web resources.

The Wizard also installs required JAR files into the directory: Program Files\IBM\CICS TS 2.1\Common. Do not change the location of this directory. It is required for configuration of the web application.

**Installing the web application in a non-default location:** If your WebSphere Application Server directory does not match the default location, install the web application as follows:

1. Create a temporary WebSphere Application Server directory; for example, C:\temp\WebSphere\AppServer.
2. Start the InstallShield installer. When it prompts for a WebSphere Application Server directory, use the temporary WebSphere\AppServer directory you created and allow the installation to proceed to create sub directories and place files in those directories.
3. When the installer has completed, copy the installed Cics_ejb directory and all sub folders from the temporary WebSphere\AppServer directory to the default_host directory under the true WebSphere\AppServer directory.

**Configuring the web application**

Once you have installed the necessary files on your server platform, you need to perform some configuration tasks on the WebSphere Application Server before you can use the tool.

**Step 1**

Start the WebSphere Application Server Administrative Console and ensure the view is set to Topology. If it isn’t, click the Topology button on the tool bar or choose Topology from the View menu. Expand the WebSphere Administrative domain tree until you can see the web applications under the servlet engine you want to use.

**Step 2**

Choose Create a Web application from the Console/Tasks menu or from the Wizards drop-down list to display the Create a Web application dialog. Complete it as follows:

1. Enter the name of the web application as CICS Dev Deploy Tool.
2. Check the Enable File Servlet check box.
3. Ensure that the Serve Servlets By Classname check box is clear.
4. Ensure the JSP version to use is set to Enable JSP .91.

Click the Next button.

**Step 3**

Expand the tree nodes and select the servlet engine you want to install the web application into (typically Default Servlet Engine). Click the Next button.

**Step 4**

Change the web application path to /Cics_ejb This is the unique web path used to access the web application. Leave the other fields as they are and click the Next button.
Step 5

On the final screen, in the Document Root field enter the full path to the Cics_ejb\web directory. In the Classpath enter the full path to the Cics_ejb\servlets directory. Click the Finish button. A dialog box should now be displayed confirming the creation of the web application. The web application should appear in the topology tree.

Step 6

Edit the classpath of the web application as follows. Select the web application CICS Dev Deploy Tool in the topology tree and click the Advanced tab. You need to edit the path of the servlets directory and add the following JAR files to your classpath:
- dfjadwas.jar
- xml4j.jar
- log.jar,
- j2ee.jar

The dfjadwas.jar file is located in the Cics_ejb\servlets directory. The other three files are in the Common directory. The order of these entries is important. The j2ee.jar file must be the last entry. Here is an example:
c:\WebSphere\AppServer\hosts\default_host\Cics_ejb\servlets\dfjadwas.jar
c:\WebSphere\AppServer\hosts\default_host\Cics_ejb\servlets\xml4j.jar
c:\Program Files\IBM\CICS TS 2.1 Tools\Common\log.jar
c:\Program Files\IBM\CICS TS 2.1 Tools\Common\j2ee.jar

Click the Apply button to include the added jar files to the Classpath in Use frame.

Adding the CICS development deployment servlet to the web application

This section describes the procedure for installing the development deployment tool’s own servlet into the WebSphere Application Server.

Step 1

Start the WebSphere Application Server Administrative Console, open the Console/Tasks menu and choose the Add a Servlet task. Alternatively, click the Wizards button on the tool bar and choose Add a servlet. A dialog box is displayed, which gives you the choice of selecting an existing JAR file or a directory containing servlet classes.

Step 2

Click the option Yes, which indicates that you want to select an existing servlet JAR file, then click the Next button.

Step 3

Expand the topology tree and select the CICS Dev Deploy Tool web application in which to add the servlet. Click the Next button.

Step 4

Click the Browse button on the displayed dialog box. Set the file type to .jar and locate the dfjadwas.jar file in the servlets directory, which is under the Cics_ejb directory, and select it. Click the Select button to close the dialog box. The File Selected path should look like the following example:
c:\WebSphere\AppServer\hosts\default_host\Cics_ejb\servlets\dfjadwas.jar

---

5. The j2ee.jar file is needed because of the connection between the EJB 1.0 client and the EJB 1.1 server. This would not be required if both client and server were at the EJB 1.1 level.
Click the **Next** button.

**Step 5**

In the displayed dialog box, select the **Create User-Defined Servlet** option. Complete the displayed dialog box as follows:

1. In the **Servlet Name** field enter a name by which the servlet can be identified in the WebSphere Application Server Administrative Console and log files, for example, `CicsEjbAd`.
2. In the **Web Application** field select CICS Dev Deploy Tool, and optionally, enter a description.
3. In the **Servlet Class Name** field, enter:
   `com.ibm.cics.addeploy.servlet.CicsEjbAdServlet`

Click the **Add** button.

**Step 6**

In the displayed dialog box, append a unique name to the web application’s web path; for example, `CicsEjbAd`. This is the web path used to invoke the web application when using the tool. Click **OK** to accept the entry. The servlet web path is displayed in the **Servlet Web Path** List, for example:

`default_host/Cics_ejb/CicsEjbAd`

The name of the servlet in this field does not have to match the servlet Name or the servlet class name from step 5. Click the **Next** button to continue.

**Step 7**

If you haven’t yet created a deployment configuration file for this servlet, leave the **Init Parameters** fields blank and move to the next step. Otherwise, in the first **Init Parm Name** cell, enter the parameter name `configDefLoc`. In the corresponding **Init Parm Value** cell, enter the complete file path to the deployment configuration file associated with this servlet, for example:

`C:\WebSphere\Appserver\hosts\default_host\Cics_ejb\dcf\DCF_01.xml`

Set **Debug mode** to **False**. Set **Load at Startup** to **True** if you want the first invocation of the servlet to respond quickly. You can leave it at **False** for now but this delays the first browser request. Click the **Finish** button. A dialog is displayed after a few seconds indicating completion of the Create Servlet command. The new servlet should appear in the topology tree. At this point the servlet is in the stopped state.

**Step 8**

If you haven’t already done so, create a deployment configuration file (DCF) for the servlet. Copy one of the supplied sample files from the dcf directory and rename it to, for example, `DCF_01.xml`. Edit the new DCF file to suit your development environment. See “Configuring the deployment configuration file” on page 162 for guidance on this. In particular, ensure that the `LocalJARBase` and `TraceLogPath` attributes include full paths and file names that reside on the local disks. Set the `ServerName` attribute to the hostname in which CICS is running.

If you didn’t set the `configDefLoc` initialization parameter when adding the servlet, do so now. To set this parameter select the servlet in the topology tree. The **Init Parm** fields are on the **Advanced** tab.

**Step 9**

Copy the `CicsEjbAd.servlet` file from the `Cics_ejb/servlets` directory and...
rename it with the name you gave the servlet in Step 5 (CicsEjbAd.servlet in the example). Put this file back in the CicsEjbAd.servlets directory.

To start the development deployment tool web application and its servlets, right-click its entry in the topology tree and choose Restart Web App.

If you intend to add more than one servlet to your web application, see "Setting up multiple development deployment tool servlets" on page 176 for guidance.

Configuring the deployment configuration file

The deployment configuration file (DCF) provides the means to control the deployment process. This is a task that is best carried out by a CICS system programmer, or at least under the supervision of one, as it requires information about your CICS configuration. Before you start editing this file, you need to understand the basic operation of the development deployment tool, and in particular the way the tool generates CICS resource definitions. See "The CICS development deployment tool for EJB technology" on page 181 for information about these topics.

Each development deployment tool servlet installed in your web application needs to have associated with it a DCF configured to suit the development environment. Once a DCF has been configured and tested it probably won’t need much further editing but you might require several DCFs (and therefore several servlets) configured to test different deployment environments.

DCF files are in XML format and can be edited in a standard text or XML editor. After editing a DCF, you must restart the deployment tool’s web application before any changes can take effect. Two sample DCF files are supplied with the installation of the development deployment tool. Both are located in the dcf folder of the web application on the WebSphere Application Server platform, for example:

C:\WebSphere\Appserver\hosts\default_host\Cics_ejb\dcf

The samples are called DCF_Skeleton.xml and DCF_Sample.xml.

DCF_Skeleton.xml is shown in Figure 23 on page 163. This can be used as a starting point in tailoring your own files. Most of the fields are empty.
DCF_Sample.xml is shown in Figure 24 on page 164. It contains sample data showing support for 4 users, 10 CICS CorbaServers (2 of which point to the same server but have different transaction IDs associated with them), and 2 ResourceMapping elements. Three of the four users have a CorbaServer, which is intended for his or her exclusive use. Preventing users from sharing CorbaServers is generally good practice. It makes beans more easily identifiable and avoids the need to administer a naming convention for beans.
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE DeploymentConfig SYSTEM "DCF.DTD">
<DeploymentConfig>
  <ConfigDefaults MaxJARSize="1000" LocalJARBase="E:/DJARS" AdminContact="John Brown, extension 123455"
MasterTrace="ALL" TraceLogPath="C:/temp/logfile.log" MaxActionWaitPeriod="600"/>
  <OS390Server DeployJarBase="/usr/deployedJARs" ServerName="winmvs2c.hursley.ibm.com"
UserIDIgnoreCase="true" FTPPort="21" NamingServiceURL="iiop://pushmepullyou.hursley.ibm.com:900/"
JNDIPrefix="DFHD"/>
  <CorbaServers>
    <CorbaServer CICSName="CON1" FriendlyName="CorbaServer 1" TransID="TRN1"/>
    <CorbaServer CICSName="CON1" FriendlyName="CorbaServer 2" TransID="TRN2"/>
    <CorbaServer CICSName="CON3" FriendlyName="CorbaServer 3" TransID="TRN3"/>
    <CorbaServer CICSName="CON4" FriendlyName="CorbaServer 4" TransID="TRN4"/>
    <CorbaServer CICSName="CON5" FriendlyName="CorbaServer 5" TransID="TRN5"/>
    <CorbaServer CICSName="CON6" FriendlyName="CorbaServer 6" TransID="TRN6"/>
    <CorbaServer CICSName="CON7" FriendlyName="CorbaServer 7" TransID="TRN7"/>
    <CorbaServer CICSName="CON8" FriendlyName="CorbaServer 8" TransID="CIRP"/>
    <CorbaServer CICSName="CON9" FriendlyName="CorbaServer 9" TransID="CIRP"/>
    <CorbaServer CICSName="TEST" FriendlyName="Test CorbaServer" TransID="TRNT"/>
  </CorbaServers>
  <Users>
    <User Userid="DSMITH" Trace="OFF">
      <CorbaServerRef Name="CON1"/>
    </User>
    <User Userid="MGREEN" Trace="1">
      <CorbaServerRef Name="CON2"/>
    </User>
    <User Userid="SWHITE" Trace="1,2,3">
      <CorbaServerRef Name="TEST"/>
    </User>
    <User Userid="JBROWN" Trace="1,3">
      <CorbaServerRef Name="CON1"/>
      <CorbaServerRef Name="CON3"/>
      <CorbaServerRef Name="CON5"/>
      <CorbaServerRef Name="CON7"/>
      <CorbaServerRef Name="CON9"/>
      <CorbaServerRef Name="TEST"/>
    </User>
  </Users>
  <Bindings>
    <ResourceMapping LogicalName="Resource1" Value="jdbc/Resource1"/>
    <ResourceMapping LogicalName="Resource2" Value="jdbc/Resource2"/>
  </Bindings>
</DeploymentConfig>

Figure 24. Deployment configuration file example

Note: The contents of the DCF is limited to ASCII characters only. The inclusion of any non-ASCII characters results in a parsing error.

The DCF structure must match that specified in the document type definition (DTD) in Figure 25 on page 165, named DCF.DTD. This DTD defines the XML tags that must be used in the DCF, and the order in which they appear. If these rules are violated, parsing and validation fails and the deployment tool cannot start.
The DCF.DTD file should reside in the same directory on the WebSphere Application Server platform as the DCF file itself. Do not change the DTD in any way as this would cause DCF validation to fail.

Table 9 describes the meaning of the XML tags defined in DCF.DTD.

Table 9. DCF.DTD XML tags

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AdminContact</td>
<td>If supplied, this string is used when issuing certain messages to indicate that the system programmer should be contacted.</td>
</tr>
<tr>
<td>Bindings</td>
<td>Element containing one or more ResourceMapping tags.</td>
</tr>
<tr>
<td>CICSName</td>
<td>The name for the CorbaServer in CICS. This must be defined as a CICS resource definition.</td>
</tr>
<tr>
<td>ConfigDefaults</td>
<td>Settings for a number of environmental variables.</td>
</tr>
<tr>
<td>CorbaServer</td>
<td>Details of a CorbaServer into which beans can be deployed.</td>
</tr>
<tr>
<td>CorbaServerRef</td>
<td>Reference of a CorbaServer available for a user to deploy beans into.</td>
</tr>
<tr>
<td>CorbaServers</td>
<td>Element containing the available CICS CorbaServers.</td>
</tr>
<tr>
<td>Field</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>DeployJarBase</td>
<td>Path name of the base directory used to hold the uploaded JAR files on MVS™. This path name is extended by the web application to include additional directories based on the CorbaServer, user ID and applid of the CICS region.</td>
</tr>
<tr>
<td>DeploymentConfig</td>
<td>Required contents of the DCF.</td>
</tr>
<tr>
<td>FriendlyName</td>
<td>The name used when displaying a list of CorbaServers to the user. The friendly name must be unique.</td>
</tr>
<tr>
<td>FTPPort</td>
<td>The FTP port used on the MVS machine for transferring the JAR file to HFS storage.</td>
</tr>
<tr>
<td>JNDIPrefix</td>
<td>Look up string used to find the location of the bean within the naming service. This must match the JNDI prefix in the CICS CORBASERVER definition (DFHD).</td>
</tr>
<tr>
<td>LocalJARBase</td>
<td>Path name of the directory used to hold the uploaded JAR files on the WebSphere machine</td>
</tr>
<tr>
<td>LogicalName</td>
<td>Logical name as found in the DD. This is the ejb-ref name or resource-ref name.</td>
</tr>
<tr>
<td>MasterTrace</td>
<td>Master trace flag controlling how much information is traced by the web application. This can be OFF, 1, 2, 3, ALL or a combination of 1,2 and 3. 3 provides the greatest amount of trace detail and 1 the least. MasterTrace also controls the permitted settings of the Trace tag. Use trace sparingly and ensure that trace logs are deleted periodically to avoid running out of disk space.</td>
</tr>
<tr>
<td>MaxActionWaitPeriod</td>
<td>Period in seconds after which the web application stops the current action and times out. Once the web application has passed control to the bean component, it will not time out until a response is received. A value of 0 indicates that the web application should not time out.</td>
</tr>
<tr>
<td>MaxJARSize</td>
<td>Maximum size of an uploaded JAR file, in kilobytes. A value of zero removes any size restriction.</td>
</tr>
<tr>
<td>Name</td>
<td>CICSName of the CorbaServer as defined in a CorbaServer element.</td>
</tr>
<tr>
<td>NamingServiceURL</td>
<td>URL of the nameserver used to look up the location of the bean used by this tool. This must be the nameserver used when the bean is published.</td>
</tr>
<tr>
<td>OS390Server</td>
<td>Settings of the OS/390 server to be used.</td>
</tr>
<tr>
<td>ResourceMapping</td>
<td>Element containing LogicalName and Value used in binding or mapping an ejb-ref or a resource-ref name to a JNDI string where one does not exist. See [How the CICS development deployment tool resolves bindings](page 184) for more information about binding.</td>
</tr>
<tr>
<td>ServerName</td>
<td>DNS registered name of the MVS machine in which the CICS region is running.</td>
</tr>
<tr>
<td>Trace</td>
<td>User trace flag. This can be OFF, 1, 2, 3, ALL or a combination of 1,2 and 3. 3 provides the greatest amount of trace detail and 1 the least. User trace is produced only if the MasterTrace flag is set to the same value.</td>
</tr>
<tr>
<td>TraceLogPath</td>
<td>Path and base file name of the generated trace log files. Each generated log file uses the base file name with an added numeric identifier, for example devdeploy1.log, devdeploy2.log and so on.</td>
</tr>
</tbody>
</table>
### Table 9. DCF.DTD XML tags (continued)

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TransID</td>
<td>The CICS transaction ID to be used on any request models generated for a bean deployed into the CorbaServer. This can be CIRP (the default TransID) or any other transaction ID defined on CICS to run the DFHIIRP program. See Request model generation and the CICS development deployment tool on page 183 for more information about transaction IDs and request model generation.</td>
</tr>
<tr>
<td>User</td>
<td>Element containing user information.</td>
</tr>
<tr>
<td>Userid</td>
<td>MVS user ID used when FTPing the JAR file to the MVS machine. If applicable, the case should match.</td>
</tr>
<tr>
<td>UserIDIgnoreCase</td>
<td>Determines whether or not the case of the user ID entered by the application programmer is used when matching with the user ID information in the DCF.</td>
</tr>
<tr>
<td>Users</td>
<td>Element containing the list of users able to use the tool.</td>
</tr>
<tr>
<td>Value</td>
<td>The JNDI string used to resolve a logical name.</td>
</tr>
</tbody>
</table>

### Setting up the CICS components of the CICS development deployment tool

To set up the CICS unit test environment used to run enterprise beans you need to configure a CICS EJB server. This is described in Chapter 13. Setting up an EJB server on page 113. You should set up all of the CorbaServers and transaction IDs that are specified in the deployment configuration file of your WebSphere servlet. In order to keep track of deployed beans it is suggested that each user of the development deployment tool be allocated separate Corbaservers. Alternatively, a naming convention for bean names should be established.

CICS definitions for the development deployment tool are in 3 groups:

**DFHADPD**

This is the program definition group. It contains definitions to the required CICS programs and transactions. See CICS Resource Definition Guide for guidance on how to install program definitions.

**DFHADFD**

This is the file definition group. It contains the DFHADJM file. DFHADJM is a file-control-managed VSAM key-sequenced data set (KSDS) used by the tool to store mappings between CICS DJARs and JAR files on HFS. This group is unlocked so you can redefine the DSName of the file. Alternatively, this can be defined by a deployment descriptor card in the JCL in the same way as other CICS data sets. DFHADJM should not be shared across regions. See CICS Resource Definition Guide for guidance on how to install file definitions. See CICS System Definition Guide for guidance and sample JCL for defining the DFHADJM data set.

**DFHADBD**

This is the bean definition group. It contains the bean runtime requirements. It is not locked to allow you to redefine the definitions. It contains:

**DFHADTCP**

This is the TCP/IP service used to receive requests for the bean component of the tool. You can alter the port number as required, but it must match the port number of CorbaServer DFHD. You can also specify the IIOP security user replaceable module (URM).
URM can be used to set the CICS user ID to be used for the execution of the deployment tool. See CICS Resource Definition Guide for more guidance.

DFHD This is the CorbaServer used to hold the DJAR containing the bean component. You need to alter the host name of the MVS system on which the CICS is running. Also, ensure that the port number matches that defined in TCP/IP service DFHADTCP. You can alter the JNDI prefix, but if you do this, it must match the JNDI prefix stated in the JNDIPrefix element of the DCF. You can also change the shelf. See CICS Resource Definition Guide for more guidance.

DFHADJAR This is the DJAR definition of the jar containing the bean component. The HFS file defaults to match the default installation of the UNIX System Services component of CICS. If the dfjadmv<tab>s.jar file is not located in the default location, adjust the HFS file to match. See CICS Resource Definition Guide for guidance on how to install DJAR definitions.

DFHADR M This is the request model definition for the CICSDDT bean. One request model is used for all operations. You should not need to alter this unless you alter the CorbaServer used to hold the DJAR containing the bean component. See CICS Resource Definition Guide for guidance on how to install request model definitions.

These groups are not included in the default CICS startup group list, DFHLIST, so they are not automatically installed on start up. Be aware that as the definitions are in a DFH group, any DFHCSDUP upgrades will overwrite any updated information with the supplied defaults. To avoid this possibility copy DFHADB D and DFHADF D to another group before you edit them. Once the definitions have been tailored, edit your start up JCL to install this group list along with any existing lists.

Once the tool has been installed, the bean needs to be published to the JNDI name server. Before you do this make sure that you have specified the nameserver on the NamingServiceURL field in the DCF. You need to publish the bean only once during the lifetime of the JNDI using the command CECI PERFORM DJAR(DFHADJAR) PUBLISH. This is not done automatically as part of the CICS start up procedure. See CICS System Programming Reference for more guidance on using this command.

Setting up user IDs and security for the development deployment tool
All deployment operations relating to a CICS region use a single predetermined user ID. This is achieved using a user replaceable module (URM) that can be defined on the TCP/IP service definition related to the enterprise bean’s specified CorbaServer. See CICS Resource Definition Guide for more guidance. If the URM is not included on the TCP/IP service definition, the CICS default user ID is used

The supplied sample URM is called DFHADURM. DFHADURM sets the CICS user ID to ADTEST. You can modify this sample URM to set the CICS user ID to one appropriate for your installation. The user ID you specify must be a valid CICS user ID with appropriate security authorization. This means that if the CICS system is started with SEC=YES and other security check options (for example, XCMD=YES and/or XFCT=YES), you need to define the user ID with the appropriate authorization for the tool to execute. The tool performs the following functions on CICS resources.
You should set the security authorization for the user ID accordingly.

A further level of security is provided by the development deployment web application. Application developers using the development deployment tool are required to have an entry in the DCF matching an MVS user ID. This entry specifies the CorbaServers that can be used for deployment by that user.

To set up these MVS user IDs you need to do the following:

• Set up a RACF group and add a GID (group ID) to give full access to UNIX System Services.
• Set up the CICS user ID and add it to the RACF group so that the required HFS directories can be created by the tool. See CICS Transaction Server for z/OS Installation Guide for guidance on this.
• Set up MVS user IDs for all users who have access to the CICS region for the purpose of deploying and running enterprise beans and add them to the RACF group.
• Ensure that FTP services are available on OS/390 UNIX System Services. See the OS/390 IBM CS IP User’s Guide, GC31–8514 for information on MVS FTP services.

The system administrator can control whether or not a particular connection should use SSL certificates. Use of SSL certificates is recommended in order to prevent user ID and password information being sent over an insecure connection. See "Chapter 21. Managing security for enterprise beans" on page 227 for more information about SSL certificates.

### Setting up more complex development deployment tool configurations

This section gives some guidance about setting up configurations involving several development deployment tool servlets or more than one CICS region, or both.

You may decide to configure several development deployment tool servlets in order to use different deployment configurations as defined in each servlet’s DCF.
Furthermore, if you intend to deploy beans to more than one CICS system, you have to set up at least one servlet per region. The basic ground rules are as follows:

**Multiple servlet configurations, same CICS region**
- Always create a separate DCF for each servlet. DCFs must not be shared.
- The DCF for each servlet must contain a NamingServiceURL and JNDIPrefix that resolve to the same CICSDDTbean.
- A particular user ID should use only one servlet (in order to prevent the possibility of ejb-jar files accidentally overwriting each other).

**Multiple servlet configurations, several CICS regions in an MVS image**
- You have to set up at least one servlet for each CICS region.
- If a user ID appears in more than one DCF make sure that the associated servlets access CICSDTbeans in different CICS regions.
- Always create a separate DCF for each servlet. DCFs must not be shared.
- The DCF for each servlet must contain NamingServiceURL and JNDIPrefix elements that allow the correct CICSDTbean to be located:
  - Each CICSDTbean published to the same JNDI nameserver must have a unique JNDI prefix in the CORBASERVER DFHD definition, which must match the JNDIPrefix element in the DCF.
  - The NamingServiceURL element in the DCF should specify the same name server as that used when publishing the bean.
- In your CICS resource definitions, the TCPIPSERVICE (DFHADTCP) and CORBASERVER (DFHD) port numbers need to be unique for each region.
- The DFHADJM file must not be shared across CICS regions. The CICSDTbean must be in the same CICS region as the associated DFHADJM file.

**Setting up multiple development deployment tool servlets**
The procedure for adding additional servlets is much the same as that described in "Adding the CICS development deployment servlet to the web application" on page 160. The servlet class is always the same (com.ibm.cics.addeploy.servlet.CicsEjbAdServlet) but the following must be unique for each servlet:
- Name
- Deployment Configuration File (DCF) containing appropriate TraceLogPath, LocalJARBase, DeployJarBase, DeployBeanRef, CorbaServers, Bindings and User values
- A configDefLoc initialization parameter pointing to the specified DCF
- A web path that matches the servlet’s name
- A page list configuration file. This can be a renamed copy of the CicsEjbAd.servlet file.

**Installing and configuring the CICS production deployment tool for EJB technology**
The workstation and WebSphere components of the deployment tools are supplied on a CD-ROM as a set of InstallShield packages.
Use the InstallShield Wizard application to install on your workstation the files for the CICS production deployment tool. Select **Complete** to install all the CICS deployment tools or **Custom** to control which of the tools and features to install on your workstation. Just follow the prompts to complete the installation.

During installation the InstallShield Wizard checks for required levels of some Java software. In particular you need to have installed IBM Developer Kit for Windows, Java 2 Technology Edition, Version 1.3, at at Service Release 6 (or later), plus the EJB 1.1 standard interface classes, provided in `javax.ejb.zip` and `j2ee.jar`. (The IBM Developer Kit for Windows, Java 2 Technology Edition, Version 1.3 at Service Release 6, is supplied with CICS TS on a CD-ROM labelled CICS Tools for EJB Technology, LCD4-4355.)

If the InstallShield Wizard finds that these required levels are not installed, it presents you with a check-box option to install them after the deployment tools installation process is finished. Mark the check-box to indicate that you would like the install process for IBM Developer Kit to be started automatically when the current install process (for the deployment tools) completes. There is no other configuration activity associated with the installation of the CICS production deployment tool.
Chapter 18. Deploying enterprise beans

The concept of deployment is introduced in "Deploying enterprise beans—overview" on page 101. This section explains the process of deploying enterprise beans into the CICS EJB server in more practical terms.

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, the CICS EJB server).

Some of the steps involved might be regarded as application programming tasks (for example, code generation), and some as systems programming tasks (for example the modification of runtime options). Here, that distinction is ignored, and they are considered as steps in the deployment task.

Deployment begins with a JAR file that contains Java classes and can contain one or more enterprise beans. Typically a JAR file contains many enterprise beans. The steps in the deployment process are as follows:

1. **Step 1**
   Preparing an EJB 1.1 deployment descriptor in the JAR file.

2. **Step 2**
   Adding generated code and CICS-specific bindings.

3. **Step 3**
   Producing CICS resource definitions (suitable for a CICS CSD, or for a CICSPlex SM data repository, or suitable for the CICS CREATE command to install).

4. **Step 4**
   Making the enterprise beans accessible to CICS by storing the ejb-jar file in the HFS repository on OS/390.

5. **Step 5**
   Publishing the CorbaServer name and enterprise bean names to an external namespace using JNDI.

At the end of this process the ejb-jar file is deployed to the CICS system, and available to be used.

---

**Using CICS deployment tools for EJB technology**

CICS provides a set of tools to facilitate the deployment process. These are illustrated in Figure 26 on page 174.
The starting point is a JAR file containing one or more enterprise beans with or without a deployment descriptor. If you are using an integrated development environment, you can generate a deployment descriptor automatically. VisualAge for Java Enterprise Edition version 3.5, for example, incorporates an EJB development environment that can create ejb-jar files with EJB version 1.0 deployment descriptors.

The first part of the deployment process is the preparation of the deployment descriptor. The CICS JAR development tool for EJB technology is an editing tool designed to help you do this. It provides a GUI interface that enables you to create or edit the ejb-jar file’s deployment descriptor together with some optional CICS specific customizations. It also converts EJB 1.0 serialized deployment descriptors into the EJB 1.1 XML equivalents required for the next stage of deployment.

The next stage is code generation. Incorporated into the CICS JAR development tool is the CICS code generation utility for EJB technology. This tool automatically generates the code required to tailor a generic enterprise bean definition into one that can run in an EJB server. This code includes the CORBA stubs and ties needed for RMI/IIOP communication. It can optionally produce a ClientJAR file, which contains only the home and remote interfaces for the
enterprise beans in the JAR file, together with their RMIC stub classes. This enables client applications to install only this smaller ejb-jar file instead of an instance of the whole output ejb-jar file.

The CICS code generation utility can also be run separately from an MS-DOS command prompt window and can be used in a batch process. The result of this process is one or more EJB 1.1 JAR files.

The final stages of deployment involve the creation of CICS resource definitions, publishing bean references to an external namespace, and making the ejb-jar file accessible to CICS. There is a choice of tools for these operations.

- **The CICS development deployment tool for EJB technology** provides a route that simplifies the creation of the CICS definitions in order for application programmers with a minimum of CICS expertise to test enterprise beans in a CICS test environment. The tool stores the ejb-jar file in HFS on OS/390 and creates a set of generic CICS resource definitions using EXEC CICS CREATE commands. It does not update the CICS CSD, so the definitions are lost if the CICS region is restarted. CICS resource definition information is stored in the ejb-jar file for possible later reuse by this or the alternative deployment tool. It also uses the CICS API to store a reference to each enterprise bean in an external namespace, for example, on the WebSphere Application Server.

- An alternative is the **CICS production deployment tool for EJB technology**. This tool enables you to specify in detail the necessary CICS resource definitions. This tool can create ejb-jar files ready to store on HFS. It also produces a DFHCSDUP input stream that you can use to define the necessary CICS resource definitions on the CICS CSD or a BATCHREP input stream to create definitions on the CICSplex SM data repository. Like the CICS development deployment Tool it stores CICS resource definition information in the ejb-jar file for use in a later deployment process. You can run this tool either via a GUI on your workstation, or as an offline utility. The CICS production deployment tool does not apply the CICS resource definitions that it generates, nor does it publish a reference to each enterprise bean in an external namespace on the WebSphere Application Server. It can place the ejb-jar file in HFS, where it is available to CICS.

- Or, the necessary tasks can be performed without the assistance that these two tools offer.

Enterprise bean development might involve successive uses of the CICS development deployment tool and the CICS production deployment tool. JAR files created using the CICS development deployment tool and run on a CICS test region can be passed to the CICS JAR development tool or the CICS production deployment tool for refinement before deployment to a CICS production environment. It is also possible to pass ejb-jar files from the CICS production deployment tool to the CICS JAR development tool or the CICS development deployment tool for modification or further testing. Both deployment tools can reuse CICS resource definitions stored in the ejb-jar file in previous sessions.

---

**The CICS JAR development tool for EJB technology**

The CICS JAR development tool is a Java-based GUI editor that runs on your workstation. It is used to edit an ejb-jar file containing one or more enterprise beans in order to prepare a deployment descriptor. This tool can also invoke the CICS Code Generation Utility, which generates the additional code required for deployment in the CICS EJB server.
For each enterprise bean, the ejb-jar file must include the following:

- The enterprise bean implementation
- The remote interface
- The home interface

The ejb-jar file can also contain the class files for all the classes and interfaces that the enterprise bean class, and the remote and home interfaces depend on. This includes their super-classes and super-interfaces, and the classes and interfaces used as method parameters, results, and exceptions. These class files can be made available from another ejb-jar file, standard jar file or directory structure, using the CLASSPATH system variable.

Before the ejb-jar file can be deployed to the CICS EJB server it must also contain the deployment descriptor, bindings for any resources and references contained in the deployment descriptor, and all the generated code required for RMI/IIOP and EJB operation. You can use the CICS JAR development tool to create this extra information.

The deployment descriptor

The deployment descriptor is an important feature of the EJB specification. It allows the declarative specification of run time attributes for the enterprise beans avoiding the need to code the equivalent information into the beans themselves. This means that you can change these values at deploy time without having to rewrite and compile the enterprise bean classes.

In the EJB version 1.0 specification a deployment descriptor consists of a number of serialized Java objects (one per enterprise bean) stored in the ejb-jar file. In EJB 1.1, a deployment descriptor is a single XML document containing the deployment data for all the enterprise beans in the ejb-jar file. The CICS JAR development tool creates only EJB 1.1 style deployment descriptors but it can read either style. You can use it to create or edit deployment descriptors, and to convert EJB 1.0 deployment descriptors to the EJB 1.1 equivalent.

A deployment descriptor contains two kinds of information:

- Structural information that describes the structure of an enterprise bean and declares the bean’s external dependencies.
- Application assembly information that describes how the enterprise bean (or beans) in the ejb-jar file is composed into a larger application deployment unit.

You should not normally change structural information because doing so could alter the enterprise bean’s function. You can change assembly level information without altering the enterprise bean’s function, although doing so may alter the behavior of an assembled application.

Code generation

Code generation is the process of creating in the ejb-jar file the following generated code:

- RMI/IIOP stub classes based on the home and remote interfaces within the source ejb-jar file. These interfaces allow enterprise beans to be invoked remotely.
- Additional classes used by the container to honor the information stored in the deployment descriptor including the classes that implement the bean’s home and remote interfaces.
- The communication stubs used by the client.
This additional EJB and RMI wrapper and stub code is needed at runtime.

Code generation is carried out by the CICS code generation utility. The utility opens an ejb-jar file, verifies each Enterprise bean in the file, then creates the additional EJB and RMI classes for each Enterprise bean. It also excludes entity beans from the output ejb-jar file.

There are two ways to invoke this utility:
- From the CICS JAR development tool GUI using the Generate command on the File menu.
  See "Using the CICS code generation utility for EJB technology from the CICS JAR development tool GUI" on page 179 for more information about using the CICS code generation utility from the CICS JAR development tool GUI.
- Directly from an MS-DOS command prompt window, or using the Run option from the Windows START menu.
  See "Using the CICS code generation utility for EJB technology as a stand-alone tool" on page 179 for more information about using CICS code generation utility as a stand-alone tool.

Using the CICS JAR development tool for EJB technology

To invoke the CICS JAR development tool, use the menu option of the Windows NT (or Windows/2000) Start menu, Programs, IBM CICS TS 2.1 Tools, CICS Jar Development Tool for EJB Technology.

Alternatively the CICS JAR development tool may be started from an MS-DOS command prompt window using the command:

```
CICSJDT [filename.jar [-auto]] [-font(xx)]
```

where:

**CICSJDT**

is the batch file to be run. Before using the CICSJDT command, ensure that the directory which contains CICSJDT.BAT file is available on your system’s PATH statement. It is normally found in C:\Program Files\IBM\CICS TS 2.1 Tools\\Jar Development Tool\bin

**[filename.jar]**

is your input JAR file. If you do not specify it on the command line you can specify it in the GUI.

**[-auto]**

indicates that the CICS JAR development tool is to load and then immediately save the specified JAR file. This can be useful for converting any EJB 1.0 deployment descriptors in the jar file to an EJB 1.1 XML deployment descriptor without user interaction. The parameter is optional, but only available if you specify an input JAR file. It is only relevant if the CICS JAR development tool is run from the command prompt.

**[-font(xx)]**

indicates the font size which is to be used by the GUI instead of the default size. xx is the desired font size, the minimum value permitted is 10. The parameter is optional. It is only available if the CICS JAR development tool is run from the command prompt.
The GUI

The GUI main panel has options to load and save ejb-jar files, and it displays a scrollable list of the enterprise beans in the current loaded ejb-jar file. Each enterprise bean is identified by its bean name. The panel has buttons providing the following operations:

- **New**: Create a new enterprise bean entry in the deployment descriptor, with default properties.
- **Copy**: Create a new enterprise bean entry in the deployment descriptor, with properties copied from another bean in the jar file.
- **Edit**: Edit the properties of an enterprise bean entry via its deployment descriptor.
- **Delete**: Delete a deployment descriptor.
- **JAR Details**: Edit JAR file level attributes in the deployment descriptor.
- **Bindings**: Show a panel that allows a JNDI binding to be specified for any enterprise bean reference or resource in the deployment descriptor.
- **CICS Options**: Show a panel that allows you to specify a CICS TRANSID for any part of the deployment descriptor that requires a REQUESTMODEL resource definition later in the deployment process.

The panel also has a message and status line, with a button that opens a display of the status log showing a scrollable message history. This can be saved to a file if required.

**the Deployment descriptor**

The principal properties of an EJB Deployment Descriptor which are supported are:

- `<ejb-jar>`: Enterprise-bean descriptor
- `<description>`: of the JAR file
- `<enterprise-beans>`
- `<session>`
  - `<description>`: of the bean
  - `<ejb-name>`
- **Classnames**: (home interface, remote interface, enterprise bean class)
- `<session-type>`: (STATELESS, STATEFUL)
- `<transaction-type>`: (BEAN_MANAGED, CONTAINER_MANAGED)
- `<env-entry>`: (name/type/value)
- `<ejb-ref>`: dependencies on other enterprise beans (name/type/home/remote)
- `<resource-ref>`: External resources required (description/name/type/authority)
- `<assembly-descriptor>`
The full details are available in the *Deployment descriptor* chapter of Sun Microsystems’s *Enterprise JavaBeans™ Specification, Version 1.1*, which is available at [http://www.javasoft.com/products/ejb](http://www.javasoft.com/products/ejb). Remember however, that CICS TS 2.1 provides partial support for Sun Microsystems Enterprise JavaBeans (EJB) architecture, Version 1.1, see [Chapter 12. What are enterprise beans?](#) on page 87.

---

### Using the CICS code generation utility for EJB technology from the CICS JAR development tool GUI

You can invoke the CICS code generation utility from the CICS JAR development tool GUI using the **Generate** command on the **File** menu.

When invoked from the CICS JAR development tool, the **inputjarfile** value is supplied by the CICS JAR development tool. The utility enables you to specify the **outputjarfile**, and a **clientjarfile**. It uses a **workingdir** (working directory) although you do not have an opportunity to specify the name of the working directory. You can specify the **verbose** and **keep** options if required. The remaining options, that are available to the user of the stand-alone tool, take their default values.

All the inputs to the CICS code generation utility are described in [Using the CICS code generation utility for EJB technology as a stand-alone tool](#). They have the same meaning for both methods of invoking the utility.

The tool issues error messages if it encounters entity beans in the ejb-jar file, but it continues to process any session beans.

### Using the CICS code generation utility for EJB technology as a stand-alone tool

You can run the CICS code generation utility from an MS-DOS command-prompt window on your workstation, or from the run option on the Windows start menu.

In either case, the command to start the CICS code generation utility as a stand-alone tool is:

```
CICSCGU inputjarfile [workingdir] [outputjarfile] [clientjarfile] [options]
```

where:

- **CICSCGU** is the batch file to be run. Before using the CICSCGU command, ensure that the directory which contains CICSCGU.BAT file is available on your system’s PATH statement. It is normally found in `C:\Program Files\IBM\CICS TS 2.1 Tools\JAR Development Tool\bin`
- **inputjarfile** is the filename of the input ejb-jar file.
[workingdir] is the directory name for temporary storage of extracted files and generated classes. It is not required if the -analyze option is selected.

[outputjarfile] is the filename of the output ejb-jar file, which contains the additional code generated by this tool. It is not required if the -analyze option is selected.

[clientjarfile] is the filename of an output ejb-jar file, which contains only the home and remote interfaces for the enterprise beans in the JAR file, together with their RMIC stub classes. It is not required if the -analyze option is selected, and is only required if you want to produce a ClientJAR file.

The ability to produce a ClientJAR file means that client applications no longer need to have the whole output ejb-jar file installed on their client machines with their client applications, just this smaller ejb-jar file.

The ClientJAR file needs no further processing by the CICS deployment tools for EJB technology. It is ready to be placed in the client according to the requirements of the client environment.

[Options]

-analyze only verify Enterprise beans in input jar file, do not generate additional classes. If you specify -analyze, [workingdir] [outputjarfile] and [clientjarfile] are not required.

Default action is: Both analyze and generate classes.

-codegen only generate EJB server implementation classes, do not verify Enterprise beans

Default action is: Both analyze and generate classes.

-force ignore verification errors

Default action is: On finding errors, issues messages and does not proceed with code generation.

-J<suboption> pass suboptions to the JVM which is used to invoke the RMIC compiler (for example: —Jmx128M, to set maximum heap size)

Default action is: No additional options are passed to the JVM.

-keep preserve the working directory and generated files

Default action is: Delete the working directory and contents.

-nocompress do not compress the deployed jar file

Default action is: Compress the JAR file.

-noRMIC do not generate RMIC stubs or ties

Default action is: Generate the RMIC stub and ties.
The tool issues error messages if it encounters entity beans in the ejb-jar file, but it continues to process any session beans.

The CICS development deployment tool for EJB technology

The CICS development deployment tool provides a method of deploying a 1.1 ejb-jar file into CICS directly from a development workstation without the need to define detailed CICS resource definitions. The purpose of this is to allow EJB application developers with minimal or no CICS knowledge to deploy enterprise beans into a CICS unit test environment. It is designed as an alternative to the CICS production deployment tool but is not appropriate for deploying enterprise beans into a CICS production environment.

The interface between the application programmer and this tool is a standard web browser. The application programmer enters user ID and password information, the name and location of the 1.1 ejb-jar file to be deployed, and selects the CICS CorbaServer into which the enterprise beans are to be deployed.

The system programmer maintains control of the environment by means of the deployment configuration file (DCF). This is an XML file which specifies user and CorbaServer information as well as a number of other parameters that relate to the operation of the tool.

In operation the tool starts by initializing a web application running on the WebSphere Application Server. The web application parses the DCF to load and validate information specified by the system programmer. Once this is successful, the application developer can connect to WebSphere to communicate with the web application. Information entered by the application developer is validated and the enterprise beans are uploaded to the WebSphere platform.

The web application extracts from the DCF the transaction ID associated with the selected CorbaServer. The binding information is extracted from the ejb-jar file together with any CICS DJAR and REQUESTMODEL resource definition information that might be present. If there is no CICS resource definition information in the ejb-jar file, the tool creates it using default values and then stores it in the ejb-jar file for use in subsequent deployment sessions. See "DJAR generation and the CICS development deployment tool" on page 183 and "REQUESTMODEL generation and the CICS development deployment tool" on page 183 for details about the generation of resource definitions. See "How the CICS development deployment tool resolves bindings" on page 184 for more information about binding.

The web application uses RMI to pass all of this information, along with the HFS name, to the CICS component of the CICS development deployment tool; the CICSDDTBean session bean. The ejb-jar file is transferred to HFS storage on OS/390 using FTP.

DJAR naming is handled by the CICS application, DFHADJAR, which is called using the JCICS API. The CICS applications DFHADSTR and DFHADINS carry out resource definition and REQUESTMODEL naming using the EXEC CICS CREATE API. The results of these CREATE commands are returned to CICSDDTBean and then passed back to the web application where they are displayed for the application developer on the web user interface. CICSDDTBean also stores a copy of the successful CICS resource definitions in the copy of the ejb-jar file on OS/390.
This deployment process is summarized in Figure 27.

DJAR generation and the CICS development deployment tool

The CICS development deployment tool generates a DJAR resource definition to represent each JAR file that is deployed using the tool. The generated DJAR definition has the following attributes:

**NAME** is generated by the tool unless either the CICS development deployment tool or the CICS production deployment tool has already been used to deploy the JAR file, in which case the existing name is reused.

**CORBASERVER** is found in the DCF and relates to the CorbaServer selection made by the user.

**HFSFILE** is generated by the tool based on the path of the uploaded JAR file on the server. It is defined by:

- The DeployJarBase setting in the DCF
- The APPLID of the CICS region
- The user ID
- The CORBASERVER name
- The name of the JAR file.

See Deployed JAR file resource definitions for more information about DJAR resource definitions.
Request model generation and the CICS development deployment tool

The CICS development deployment tool uses any user-defined request models already created with the CICS Production Deployment Tool. If there are none, the tool can generate the required request models based on information described in the deployment descriptor elements and on the DCF. However if the transID associated with a CorbaServers defined in the DCF is the default CIRP transID, the tool does not need to generate a request model. Instead it uses the existing defaults.

The data found in any existing user-defined request model is used as is, except for the CorbaServer. This is replaced with the CorbaServer found in the DCF, which relates to the CorbaServer selection made by the user.

The CICS development deployment tool generates request models only for session beans; that is beans defined within session elements.

Request models generated for bean-managed session beans are slightly different from those generated for container-managed session beans:

- If a session bean is bean-managed, that is it has a transaction type of Bean, the tool generates a request model for each session bean element.
- If a session bean is container-managed; that is it has a transaction-type of Container, no request model is generated for the session bean element, but request models are generated according to the information found in the container-transaction elements in the application-descriptor section of the deployment descriptor. The tool generates a request model for each of the container-transaction elements that have method elements relating to a session bean (as indicated by the ejb-name tag).

The tool generates a request model with the following attributes:

**NAME** This is generated by the tool.

**CORBASERVER** This is found in the DCF and relates to the CorbaServer selection made by the user.

**TYPE** This is always EJB.

**BEANNAME** This is taken from the ejb-name of the session bean.

**INTFACETYPE** This is always REMOTE.

**OPERATION**
- If the bean is bean-managed, OPERATION is always an asterisk (*).
- If the bean is container-managed, OPERATION is derived from the method-name and the method-params. The OPERATION is an asterisk (*) if the method-name is an asterisk, otherwise the OPERATION is the same as the mapped IDL version of method-name. If there are any method-params elements, the IDL mapping of these elements are appended to the end of OPERATION. If no method-params tags exist, it is assumed that the method element is describing an overloaded method and an asterisk is appended to the end of OPERATION.

**TRANSID** is either found in the DCF and relates to the CorbaServer selection made by the user, or it is defined using the CICS Jar Development Tool.
See [CICS Resource Definition Guide](#) for more information about defining request models.

**How the CICS development deployment tool resolves bindings**

This section describes how the CICS development deployment tool handles references from an enterprise bean to another enterprise bean or to another resource such as a database. This involves mapping the reference to a JNDI string. It is called binding.

References to other enterprise beans are specified using the `ejb-ref` element in the deployment descriptor. References to resources are specified with the `resource-ref` element. These two types of references are handled somewhat differently.

The tool ignores any bindings that have already been resolved. It then resolves the remaining bindings in the following way:

**ejb-ref binding**

- It first attempts a link, checking if the referenced bean matches another session bean in the same `ejb-jar` file.
- If the link fails, the tool uses any matching `ResourceMapping` elements in the DCF file. See [Configuring the deployment configuration file](#) on page 162 for guidance on specifying ResourceMapping in the DCF.
- If both the link and ResourceMapping methods fail, the tool searches for a matching reference in other beans in the `ejb-jar` file (that is for bindings with the same `ejb-ref-name`, `home`, and `type` elements). These matching bindings could have been created using the CICS JAR development tool or by previous runs of the CICS development deployment tool.

**resource-ref binding**

- The tool first uses any matching `ResourceMapping` element in the DCF file. See [Configuring the deployment configuration file](#) on page 162 for guidance on specifying ResourceMapping in the DCF.
- If the ResourceMapping method fails, the tool searches for a matching reference in other beans in the `ejb-jar` file. (that is for bindings with the same `res-ref-name`, `res-type`, and `res-auth` elements). These matching bindings could have been created using the CICS JAR development tool or by previous runs of the CICS development deployment tool.

---

**Using the CICS development deployment tool for EJB technology**

Using the CICS development deployment tool is straightforward. However before you can use this tool, make sure that the necessary components have been set up and the deployment configuration file correctly configured. See [Configuring the deployment configuration file](#) on page 162 for guidance on this.

1. To start the tool open your Web browser and:
   a. set the browser so that it does not cache Web pages:
      - For Microsoft Internet Explorer, version 5.00:
        - Click Tools
        - choose Internet Options...
        - in the Temporary Internet Files panel click Settings...
        - select the Every visit to the page button in the Check for newer versions of stored pages panel.
      - For Netscape:
– Click Edit
– choose Preferences...
– select Advanced, then Cache
– Select the every time button in the Document in cache is compared to document on network section.

If you alter this setting for other uses of your browser, you will need to do this each time you use the CICS development deployment tool. If not, you can treat this instruction as a setup instruction and need not repeat it for each use of the tool.

b. Enter the URL of the Web application. (In the example from the deployment tools' setup section “Adding the CICS development deployment servlet to the web application” on page 167, this is http://default_host/Cics_ejb/CicsEjbAd.) This opens the User Login page as shown in Figure 28.

---

Figure 28. The CICS development deployment tool User Login page

---

Enter your user ID and password. This user ID must match the one you use to log on to MVS and must be set up in the deployment configuration file. See “Configuring the deployment configuration file” on page 162 for guidance on how to do this. The user ID must also have access to the required HFS shelf directories. See “Setting up user IDs and security for the development deployment tool” on page 168 for guidance on this.

c. Check the Save details box if you prefer not to enter your user details for every JAR deployment. However this is valid only for the lifetime of the current web browser session. If the current browser session expires, you have to log in again to access the tool.
2. Click **Submit** to proceed. This opens the **Deployment Information** page, which is shown in **Figure 29**.

![Figure 29. The Deployment Information page.](image)

Complete this page as follows:

a. Choose the operation to perform on the selected ejb-jar file by choosing **Deploy** or **Undeploy** from the drop-down menu.

b. Select a target CorbaServer from the list of those available. The selected CorbaServer specifies the run time environment within the CICS EJB server in which the enterprise beans in your selected JAR are to be run. You can deploy multiple instances of an enterprise bean into CICS but you must deploy each instance into a separate CorbaServer. The CorbaServers listed are those that have been specified by your system programmer in the deployment configuration file.

c. If it is not already shown, enter the full path of the ejb-jar file located on your workstation into the **JAR file path** field.

3. Click on the **Deploy** or **Undeploy** button to start the operation.

The deploy operation leads to the creation of the necessary CICS resource definitions. It also publishes the name of the ejb-jar file to a nameserver on WebSphere Application Server. The resource definitions are stored in the ejb-jar file in XMI format for possible later use. The ejb-jar file is uploaded from the client to WebSphere, where it is validated, then transferred by FTP to HFS storage on OS/390.

Undeploy removes the CICS resource definitions associated with a previously deployed ejb-jar file, but it does not remove the ejb-jar file itself from HFS.

Once the operation is complete, the **Deploy Results** page is displayed as shown in **Figure 30 on page 187**.
This page summarizes the result of the deployment or undeployment operation. The button on the page acts as a toggle, allowing you to turn on or off the information relating to any CICS resource definitions created.

If a problem occurs during deployment a message is issued. Messages are color coded — blue for warning, pink for an error. Select the ? icon to the right of the message to display further information about the problem.
The CICS production deployment tool for EJB technology

The CICS production deployment tool is the tool which system administrators use to perform the tasks required to deploy Enterprise JavaBeans in an ejb-jar file into CICS systems.

The CICS production deployment tool requires as input an ejb-jar file that has an EJB 1.1 deployment descriptor. This is sufficient to let you edit the deployment descriptor and the CICS information. If in addition the ejb-jar file also includes the generated code (such as would be added by the CICS JAR development tool and the CICS code generation utility or by the CICS development deployment tool), you will be able to deploy it after using the CICS production deployment tool.

While with the tool you can edit the deployment descriptor and CICS information, you can't actually deploy it unless the jar also includes the generated code. i.e. output from the CICS development deployment tool or the CICS JAR development tool.

When application developers wish to deploy ejb-jar files as part of their testing, it is expected that they will use the CICS development deployment tool.

The CICS production deployment tool can work with a local ejb-jar file (one stored on the workstation where the CICS production deployment tool is executing), or it can establish a connection with HFS on the OS/390 host system. The HFS data can be accessed via FTP, or as a Network drive. It is only possible to work with local and HFS files at the same time and thus move or copy files between the two storage locations when HFS is accessed as a Network drive.

The CICS production deployment tool makes the required changes to the ejb-jar file and is able to produce as output:

- a deployed ejb-jar file, containing all the modifications required before it may be used. If stored in HFS this ejb-jar file is ready to be used.
- a DFHCSDUP input stream to define in the host system CSD the CICS resource definitions relevant to the Enterprise JavaBeans in this ejb-jar file.
- a BATCHREP input stream which may be used in a CICSPlex SM environment to fulfil an equivalent function to the DFHCSDUP input stream.

Each use of the CICS production deployment tool can generate either a DFHCSDUP input stream, or a BATCHREP input stream, or both, as the user chooses.

The BATCHREP output produced by the CICS production deployment tool uses the facilities of CICSPlex SM Business Application Services (BAS) to produce a self-contained set of resource definitions. However, as CICSPlex SM handles these resources in a different way to CICS resource definitions which are stored in the CSD, you are advised to familiarize yourself with the implications of these differences before attempting to install them into your systems. For more information, see the CICSPlex System Manager Concepts and Planning manual.

Note: It is possible to define the required resources to CICS without the use of the CICS production deployment tool, but you are recommended to make use of the tool's ability to generate appropriate resource definition commands.

The CICS production deployment tool is provided in two forms:

- As a Graphical User Interface (GUI) enabled as a Java and XML application.
Using the CICS production deployment tool for EJB technology

The CICS production deployment tool may be used as a GUI, see "The CICS production deployment tool as a GUI" or as an offline utility, see "The CICS production deployment tool as an offline utility" on page 193.

The CICS production deployment tool as a GUI

To run the CICS production deployment tool as a GUI on your workstation:

1. Install the tool on your workstation, as described in the "Installing and configuring the CICS production deployment tool for EJB technology" on page 170.

2. Then,
   - Use the menu option of the Windows NT (or Windows/2000) Start menu, Programs, IBM CICS TS 2.1 Tools, Production Deployment Tool for EJB Technology.
   - Alternatively the CICS production deployment tool may be started from an MS-DOS command prompt window using the command:
     
     EYUDEPLOY

     where:

     EYUDEPLOY

     is the batch file to be run. Before using the EYUDEPLOY command, ensure that the directory which contains EYUDEPLOY.BAT file is available on your system’s PATH statement. It is normally found in C:\Program Files\IBM\CICS TS 2.1 Tools\Production Deployment Tool

3. Two windows open on your workstation:
   - A Console window that displays startup and shutdown of the GUI window.
     On starting it displays:

     Updating current environment.
     CICS Production Deployment Tool for EJB Technology Starting.

     No user action is required in this window, except on closing the GUI window. The console window displays:

     Shutdown in progress, please wait...
     CICS Production Deployment Tool for EJB Technology Completed. Restoring original Environment. Press any key to continue . . .

     and you should then press any key to close this console window.
   - The GUI window.

4. The first panel, see Figure 31 on page 190, of the GUI has General information about the tool, only a little of which is shown by the screen image included here.
   - The full text of this introductory information is given in Figure 32 on page 191. Follow the guidance in the Getting started and Navigation topics to navigate through the application.
   - The fully expanded navigation panel is shown in Figure 33 on page 192 topics that are not available to you are greyed-out until you take whatever prerequisite action is needed.
Figure 31. The first panel of the CICS production deployment tool GUI
Introduction

The deployment tool:
- works with JAR files that conform to the EJB 1.1 specification.
- enables you to provide or change values for EJB environment entries.
- enables you to provide or change values for Container Bindings.
- enables you to add CICS resource definitions (CORBASERVERs, DJARs and REQUESTMODELs) to the DTD of your JAR file, or to change the attributes of those already there.
- can store your edited JAR files in HFS (but see the topic "Jar file tasks" for details).
  Jar files must be stored in HFS to be used.
- can prepare for you the batch input streams for CICS resource definition in either DFHCSQUP or BATCHREP form.

Getting started

To reveal the navigation panel:
- click on the words "CICS EJB Production Deployment" in the upper left part of this panel
- then click on the words "JAR file Tasks" to reveal the full navigation tree.
- For more detail, see the "Navigation" topic below.

Hints and Tips

- Throughout this tool, most fields have two levels of help information.
- If you allow the cursor to rest over a field description, or over an input field, a short message that might provide a "hint" about the use of this field appears.
- If you click on the Tips button in the lower right corner of most panels,
  a fuller description of the field that is currently "in focus" is seen in the Tips popup window.
  Click again to close the Tips window. Each time, the double arrow-head in the button changes direction from up to down and back again.
- You can alter the size of the Tips window by dragging its top bar up or down. You can also close the Tips window by dragging the top bar as far down as possible. From this position you can then open the window simply by dragging its top bar upwards.

Navigation

- On the left hand side of each window that the application presents, there is a navigation panel.
  To reveal the navigation panel, click on the words "CICS EJB Production Deployment" which appear in the upper left part of every navigation panel.
- You can drag the right hand border of the navigation panel across the application window to narrow or widen the panel.
- The Navigation panel has two tabs "Tasks" and "Resources". When the Navigation panel is too narrow to show both tabs, a left/right scroll bar appears.
- Under the Resources tab: by clicking on the words "Launch CICS Web Site", or the symbol in front, you can start your Web browser at the CICS Transaction Server web site.
- Under the Tasks tab: when you first click on the words "JAR file Tasks", the full navigation tree appears.
  Thereafter you may collapse or expand parts of the tree using the "twisties".
- Each item that has subordinate tasks is marked by a blue "twistie" symbol. The "twistie" is a round blue symbol with a single line pointing from it, either to the right or downwards. An example may be seen in front of the words "JAR file tasks" after your first click on those words has revealed the full navigation tree.
- You can expand each item by clicking the "twistie" to reveal the subordinate tasks. The "twistie", which previously pointed to the right, now points downward, and subordinate tasks appear. Click again to reverse this. Available tasks are shown in dark type, others appear in grey text.
- On each task line, left click on the text starts that function, and right click gives the option to start it, or start it in a new window.
- On most panels you make choices, or provide input, and then click the "OK" button to signal that you are ready to move on. Don't forget the "OK" button
- At the bottom of the list of tasks is a line with the words "Trace Information - use only when instructed".

The IBM logo

Every panel has an IBM logo in the top left corner. Clicking on this starts your web browser at the IBM home page.
5. The first question to answer is about the location of your JAR files.

   - You can work with local files or with files stored in an HFS directory on your OS/390 host system.

   - Files in HFS on your host system can be accessed using FTP, or the HFS directory can be defined to Windows on your workstation as a network drive using the "Map Network Drive" function of Windows. This defining of the HFS directory as a network drive is done outside the CICS production deployment tool, not by the tools.

   - Files in HFS on your host system, when accessed by defining the HFS directory to Windows as a network drive, are treated as local files for the purpose of this discussion.

   - If you work with files in HFS on your host system using FTP, you can only work with them, and you cannot access any local files, until you choose otherwise.

   - If you work with local files (including any on a network drive), you can only work with them, and you cannot access any files in HFS on your host system using FTP, until you choose otherwise.

   - When you choose to change from one kind of file access to the other, you start a new JAR file edit session within this use of the CICS production deployment tool.

   - It follows from all this that you can only transfer JAR files from workstation to host (or the reverse) within the CICS production deployment tool, if you also use the "Map Network Drive" function of Windows.

6. Whichever form of file access you select, the next panel asks you to identify the input and output files that you will use. File names and paths maybe entered directly, or selected by browsing. In either case they can, if required, then be modified by overtyping. Apart from input and output JAR files, you can also
identify a priming JAR file. A priming JAR file is one which you use as a model for changing the EJB information or the CICS information or both of them, when processing this source JAR file.

In the Specify files panel of the tool, open the Tips window, and place your cursor in the priming JAR file input box, more information on priming is presented there.

7. Each panel that requires input is completed by clicking the OK button at the foot of the panel (or the cancel button if that is appropriate). This changes the items that are in black type, or greyed-out, in the navigation panel if their availability depended on your actions in the panel that you have just ended. You must select the next panel to work on from the navigation panel, by clicking on the caption of that item. Panels are not presented to you automatically.

8. When setting CICS resources, you are providing values for parameters which are defined in the CICS Resource Definition Guide. You can have many REQUESTMODELS, but only one instance each of a CORBAServer definition and a DJAR definition.

See the CICS Resource Definition Guide for more information about defining CORBAServers, DJARs, and REQUESTMODELS.

The CICS production deployment tool as an offline utility

To run the CICS production deployment tool as an offline utility:

1. Prepare a script, such as the example in Figure 34 on page 194 and store it as EYU9EJDD in a directory in your PATH.

A variation of this script is available to you in

/usr/lpp/cicsts/lib/offline/EYU9EJDD

(your installation may be using a different directory naming convention).

Both versions, the one in this file, and the one shown in Figure 34 achieve the same purpose. The version in Figure 34 differs only because it has been rearranged to fit the printed or displayed page. It has extra records and variables so that it eliminates the longer records of the version in /usr/lpp/cicsts/lib/offline/EYU9EJDD which would not display so clearly. It also has extra explanatory comments.
2. Run the offline utility from the OS/390 UNIX System Services environment, by issuing the command:

EYU9EJDD input.xml
In this command:

**EYU9EJDD**

is the name of the offline utility script which you prepared.

**input.xml**

is the name of your input stream to be processed by EYU9EJDD. Figure [Figure 36 on page 196](#) shows a sample XML input file to use with EYU9EJDD.

EYU9EJDD expects to find this file in your default directory, unless you provide an *input.xml* which identifies the directory in which it is located.

3. or submit a batch job to run the offline utility script using BPXBATCH.

   Figure [Figure 35](#) shows sample JCL for a batch job to use BPXBATCH to run the offline utility.

```
//EYU9EJDD JOB accounting info,name,MSGLEVEL=1
//*
//EYU9EJDD EXEC PGM=BPXBATCH,REGION=0K,
//   PARM='SH EYU9EJDD /u/anyuser/myeyu9.xml'
//STDOUT DD PATH='/u/anyuser/stdout.out',PATHMODE=SIRWXU,
// PATHOPTS=(OWRONLY,OCREAT,OTRUNC)
//STDENV DD *
//PATH=.:/usr/lpp/java130/J1.3/bin:/bin:/usr/bin:/u/anyuser/bin:/
//*
```

In this sample:

**PARM** includes the name *(EYU9EJDD)* of the script that you want to process, and the name (/u/anyuser/myeyu9.xml) of the HFS file containing the input that the script is to process. It is this XML file, described in [XML input for the offline utility](#) on page 196, that identifies the input, output and other files that are used by the CICS production deployment tool offline utility, and the actions that are required of the utility on this occasion.

**STDOUT** identifies the HFS file that you wish to use to record information messages from BPXBATCH

**STDENV** identifies the PATH required. In this example the 4 elements are:

- `/usr/lpp/java130/J1.3/bin` the directory which contains the Java executables at your installation.
- `/bin` your installation's general directory for executables. The directory that contains the *cp* function.
- `/usr/bin` any other directory or directories that you need to define, according to your installation's conventions.
- `/u/anyuser/bin/` your own directory, the one in which you chose to store the script EYU9EJDD from Figure 34 on page 194.

Figure 35. sample JCL for a batch job to use BPXBATCH to run the offline utility
XML input for the offline utility

Figure 36 shows a sample XML input file to use with EYU9EJDD.

```xml
<?xml version="1.0" encoding="ebcdic-cp-us"?>
<!DOCTYPE eyu SYSTEM "EYU9EJDD.dtd">
<eyu name="EYU9EJDD" xml:lang="en">
<!-- The heading section above remains unchanged apart from the encoding value -->
<!-- there is NOT a function to accept values other than en for lang. -->
<!-- The global section defines Source and Target JAR files, and the DFHCSDUP -->
<!-- and CICSplex SM BATCHREP options. -->
<global>
  <source-jar>myinput.jar</source-jar>
  <target-jar>myoutput.jar</target-jar>
  <cics-info>
    <output-destination>/usr/anyuser/mycsdup</output-destination>
    <csd-list>csdpay</csd-list>
    <csd-group>paycsdgr</csd-group>
  </cics-info>
  <cpsm-info>
    <output-destination>/usr/anyuser/mybatrep</output-destination>
    <context>payroll</context>
    <resdesc>payrrdsc</resdesc>
    <resgroup>payrrgrp</resgroup>
  </cpsm-info>
</global>
<!-- Now set the environment entry envvar1 for bean EnviBean in -->
<!-- this JAR file to OFFLINE. -->
<set>
  <bean-name>EnviBean</bean-name>
  <env-entry>
    <env-entry-name>envvar1</env-entry-name>
    <env-entry-value>OFFLINE</env-entry-value>
  </env-entry>
</set>
<!-- Next define the CICS resources for this JAR. -->
<!-- A CORBAServer, 2 DJARs and a Request Model are shown here. -->
<define>
  <CICS>
    <cts-corba-server>
      <name>CS01</name>
      <description>Sample CORBAServer 1</description>
      <host>mvs.acme.com</host>
      <jndi-prefix>CICSHT61</jndi-prefix>
      <port>12345</port>
      <sess-bean-timeout>000010</sess-bean-timeout>
      <shelf>/var/cicsts/myshelf</shelf>
      <ssl>NO</ssl>
      <sslport>NO</sslport>
      <certificate>cert-id</certificate>
    </cts-corba-server>
    <cts-djar>
      <name>DJAR01</name>
      <description>DJAR DESC</description>
      <corba-server>CS01</corba-server>
      <hfs-file>/usr/anyuser/sample.jar</hfs-file>
    </cts-djar>
  </CICS>
</define>
```

Figure 36. a sample XML input file to use with EYU9EJDD (Part 1 of 2)
The values that you supply are shown thus: value. The structure of the input stream will vary from this sample to reflect the needs of the input JAR file that you are processing. Each statement type is explained in "XML input for the offline utility" on page 196. It also indicates which parts of the example are required, which are optional, and which can be repeated.

This tool is particularly suited to the processing of large numbers of JAR files. However you are recommended to gain familiarity with the CICS Production Deployment tool for EJB Technology by using first the interactive version of the tool. With its hints and tips facility, the interactive version of the tool will give you an appreciation of the Inputs and outputs. This may lead you more readily to an appreciation of the purpose of each of those parts of the XML file that you might need to alter.

The XML statements are explained below. Only start tags <nnn> are described. As you will see from the example, end tags </nnn> are also required except for the 2 statements at the beginning of the file.

![XML code](image)
<?xml version="1.0" encoding="ebcdic-cp-us"?>

This is the first record in the XML file. Code it just as it appears here, but make sure that the value for the encoding parameter matches the codepage which is in effect when the file is stored. Two of the more frequently used values are:

iso-8859-1

is valid if the file containing the XML is in ASCII. If the file has only been edited on the workstation and never on the OS/390 host system, this is likely to be the case.

ebcdic-cp-us

is valid if the file containing the XML is in EBCDIC. If the file has been edited on the OS/390 host system, this is likely to be the case.

<!DOCTYPE eyu SYSTEM "EYU9EJDD.dtd">

This is the second record in the XML file. Code it just as it appears here.

<bean-name>

1. Appears within a <set> group of statements and is followed by an <env-entry> statement. It identifies the bean-name to which the following <env-entry> statement relates. You may code as many <bean-name>/<env-entry> pairs as you need.

2. Is a parameter of the REQUESTMODEL resource definition. See the description of <cts-rqmodel> for a link to the details.

3. Appears within a <bind> group of statements, where it is followed by a <binding> statement. It identifies the bean-name to which the following <binding> statement group relates.

<bind>

Must follow the <define> statement. It is a wrapper for resource and reference binding information: the <bean-name> statement and the <binding> statement. The <bind> group is optional.

<binding>

Must follow the <bean-name> statement within a <bind> group. It is a wrapper for resource and reference binding information: the <reference> statement, the <link> statement and the <JNDI-binding> statement. The <binding> group is required if a <bind> group is used.

<certificate>

is a parameter of the CORBASERVER resource definition. See the description of <cts-corba-server> for a link to the details.

<CICS>

Must follow the <define> statement. It is a wrapper for the CICS resource definition information: the <cts-corba-server> statement the <cts-djar> statement and the <cts-rqmodel> statement. The <CICS> group is optional.

<cics-info>

Follows the <target-jar> statement. It is a wrapper for a group of statements about the output file for CICS Resource definition statements in DFHCSDUP format.

This group of statements is optional. If you include this group of statements in the XML file, it indicates that an output in DFHCSDUP format is produced. The output, if selected contains the commands required by DFHCSDUP to add the resource definitions described by the <define> group of XML statements.

Including the <cics-info> does not run DFHCSDUP to add these definitions to the CSD, nor does it INSTALL those definitions in the active CICS.

<context>

Follows the <output-destination> statement within <cics-info>. The
statement is required if the parent group is coded. It identifies the CICSpix SM context information. This name is not case sensitive.

<corba-server>
is a parameter of the DJAR resource definition, and the REQUESTMODEL resource definition. See the description of the relevant group of statements for a link to the details.

<cpsm-info>
Follows the <target-jar> statement or the <cics-info> statement. It is a wrapper for a group of statements about the output file for CICS Resource definition statements in CICSpix SM BATCHREP format.

This group of statements is optional. If you include this group of statements in the XML file, it indicates that an output in BATCHREP format is produced. The output, if selected contains the commands required by BATCHREP to add the resource definitions described by the <define> group of XML statements.

Including the <cpsm-info> does not run BATCHREP.

<csd-group>
Follows the <csd-list> statement within <cics-info>. The statement is required if the parent group is coded. It identifies the CSD group which is to be updated when the <output-destination> file is processed by DFHCSDUP. This name is not case sensitive.

<csd-list>
Follows the <csd-name> statement within <cics-info>. The statement is required if the parent group is coded. It identifies the CSD list which is to be updated when the <output-destination> file is processed by DFHCSDUP. This name is not case sensitive.

<cts-corba-server>
The <cts-corba-server> group defines the parameters of a CORBASERVER resource definition. These are described in the CICS Resource Definition Guide. The <cts-corba-server> group is optional, but if used, all its subordinate XML statements are required. If they are to take no value, they must appear as <tag></tag>.

<cts-corba-server-name>
Appears within the <discard> group of statements. It identifies a CICS resource for which resource definition statements to discard the resource are to be generated.

<cts-djar>
The <cts-djar> group defines the parameters of a DJAR resource definition. These are described in the CICS Resource Definition Guide. The <cts-djar> group is optional, but if used, all its subordinate XML statements are required. If they are to take no value, they must appear as <tag></tag>.

<cts-djar-name>
Appears within the <discard> group of statements. It identifies a CICS resource for which resource definition statements to discard the resource are to be generated.

<cts-rqmodel>
The <cts-rqmodel> group defines the parameters of a REQUESTMODEL resource definition. These are described in the CICS Resource Definition Guide. The <cts-rqmodel> group is optional, but if used, all its subordinate XML statements are required. If they are to take no value, they must appear as <tag></tag>.

<cts-rqmodel-name>
Appears within the <discard> group of statements. It identifies a CICS resource for which resource definition statements to discard the resource are to be generated.
<define>
Must follow the <set> statement. It is a wrapper for the <CICS> statement.
The <define> group is optional.
<description>
is a parameter of the CORBASERVER resource definition, the DJAR resource definition, and the REQUESTMODEL resource definition. See the description of the relevant group of statements for a link to the details.
<discard>
Follows the <define> statement. It is a wrapper for the <cts-corba-server-name> statement, the <cts-djar-name> statement, and the <cts-rqmodel-name> statement. The <discard> group is optional.
<env-entry>
Appears within a <set> group of statements and follows a <bean-name> statement. It contains pairs of <env-entry-name> and <env-entry-value> statements. You may code as many <bean-name>/ <env-entry> pairs as you need.
<env-entry-name>
Appears within an <env-entry> group of statements and is followed by a <env-entry-value> statement. It identifies the environment entry that the <env-entry-value> updates. You may code as many <env-entry-name>/ <env-entry-value> pairs as you need.
<env-entry-value>
Appears within an <env-entry> group of statements and follows an <env-entry-name> statement. It provides the new value for the environment entry that the <env-entry-name> identifies. You may code as many <env-entry-name>/ <env-entry-value> pairs as you need.
<eyu name="EYU9EJDD" xml:lang="en">
This is the third record in the XML file. Code it just as it appears here. There is not a facility to change the xml language in this implementation.
<global>
Must follow the <eyu ........> statement. It is a wrapper for statements about the Input JAR file and the various output files for this use of EYU9EJDD. There is not a facility to identify a priming JAR file.
<hfs-file>
is a parameter of the DJAR resource definition. See the description of <cts-djar> for a link to the details.
<host>
is a parameter of the CORBASERVER resource definition. See the description of <cts-corba-server> for a link to the details.
<interface>
is a parameter of the REQUESTMODEL resource definition. See the description of <cts-rqmodel> for a link to the details.
<interface-type>
is a parameter of the REQUESTMODEL resource definition. See the description of <cts-rqmodel> for a link to the details.
<JNDI-binding>
Appears within a <binding> group of statements and following a <bean-name> statement. In the group, the <JNDI-binding> follows the <link>. It is used to supply the value for the <JNDI-binding> appropriate for the <reference> of type <link>.
<jndi-prefix>
is a parameter of the CORBASERVER resource definition. See the description of <cts-corba-server> for a link to the details.
<link>
Appears within a <binding> group of statements and following a
<bean-name> statement. In the group, the <link> follows the <reference>. It identifies the type of resource or reference for which the <JNDI-binding> is needed.

<module>
   is a parameter of the REQUESTMODEL resource definition. See the description of <cts-rqmodel> for a link to the details.

<name>
   is a parameter of the CORBASERVER resource definition, the DJAR resource definition, and the REQUESTMODEL resource definition. See the description of the relevant group of statements for a link to the details.

<operation>
   is a parameter of the REQUESTMODEL resource definition. See the description of <cts-rqmodel> for a link to the details.

<output-destination>
   Appears first in the <cics-info> group of statements and in the <cpsm-info> group. The statement is required if the parent group is coded. It identifies the file that will hold the DFHCSDUP commands (for <cics-info>) or the BATCHREP commands (for <cpsm-info>). <output-destination> is case-sensitive.

<port>
   is a parameter of the CORBASERVER resource definition. See the description of <cts-corba-server> for a link to the details.

<reference>
   Appears within a <binding> group of statements and following a <bean-name> statement. It identifies the resource or reference within the bean that requires a <JNDI-binding>.

<resdesc>
   Follows the <context> statement within <cics-info>. The statement is required if the parent group is coded. It identifies the CICSPlex SM resource description. This name is not case sensitive.

<resgroup>
   Follows the <resdesc> statement within <cics-info>. The statement is required if the parent group is coded. It identifies the CICSPlex SM resource group. This name is not case sensitive.

<sess-bean-timeout>
   is a parameter of the CORBASERVER resource definition. See the description of <cts-corba-server> for a link to the details.

<set>
   Must follow the <global> statement. It is a wrapper for pairs of <bean-name> and <env-entry> statements. The <set> group is optional.

<shelf>
   is a parameter of the CORBASERVER resource definition. See the description of <cts-corba-server> for a link to the details.

<source-jar>
   The first statement inside a <global> block. It identifies the JAR file that you want to process. It is required, and case-sensitive.

<ssl>
   is a parameter of the CORBASERVER resource definition. See the description of <cts-corba-server> for a link to the details.

<sslport>
   is a parameter of the CORBASERVER resource definition. See the description of <cts-corba-server> for a link to the details.

<target-jar>
   Follows the <source-jar> statement. It identifies the output JAR file that you intend to write. It is required, and case-sensitive.

<transid>
   is a parameter of the REQUESTMODEL resource definition. See the description of <cts-rqmodel> for a link to the details.
<type>
is a parameter of the REQUESTMODEL resource definition. See the
description of <cts-rqmodel> for a link to the details.

The final stages of deployment

If you use the CICS development deployment tool for EJB technology, the following
tasks are unnecessary. However if you use the CICS production deployment tool,
there are three more tasks to complete.

Applying generated resource definitions

The resource definition statements that were generated by the CICS production
deployment tool, either as a DFHCSDUP input stream, or as a BATCHREP input
stream, (or both, if you so specified), and which were stored in HFS, now have to
be applied to the systems for which they are intended. This is done by use of
DFHCSDUP or BATCHREP.

DFHCSDUP

It is necessary to copy the resource definition statements from HFS to an
MVS data set before running DFHCSDUP. One way of doing so and then
running DFHCSDUP is shown in Figure 37. In that example, HFSPATH
identifies the HFS file that contains the resource definition statements that
were generated by the CICS production deployment tool.

```
//CSDJOB JOB accounting info,name,MSGLEVEL=1
//*
//PARMS SET HFSPATH='''/usr/anyuser/mycsdup''',
// CSDNAME=CICSTS21.CICS.csd
//*
//OCOPY EXEC PGM=IKJEFT01,REGION=0M
//CSDIN DD DSN=&CSDIN,DISP=(NEW,PASS),UNIT=SYSDA,
// SPACE=(CYL,(8,64)),DCB=(LRECL=80,BLKSIZE=6080,
// RECFM=FB)
//HFSIN DD PATH=&HFSPATH
//SYSTSPRT DD SYSTSPRT
//SYSTSIN DD *
OCOPY INDD(HFSIN) OUTDD(CSDIN)
/*
//CSDUP EXEC PGM=DFHCSDUP,REGION=0M,COND=(0,NE)
//STEP1 DB DD DSN=CICSTS21.CICS.SDFLOAD,DISP=SHR
//DFHCSD DD DSN=&CSDNAME,DISP=SHR
//SYSUT1 DD UNIT=SYSDA,SPACE=(1024,(100,10))
//SYSPRINT DD SYSOUT=A
//SYSDUMP DD SYSOUT=A
//SYSABEND DD SYSOUT=A
//CBOOUT DD SYSOUT=A
//SYSSIN DD DSN=*.OCOPY.CSDIN,DISP=SHR
//*/
```

Figure 37. Sample JCL to copy DFHCSDUP commands from HFS to an MVS data set and
then to apply those commands using DFHCSDUP.

BATCHREP

It is necessary to copy the resource definition statements from HFS to an
MVS data set before running BATCHREP. One way of doing so is shown in
Figure 38 on page 203. In that example, HFSPATH identifies the HFS file
that contains the resource definition statements that were generated by the
CICS production deployment tool.
The resource definition statements are then available for use by BATCHREP. For information on submitting a batched repository update job (BATCHREP), see CICSPlex System Manager Administration.

You could of course use any other available method to copy the HFS file to an MVS data set, including online use of the OCOPY command.

The BATCHREP output produced by the CICS production deployment tool uses the facilities of CICSPlex SM Business Application Services (BAS) to produce a self-contained set of resource definitions. However, as CICSPlex SM handles these resources in a different way to CICS resource definitions which are stored in the CSD, you are advised to familiarize yourself with the implications of these differences before attempting to install them into your systems. For more information, see the CICSPlex System Manager Concepts and Planning manual.

Publishing names to the JNDI

Finally the names of the DJARs and the CORBASERVERs involved must be published to the JNDI. See Creating object references in the namespace on page 143 for details.

Ensuring that the JAR file is stored in HFS

If you did not use the CICS production deployment tool to store the output ejb-jar file in the HFS of the MVS image in which the required CICS region is run, you must now make sure that it is stored there, either by reuse of the CICS production deployment tool, or otherwise.

The ejb-jar file is then deployed to the CICS system, and is available to be used.
Chapter 19. The CICS Connector for CICS TS

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

- “Overview of the CICS Connector for CICS TS”
- “Using the CICS Connector for CICS TS’s CCF interface” on page 210
- “Introducing VisualAge for Java Enterprise Access Builder” on page 213
- “Data conversion and the CICS Connector for CICS TS” on page 215
- “The CICS Connector for CICS TS—using the CTG API” on page 219
- “Benefits of the CICS Connector for CICS TS” on page 209
- “Requirements for the CICS Connector for CICS TS” on page 220
- “Restrictions and recommendations for the CICS Connector for CICS TS” on page 219
- “Installing the CICS Connector for CICS TS” on page 220
- “CICS Connector for CICS TS messages” on page 221
- “Tracing the CICS Connector for CICS TS” on page 221

Overview of the CICS Connector for CICS TS

The CICS Connector for CICS TS helps you to build Enterprise JavaBean (EJB) server components that make use of existing CICS programs.

Previous releases of CICS support CICS connectors that enable a Java client program, running outside CICS (on Windows NT, OS/2, Unix, or native OS/390), to connect to a specified program on a CICS server. CICS Transaction Server for z/OS introduces a new CICS connector—the CICS Connector for CICS TS—that enables a Java program running on CICS Transaction Server for z/OS to connect to a specified program on a CICS server.

What are CICS connectors?

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 applets or servlets.

A CICS connector is supported on each of the following platforms: AIX, OS/2, Windows NT, Solaris, and OS/390. The Java client application may run on the same platform as the connector, or it may run on any Java-enabled platform and drive the connector by means of a gateway process running on one of the listed platforms.

The CICS connectors that run on AIX, OS/2, Windows NT, and Solaris are shipped as part of the CICS Transaction Gateway (CTG) product and connect to CICS on OS/390 using an SNA or TCP62 connection. The CICS connector that runs on native OS/390 is shipped as part of the CICS Transaction Gateway for OS/390 product and connects to CICS on OS/390 using an external CICS interface (EXCI) connection.

In every case, the Java client application using the connector is coded using one of two application programming interfaces (APIs):

1. A lower-level, CICS-specific, API known as the CTG API. The CTG API consists of the external call interface (ECI) and the external presentation interface (EPI).
2. A higher-level API known as the Common Connector Framework (CCF) Client Interface.
The Common Connector Framework is an IBM architecture that defines a standard way for a Java program to interact with an application server such as CICS, IMS, or SAP. One of the advantages of this architecture is that it provides a client API with the same style, regardless of the application server that it drives. This enables application development tools, such as VisualAge for Java (VAJ) Enterprise Access Builder, to provide generic tooling independent of the type of application server being accessed.

All the CICS connectors support the CCF Client Interface.

The CICS Connector for CICS TS runs on CICS TS z/OS. Like the other CICS connectors, it provides both the CTG API and the CCF Client Interface.

The recommended way to create a Java application or bean that uses a CICS connector is to use the VAJ Enterprise Access Builder, or a product that offers similar function, to program to the connector’s CCF Client Interface. If you do not have such a product, you must use the CTG API.

The Java applications, or beans, that you create are portable across the set of CICS connectors. Thus, for example, a bean that invokes a CICS program on a CICS OS/390 server could be used from:

- An applet in a browser, connecting by means of a gateway process running on any of the platforms supported by the CTG
- A servlet in a Web server running on any of the platforms supported by the CICS connectors
- An enterprise bean in an EJB server running on any of the platforms supported by the CICS connectors
- A CICS Java application or enterprise bean running in CICS TS z/OS.


The CICS Connector for CICS TS

The CICS Connector for CICS TS allows a Java program or enterprise bean running on CICS TS z/OS to link to a CICS server program. It allows you, for example, to create powerful EJB components that make use of existing CICS programs.

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 OS/390 region, but optionally may be on the same CICS region as the Java program or bean.

The background—accessing CICS programs from Java

Frequently, new Java applications can be developed more quickly and reliably by harnessing the power of existing (non-Java) CICS programs. Typically, the Java application is network-based, perhaps started from a browser, and the CICS program is written in a language such as COBOL. This section reviews the several ways in which existing CICS OS/390 programs can be accessed from Java code, and shows how the CICS Connector for CICS TS fits into this pattern.
From Java programs outside CICS

From the network, a Java client application or applet can use the CICS connector interface—that is, either the CCF Client Interface or the CTG API—to link to a CICS OS/390 program.

This method is shown in Figure 39. In this example, because the client applet is not running on the same host as the CICS connector, the CICS Transaction Gateway for OS/390 is used to communicate with the connector. The connector uses EXCI to pass requests to CICS.

The picture also shows a Java servlet. It too uses the CICS connector interface to connect to a CICS OS/390 server program. Because the servlet is running on the same host as the connector, it uses the “local” protocol to communicate with the latter. The CICS Transaction Gateway for OS/390 is bypassed.

The CICS connector for native OS/390 supports the external call interface but not the external presentation interface. Thus, ECI but not EPI calls are supported.

This method is shown in Figure 39.

Figure 39. Java clients connect to a CICS server program from outside CICS. A Java applet, running on a browser, uses the CICS connector interface to link to a CICS OS/390 server program. Because the client applet is not running on the same host as the CICS connector, the CICS Transaction Gateway for OS/390 is used to communicate with the connector. The connector uses EXCI to pass requests to CICS. (To CICS, these appear to be ECI calls. Because EXCI is used, the CICS server region must be on the same OS/390 operating system, or Parallel Sysplex, as the connector.)

The picture also shows a Java servlet. It too uses the CICS connector interface to connect to a CICS OS/390 server program. Because the servlet is running on the same host as the connector, it uses the “local” protocol to communicate with the latter. The CICS Transaction Gateway for OS/390 is bypassed.

A variation is shown in Figure 40 on page 208. In this scenario, the CICS Transaction Gateway runs on an intermediate Windows NT, OS/2, or Solaris machine. On these platforms, the CICS connector uses a CICS Universal Client to pass requests to a back-end CICS OS/390 region. In this setup, the full CTG API (including both ECI and EPI functions) is supported. In other words, the Java client
can access 3270-based CICS programs, as well as CICS programs that use a suitable communications area.

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

From Java programs inside CICS
There are two methods by which Java programs running within CICS can access non-Java CICS programs:

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

The Java programmer requires a detailed knowledge of CICS.

**Using the CICS Connector for CICS TS:** In CICS Transaction Server for z/OS only, a Java program or enterprise bean running on CICS can use the CICS Connector for CICS TS to link to a suitable CICS server program. The connector uses a CICS LINK call, rather than EXCI, to access the back-end server program. Link and distributed program link (DPL) calls are supported. This scenario is shown in Figure 41 on page 209.

The CICS Connector for CICS TS uses the connector classes provided with the CICS Transaction Gateway for OS/390; however, because the client program or bean runs on the same host as the connector, it is not necessary, to use the connector, to run the CICS Transaction Gateway for OS/390 as a server application in its own address space.
There are two ways of using the CICS Connector for CICS TS:

1. Program to the connector’s CCF Client Interface, using VisualAge for Java Enterprise Access Builder or a similar product. This is the recommended method.

2. Program to the connector’s CTG API. Normally, you would use this method only if you do not have access to VAJ EAB or a similar product.

To use the CICS Connector for CICS TS to create an enterprise bean, the Java programmer requires a reasonable knowledge of CICS (although somewhat less than if he were using JCICS). However, the enterprise beans that he creates, using either the connector or the JCICS classes, can be used by Java programmers who have little knowledge of CICS.

Benefits of the CICS Connector for CICS TS

1. The CICS Connector for CICS TS helps you to build powerful enterprise beans that make use of existing CICS programs.

2. The connector enables you to exploit VisualAge for Java Enterprise Access Builder, and similar tools, to develop enterprise beans rapidly.

3. The enterprise beans that you build:
   - 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 used by Java client applications, applets, and servlets running on many platforms.

A sample application

The EJB CICS sample application uses the CICS Connector for CICS TS. The sample implements an enterprise bean that uses the connector to link to back-end CICS COBOL programs. The EJB CICS sample application is described in “The EJB CICS sample application” on page 133.
Using the CICS Connector for CICS TS’s CCF interface

The recommended way to create a Java application that uses any CICS connector—including the CICS Connector for CICS TS—is to use the VisualAge for Java Enterprise Access Builder, or a similar product, to program to the connector’s CCF Client Interface. VAJ is described in “Introducing VisualAge for Java Enterprise Access Builder” on page 213. The rest of this section describes what you need to know about the CCF Client Interface in order to use the Enterprise Access Builder effectively.

The CCF Client Interface consists of the following classes:

**ConnectionSpec**

A `ConnectionSpec` object holds all the connection-relevant attributes (for example, hostname and TCP/IP port number) necessary to drive an interaction with a server. It identifies a unique connection.

It is the factory for a `Communication` object.

The CICS Connector for CICS TS’s `ConnectionSpec` class is called `CICSConnectionSpec`.

**InteractionSpec**

An `InteractionSpec` object holds all the interaction-relevant attributes (for example, the name of the target program and the mode of the interaction—send or receive) necessary for an interaction with a server. It is passed as an argument to a `Communication` object when a particular interaction is to be carried out.

The CICS Connector for CICS TS’s `InteractionSpec` class is called `ECIInteractionSpec`.

**Communication**

The `execute` method of a `Communication` object allows you to drive an interaction with a server. The `execute` method takes three arguments—an `InteractionSpec` that specifies the type of interaction, and `input` and `output` objects that carry the exchanged data.

**input/output**

`Input` and `output` objects are beans that hold the data exchanged with the target program. The data is accessible via the bean’s property access methods.

The implementation of the beans may be based on the Java Record Library, or may be proprietary to a particular CCF connector.

VAJ Enterprise Access Builder provides tools to import a data descriptor (such as a COBOL copy book) that represents the communications area of the target program. From the data descriptor, you can construct record beans that are used to build and decode the target program data.

Here’s an outline of the program logic needed for a single interaction with an application server, using the standard CCF Client Interface classes supported by a CICS connector. This is the logic a CICS enterprise bean would use to access a back-end CICS program.

1. Create a `CICSConnectionSpec` object that includes a URL, which (in our case, because the CICS enterprise bean is running on the same operating system as the CICS Connector for CICS TS) should be set to “local://”. 

210 CICS Transaction Server: Java applications in CICS
Note: How to set the attributes of the CCF Client Interface objects is described in more detail in "Setting the CCF interface attributes".

2. Create a Communication object from the CICSConnectionSpec object, and execute its connect method.

3. Create an ECIInteractionSpec object that includes the name of the target program.

4. Run the execute method of the Communication object, passing the ECIInteractionSpec, and the input and output record beans, as arguments.

5. Retrieve the data returned by the target program from the output record bean.


This sequence is illustrated in Figure 42.

Figure 42. Using the CCF Client Interface classes

Setting the CCF interface attributes

This section describes how to set the attributes of the CICSConnectionSpec and ECIInteractionSpec classes that form part of the CICS Connector for CICS TS's
Client Interface. (The Communication, input, and output classes are programmed in the same way as for other CICS CCF connectors.)

**Note:** Many of the attributes are ignored by the CICS Connector for CICS TS. This is because the CICSCConnectionSpec and ECIInteractionSpec classes are also used by “traditional” CTG client applications, for some of which the attributes are meaningful. Ignoring the attributes, rather than introducing new types of CICSCConnectionSpec and ECIInteractionSpec, means that existing CTG client applications can be ported more easily to CICS OS/390.

**CICSCConnectionSpec**
Set the attributes of your CICSCConnectionSpec object as follows:

**CICSServer**
The SYSID of the CICS region which owns the program to be linked to. You can set this value explicitly. However, the recommended method is to set a null value here, and to rely on the PROGRAM definition to specify the location of the server program, and whether or not dynamic routing should occur.

**ClientSecurityClassName**
Ignored by CICS.

**connectionTimeout**
Ignored by CICS.

**GatewayURL**
Set as follows:

- **auto://tcpipaddr:portno/**
  Supported, provided that the TCP/IP address is that of the host that CICS is running on (in which case, the local protocol is used).

- **local://**
  Supported.

- **http://tcpipaddr/**
  Not supported.

- **tcp://tcpipaddr:portno/**
  Not supported.

Normally (because the CICS enterprise bean you are creating will run on the same host as the CICS Connector for CICS TS) you would set strGatewayURL to “local://”.

**logonLogoff**
Ignored by CICS.

**maxConnections**
Ignored by CICS.

**minConnections**
Ignored by CICS.

**reapTime**
Ignored by CICS.

**ServerSecurityClassName**
Ignored by CICS.

**terminalModel**
Ignored by CICS.
unusedTimeout
Ignored by CICS.

ECIInteractionSpec
Set the attributes of your ECIInteractionSpec object as follows:

CICSELUW
Set as follows:
- **False**  The ECI call is not part of the CICS extended unit of work (UOW).
- **True**   The ECI call is part of the CICS extended UOW.

**Note:** In “traditional” CTG applications, the CICS extended UOW encompasses a series of one or more ECI requests to a server program, each executed with SYNCONRETURN set off, followed by a final ECI request (to the same server program) that is executed with SYNCONRETURN set on. This final ECI call causes CICS to take a syncpoint on successful completion of the server program, and any changes to resources made by the server program to be committed.

When using the CICS Connector for CICS TS, all the LINK requests in the CICS extended UOW, including the last, are executed with SYNCONRETURN set off. Any changes to resources made by the server program are committed by CICS at end of task or if the application issues a syncpoint. This behaviour is consistent with that of CICS DPL.

ECITimeout
Ignored by CICS.

Mode
Only sync (MODE_SEND_RECEIVE) is supported.

Password
Ignored by CICS.

ProgramName
Set to the name of the program to be linked to.

TPNTransactionName
Ignored by CICS.

TransactionName
Optionally, can be set to the name of the transaction to be used as the mirror transaction on the remote region. The default is an empty string (""").

Userid
Ignored by CICS.

---

**Introducing VisualAge for Java Enterprise Access Builder**

The VisualAge for Java Enterprise Access Builder provides visual tools and high-level constructs for programming the connector’s CCF Client Interface. In particular, the Enterprise Access Builder provides a construct called a *Command*. A Command represents a single interaction with an application server. It is a Java bean with properties and an execute method—see Figure 43 on page 214. A Command is a composition—it encapsulates all the CCF objects necessary for carrying out an interaction.
The Enterprise Access Builder also provides a high-level construct called a Navigator, which implements a sequence of interactions with an application server. A Navigator is simply a composition of Commands and Navigators. Figure 44 on page 215 shows a Navigator produced by the Enterprise Access Builder.

Figure 43. A Command bean produced by VisualAge for Java EAB
The EAB Command Editor tool allows you to build Commands and Navigators visually.

Data conversion and the CICS Connector for CICS TS

Java programs always use the Unicode character set. However, the communications area passed to the target program on the back-end CICS OS/390 region must be in EBCDIC. When writing your enterprise beans, you can handle data conversion using either of two methods:

Method 1

1. Convert from Unicode to ASCII in the input record bean.6
2. Use the CICS conversion program, DFHCCNV, to convert between ASCII and EBCDIC. The connector calls DFHCCNV, before and after the program link call, provided that you have included a conversion template for the program’s communication area in the DFHCNV conversion table. Figure 45 on page 216 shows an example conversion template, coded using DFHCNV macros, for the communications area of a program named server_program.

---

6. We’re assuming that you’re programming the connector’s CCF Client Interface.
For detailed information about the DFHCCNV conversion program, conversion templates, the DFHCNV conversion table, and the syntax of DFHCNV macros, see the CICS Family: Communicating from CICS on System/390.

Note: The connector calls DFHCCNV on the local CICS region—the region on which the enterprise bean runs—even if the target program is remote. So you must add your conversion template to the conversion table on the local region.

3. Convert from ASCII to Unicode in the output record bean.

Method 2
1. Convert directly from Unicode to EBCDIC in the input record bean.
2. Convert directly from EBCDIC to Unicode in the output record bean.

This method is more efficient because it misses out the intermediate conversion from Unicode to ASCII. However, the record beans must be coded to contain all the conversion logic.

The CICS Connector for CICS TS—using the CTG API

If you do not have VisualAge for Java Enterprise Access Builder or a similar product, your enterprise beans can use either JCICS or the CICS Transaction Gateway API to link to a CICS server program.

This section tells you how to program the CICS Connector for CICS TS's non-CCF interface, using the CTG API.

Using the CTG API to link from a CICS OS/390 Java program or enterprise bean to another, possibly remote, CICS program is possible only in CICS TS z/OS. (In previous releases of CICS, you could only use the CTG API to link to a CICS OS/390 program from a non-CICS Java client application.)

The CTG API is described in the CICS Transaction Gateway Programming Guide, SC34-5594-00, and in the HTML documentation shipped with the CICS Transaction Gateway. You should refer to those sources for general guidance. The rest of this section describes how to use the ECIRequest and JavaGateway classes with the CICS Connector for CICS TS.

Note: Many of the attributes of the ECIRequest and JavaGateway classes are ignored by the CICS Connector for CICS TS. This is because these classes are also used by “traditional” CTG client applications, for some of which the attributes are meaningful. Ignoring the attributes, rather than introducing new types of ECIRequest and JavaGateway, means that existing CTG client applications can be ported more easily to CICS OS/390.
ECIRequest

Set the attributes of your ECIRequest object as follows:

**Abend_Code**
Ignored by CICS.

**Call_Type**
Set to one of the following:

- **CICS_EciListSystems**
  The connector returns a list of CICS server regions.

  **Note:** The list returned by the CICS Connector for CICS TS contains only one item—see the description of the SystemList attribute.

- **ECI_STATE_SYNC**
  The connector returns the status of the connection.

- **ECI_SYNC**
  The connector issues an EXEC CICS LINK SYSID() PROGRAM() call.

- **ECI_SYNC_TPN**
  The connector issues an EXEC CICS LINK SYSID() PROGRAM() call.

**Cics_Rc**
CICS returns a suitable return code on each request.

**Commarea**
Set to a communications area suitable for the server program to be linked to.

**Commarea_length**
Set to the length of the communications area.

**ConnectionType**
Returned by CICS on an ECI_STATE_SYNC call.

**CicsClientStatus**
Returned by CICS on an ECI_STATE_SYNC call.

**CicsServerStatus**
Returned by CICS on an ECI_STATE_SYNC call.

**Extended_Mode**
Supported by CICS.

  **Note:** In “traditional” CTG applications, extended mode encompasses a series of one or more ECI requests to a server program, each executed with SYNCONRETURN set off, followed by a final ECI request (to the same server program) that is executed with SYNCONRETURN set on. This final ECI call causes CICS to take a syncpoint on successful completion of the server program, and any changes to resources made by the server program to be committed.

  When using the CICS Connector for CICS TS, all the LINK requests in the CICS extended UOW, including the last, are executed with SYNCONRETURN set off. Any changes to resources made by the server program are committed by CICS at end of task or if the application issues a syncpoint. This behaviour is consistent with that of CICS DPL.

**Luw_Token**
In extended mode, CICS returns a unique number denoting the unit of work (UOW) token.
Message_Qualifier
    Ignored by CICS.

Password
    Ignored by CICS.

Program
    Set to the name of the program to be linked to.

Server
    Set to the SYSID of the CICS region which owns the program to be linked to. You can set this value explicitly. However, the recommended method is to set a null value here, and to rely on the PROGRAM definition to specify the location of the server program, and whether or not dynamic routing should occur.

SystemList
    Contains a list of CICS server regions, each CICS denoted by its SYSID. This list is returned by the connector on a CICS_EciListSystems request.

    The CICS Connector for CICS TS returns a list consisting of only one item—the “CICS default list entry”, which has a SYSID of “ ” (4 spaces). CICS treats this SYSID as null; setting the Server attribute of the ECIRequest object to this value causes CICS to run the server program locally.

Transid
    Optionally, can be set to the name of the transaction to be used as the mirror transaction on the remote region. The default is an empty string (“”).

Userid
    Ignored by CICS.

JavaGateway

    Set the attributes of your JavaGateway object as follows:

Port
    Ignored by CICS.

Server
    Set as follows:

    auto://tcpipaddr/
        Supported, provided that the TCP/IP address is that of the host that CICS is running on.

    local://
        Supported.

    http://tcpipaddr/
        Not supported.

    tcp://tcpipaddr/
        Not supported.

    Normally (because the CICS enterprise bean you are creating will run on the same host as the CICS Transaction Gateway) you would set strGatewayURL to “local://”.

SetClientSecurity
    Ignored by CICS.

SetServerSecurity
    Ignored by CICS.
Restrictions and recommendations for the CICS Connector for CICS TS

The following restrictions and recommendations apply to the CICS Connector for CICS TS:

1. **The CCF Interface**
   The values you can assign to some attributes of CCF objects are restricted:
   - **ECIInteractionSpec objects**
     - **Mode** must be set to **MODE_SEND_RECEIVE**.
     - **ProgramName** must be set.

     This request maps to an EXEC CICS LINK PROG() {SYSID()} {SYNCONRETURN} command.
   - **CICSCConnectionSpec objects**
     - **GatewayURL** must be either of the following:
       - `local://`
       - `auto://tcpipaddr:portno/`  
       where `tcpipaddr` is the TCP/IP address of the host CICS.

2. **CTG API**
   The values you can assign to some attributes of CTG objects are restricted:
   - **ECIRequest objects**
     - **Call_Type** must be one of the following:
       - **ECI_STATE_SYNC**
         Returns the status, which is a single value made up of the following components:
         - **ECI_CONNECTED_TO_SERVER**
         - **ECI_SERVERSTATE_UP**
         - **ECI_CLIENTSTATE_INAPPLICABLE**

       **ECI_SYNC**
       Has the same semantics as ECI_SYNC_TPN. The program name must be set and, optionally, the server name (which is used as the SYSID).

       This request maps to an EXEC CICS LINK PROG() {SYSID()} {SYNCONRETURN} command.
       **ECI_SYNC_TPN**
       Has the same semantics as ECI_SYNC. The program name must be set and, optionally, the server name (which is used as the SYSID).

       This request maps to an EXEC CICS LINK PROG() {SYSID()} {SYNCONRETURN} command.
   - **JavaGateway objects**
     - **Server** must be either of the following:
       - `local://`
       - `auto://tcpipaddr:portno/`  
       where `tcpipaddr` is the TCP/IP address of the host CICS.

3. **Transaction commit point**
   - If the target server program is local to the region on which the connector runs, or is remote and the request mode is extended (**ECI_EXTENDED**):
     - The transaction is committed at end-of-task or **SYNCPOINT**
     - Can be backed out by a **SYNCPOINT ROLLBACK** request.
If the target server program is remote and the request mode is not extended (ECI_NO_EXTEND):
- The transaction at the backend is committed on return (SYNCONRETURN).
- The local transaction is committed at end-of-task or SYNCPOINT, or can be backed out by a SYNCPOINT ROLLBACK request.

4. The connector does not support explicit commit or backout commands.
5. Connector applications must be in JVM mode.
6. It is recommended that, for improved performance, you run the CTG classes on the trusted middleware classpath.
7. All connector requests must be made on the primary thread for the process.

---

Installing the CICS Connector for CICS TS

Requirements for the CICS Connector for CICS TS

The software requirements for the CICS Connector for CICS TS are:
1. The CICS Transaction Gateway for OS/390 Version 3.12. This is shipped with CICS.
2. If the CICS server program is to run on a separate back-end CICS region (the usual case), the back-end region must support distributed program link (DPL) calls.

Installing the connector

The CICS Connector for CICS TS consists of two parts:
1. A platform-specific library of native functions, libCTGJNI.so
2. The CICS Transaction Gateway for OS/390 Java classes.

Both the libCTGJNI.so library and the CICS Transaction Gateway for OS/390 Java classes are automatically installed when you install CICS.

Check that the library path used by the CICS JVM includes the directory which contains the libCTGJNI.so library. CICS installs libCTGJNI.so into the /usr/lpp/cicsts/cicsts21/lib directory (where cicsts21 is the value of the CICS_DIRECTORY variable used by the DFHIJVMJ job during CICS installation).

If you need to modify the library path used by the CICS JVM, edit the LIBPATH statement in your JVM profile.

Note: The LIBPATH statement in the default JVM profile, DFHJVMPR, already includes the /usr/lpp/cicsts/cicsts21/lib directory. If you are using the default profile no changes are necessary.

CICS installs the CICS Transaction Gateway for OS/390 Java classes in an HFS directory that is in the “trusted middleware” classpath used by the CICS JVM. (For CICS JVM purposes, the CICS Connector for CICS TS is classified as trusted middleware.)

To complete the installation, ensure that the ccf.jar file is in the trusted middleware classpath. The ccf.jar file contains utility classes required by the CTG for OS/390 Java classes. It is supplied with VisualAge for Java Enterprise Access Builder and as part of the EJB CICS sample application. If you have already installed the EJB CICS sample application you need do nothing more. If not, you can find the ccf.jar file in the EJB CICS sample directory on HFS. The default EJB CICS sample directory is:
/usr/lpp/cicsts/cicsts21/samples/bankaccount

where cicsts21 is the value of the CICS_DIRECTORY variable used by the DFHIJVMJ job during CICS installation.

Note: The recommended way to add files and directories to the trusted middleware classpath is to specify them on either the TMPREFIX or TMSUFFIX initialization option in your JVM profile. For information about TMPREFIX and TMSUFFIX, see the CICS System Definition Guide.

CICS Connector for CICS TS messages
CICS messages related to the connector are in the range DFHCZ0150—DFHCZ0159. These are described in the CICS Messages and Codes manual.

Tracing the CICS Connector for CICS TS
Trace data can be output from either or both parts of the CICS Connector for CICS TS:
1. From the library of CICS native functions, libCTGJNI.so. Use this trace information to help debug a failure within CICS.
2. From the CICS Transaction Gateway for OS/390 Java classes. Use this trace information to help debug a failure within the connector classes.

CICS trace
The CICS trace points related to the connector are in the range AP 21E7—AP 21E9. 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.

CTG for OS/390 trace
How to switch on CICS Transaction Gateway for OS/390 tracing for the connector depends on which of the connector’s interfaces you are using—the CCF interface or the CTG API.

Tracing the CCF interface
How to switch on CICS Transaction Gateway for OS/390 tracing when using the connector’s CCF interface is described in the HTML documentation supplied with VisualAge for Java Enterprise Access Builder.

Tracing the CTG API
To switch on CICS Transaction Gateway for OS/390 tracing for the CTG API, append the following lines to the JVM properties file (the file pointed to by the JVMPROPS statement in your JVM profile).
gateway.T=on
gateway.T.entry=on
gateway.T.lines=on
gateway.T.exit=on
gateway.T.stack=on
gateway.T.trace=on
gateway.T.timing=on

Notes:
1. You can set some of these switches to off, as required. The meanings of the switches are as follows:
gateway.T=on/off
  Set all debugging on or off.
gateway.T.entry=on/off
  Set entry points on or off.
gateway.T.lines=on/off
  Set general lines on or off.
gateway.T.exit=on/off
  Set exit points on or off.
gateway.T.stack=on/off
  Set stack dumps on or off.
gateway.T.trace=on/off
  Set product-level tracing on or off.
gateway.T.timing=on/off
  Set timing on or off.

2. In the CICS-supplied sample JVM properties files, dfjjvmpr.props (pointed to by the default JVM profile, DFHJVMPR) and dfjjvmps.props, the above lines are already present, with each switch set to off. If you use one of these property files, simply turn on the switches that you require.

3. The switches could be set by the Java application program. For example:
   System.setProperty("gateway.T","on"), and so on.

When trace is switched on, trace records are written to stderr. By default—that is, unless the CICS JVM profile specifies otherwise—the JVM directs stderr to dfhjvmerr.applid.time.taskid.txt. The dfhjvmerr file is written to the sub-directory defined by the WORK_DIR entry in the JVM profile.
Chapter 20. 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. It includes the following topics:

- "CICS enterprise bean set-up problems"
- "CICS deployment tools’ online help"
- "CICS enterprise bean deployment problems"
- "Using EJB server runtime diagnostics" on page 224
- "Using EJB client runtime diagnostics" on page 225

See the CICS Problem Determination Guide for guidance on the more general aspects of CICS problem determination and diagnostics.

CICS enterprise bean set-up problems

If you have difficulties setting up the CICS EJB server, the problem could be related to your basic CICS Java set up. Try running the Java HelloWorld sample. If this also fails it points to a problem with the set up of your JVM rather than anything else.

CICS deployment tools’ online help

The CICS JAR Development Tool for EJB Technology includes online documentation in the form of HTML pages. When requested via the Help pulldown, these HTML pages are displayed using the default browser. If your default browser is Netscape, and the documentation fails to display, or displays with error messages, check the browser settings as follows:

1. Click Start->Settings->Folder Options.
2. Select the File Types tab.
3. Find the file type that has the HTML extension. It is registered file type Netscape Hypertext Document or similar.
4. Click Edit, then select the open action and click Edit.
5. Ensure that the Application field is set to NSShell (it may have been set to Netscape).
6. Click OK and Close.

CICS enterprise bean deployment problems

This section concerns possible problems in using CICS EJB deployment tools. See "Using CICS deployment tools for EJB technology" on page 173 for guidance on using these tools and "Chapter 17. Installing and configuring CICS deployment tools for EJB technology" on page 155 for guidance on installation.

You can download any service updates to the CICS EJB tools from the following IBM web site:

http://www.software.ibm.com/software/cics/support/

Error handling for the Development Deployment Tool

Problems in the CICS Development Deployment Tool are handled as follows. When the tool detects an error, it generates an exception. This causes the web application to issue a message, which includes any associated variables. Error conditions and error objects are traced to CICS if the error occurs in an enterprise bean. If the
error is in the web application, trace is sent to a file determined in the DCF and messages are sent to the WebSphere console.

Any problems in the CICS components of the development deployment tool, cause an error to be returned to the caller indicating the failing instructions and the associated RESP and RESP2 values. These values are displayed, or translated to a more meaningful message where the error could be expected (such as the DFHADJM file not being available, or there being no available DJAR names remaining).

All error conditions are traced along with any available data in order to aid debugging.

---

**Using EJB server runtime diagnostics**

This section includes the following topics:

- [CICS enterprise bean errors and messages](#)
- [JVM trace](#)
- "Debugging Java applications in CICS" on page 225

**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 is a facility to aid in the diagnosis of problems in the java virtual machine. It is controlled using the `ibm.dg.trc.external` system property, which is set in the JVMPROPS file for the Java program but should be used only under the direction of IBM support personnel. It has to be used with care as JVM trace can produce huge amounts of data in a very short time. Before running JVM trace, careful consideration must be given as to what information is required to resolve the problem.

JVM trace is issued within the CICS trace point ID AP 4D01. If the JVM trace facility fails, CICS issues the trace point AP 4D00.
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 in the CICS JVM" on page 47 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 website 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.

**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.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

**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.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.
Chapter 21. 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" for guidance on using this type of security control.

**Secure Sockets Layer (SSL) security**

The secure sockets layer (SSL) is an architecture allowing servers and clients that use TCP/IP to communicate to authenticate each other and to encrypt the data flowing between them. When a server uses SSL it must hold three pieces of data: a private key, a public key, and a server certificate. Before you can use SSL with CICS you need to create a RACF key ring, which contains these three items of data. For information about using SSL with enterprise beans see "Authentication of IIOP requests" on page 65 and see CICS RACF Security Guide to build a key ring in RACF.

**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 CICS RACF Security Guide.

---

**Protecting Java applications in CICS**

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.

A security policy is enforced by a security manager and is effective at JVM startup. Only one policy is in effect for the JVM at any given time, but this policy can be the result of processing one or more policy configuration files.

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. To protect a Java application from performing a potentially unsafe action, a security manager must be installed so that a security policy can be enforced. A security policy provides the security manager with a set of permissions (system access privileges) which are assigned to code sources. Every time the Java virtual machine (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.

A default security manager and policy are supplied with the Java 2 platform. The default security manager allows you to specify a security policy in a policy file that is separate from the application program. This section explains how the default security policy mechanism is used in CICS with the sample CICS enterprise beans security policy file.
Specifying JVM system properties

The Java 2 security policy is defined in terms of a series of JVM system properties. The system properties are specified in an HFS file referenced by the JVMPROPS parameter in the JVM profile.

The profile is a member of the partitioned data set specified by the DFHJVM data definition statement in the CICS JCL. The name of the member to be used as the JVM profile for CICS enterprise beans is specified in the JVMPROFILE attribute of the Java PROGRAM definition, or, for enterprise beans and IIOP applications, the definition of the initial program used by the request processor transaction definition (which is by default CIRP). This program definition is usually DFJIIRP, and the JVMPROFILE for it is usually DFHJVMPR. See “Defining resources for the JVM” on page 47.

Within the CICS JVM profile, you specify parameters such as:

**JVMPROPS**
  - which specifies the name of an HFS file that contains the JVM system properties

The JVM system properties themselves are contained within the HFS file referenced by the JVMPROPS parameter. CICS provides two sample properties files, dfjjvmpfr.props and dfjjvmpr.props. Examples of system properties you can specify are:

*java.security.manager*
  - which indicates the Java security manager to be used

*java.security.policy*
  - which describes the location of the Java security policy file

For example, if the CICS JCL contains

```
//DFHJVM DD DSN=CICSTS21.CICS.JAVAPROF,DISP=SHR
```

and the CICS-supplied definitions of CIRP and DFJIIRP are used, then the JVM profile is member CICSTS21.CICS.JAVAPROF(DFHJVMPR), which might contain the following:

```
JVMPROPS=/u/cicsts/dfjjvmpfr.props
```

while the `/u/cicsts/dfjjvmpfr.props` file might contain:

```
java.security.manager=default
java.security.policy=/usr/lpp/cicsts/cicsts21/lib/security/dfjejbpl.policy
```

Where *cicsts21* is your chosen value for the USSDIR installation parameter that you defined when you installed CICSTS.

Configuring the JVM to use the Java2 security policy

A security policy is enforced by a security manager and is effective at JVM startup. Only one policy is in effect for the JVM at any given time, but this policy can be the result of processing one or more policy configuration files. A default policy file is provided with the JVM in `/usr/lpp/java130s/J1.3/lib/security/java.policy`. Policy file locations are specified in the **security properties file** and are combined with any user policy files defined by the `java.security.policy` JVM system property. The security properties file is provided by the JVM in:
Where the java130s/J1.3 subdirectory names are the default values when you install the IBM Developer Kit for OS/390, Java 2 Technology Edition.

The following JVM system properties must be specified to enable CICS Java applications and enterprise beans to run with the Java 2 security mechanism in effect. The use of these properties is described in more detail in Enabling the default Java 2 security manager:

- java.security.manager
- java.security.policy

Note: If no security manager is specified, then by default, the JVM runs without Java 2 security enabled.

**Enabling the default Java 2 security manager**

To run an application with the default Java 2 security manager enabled in the CICS JVM, one of the following values must be specified in the JVM system properties file (all these values have the same effect):

```
java.security.manager=default
```

or

```
java.security.manager=""
```

or

```
java.security.manager=
```

If the `java.security.manager` property is specified without a security policy, the policy permissions supplied in the default Java security properties file are used. To specify a policy file that will be used in addition to the ones specified in this default security properties file, the `java.security.policy` property must also be specified, giving the name of the file containing the additional security policy. See Defining a security policy.

**Defining a security policy**

The security properties file, `java.security`, is a configuration file which defines the default Java security properties for the JVM. A user-defined security policy file may be applied in addition to, or by modifying, the policy files specified in the default security properties file.

Additional security policies can be enabled in either of the following ways:

- By specifying the additional policy file name in the `java.security.policy` system property passed to the JVM at runtime, or
- By adding a policy url entry in the default security properties file (`java.security`). See Policy file entries on page 230

Any security policy defined in the default security properties file (`java.security`) can be ignored by using a double equals sign (==) to specify the java.security policy JVM system property. For example:

```
java.security.policy==/usr/lpp/cicsts/cicsts21/lib/security/dfejbpl.policy
```
Policy file entries

Policy file entries in the java.security, security properties file, 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, which implies that 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 java.security file contains these two entries:

```
policy.url.1=file:${java.home}/lib/security/java.policy
policy.url.2=file:${user.home}/.java.policy
```

The following line would need to be added for the sample enterprise bean policy dfjejbpl to be considered. (Since two policy URLs are already defined by default in java.security):

```
policy.url.3=file:/usr/lpp/cicsts/cicsts21/lib/security/dfjejbpl.policy
```

For more information about Java 2 security, refer to the Java 2 documentation.

CICS-supplied enterprise beans security policy

To prevent unauthorized access to system resources by enterprise beans, you are recommended to use the the default Java security manager, configured to use the CICS-supplied enterprise beans security policy. Note that if you activate the security manager but do not provide a security policy, then the container code may become inaccessible, and therefore, no CorbaServers can be initialized.

The following JVM system properties can be defined to apply the supplied sample enterprise bean policy:

```
java.security.manager=default
java.security.policy=/usr/lpp/cicsts/cicsts21/lib/security/dfjejbpl.policy
```

Where `cicsts21` is your chosen value for the USSDIR installation parameter that you defined when you installed CICSTS.

This policy is based upon the security policy recommended in the Sun Microsystems's *Enterprise JavaBeans Specification, Version 1.1*, which is available at [http://www.javasoft.com/products/ejb](http://www.javasoft.com/products/ejb). This sample policy is shown in

The sample policy file will be loaded in addition to all the policy files specified in the JVM default security properties file (located in `/usr/lpp/java130s/J1.3/lib/security/java.security`), where the `java130s/J1.3` subdirectory names are the default values when you install the IBM Developer Kit for OS/390, Java 2 Technology Edition.

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 enterprise beans supplied 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 46.
2. Grants any code source only the permissions outlined in the Enterprise JavaBeans specification, Version 1. See the default ‘grant’ block in Figure 46:
   - 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

A security policy is stored in text format, and can be modified and displayed by any standard text editing tool. Users can modify the supplied policy or create their own. However, it is recommended that the policy file is made secure with update authority restricted to system administrators. See [Defining file access permissions for enterprise beans on page 232](#) for more information.

**Note:** The DB2 JDBC 1.2 driver does not support the Java 2 security policy mechanism. Therefore, CICS Java applications and enterprise beans that use JDBC or SQLJ with the Java security manager configured to use the CICS supplied security policy will fail.

```java
// permissions granted to CICS enterprise beans Container codesource protection domain
grant codeBase "file:usr/lpp/cicsts/cicsts21/-" {
    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 46. 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. Please note, CICS TS 2.1 does not support EJB security roles. Use of isCallerInRole() and method permission checks based on security role information defined in the XML deployment descriptor will always return true in CICS TS 2.1.

**java.security.Principal getCallerPrincipal()**

This method is used to determine who invoked the current bean method. The getCallerPrincipal method is fully supported in CICS TS 2.1 and details of the way that the identity of the current caller is determined are shown in [Deriving distinguished names on page 234](#).
boolean isCallerInRole(String rolename)

This method is used to test whether the current caller is assigned to a given security role. Use of this method in CICS TS 2.1 will always return TRUE, whether CICS security is active or not.

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 in CICS TS 2.1.

- java.security.Identity getCallerIdentity()
- boolean isCallerInRole(java.security.Identity role)

**Note:** Note that enterprise beans developed using VisualAge for Java V3.5, which supports only the Enterprise JavaBeans (EJB) 1.0 specification, need to be migrated to the Enterprise JavaBeans 1.1 specification level using the supplied JAR development tool [The CICS JAR development tool for EJB technology on page 175](#). See [Chapter 16. Writing enterprise beans on page 145](#) 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 [Authorizing CICS region userids to UNIX system services on page 44](#).
Access to HFS files used by enterprise beans

Table 10. File access permissions required for CICS enterprise beans

<table>
<thead>
<tr>
<th>File/Directory structure</th>
<th>Minimum permission</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORBASERVER Shelf directory (for example, /var/cicsts/)</td>
<td>Read, write and execute</td>
<td>The shelf is accessed during CORBASERVER and DJAR installation, and each CICS needs to create unique subdirectories (see note 1).</td>
</tr>
<tr>
<td>/usr/lpp/cicsts/cicsts21 directory structure and classes</td>
<td>Read and execute</td>
<td>Contains the CICS-supplied Java code (see note 2).</td>
</tr>
<tr>
<td>/usr/lpp/java130s/J1.3/bin and /usr/lpp/java130s/J1.3/bin/classid directories</td>
<td>Read and execute</td>
<td>Contain the IBM Java 2 persistent reusable JVM code (see note 3).</td>
</tr>
<tr>
<td>CICS working directory</td>
<td>Read, write and execute</td>
<td>Used to create stdin, stdout, and stderr files (see note 4).</td>
</tr>
<tr>
<td>Deployed jar file</td>
<td>Read</td>
<td>Used during DJAR installation by the deployment process.</td>
</tr>
<tr>
<td>Security policy file (if required)</td>
<td>Read</td>
<td>Required if the java.security.policy property is specified in the JVM system properties file.</td>
</tr>
<tr>
<td>System properties file</td>
<td>Read</td>
<td>Required by CICS when creating a JVM (see note 5).</td>
</tr>
</tbody>
</table>

Notes:
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. cicsts21 is your chosen value for the USSDIR installation parameter that you defined when you installed CICSTS.
3. The java130s/J1.3 subdirectory names are the default values when you install the IBM Developer Kit for OS/390, Java 2 Technology Edition.
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 the OS/390 UNIX System Services Command Reference, SC28–1892.

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 47 on page 234 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 'CICSTS21.CICS.DFHEJDIR' NOTIFY(cics_sys_admin_id) UACC(NONE)
PERMIT 'CICSTS21.CICS.DFHEJDIR' ID(cics_id1,...,cics_group1,...,cics_groupn)
ACCESS(UPDATE)

ADDSD 'CICSTS21.CICS.DFHEJOS' NOTIFY(cics_sys_admin_id) UACC(NONE)
PERMIT 'CICSTS21.CICS.DFHEJOS' ID(cics_id1,...,cics_group1,...,cics_groupn)
ACCESS(UPDATE)

Figure 47. An example of commands to authorize access to CICS data sets

See [CICS RACF Security Guide](#) for information about using RACF.

**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 TS 2.1 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, `CICSTS210.CICS.SDFHSAMP`, as:

- DFHEJDN1 for Assembler language
- DFHEJDN2 for C language
Chapter 22. CICSPlex SM with Enterprise JavaBeans

This chapter describes the following:

- "CICSPlex SM support for enterprise beans"
- "CICSPlex SM definition support for enterprise beans"
- "BAS logical scope considerations" on page 238
- "Enterprise JavaBean component migration" on page 239
- "CICSPlex SM inquiry support for enterprise beans" on page 233
- "Types of inquiry available for enterprise bean objects" on page 240
- "Using CICSPlex SM to manage Enterprise JavaBean workloads" on page 240
- "Workload balancing" on page 241
- "Workload separation" on page 241
- "CICSPlex SM resource monitoring considerations for Enterprise JavaBeans" on page 242
- "CICSPlex SM real-time analysis considerations for Enterprise JavaBeans" on page 242

CICSPlex SM support for enterprise beans

The management of enterprise beans may be undertaken at a CICSpelix 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 JavaBean execution

The CICSPlex SM areas that cover these facilities are:

- The application programming interface (API) - to allow the definition, enquiry and management of Enterprise JavaBean 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 JavaBean 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 JavaBean 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 CPSM Managing Business Applications. The BAS objects that are specific to Enterprise JavaBeans are:

- EJCODEF - Enterprise JavaBean CorbaServer definition
- EJDJDEFF - Enterprise JavaBean 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 the 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 the *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 JavaBean operation:

- TCPDEF - TCP/IP Service definition
- RQMDEF - Request Model definition
- TRANDEF - CICS Transaction definition
- PROGDEF - Program definition

Enterprise bean execution requests from Enterprise JavaBean 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 72 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 82 for information.

In a distributed Enterprise JavaBean 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 Enterprise JavaBean 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 the *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 JavaBean objects, separate DJARs should be employed to contain enterprise beans discrete to the scoped business applications.

**Enterprise JavaBean component migration**

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 Enterprise JavaBean 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 237. 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** - Enterprise JavaBean CorbaServer instances
- **EJDJAR** - Enterprise JavaBean 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 JavaBeans 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 - TCP/IP Service instances
- RQMODEL - Request Model instances
- LOCTRAN - Local Transaction instances
- UOWORK - Unit of Work instances
- UOWLINK - Unit of Work Link instances
- PROGRAM - Program instance

All of these objects include attributes which have relevance to the management and execution of Enterprise JavaBeans.

Types of inquiry available for enterprise bean objects

As stated previously, there are three paths of inquiry regarding the state of your Enterprise JavaBean 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 Enterprise JavaBean 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 Enterprise JavaBean 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 Enterprise JavaBean 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 JavaBean, 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 Enterprise JavaBean environment
- The enterprise beans have been deployed and are INSERVICE

When the listed criteria have been met, the implementation of Enterprise JavaBean 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 Enterprise JavaBean 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 Enterprise JavaBean 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 JavaBeans

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 Enterprise JavaBean 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 241. 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 JavaBeans

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 CICSpex 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. It covers the following topics:

- "Chapter 23. Stateless CORBA objects" on page 247
- "Developing stateless CORBA objects" on page 247
Chapter 23. Stateless CORBA objects

From the client point of view, a stateless CORBA object in a CICS IIOP ORB is just a collection of methods, that is, a stateless object. Each method represents a piece of logic that may make one or more CICS API calls, including CICS LINKs, to existing CICS programs. At the end of the method, no data is stored in attributes.

This implies that every method must be passed sufficient information in its parameter list to enable it to complete its work. No information is passed to the server by virtue of the object reference, except the object type, which is used to find the class and methods of the implementation. 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.

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 transaction flows as an extra parameter on the IIOP call. If a target stateless CORBA object implements CosTransactions::TransactionalObject, then the object will be treated as transactional.

CICS stateless CORBA objects are supported in JVM mode only. The VisualAge for Java, Enterprise Edition for OS/390 bytecode binder cannot be used.

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 invocations of methods; state is initialized at the start of each method call and referenced by explicit parameters.

In the server programming model, each method is a subroutine. The parameters passed allow you to establish temporary variables from various existing databases or applications, to perform business logic, to store data in the existing databases or applications, and to return results when the subroutine returns.

To develop a stateless CORBA object, you need to perform the following steps:

- Write the Interface Definition Language (IDL) defining interfaces and operations
- Write a client program that makes calls to the server based on the IDL definition
- Write a server program (the stateless CORBA object) that implements the interfaces defined in the IDL
- Define CICS resources for the server execution

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 JCICS Javadoc class Reference for details of the JCICS classes, and Chapter 4. Java programming in CICS on page 13 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, and can be found at JCICS Javadoc class Reference.
Obtaining an interoperable object reference

In order to access a server object, the client program requires a reference to it. In a distributed environment, an object reference, known as an Interoperable Object Reference (IOR), is more than just a storage address obtained using new. 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.

You should use the PERFORM CORBASERVER PUBLISH command to publish the CORBASERVER resource definition defining the execution environment for this IIOP request. A stringified IOR of the GenericFactory class is then created and stored on the shelf (an HFS directory associated with the CorbaServer), and published to the nameserver.

You can use the CICS CEMT master terminal command ([CICS Supplied Transactions](#)) to issue the PERFORM command, or you can issue the EXEC CICS PERFORM ([CICS System Programming Reference](#)) command from a CICS application.

The genfac.ior file is written to the CORBASERVER's shelf directory:

/shelf/applid/corbaserver/

where:

- **shelf** is the SHELF directory name specified in the CORBASERVER resource definition, defaulting to /var/cicsts/
- **applid** is the 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, or your client can use the JNDI interface to obtain the IOR from the nameserver.

An implementation of the CosLifeCycle GenericFactory is provided for object creation. (Note that for a stateless object, the GenericFactory is completely adequate; there is no value in allowing more powerful factories such as application specific factories).

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)

Before you write a CORBA client or a stateless CORBA object server application, you must first 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.

The proxies and skeletons provide the object-specific information needed for an ORB to distribute a method invocation.

**Note:** To build the skeleton code, you should use the IDL compiler "idlj" command which is part of the SDK.

The following diagram shows how the same IDL file is used to generate different classes used by the client and the server.
Developing an IIOP server program

The server program can be developed on any platform that supports Java. For example, an NT workstation, AIX®, or the OS/390 Unix Systems Services environment. 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 SDK.

The IDL compiler can be invoked as follows:

```java
java 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.

- `-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.

-m Generate information to be included in a make description file; output goes to a .u file.

-sep <string>
Replace the file separator character with <string> in the file names listed in the .u file, if -m is specified.

-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.

-bean Generate classes that can be used as Java beans.

-stateful
Parse stateful interface objects (used for Objects-by-value). Note that this is non-standard IDL and is not supported by CICS.

3. Write your server implementation in Java code. The idl compiler will generate an abstract class called _interfacenameImplBase. Your program must extend this. If objects of this type are to be created by the Generic Factory, it must be called _interfacenameImpl. For example:

    public class _BankAccountImpl extends _BankAccountImplBase

This requires the CORBA classes from dfjorb.jar and may use the CICS API Java classes from dfjcics.jar supplied by CICS.

dfjcics.jar and dfjorb.jar are stored in the OS/390 UNIX System Services HFS during installation of CICS, in the following directory:

    /usr/lpp/cicsts/<username>/lib

Where username is a name you can choose during CICS installation, defaulting to cicsts21

4. Compile your program and the output from step 2, with the javac compiler or an equivalent, such as VisualAge for Java, with the following files in your CLASSPATH:

    • dfjorb.jar
    • dfjcics.jar (if required)
Ensure that the location of the output files is added to the end of the 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:

```java
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 `bank` Java package. You can see full details of this implementation in the `BankAccount` sample application distributed in:

```
/usr/lpp/cicsts/<username>/samples/dfjcorb
```

Where `username` is a name you can choose during CICS installation, defaulting to `cicsts21`.

**Resource definition for example**

The following REQUESTMODEL example associates the inbound request with a TRANSID that gives the request the right execution characteristics.

```java
DEFINE REQUESTMODEL(DFJ$IIRB)
    GROUP(DFH$IIOP)
    CORBASERVER(IIOP)
    TYPE(Corba)
    DESCRIPTION(Bank account sample)
    MODULE(bank)
    INTERFACE(BankAccount*)
    OPERATION(*)
    TRANSID(BNKS)
```

The BNKS transaction defines execution characteristics for query and update requests received using IIOP. It runs the DFJIIRP program, which invokes the methods in `_BankAcctImpl`. 

252  CICS Transaction Server: Java applications in CICS
Developing the IIOP client program

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 it (in ascii mode) from the shelf defined in the CICS CORBASERVER, where it was created when the CORBASERVER resource definition for this request type was installed.

   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 nameservers" on page 72. 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/OrgFactory

   See CICS Resource Definition Guide and CICS Supplied Transactions.

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.

Note: You need dfjorb.jar in the CLASSPATH when generating server side (CICS) applications, and your client ORB vendor’s classes in the CLASSPATH when generating client side applications.

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, then you can build them using the client ORB's IDL-to-Java compiler, from the IDL given in the CORBA specification, or alternatively, use the IDL subset provided in /usr/lpp/cicsts/<cicsts21>/samples/dfjcorb/

Where cicsts21 is your chosen value for the USSDIR installation parameter that you defined when you installed CICSTS.

These bindings should be downloaded in ascii mode.

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.

```java
import java.io.*;
import org.omg.CORBA.*;
import org.omg.CosLifeCycle.*;
import org.omg.CosNaming.*;
public class bankLineModeClient{
```

Chapter 23. Stateless CORBA objects 253
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.
Chapter 24. Migrating IIOP applications from CICS TS 1.3

CICS has implemented an enhanced CORBA ORB in CICS TS 2.1. This means that you can exploit some new function, but you will also need to make some changes to existing IIOP applications, or the execution environment. You will need to make the following changes:

Environment
CICS has replaced dfjcorb.jar by dfjorb.jar in CICS TS for z/OS Version 2.1. See "Chapter 10. Configuring CICS for IIOP" on page 71 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 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 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 62 for more information about using DNS.

There are new SSL options. See "Authentication of IIOP requests" on page 65 for more information about the use of SSL. See 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 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 10, Configuring CICS for IIOP on page 71 for more information about setting up your IIOP environment.

Security URM
You need to change any IIOP security user-replaceable modules (URMs) 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 USERID on page 79 for more information about supplying a URM.

IDL
CICS does not provide dfjcidl.jar in CICS for z/OS Version 2.1. Instead you should 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 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 name server. All existing stringified IOR files need to be recreated. For more information, see Defining nameservers on page 72.

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 new request streams logic manages these messages in a different way in CICS TS 2.1, and these TS queues are no longer needed.

JVM
IIOP applications execute in the JVM; they cannot be 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 7, Configuring the JVM on page 47, and define your programs as JVM programs.
Chapter 25. Using the IIOP samples

The following sample applications demonstrate the use of IIOP applications (stateless CORBA objects) and the CICS Java programming support (JCICS). They execute only with the CICS JVM and will not run with the VisualAge for Java, Enterprise Edition for OS/390:

**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:
- An OS/390 Version 2.8 system or later, with UNIX Systems Services and HFS
- Language Environment configured and active
- CICS TS Release 2.1
- The IBM Developer Kit for OS/390, Java 2 Technology Edition, with the special enhancement that provides a persistent, reusable Java Virtual Machine (JVM).
  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

   **MAXOPENTCBS**
   
   40 minimum is recommended

   **TCP/IP**
   
   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.

   **DFHJVM**
   
   A partitioned dataset containing the JVM profiles that set environment variables to control the initialization of the Java Virtual Machine (JVM). See "Chapter 7. Configuring the JVM on page 47" for more information about the content of JVM profiles and setup requirements for the JVM.

   Sample local VSAM data set definitions for these files are provided in the CICS-supplied RDO group DFHEJVS. These data sets must authorized with RACF for UPDATE access. See CICS RACF Security Guide for guidance.

4. Setup the JVM as described in "Chapter 7. Configuring the JVM on page 47".

5. Create a shelf directory on HFS and give the CICS region userid full access to it. See "Authorizing CICS region userids to UNIX system services" on page 44 for guidance.

6. Define the following environment variables in the profile of the server side application:

   **CICS_HOME**
   
   The installation directory prefix of CICS TS:

   /usr/lpp/cicsts/cicsts21/

   where cicsts21 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/java130s/J1.3/
where `/java130s/J1.3` is defined when you install the IBM Developer Kit for OS/390.

7. Ensure that the following files are in your CLASSPATH:
   - The sample Java source and makefiles that are stored in the OS/390 UNIX System Services HFS during CICS installation, in the following directories:
     - `$CICS_HOME/samples/dfjorb/HelloWorld`
     - `$CICS_HOME/samples/dfjorb/BankAccount`
   - The location where you have compiled the classes for the server side applications.

8. The CICS supplied groups DFHIIOP and DFH$IIOP must be installed before you run the samples. 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.

   **Note:** Before using CEDA, you may need to use CEOT to enable your terminal to handle mixed case input— for details, see `CICS Supplied Transactions`.

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

and 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 changed 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 `CICS Resource Definition Guide` and `CICS Resource Definition Guide`.

9.
Translate and compile the following CICS C language programs and mapset and include them in a library in the CICS DFHRPL concatenation. They are stored in SDFHSAMP during CICS installation. The order of compilation is important. Both DFH$IIIBI and DFH$IIICC can be compiled independently, but the BMS mapset DFH$IIIMA must be compiled before compiling DFH$IIIBQ. See [CICS Application Programming Guide](#) for guidance on translating, compiling and linking CICS application programs. The file DFH$IIIMA contains one mapset BANKINQ with two maps. Compile and link the mapset BANKINQ. See [CICS Application Programming Guide](#) for guidance on compiling and linking BMS maps.

**DFH$IIIBI**
C program that initializes the BANKACCT file. Run by the BNKI transaction.

**DFH$IIIBQ**
C program that queries the accounts held in BANKACCT.

**DFH$IIICC**
C program that performs a credit check. This is called by DFH$IIIBQ.

**DFH$IIIMA**
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. Obtain Java bindings for part of the CORBA CosLifeCycle and CosNaming modules. If they are not provided by the client ORB, then you can build them using the client ORB’s IDL-to-Java compiler, from the IDL given in the CORBA specification, or alternatively, use the IDL subsets provided in `$CICS_HOME/samples/dfjcorb/`. If these files are to be used on the client, they need to be downloaded as ASCII.

Note: You may need to change the import statements in the client code to correspond with the package name of the bindings generated by your ORB’s IDL compiler. Alternatively, use your client ORB IDL compiler’s equivalent of the `-pkgPrefix` option to set the package name to that required by the Java program’s import statement.

11. You will need to obtain a `genfac.ior` file containing an object reference to your server’s generic factory, and place it in the current directory. This is created when you issue a PERFORM CORBASERVER PUBLISH command for the installed sample IIOP CORBASERVER resource definition. The genfac.ior file 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 (CICS Supplied Transactions) to issue the PERFORM command, or you can issue the EXEC CICS PERFORM (CICS System Programming Reference) command 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"
- "Building the client side HelloWorld application"
- "Running the HelloWorld sample application"

Building the server side HelloWorld application

The makefile in $CICS_HOME/samples/dfjcorb/HelloWorld/server builds everything required for the server side application.

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. After compiling the IDL to create the sub-directory, hello, the java file must be moved into this sub-directory. Then, this can be compiled from the current directory, as follows:
   
   ```
javac hello\HelloWorldClient.java
   ```

3. Compile the client application, ensuring that the Java classes produced in the previous step are available through the CLASSPATH environment variable. $CICS_HOME/samples/dfjcorb/HelloWorld/server should be added to the end of the classpath, ibm.jvm.shareable.application.class.path, in the default JVM properties file, dfjjvmpr.file.

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 263

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.

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. $CICS_HOME/samples/dfjcorb/BankAccount/server should be added to the end of the classpath, ibm.jvm.shareable.application.class.path, in the default JVM properties file, dfjjvmpr.file
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
   java bank.bankLineModeClient
   ```

This Part tells you what you need to know to use VisualAge for Java, Enterprise Edition for OS/390. It covers the following topics:

- “Chapter 26. VisualAge for Java, ET/390” on page 267
- “Chapter 27. Java hot-pooling concepts” on page 271

The normal CICS program execution model is used, where the environment is initialized for every task, rather than a Java Virtual Machine (JVM). This Java language application support is similar to CICS language support for COBOL or C++.

You can develop Java programs for CICS on a workstation, or in the OS/390 UNIX System Services shell, using an editor of your choice, or in a visual composition environment such as VisualAge. You then compile your program using a compiler such as VisualAge for Java, or javac. The Java byte-code produced by the compiler is then transferred (if necessary) to OS/390 UNIX System Services, and processed by the ET/390 bytecode binder, to produce an OS/390 Java executable file (jll or exe) that is called a Java program object. The Java program object is stored in OS/390 PDSE libraries and can be loaded and executed by CICS.

CICS loads the program object from the PDSE and executes it in a Language Environment (LE) run-unit, or enclave, similarly to C++, using run-time support in the CICS region provided by the Java run-time component of ET/390.

To improve performance, frequently used Java program objects can be run in a preinitialized OS/390 Language Environment enclave that is reused by multiple invocations of the program. The program is not executed under control of the quasi-reentrant (QR) OS/390 task control block (TCB) used by other CICS tasks, but uses a special class of TCB, known as an H8 TCB. This feature is known as hot-pooling.

See [Chapter 27. Java hot-pooling concepts on page 271] for more information about hot-pooling.

The following diagram shows the development process and components involved in creating a Java program object using VisualAge for Java, Enterprise Edition for OS/390.
Note: CICS TS for z/OS Version 2 supports hpj-bound Java program objects that were developed using CICS TS OS/390 Version 1 Release 3 and its associated tooling. Such program objects will run in CICS TS for z/OS Version 2 unchanged, but CICS TS for z/OS Version 2 CICS TS V2 provides no support for developing new Java program objects nor for modifying existing Java program objects. See the CICS TS OS/390 Version 1 Release 3 CICS Application Programming Guide for information about developing Java program objects for CICS.

Running a Java program object in CICS

Java program objects can be invoked in CICS:
- By EXEC CICS LINK commands issued by other CICS non-Java programs
- By other Java programs using a link method on a program.
- By entering a TRANSID on a CICS-attached terminal, such as a 3270

Run-time requirements

Language Environment
Language Environment (LE) is required to execute Java programs bound by the ET/390 binder.

Storage
Memory requirements to run Java programs using ET/390 are higher than for conventional programs. You should ask your CICS systems programmer to set the system initialization parameter EDSALIM to a high value (such as 100MB) when starting CICS, otherwise a Short-on-Storage condition may occur. Note that this must be set in the startup job control commands and can not be changed during CICS execution using CEMT SET commands.

DFHRPL
The following libraries must be concatenated with DFHRPL at run-time:
- PDSEs containing your Java program objects, bound with the ET/390 binder
- PDSEs containing Java resource files (such as JavaBeans)
• The **SDFJLOAD** PDSE library, containing the JCICS run-time support. These programs are stored in SDFJLOAD, and bound with the ET/390 binder, during installation of CICS.

• The **HPO.SHPOMOD** PDSE containing the ET/390 run-time library. This PDSE is built during the installation of VisualAge for Java, Enterprise Edition for OS/390

**Resource Definition**

All program resources must be defined to CICS using **CEDA**, and **INSTALL**ed before use (or autoinstalled). See the **CICS Resource Definition Guide** for information about defining CICS resources.

**Note:** Before using CEDA, you may need to use **CEOT** to enable your terminal to handle mixed case input— for details, see [CICS Supplied Transactions](#).

• Define a TRANSACTION, PROGRAM and GROUP for your executable program object. User programs must be defined to CICS by the short 8-byte name, as LE PROGRAMs. For example, using the CICS supplied CEDA transaction:

  `ceda define transaction (andy) group(newgroup) program(cichell)`
  `ceda define program(cichell) group(newgroup)`

• VisualAge for Java, Enterprise Edition for OS/390 run-time dlls must be defined to CICS. Member HPOSCSD in the SHPOJCL dataset supplied with ET/390 should be used as input to DFHCSDUP to define the ET/390 run-time dlls. The HPOJAVA predefined group is supplied. You should add it to GRPLIST or install it before using ET/390.

• JCICS run-time programs are defined in the DFHJAVA group. This is already included in DFHLIST.

• Install the group, for example:

  `ceda i group(newgroup)`
Chapter 27. Java hot-pooling concepts

When you execute a Java program object that has been bound by the VisualAge for Java ET/390 bytecode binder (hpj), the Language Environment run-unit or enclave is built and initialized for each invocation. You can reduce this performance overhead for frequently-run Java program objects by requesting that a preinitialized and persistent enclave is reused for multiple invocations of the program.

This feature is known as hot-pooling.

CICS uses the PIPI preinitialization services of OS/390 Language Environment to build the enclave, and executes the Java program object in the CICS region under the control of an open transaction environment (OTE) task control block (TCB) that is reserved for hot-pooling requests. Work done under these TCBs (known as H8 mode TCBs) runs concurrently with work done under other OTE TCBs, and the usual CICS quasi-reentrant TCB (QR) where most CICS transactions are executed.

See the OS/390 Language Environment Programming Guide SC28-1939 for a description of the PIPI services.

Open transaction environment TCBs

In earlier releases of CICS, user applications operate in a restricted, or closed, environment. Although the applications can use the functionally-rich CICS programming interfaces, direct invocation of other services is not supported. This is because CICS runs user transactions under a single OS/390 TCB, known as the CICS quasi-reentrant (QR) TCB. Direct invocation of other services outside the scope of the CICS permitted interfaces could interfere with CICS own use of its QR TCB. In particular, services that result in the suspension (“blocking”) of the QR TCB would cause all CICS tasks to wait.

In CICS TS 1.3, the open transaction environment (OTE) was introduced, providing support for a new type of open TCB, allowing CICS transactions to use non-CICS services within the CICS address space, without interference with other transactions.

OTE distinguishes several kinds of open TCB modes, each given 2-character identifiers. For example, J8 mode TCBs are created to run interpretive Java programs under a Java Virtual Machine (JVM) and H8 mode TCBs are created to run hot-pooled Java program objects. Each mode has a specific purpose, and is handled by CICS in a different way. CICS decides which mode to use, not the application program. A CICS task is allowed at most one TCB of each mode, which it keeps from the time it is allocated to the end of the task. The TCB then becomes free, and CICS may allocate it to another task, or destroy it.

The CICS system initialization parameter MAXOPENTCBS limits the total number of OTE TCBs of all modes. There can be any number of H8 mode TCBs up to the MAXOPENTCBS limit, as long as CICS has enough storage to meet their demands.

The following Venn diagram shows the possible states for H8 mode TCBs, and how they relate to other OTE TCBs. An H8 mode TCB is allocated and active if a program is running on it; it is idle if the program has ended but the TCB still belongs to the task.
H8 mode TCB allocation for hot-pooling

The following steps show how CICS chooses an H8 mode TCB for an invocation of program progname in transaction transid. CICS associates progname and transid with an H8 mode TCB when it is allocated. In choosing a TCB to allocate, CICS searches first for an exact match of transid and progname, then attempts to create a new TCB, then accepts a best match on progname only, then accepts no match at all.

1. If the transaction already has an H8 mode TCB allocated, it is used.
2. If there is a free H8 mode TCB, which has both progname and transid, it is allocated. This is an exact match.
3. If there are currently fewer than MAXOPENTCBS, a new H8 mode TCB is created, and associated with progname and transid. This is new.
4. If there is a free H8 TCB that matches progname only, it is used. This is a partial match.
5. If there is a free H8 TCB that does not match progname or transid, it is used. This is no match.
6. If there is a free OTE TCB of another mode, it is destroyed and an H8 TCB is created. This is called stealing and, although costly on performance, it allows the request to be satisfied.
7. If there is no free TCB of any mode, the task is suspended until one becomes free.

Note: This allocation sequence implies the following possibilities:

1. There can be several H8 mode TCBs associated with the same program and transaction names. This is useful when there is a high throughput of the same transaction.
2. There can be H8 mode TCBs with several independent programs loaded into their enclaves, which are therefore capable of running one of several transactions.
3. A long-running task can keep an idle TCB from being used by other tasks.

**Using hot-pooling with ET/390**

A hot-pooled enclave is initialized the first time that CICS attempts to load a Java program object that has been defined with the Hotpool attribute. An H8 TCB is allocated to execute the program, as described above. Subsequent invocations of the same program can reuse the enclave, but not, for example, if the original transaction is still active, or the subsequent programs are running under different trandis. Subsequent invocations can run under the same TCB until it is terminated for one of the following reasons:

- The Language Environment storage *heap* grows as programs reuse the enclave until the predefined *heap* size limit approaches. CICS then terminates the enclave when the current program completes. You can extend this threshold by adjusting the *heap* size in the Language Environment RUNOPTS options that are used at enclave creation. CICS provides a user replaceable module, DFHAP8HO, which you can modify to reset RUNOPTS options. See [Defining Language Environment run-time options](#) for more information about DFHAP8HO.

- When CICS recognizes a short-on-storage (SOS) condition, each H8 TCB is terminated when the owning task ends.

- When SET PROGRAM NEWCOPY, PHASEIN, NOTHOTPOOL, or DISABLED is issued for any program (even non-Java programs), all H8 TCBs are terminated when the current task ends, because CICS does not track all programs that are loaded into an enclave. Subsequent invocation of a HOTPOOL program will cause a new hot-pooled enclave to be built and a new H8 TCB allocated.

- The program running on the TCB abends.

**Defining Language Environment run-time options**

Language Environment run-time options are parameters used during enclave initialization to control certain aspects of the program execution. Default values are set during installation, but you can change these for hot-pooling initialization using the user supplied module DFHAP8HO. This module is called during initialization of the Language Environment PIPI enclave. See the OS/390 Language Environment Programming Guide SC28-1939, and the source of DFHAP8HO in DFHSAMP for details of the Language Environment options that can be reset.

See also the comments in the CICS-supplied DFHAP8HO module for examples of how to set these options.

**Note:** Java Garbage collection is always set to ‘OFF’ for hot-pooled Java program objects.

See the [CICS Customization Guide](#) for more information about user replaceable modules.

CICS programs can include a CEEUOPTS CSECT to supply Language Environment run-time options to control the program’s execution. CEEUOPTS is ignored for programs run in a hot-pooling environment, because the enclave is preinitialized and run-time options only take effect during initialization.

**Allocating storage**

The number of H8 TCBs that can be supported in any one CICS region is dependent on the amount of storage available. The main restriction is below-the-line...
storage. Using the default LE options set in DFHAPH80, an H8 TCB and its
associated LE Enclave storage requires 20K of below the line DSA and 6K of below
the line MVS storage. This storage is in addition to any other current requirements.

You can calculate the number of TCBs that your CICS region can support by
looking at the current peak storage requirements. You can do this by running the
supplied STAT transaction, which shows storage values that can be used to
calculate the current availability of DSA and MVS storage. For example:

- Current DSA Limit ........ 6,144K (set by DSALIM in the SIT)
- Current DSA Used .......... 3,596K

(DSA available) = (Current DSA Limit) - (Current DSA Used) = 6144K - 3596K = 2548K

The above figures show the amount of CICS DSA storage available to be used for
TCBs (at 20K per TCB). You should not use all of this available storage when
calculating how many TCBs can be supported. Leave a buffer of about 500K. For
the example shown we would recommend a maximum of 100 TCBs.

You must also ensure that there is enough MVS storage to support the requirement
of 6K per TCB (600K for our example) in the private region. This storage can also
be displayed by the STAT transaction as shown below:

- Private Area storage available below 16Mb ..........: 1,990K

In this case there is 1990K available which will support the 600K requirement for
100 TCBs in our example. Again do not use all of the available storage in this area,
leave a buffer of 500K. The number of TCBs needs to be adjusted so that you do
not over commit storage demands in either the DSA or the MVS private region. You
may find that there is not enough available DSA because the DSA limit has been
set too low. Using the above statistics you can calculate how much higher the DSA
limit can be set. As you increase the DSA limit, the Private Area storage below
16Mb, decreases. Adjust the DSA limit to get the right balance between DSA and
Private area storage. Remember to take these statistics at peak time because there
are other users such as DB2 who take storage from this Private area.

In summary, adjust the DSA limit the best you can and then set the number of
TCBs to the lesser of (DSA available - 500K)/20K or (Private available - 500K)/6K.
If you still need more TCBs to satisfy your throughput requirements, you may need
to spread the applications across multiple CICS regions.

**Defining hot-pooled Java program objects**

You define a Java program object as eligible for hot-pooling by specifying the
Hotpool (Yes) attribute on the PROGRAM resource definition.

The Hotpool attribute specifies whether a Java program object main program is to
be run in a preinitialized enclave under an H8 TCB. The default is NO, which
causes the Java program object to be run under the QR TCB. Programs defined
with Hotpool must be built as executables, not load libraries.

If Hotpool (Yes) is specified, then the following PROGRAM attributes are ignored.
- JVM attributes
- RELOAD
- USAGE
- EXECKEY
- REMOTENNAME
- REMOTESYSTEM

274  CICS Transaction Server: Java applications in CICS
The Hotpool option is not supported by autoinstall. It is not possible to modify the Hotpool option in the autoinstall URM, and also any Hotpool option in the autoinstall model will be ignored. In effect, all autoinstalled programs are treated as Hotpool(No).

You are recommended to define hot-pooled programs in advance and pre-load them before they are required, to achieve better performance.

**THREADSAFE**
Programs must be THREADSAFE to use Java hot-pooling. Any native programs called by hot-pooled Java programs must be defined as THREADSAFE.

### Programming for Java hot-pooling
Because of the way that hot-pooled programs are run and reused in a preinitialized environment, there are some restrictions and modification you may need to make.

### Static storage and static initializers
Some Java programs may need modification for use with hot-pooling, as hot-pooling keeps programs loaded in an enclave and reuses them without reinitializing the static storage for each invocation of the program. This means that your program must reinitialize its own static storage, if it depends on the state of a changeable class field. Static initializers are run only at load time and are not rerun for each program invocation.

With hot-pooling, class variables exist through a number of CICS tasks, and their value may not be predictable. This is true for class variables in all classes, both application and system classes, and includes classes which may affect the application, but are not used explicitly (including those used in static initializers). For example:

- A stock exchange program may 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 will use the new default time-zone, which may not be appropriate.
- A program may update the system properties object in java.lang.System (if not prohibited by a security manager). Subsequent invocations of the program will use the new properties which may not be appropriate in this case.

### Avoiding static storage reuse problems
Review and modify existing programs to identify and eliminate changeable class fields and static initializers. 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.

**Note:** CICS provides a security manager, so you cannot use your own to restrict access to changeable fields.
Programming interfaces

There are some restrictions when using JCICS classes in a hot-pooled Java program, or when issuing EXEC CICS LINK, LOAD, or XCTL for a hot-pooled Java program object. These restrictions are:

LINK

If you link to a program defined with Hotpool (Yes), it will be run under an H8 TCB.

You cannot nest links to hot-pooled Java program objects. INVREQ is returned if you attempt to link to a hot-pooled Java program object while there is already a hot-pooled Java program object on the link stack.

LOAD

If you issue an EXEC CICS LOAD for a hot-pooled Java program object, the command is executed, but the requested program is loaded into the current enclave. The Hotpool attribute is ignored.

XCTL

If you transfer control to a program defined with Hotpool (Yes), it will be run under an H8 TCB. INVREQ is returned if you attempt to transfer control to a hot-pooled Java program object while there is already a hot-pooled Java program object on the program stack.

Protection keys

Hot-pooled Java program objects can only run in key 8, non-subspace mode. All storage acquired in the enclave by Language Environment is in key 8. The TRANSACTION attribute TASKDATAKEY(USER) has no effect.

A hot-pooled Java program object can use JCICS to link to a key 9 program.

Exit programming interface (XPI)

The loader ACQUIRE_PROGRAM call can be used with hot-pooled Java program objects, but the Hotpool PROGRAM attribute is ignored.

The value of the Hotpool attribute can be retrieved and set using the HOTPOOL keyword of the INQUIRE_PROGRAM and SET_PROGRAM calls.

Hot-pooling exit program

The DFHJHPAT user replaceable module is called before each program is started. This is an optional program that you can use for your own purposes, such as tracing. The module must be called DFHJHPAT and must be written in the C language. The following parameters are passed to this module when it is called:

```
int jhpatFormat
char *progname
char *transid
void *env
```

Where:

jhpatFormat

This parameter is set to 1. Other values are reserved.

*progname

contains the program name in a null-terminated string.

*transid

contains the transaction identifier in a null-terminated string.
*env

  is a pointer to the JNIEnv

Note:  The Java program object may need to call a native program to access any
variables that you set with setenv commands in this module.

Security

  CICS program security checking occurs only once when a program is first loaded
into an enclave. Subsequent reuse of the same program does not cause an
additional check.

  Native MVS resources, such as HFS files and sockets are checked by MVS using
the address space USERID, not that of the end-user.

Problem determination

  Messages generated by Language Environment are sent to stderr, which CICS
routes to the CESE transient data queue. Application output to stderr is also routed
to CESE.
Bibliography

CICS Transaction Server for z/OS

The above titles are the only unlicensed books available in hardcopy for CICS Transaction Server for z/OS Version 2 Release 1. All the remaining CICS and CICSPlex SM books are supplied in softcopy only in the CICS Information Center, which is distributed on CD-ROM.

CICS books for CICS Transaction Server for z/OS

General
- CICS User's Handbook
- CICS Transaction Server for z/OS Glossary

Administration
- CICS System Definition Guide
- CICS Customization Guide
- CICS Resource Definition Guide
- CICS Operations and Utilities Guide
- CICS Supplied Transactions

Programming
- CICS Application Programming Guide
- CICS Application Programming Reference
- CICS System Programming Reference
- CICS Front End Programming Interface User's Guide
- CICS C++ OO Class Libraries
- CICS Distributed Transaction Programming Guide
- CICS Business Transaction Services
- Java Applications in CICS

Diagnosis
- CICS Problem Determination Guide
- CICS Messages and Codes
- CICS Diagnosis Reference
- CICS Data Areas
- CICS Trace Entries
- CICS Supplementary Data Areas

Communication
- CICS Intercommunication Guide
- CICS Family: Interproduct Communication
- CICS Family: Communicating from CICS on System/390
- CICS External Interfaces Guide
- CICS Internet Guide

Special topics
- CICS Recovery and Restart Guide
- CICS Performance Guide
- CICS IMS Database Control Guide
- CICS RACF Security Guide
CICSPlex SM books for CICS Transaction Server for z/OS

General
- CICSPlex SM Concepts and Planning
- CICSPlex SM User Interface Guide
- CICSPlex SM Commands Reference Summary
- CICSPlex SM Web User Interface Guide

Administration and Management
- CICSPlex SM Administration
- CICSPlex SM Operations Views Reference
- CICSPlex SM Monitor Views Reference
- CICSPlex SM Managing Workloads
- CICSPlex SM Managing Resource Usage
- CICSPlex SM Managing Business Applications

Programming
- CICSPlex SM Application Programming Guide
- CICSPlex SM Application Programming Reference

Diagnosis
- CICSPlex SM Resource Tables Reference
- CICSPlex SM Messages and Codes
- CICSPlex SM Problem Determination

Other CICS books
- Designing and Programming CICS Applications
- CICS Application Migration Aid Guide
- CICS Family: API Structure
- CICS Family: Client/Server Programming
- CICS Transaction Gateway for OS/390 Administration
- CICS Family: General Information
- CICS 4.1 Sample Applications Guide
- CICS/ESA 3.3 XRF Guide

Note: The CICS Transaction Server for OS/390: Planning for Installation book that was part of the library for CICS Transaction Server for OS/390, Version 1 Release 3, is now merged with the CICS Transaction Server for z/OS Installation Guide. If you have any questions about the CICS Transaction Server for z/OS library, see CICS Transaction Server for z/OS Installation Guide which discusses both hardcopy and softcopy books and the ways that the books can be ordered.

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.
Index

Special Characters
“Hello World” sample application
installation
on CICS 129
on the Web server 130
prerequisites 128
supplied components 128
testing 132
what it does 127

A
abend codes
CICS EJB 224
accessing databases
from enterprise beans 97
application assembler, of EJB application 100
Application programs
Java 13
authentication
IIOP 65

B
bean-managed entity beans 94
bean provider 100
bindings 184

C
CICS code generation utility
installing and configuring 156
CICS Connector for CICS TS
benefits of 209
changes to CICS externals
messages 221
trace points 221
data conversion 215
how to use 210
installation 220
overview 205
requirements 220
restrictions 219
setting object attributes 211
using the CCF interface 210
using the CTG API 216
using with VisualAge for Java 213
CICS-deployed JAR file 237
CICS deployment tools
online help problems 223
CICS development deployment tool (continued)
REQUESTMODEL generation 183
security 168
servlet 160
user ID 168
web application
configuring 159
installing 158
with multiple CICS regions 169
CICS Development Deployment Tool
messages 224
CICS EJB security
Corbaserver 66
TCPIPSERVICE resource 67
CICS enterprise beans
file access permissions 232
CICS JAR development tool
installing and configuring 156
CICS JVM messages 224
CICS production deployment tool
installing and configuring 170
CICS security
files to be defined 233
resource definitions 66
system initialization parameters 65
CICSCGU 179
CICSplex SM support 237
CIRP transID 183
code generation tool 179
Connection optimization
registration 62
connectors
CICS Connector for CICS TS
benefits of 209
data conversion 215
how to use 210
installation 220
overview 205
requirements 220
setting object attributes 211
using with VisualAge for Java 213
container-managed entity beans 94
container plugin 52
CORBA 55
exceptions 226
the Object Request Broker 55
CORBA debug plugin 52
Corbaserver
resource in CICS EJB security 66
CorbaServer 186, 237

D
Data Access beans
described 98
DCF 162
DCF.DTD 163, 165
DebugControl interface 51
debugging
  in the JVM 47
Debugging CICS JAVA applications 47
Debugging Java applications 225
deployer, of EJB application 100
deploying enterprise beans 101, 173
code generation 176
deployment configuration file 162
deployment descriptor 176
deployment tools 173
deployment configuration file 162
deployment descriptor editing 175, 177
deployment tools 173
  CICS code generation utility 179
  CICS development deployment tool 157, 181
  CICS JAR development tool 156, 175, 177
  CICS production deployment tool 170
  installing and configuring 155
  prerequisite hardware 156
  prerequisite software 156
  service updates 155

deriving distinguished names 234
DFHABBD 167
DFHADFD 167
DFHADJAR 167
DFHADJM 167
DFHADPD 167
DFHEJDIR 59
DFHEJDIR, EJB request streams directory file 106, 114
DFHEJDIR file 233
DFHEJDNX user-replaceable module 234
DFHEJOS, EJB passivated session beans file 106, 114
DFHEJOS file 233
DFHXOPUS 79
diagnostic services
  JVM trace 224
distinguished names
  deriving 234
  obtaining 234
DJAR 237
  and the CICS development deployment tool 182
DNS
  connection optimization 62
  name resolution 63
DNS connection optimization
  resource definition 64
Domain Name System connection optimization 62

E

EJB
  abend codes 224
  migrating EJB 1.0 enterprise beans to EJB 1.1 157
EJB CICS sample application
  installation
    on the Web server 139
    on z/OS 137
  prerequisites 135
  supplied components 136
  testing 141
  what it does 134
EJB client
  messages 225
EJB container 90
EJB Installation Verification Program
  installation 123
  on CICS 123
  on Unix System Services 124
  introduction 123
  prerequisites 123
  running 125
ejb-ref binding 184
EJB server 90
EJB standard interface classes 157
EJCOBEAN 239
EJCODEF 237
EJCOS 239
EJDJAR 239
EJDJBEAN 239
EJDJDEF 237
EJDJBEAN 239
template bean 237
template beans
  accessing databases 97
  benefits 110
  CICS Connector for CICS TS 205
  client program 149
  configuring CICS server 103
  deployment 101
  deployment checklist 145
  deployment descriptor 92
  deployment tools 173
  described 89
  development roadmap 145
  EJB container 90
  EJB server 90
entity beans
  bean-managed 94
  comparison with session beans 95
  container-managed 94
  described 94
  primary key 94
  environment 92
  example pseudocode 108
  home interface 91
  in a sysplex 103
  managing transactions 96
  overview 88
  remote interface 91
  requirements 111
sample programs
  “Hello World” application 127
  EJB CICS application 133
    introduction 127
security 99
session beans
  code example 146
  comparison with entity beans 95
  described 93
enterprise beans  (continued)
session beans  (continued)
  stateful  94
  stateless  94
  writing  146
setting up a logical EJB server  105
setting up an EJB server  113
  multi-region  120
  single-region  113
  testing the server  119
use of Data Access beans  98
user tasks
  application assembler  100
  bean provider  100
  deployer  100
  system administrator  101
workload balancing  103
  writing  145
Enterprise beans
  errors and messages  224
  security  231
  security policy  230
  security roles  231
Enterprise Java domain messages  224
Enterprise JavaBean  237
entity beans
  bean-managed  94
  comparison with session beans  95
  container-managed  94
  described  94
  primary key  94
example pseudocode, for EJB clients  108

F
file access permissions
  for CICS enterprise beans  232

G
GenFacIOR migration  256

H
home interface, of enterprise beans  91

I
  IDL  249
  IDL migration  256
IIOP  (continued)
  DNS connection optimization  61, 62
  dynamic routing  83
  enterprise beans  56
  HelloWorld sample  261
  IDL  249
  IDL example  252
  locateRequest  60
  message fragments  60
  message processing  59
  MessageError  60
  messages>32k  256
  migrating from CICS TS 1.3  255
  migrating from TS 1.3  257
  Obtaining a USERID  79
  programming model  247
  request message  59
  request receiver  59
  REQUESTMODEL processing  82, 83
  sample applications  257
  sample program components  257
  security URM  68
  server development procedure  250
stateless CORBA objects  56
TCP/IP Listener  59, 72
TCPIPSERVICE  72
the ORB  55
using RACF  65
using SSL  65
IIOP client example  253
InstallShield
  installing CICS deployment tools for EJB technology  155

J
Java
  ADDRESS  20
  ASSIGN  21
  Class  14
  class library (JCICS)  13
  command arguments  15
  condition handling  18
  debugging  225
  DFHRPL requirements  268
Errors  14
Exceptions  14
hot-pooling  271
INQUIRE SYSTEM  21
INQUIRE TASK  21
INQUIRE TERMINAL or NETNAME  22
interface  14
Java Record Framework  15
JavaBeans  14
JCICS command reference  16
plugins  50
PrintWriter  16
protecting applications  227
resource definition  15
run-time requirements  268
running  268

Index  287
Java (continued)
  sample program installation 31
  sample programs 31
  System.err 16
  System.out 16
  threads 16
  transient data 26
  translation 14
  unsupported CICS services 27
Java 2 security manager 229
Java Debug Interface 48
Java Debug Wire Protocol 48
Java hot-pooling 271
  H8 TCB 272
  LE options 273
  storage usage 273
  using 273
Java Platform Debugger Architecture 48
Java support
  accessing databases 97
CICS Connector for CICS TS
    benefits of 209
    data conversion 215
    how to use 210
    installation 220
    overview 205
    requirements 220
    restrictions 219
    setting object attributes 211
    using 273
Data Access beans 98
  enterprise beans
    benefits 110
    deployment 101, 173
    deployment descriptor 92
    described 89
    EJ B container 90
    EJB server 90, 103
    entity beans 94
    environment 92
    example pseudocode 108
    home interface 91
    managing transactions 96
    overview 88
    remote interface 91
    requirements 111
    security 99
    session beans 93
    setting up an EJ B server 105
    user tasks 99
JavaBeans
  described 89
Java virtual machine
  debugging 47
  XDFHENV data set 47
Java Virtual Machine Debug Interface 48
Java2 Security 227
JavaBeans
  described 89
Javadoc 247
JCICS
  ABEND handling 17
  abnormal termination 18
  APPC 19
  BMS 19
  CANCEL command 25
  creating objects 29
  DEQ command 25
  diagnostic services 19
  ENQ command 25
  error handling 18
  exception handling 17
  file control 22
  HANDLE commands 18
  Javadoc 247
  program control 24
  RETRIEVE command 25
  sample program installation 31
  sample programs 31
  START command 25
  storage services 25
  temporary storage 25
  Terminal Control 26
  UOWs 27
  Using 28
  using objects 29
JCICS class library 13
JCICS command reference 16
JDI 48
JDWP 48
JNDI 203
  set up 156
JPDA 48
JVM
  plugins 50
  problem determination 224
  using a debugger 47
JVM initialization options 49
JVM messages 224
JVM profiles
  defining to CICS 47
  JVM trace 224
JVMDI 48
L
  load balancing 60
  logical EJB server
    described 103
    setting up 105
      a multi-region server 120
      a single-region server 113
    testing the server 119
M
  messages
    CICS Connector for CICS TS 221
Migration from TS 1.3 255
O
open transaction environment 271
ORB function 60
OTS transaction 59

P
permissions (system access privileges) 227
Plugin interface 51
POSIX characters 158
primary key, entity beans 94

R
RACF definitions
  to configure CICS for security 233
remote interface, of enterprise beans 91
request model
  and CICS deployment tools 183
request stream 59
REQUESTMODEL 82
  examples 83
  pattern matching 83
resource-ref binding 184

S
sample programs
  “Hello World” sample
    installation 129
    prerequisites 128
    supplied components 128
    testing 132
    what it does 127
EJB CICS sample
  installation 137
  prerequisites 135
  supplied components 136
  testing 141
  what it does 134
EJB IVP
  installation 123
  introduction 123
  prerequisites 123
  running 125
secure sockets layer (SSL) 99
security
  CICS development deployment tool 168
security manager
  applying a security policy 229
  enabling 229
security policy
  security manager
    application 229
session beans
  comparison with entity beans 95
  described 93
  stateful 94
  stateless 94
SSL
  system initialization parameters 65
    with IIOP 65
  stateful session beans 94
  stateless CORBA objects 247, 257
  stateless session beans 94
  system access privileges (permissions) 227
T
TCB
  for hot-pooling 271
TCP/IP Listener 72
TCPIPSERVICE
  resource in CICS EJB security 67
TCPIPSERVICE resource 72
trace points
  CICS Connector for CICS TS 221
U
undeploy 186
UNIX System Services
  and enterprise bean deployment 168
V
VisualAge for Java 213
VisualAge for Java Enterprise Toolkit for OS/390 267
W
WebSphere Application Server
  configuring the CICS development deployment tool
    web application 159
    installing and configuring 158
    prerequisites 156
  wrapper plugin 52
X
XML 162
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

The following terms are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

<table>
<thead>
<tr>
<th>Term</th>
<th>IBM</th>
<th>VisualAge</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIX</td>
<td>IBM</td>
<td>VisualAge</td>
</tr>
<tr>
<td>CICS</td>
<td>OS/390</td>
<td>WebSphere</td>
</tr>
<tr>
<td>CICSPlex</td>
<td>RACF</td>
<td></td>
</tr>
<tr>
<td>DB2</td>
<td>SecureWay</td>
<td></td>
</tr>
</tbody>
</table>

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

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Microsoft, Windows, and Windows NT, are trademarks of Microsoft Corporation in the United States, or other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.
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:
  
  User Technologies Department (MP095)
  IBM United Kingdom Laboratories
  Hursley Park
  WINCHESTER
  Hampshire
  SO21 2JN
  United Kingdom

• By fax:
  – From outside the U.K., after your international access code use 44–1962–842327
  – From within the U.K., use 01962–842327

• Electronically, use the appropriate network ID:
  – IBM Mail Exchange: GBIBM2Q9 at IBMMAIL
  – 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.